Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit im Studiengang Physik

vorgelegt von

Michael Krieger

2011

Diplomarbeit

Entwurf und Simulation eines digitalen Tail-Cancellation-Filters

Michael Krieger

September 2011

ausgeführt am

Lehrstuhl für Schaltungstechnik und Simulation

unter der Betreuung von

Prof. Dr. Peter Fischer und Tim Armbruster

Abstract

The SPADIC readout eletronics for use in the transition radiation detector system within the CBM experiment is currently under development. This work contributes to the data processing chain of the SPADIC system by introducing a concept for a digital tail cancellation filter, which is going to be realized in the next version of the chip.

A mathematical model for the process of tail cancellation has been developed, from which a method for choosing the filter coefficients is derived. The implementation of the filter as a digital circuit, in particular the synthesis of fast and compact multipliers, has been addressed. It has turned out that, with the inclusion of a new full adder cell, the fully automatic synthesis of arithmetic circuits is superior to a semi-automatic method under investigation. In order to analyze the quantization error that is introduced in the digital signal processing that takes place in the filter, a simulation software has been written and used to determine the optimal internal filter resolution.

Zusammenfassung

Für die Auslese der Übergangsstrahlungsdetektoren, die einen Teil des CBM-Experiments bilden, wird zurzeit die Ausleseelektronik SPADIC entwickelt. Als Teil der Datenverarbeitungskette des SPADIC-Systems wird in der nächsten Version des Chips ein digitales Tail-Cancellation-Filter enthalten sein, wofür in dieser Arbeit ein Konzept vorgestellt wird.

Zur Beschreibung der Tail Cancellation wurde ein mathematisches Modell entwickelt, anhand dessen eine Methode zur Wahl der Filterkoeffizienten abgeleitet wird. Weiterhin befasst sich diese Arbeit mit der Implementierung des Filters als digitale Schaltung, wobei es insbesondere um die Synthese von schnellen und kompakten Multiplizierern geht. Es hat sich herausgestellt, dass, unter Verwendung einer neuen Volladdiererzelle, die vollautomatische Synthese arithmetischer Schaltungen einem halbautomatischen Verfahren, das untersucht wurde, überlegen ist. Um den Quantisierungsfehler zu untersuchen, der bei der digitalen Signalverarbeitung im Filter entsteht, wurde eine Simulationssoftware geschrieben und eingesetzt, um die optimale interne Auflösung des Filters zu bestimmen.

Inhaltsverzeichnis

1 Einleitung

An der Beschleunigeranlage *FAIR (Facility for Antiproton and Ion Research)*, das sich beim GSI Helmholtzzentrum für Schwerionenforschung in Planung befindet, wird das *CBM*-Experiment *(Compressed Baryonic Matter)* stattfinden, bei dem neue Erkenntnisse über das Verhalten stark wechselwirkender Materie im Zustand höchster Dichte gewonnen werden sollen. Zu diesem Zweck werden mithilfe des neuen Beschleunigers schwere Ionen zur Kollision gebracht und mit einem Detektorsystem, das dafür entwickelt wird, die Reaktionsprodukte nachgewiesen [1].

Einen Teil dieses Detektorsystems bilden *Übergangsstrahlungsdetektoren* (engl. *Transition Radiation Detector* oder abgekürzt *TRD*), mit denen Elektronen von Pionen unterschieden, sowie die Spuren geladener Teilchen im Allgemeinen rekonstruiert werden sollen. Die TRDs nutzen den Effekt der *Übergangsstrahlung* aus, die beim Durchgang geladener Teilchen durch eine Abfolge von Medien mit verschiedenen Brechungsindizes (*Radiatoren* aus übereinandergelegten Folien verschiedener Materialien) entsteht, um Teilchensorten zu identifizieren, denn die Intensität der entstehenden Strahlung hängt vom Gammafaktor der Teilchen und somit bei gleichem Impuls von deren Masse ab. Um die Übergangsstrahlung sowie den Durchgang der Teilchen selbst nachzuweisen, werden Vieldraht-Proportionalkammern (*MWPC* für engl. *Multiwire Proportional Chamber*) verwendet, mit denen zusätzlich durch die Aufstellung in mehreren Lagen eine zweidimensionale Ortsauflösung senkrecht zur Strahlrichtung möglich ist [2].

Die in den MWPCs entstehenden Signale werden von dem elektronischen Auslesesystem *SPADIC (Self Triggered Pulse Amplification and Digitization ASIC)* [3] aufgenommen, das seit mehreren Jahren am Lehrstuhl für Schaltungstechnik und Simulation des ZITI Heidelberg entwickelt wird. Das SPADIC-System zeichnet sich dadurch aus, dass die Detektorsignale auf einem Chip sowohl analog verstärkt als auch digitalisiert und im digitalen Bereich weiterverarbeitet werden. Das System erkennt selbstständig die relevanten Teile (die *Hits*) des kontinuierlich eintreffenden Signals und gibt diese für die spätere Auswertung in Form von Datenpaketen in einem speziellen Protokoll nach außen weiter. In Abbildung 1.1 ist ein Prototyp des SPADIC-Chips auf der zugehörigen Platine gezeigt.

Die Eigenheiten der Signalentstehung in den MWPCs bewirken, dass es aufgrund der sogenannten *Ion Tails* zum *Pile-Up*-Effekt kommt, der störend für die Erkennung der einzelnen *Hits* ist. Daher sollte für die nächste Version des SPADIC-Chips ein digitales *Tail-*

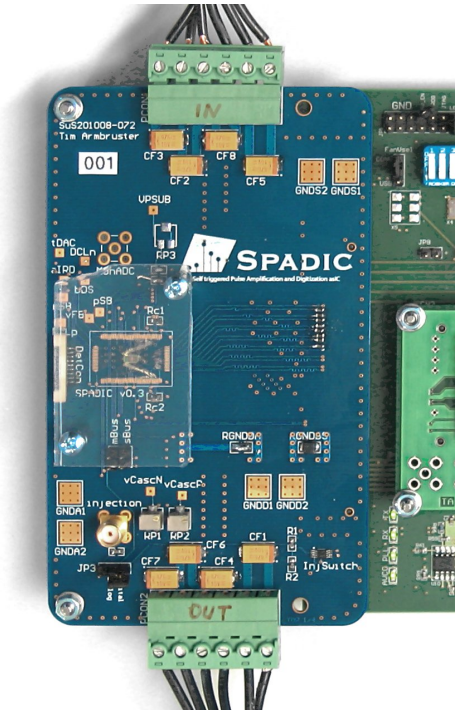

Abbildung 1.1: Prototyp des SPADIC-Chips und der zugehörigen Platine in einer Version vom Oktober 2010 (Foto: [4])

Cancellation-Filter entwickelt werden, mit dem sich dieses Problem beheben lässt. Dies ist der Inhalt dieser Arbeit.

Da die *Tail Cancellation* eine Anwendung der digitalen Signalverarbeitung ist, werden in Kapitel 2 zunächst die dafür notwendigen mathematischen Grundlagen behandelt. Das Ergebnis dieses Teils der Arbeit werden die *Pulsfolgen* sein, die zur Modellierung der Signale, sowie des Filters selbst, nützlich sind.

Anschließend werden in Kapitel 3 die Entstehung und mathematische Beschreibung der Signale der MWPCs dargestellt und eine Filterstruktur beschrieben, mit der die unerwünschten Eigenschaften der Signale unterdrückt werden können.

In Kapitel 4 wird erläutert, wie das Filter als digitale Schaltung implementiert wird. Dabei geht es zu einem großen Teil um die Synthese von Schaltungen, die die arithmetische Operation der Multiplikation durchführen können.

Die Simulation des Filters ist wichtig, um die Quantisierungseffekte, die bei der Signalverarbeitung auftreten, einschätzen zu können. Dafür wurde eine Software entwickelt, die im Kapitel 5 vorgestellt wird. In Kapitel 6 kommt die Simulation bei der Messung der Quantisierungsfehler zur Anwendung.

2 Mathematische Grundlagen

Die Vorgänge, die bei der digitalen Signalverarbeitung stattfinden, sind ohne Mathematik kaum zu begreifen. Mit Hilfe von mathematischen Konzepten ist es möglich, die Eigenschaften und das Verhalten von Signalen und signalverarbeitenden Systemen zu beschreiben und Zusammenhänge herzustellen. Deshalb wird hier auf einige Begrifflichkeiten eingegangen, die die theoretische Grundlage dieser Arbeit bilden.

Im Einzelnen geht es um Folgen, ihre Repräsentation durch die z-Transformation und zeitdiskrete Systeme. Die folgenden Abschnitte richten sich weitgehend nach den Darstellungen in [5–7], wo diese Themen in größerem Umfang behandelt werden.

Im Bereich der Theorie der zeitdiskreten Signale und Systeme gibt es viele Aspekte, die sich auf unterschiedliche Weisen ausdrücken lassen. Mit der Einführung von neuen mathematischen Objekten und deren Eigenschaften können jeweils Zusammenhänge zu bereits bekannten Objekten hergestellt werden. Mit zunehmender Anzahl von neuen Begriffen ist es klar, dass es dabei schwierig ist, den Überblick darüber zu behalten, welche Aussagen gleichbedeutend sind, unter welchen Bedingungen sie gelten und welche Folgerungen aus einer Aussage hervorgehen. Es wird versucht, dies, soweit es möglich ist, übersichtlich und strukturiert darzustellen.

2.1 Folgen

Die mathematischen Objekte, die die Eingangs- und Ausgangssignale eines Filters repräsentieren, sind *Folgen* [5, S. 10]. Da später oft die Rede von Folgen sein wird, lohnt es sich, hier zunächst einmal festzuhalten, worum es sich dabei genau handelt, und die in diesem Zusammenhang verwendeten Schreibweisen und Bezeichnungen einzuführen.

2.1.1 Definition

Unter einer Folge wird hier eine Abbildung von den ganzen Zahlen in die komplexen Zahlen verstanden: Eine Folge x ordnet jeder ganzen Zahl n eine komplexe Zahl $x[n]$ zu.

$$\begin{aligned} x &: \mathbb{Z} \to \mathbb{C} \\ n &\mapsto x[n] \end{aligned}$$

Mit x ist also die Folge als Ganzes bezeichnet, mit $x[n]$ das einzelne Folgenglied an der Stelle oder dem *Index n*. In der Interpretation einer Folge als Signal stellt n die Zeit dar, für $n_1 < n_2$ ist also $x[n_1]$ der Wert irgendeiner Größe zu einer gewissen Zeit und $x[n_2]$ der Wert dieser Größe zu einer späteren Zeit.

2.1.2 Operationen mit Folgen

Folgen können durch Verknüpfung mit anderen Folgen oder durch Anwenden von Operatoren in andere Folgen umgewandelt werden.

Skalare Multiplikation

Die Multiplikation einer Folge x mit einer komplexen Zahl a ergibt eine Folge y, deren Folgenglieder durch Multiplikation jedes Folgenglieds von x mit a entstehen:

$$y = a \cdot x \quad \Leftrightarrow \quad \forall n \in \mathbb{Z} : y[n] = a \cdot x[n] \tag{2.1}$$

Verzögerung

Eine Folge y heißt um eine ganze Zahl k gegenüber einer Folge x *verzögert*, wenn gilt:

$$\forall n \in \mathbb{Z} : y[n] = x[n-k] \tag{2.2a}$$

Fasst man die Verzögerung um k als Operator $\mathscr{D}_k$ auf, der auf die Folge x wirkt, lässt sich das so schreiben:

$$y = \mathscr{D}_k x \tag{2.2b}$$

Addition und Multiplikation

Wenn zwei Folgen x_1 und x_2 miteinander durch Addition oder Multiplikation verknüpft sind, bedeutet das, dass jeweils alle Folgenglieder paarweise addiert bzw. multipliziert werden und daraus eine neue Folge y gebildet wird:

$$y = x_1 + x_2 \quad \Leftrightarrow \quad \forall n \in \mathbb{Z} : y[n] = x_1[n] + x_2[n] \tag{2.3}$$

$$y = x_1 \cdot x_2 \quad \Leftrightarrow \quad \forall n \in \mathbb{Z} : y[n] = x_1[n] \cdot x_2[n] \tag{2.4}$$

Faltung

Zwei Folgen x_1 und x_2 können durch die *Faltung* miteinander zu einer neuen Folge $y = x_1 * x_2$ verknüpft werden. Die Folgenglieder von y entstehen durch die folgende Berechnungsvorschrift:

$$y = x_1 * x_2 \quad \Leftrightarrow \quad \forall n \in \mathbb{Z} : y[n] = \sum_{k=-\infty}^{\infty} x_1[k]\, x_2[n-k] \tag{2.5a}$$

Die Faltung ist also eine Kombination von Skalierung, Verzögerung und Addition:

$$x_1 * x_2 = \sum_{k=-\infty}^{\infty} a_k \mathscr{D}_k x_2 \quad \text{mit } a_k = x_1[k] \tag{2.5b}$$

2.1.3 Rechenregeln

Aus den Definitionen folgt, dass die Addition, Multiplikation und Faltung zweier Folgen kommutativ und assoziativ sind, d. h. für die Verknüpfungen $\bullet \in \{+, \cdot, *\}$ gilt, falls x, y und z Folgen sind:

$$x \bullet y = y \bullet x \tag{2.6a}$$
$$(x \bullet y) \bullet z = x \bullet (y \bullet z) \tag{2.6b}$$

In Kombination mit der skalaren Multiplikation mit einer Zahl $a \in \mathbb{C}$ gilt beim Addieren zweier Folgen x und y das Distributiv-, beim Multiplizieren und Falten hingegen das Assoziativgesetz:

$$a\,(x + y) = a\,y + a\,x \tag{2.7a}$$
$$a\,(x \cdot y) = (a\,x) \cdot y = x \cdot (a\,y) \tag{2.7b}$$
$$a\,(x * y) = (a\,x) * y = x * (a\,y) \tag{2.7c}$$

Die Rechenregeln im Zusammenhang mit dem Verzögerungsoperator erscheinen zunächst trivial (seien x und y Folgen, a eine komplexe Zahl),

$$\mathscr{D}_k(\mathscr{D}_l x) = \mathscr{D}_l(\mathscr{D}_k x) = \mathscr{D}_{k+l} x \tag{2.8a}$$
$$\mathscr{D}_k(a\,x) = a\,(\mathscr{D}_k x) \tag{2.8b}$$

sind aber vor allem bei der Addition, Multiplikation und Faltung zweier Folgen doch nicht

unmittelbar einzusehen und deshalb hier einmal festgehalten:

$$\mathscr{D}_k(x+y) = \mathscr{D}_k x + \mathscr{D}_k y \tag{2.8c}$$

$$\mathscr{D}_k(x \cdot y) = (\mathscr{D}_k x) \cdot (\mathscr{D}_k y) \tag{2.8d}$$

$$\mathscr{D}_k(x * y) = (\mathscr{D}_k x) * y = x * (\mathscr{D}_k y) \tag{2.8e}$$

2.1.4 Spezielle Folgen

Es gibt bestimmte Folgen, die eine besondere Bedeutung haben und deshalb nun im Einzelnen genannt werden.

Nullfolge

Die Nullfolge $\mathscr{O}$ ist die Folge, die für alle n den Wert Null hat:

$$\forall n \in \mathbb{Z} : \mathscr{O}[n] = 0 \tag{2.9}$$

Sie ist das neutrale Element der Addition, d. h. für eine beliebige Folge x ist $x + \mathscr{O} = x$:

$$\forall n \in \mathbb{Z} : (x + \mathscr{O})[n] = x[n] + 0 = x[n] \tag{2.10a}$$

Sowohl die Multiplikation der Nullfolge mit einer beliebigen komplexen Zahl a oder einer beliebigen Folge x als auch die skalare Multiplikation einer beliebigen Folge x mit der Null ergeben die Nullfolge:

$$a\mathscr{O} = x \cdot \mathscr{O} = 0\, x = \mathscr{O} \tag{2.10b}$$

Einheitsimpuls

Die Folge, die für $n = 0$ den Wert Eins und ansonsten den Wert Null hat, ist der *Einheitsimpuls* und wird mit δ bezeichnet.

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{2.11}$$

Der Einheitsimpuls ist das neutrale Element der Faltung, d. h. für eine beliebige Folge x ist $x * \delta = x$:

$$(x * \delta)[n] = \sum_{k=-\infty}^{\infty} x[k]\,\delta[n-k] = x[k]\,\delta[n-k]\Big|_{k=n} = x[n] \tag{2.12a}$$

Dieselbe Aussage anders formuliert bedeutet: *Jede beliebige Folge x lässt sich als Überlagerung von verzögerten und skalierten Einheitsimpulsen ausdrücken:*

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]\,\delta[n-k] \quad \Leftrightarrow \quad x = \sum_{k=-\infty}^{\infty} a_k \mathscr{D}_k \delta \quad \text{mit } a_k = x[k] \tag{2.12b}$$

Einheitssprung

Der *Einheitssprung u* ist die Folge, die für alle negativen Zahlen den Wert Null und ansonsten den Wert Eins hat:

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases} \tag{2.13}$$

Somit besteht zwischen dem Einheitsimpuls und dem Einheitssprung die folgende Beziehung:

$$u[n] = \sum_{k=-\infty}^{n} \delta[k] \tag{2.14}$$

Exponentialfolgen

Eine Folge x ist eine *Exponentialfolge*, falls es eine komplexe Zahl $q \neq 0$ gibt, so dass gilt:

$$\forall\, n \in \mathbb{Z} : x[n] = q^n \tag{2.15}$$

Die Zahl q heißt *Basis* der Exponentialfolge. Alle Exponentialfolgen haben an der Stelle $n = 0$ den Wert $q^0 = 1$. Wenn q eine reelle Zahl ist, sind auch alle Folgenglieder reell. Ansonsten gilt für eine beliebige Exponentialfolge x mit der Basis $q = r\,e^{i\phi}$:

$$x[n] = r^n\, e^{i\,n\phi} = r^n \left(\cos n\phi + i \sin n\phi\right) \tag{2.16}$$

Pulsfolgen

In einem späteren Abschnitt werden die *Pulsfolgen* σ_q^k eingeführt, die eine Verallgemeinerung der Exponentialfolgen und des Einheitsimpulses darstellen. Auch hier heißt die komplexe Zahl q die *Basis*. Die ganze Zahl k ist der *Grad* der Pulsfolge.

Die Definition dieser Folgen kann hier noch nicht vorgenommen werden, aber schon jetzt sei gesagt, dass die Pulsfolgen vom Grad $k = 1$ mit den Exponentialfolgen mit der Basis q im Zusammenhang stehen, und dass die Pulsfolgen mit dem Grad $k = 0$ oder der Basis $q = 0$ der Einheitsimpuls sind.

2.1.5 Eigenschaften von Folgen

Folgen können anhand bestimmter Merkmale gruppiert werden, so dass sich Aussagen formulieren lassen, die für alle Folgen gelten, die ein gewisses Merkmal aufweisen. Einige dieser Merkmale werden nun vorgestellt.

Beschränktheit

Eine Folge x ist *beschränkt*, wenn es eine positive reelle Zahl B gibt, so dass gilt [5, S. 23]:

$$\forall n \in \mathbb{Z} : |x[n]| \leq B \tag{2.17}$$

Jede Exponentialfolge $x[n] = q^n$ mit $|q| \neq 1$ ist z. B. nicht beschränkt, weil mit zunehmendem oder abnehmendem n (je nachdem, ob $|q|$ größer oder kleiner als Eins ist) die Folgenglieder jeden Wert übersteigen. Der Einheitsimpuls und der Einheitssprung sind hingegen beschränkt, weil $|\delta[n]| \leq 1$ und $|u[n]| \leq 1$ für alle n gilt.

Rechts- und Linksseitigkeit

Mit den folgenden Eigenschaften werden Folgen gekennzeichnet, die nur auf einem bestimmten Teil des Definitionsbereichs $\mathbb{Z}$ von Null verschieden sind [5, S. 192 f.].

Eine Folge x heißt *rechtsseitig*, wenn es eine ganze Zahl N gibt, so dass gilt:

$$\forall n < N : x[n] = 0 \tag{2.18a}$$

Analog dazu heißt eine Folge x *linksseitig*, wenn es eine ganze Zahl N gibt, so dass gilt:

$$\forall n > N : x[n] = 0 \tag{2.18b}$$

Davon abgeleitet gibt es die beiden weiteren Eigenschaften, die besonders benannt sind:

- Eine Folge ist *endlich*, wenn sie *rechtsseitig und linksseitig* ist. Es gibt also nur endlich viele von Null verschiedene Folgenglieder.
- Eine Folge ist *zweiseitig*, wenn sie *weder rechtsseitig noch linksseitig* ist.

Beispielsweise ist der Einheitsimpuls δ endliche Folge, der Einheitssprung u rechtsseitig und eine Exponentialfolge zweiseitig.

2.2 z-Transformation

2.2.1 Definition

Die z-Transformation ist eine Abbildung $\mathcal{Z}$, die einer Folge x eine komplexwertige Funktion $\mathcal{Z}\{x\} = X$ über einer Teilmenge D der komplexen Zahlen zuordnet [5, S. 179 ff.]:

$$\mathcal{Z} : x \mapsto (X : D \subset \mathbb{C} \to \mathbb{C}) \tag{2.19}$$

Die Funktion X heißt *z-Transformierte* von x und ihr Definitionsbereich D wird aus einem Grund, der gleich ersichtlich wird, *Konvergenzgebiet* (engl. *region of convergence*, abgekürzt *ROC*) genannt. In Abbildung 2.1 ist grafisch angedeutet, wie zwei verschiedene Folgen jeweils in eine Funktion über einer von der Folge abhängigen Teilmenge der komplexen Zahlen abgebildet werden.

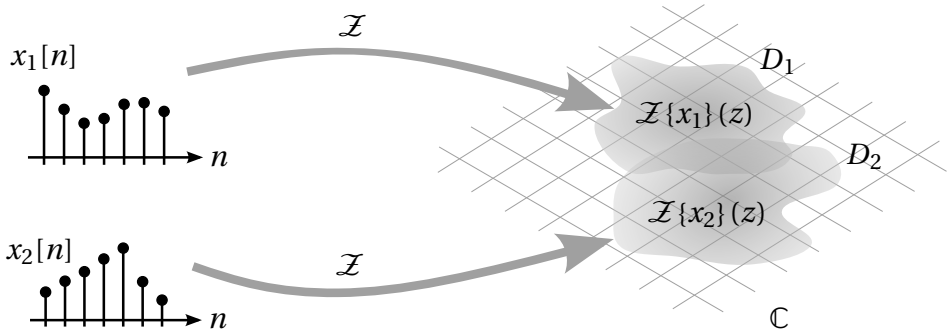

Abbildung 2.1: Die z-Transformation bildet eine Folge x in eine komplexwertige Funktion $\mathcal{Z}\{x\}$ über einer Teilmenge D von $\mathbb{C}$ ab.

Die Berechnungsvorschrift für die Funktionswerte $X(z)$ der z-Transformierten einer Folge x lautet:

$$\mathcal{Z}\{x\}(z) = X(z) = \sum_{n=-\infty}^{\infty} x[n]\, z^{-n} \tag{2.20}$$

Diese unendliche Reihe konvergiert für eine gegebene Folge x im Allgemeinen nicht an jeder Stelle $z \in \mathbb{C}$. Daher ist der Definitionsbereich der Funktion X nur diejenige Teilmenge D der komplexen Zahlen, für die die Reihe konvergiert, weswegen D das Konvergenzgebiet genannt wird.

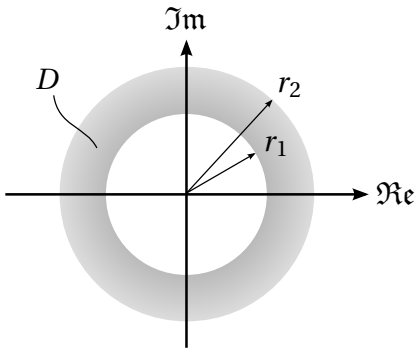

Abbildung 2.2: Das Konvergenzgebiet von $X(z)$ ist ein Kreisring in der komplexen Zahlenebene.

2.2.2 Eigenschaften

Konvergenzgebiet

Die Reihe aus der Definition (2.20) der z-Transformierten ist eine *Laurent-Reihe*. Als solche hat ihr Konvergenzgebiet die aus der komplexen Analysis bekannte Eigenschaft: [8, S. 9, S. 56]

Es gibt zwei eindeutig bestimmte reelle Zahlen r_1 und r_2 mit $0 \le r_1, r_2 \le \infty$, so dass $X(z)$ für alle z aus der Menge

$$D = \{z \mid r_1 < |z| < r_2\} \tag{2.21}$$

absolut konvergiert. Für alle z mit $|z| < r_1$ oder $|z| > r_2$ divergiert die Reihe.

Die Zugehörigkeit eines Punkts $z = r\,e^{i\phi}$ zum Konvergenzgebiet hängt also nur von r, aber nicht von ϕ ab. Wenn z in D liegt, liegen auch alle anderen Punkte z_1 mit $|z_1| = r$ in D, d. h. das Konvergenzgebiet ist ein Kreisring um den Ursprung der komplexen Zahlenebene, wie in Abbildung 2.2 gezeigt.

Es kann sein, dass $r_1 > r_2$ ist. Dann konvergiert die Reihe an keinem Punkt. Außerdem können r_1 und r_2 auch die Werte 0 und ∞[1] annehmen, was bedeutet, dass D das Gebiet innerhalb eines Kreises um den Ursprung (i. A. ohne den Ursprung selbst) oder außerhalb eines solchen Kreises sein kann.

Es gibt einen Zusammenhang zwischen der Links- oder Rechtsseitigkeit einer Folge x und dem Konvergenzgebiet ihrer z-Transformierten X:

- Wenn x rechtsseitig ist, ist $X(z)$ für alle z mit $|z| > r_1$ definiert, d. h. $r_2 = \infty$.

[1] mit $r = \infty$ ist gemeint: $\forall z \in \mathbb{C} : |z| < r$

- Wenn x linksseitig ist, ist $X(z)$ für alle z mit $|z| < r_2$ definiert (außer vielleicht $z = 0$), d. h. $r_1 = 0$.

Das kann man für eine rechtsseitige Folge so herleiten:

Sei x eine rechtsseitige Folge, es gebe also eine Zahl $N \in \mathbb{Z}$, so dass $x[n] = 0$ für alle $n < N$. Substituiert man $k = n - N$, kann man die z-Transformierte von x an der Stelle z so schreiben:

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} x[n]\, z^{-n} \\ &= \sum_{k=-\infty}^{\infty} x[k+N]\, z^{-(k+N)} \\ &= z^{-N} \cdot \sum_{k=0}^{\infty} x[k+N] \left(z^{-1}\right)^k \end{aligned}$$

Es handelt sich also um eine Potenzreihe in z^{-1}, die eine eindeutig bestimmte reelle Zahl als Konvergenzradius hat [8, S. 9]. Bezeichnet man diesen Konvergenzradius mit $\frac{1}{r_1}$, konvergiert die Reihe für alle z mit $|z^{-1}| < \frac{1}{r_1}$. Somit ist der Definitionsbereich von X die Menge $\{z \in \mathbb{C} \mid |z| > r_1\}$.

Im Fall einer linksseitigen Folge hat nach einer ähnlichen Umformung der Term für die z-Transformierte die Form:

$$X(z) = z^{-N} \cdot \sum_{k=0}^{\infty} x[N-k]\, z^k$$

Die Potenzreihe ohne den Vorfaktor z^{-N} konvergiert für alle z, deren Betrag kleiner als der Konvergenzradius r_2 ist. Falls N nicht positiv ist, ist dies gleichzeitig auch das Konvergenzgebiet von X, andernfalls ist X wegen des Vorfaktors z^{-N} für $z = 0$ nicht definiert.

In Abbildung 2.3 ist zusammenfassend gezeigt, was die Konvergenzgebiete von links- und rechtsseitigen Folgen sind.

Anfangswerttheorem

Im besonderen Fall einer rechtsseitigen Folge x, die für alle negativen Indizes den Wert Null hat, gilt das *Anfangswerttheorem* [5, S. 215]:

$$\lim_{z\to\infty} \mathcal{Z}\{x\}(z) = \lim_{z\to\infty} \sum_{k=0}^{\infty} x[k]\, z^{-k} = x[0] \tag{2.22}$$

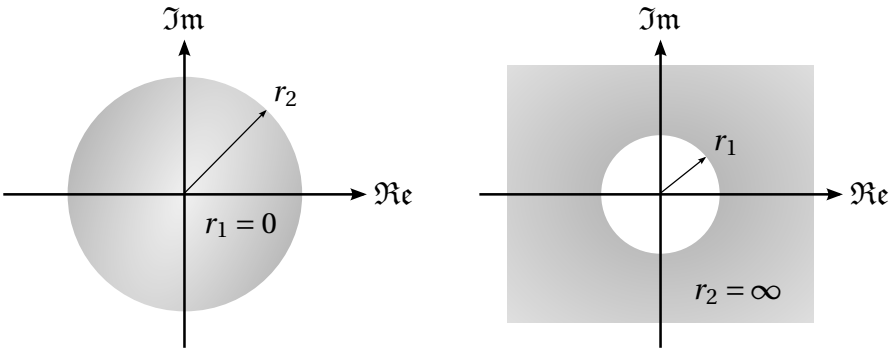

Abbildung 2.3: Konvergenzgebiet der z-Transformierten einer linksseitigen Folge (links) und einer rechtsseitigen Folge (rechts)

2.2.3 Rechenregeln

Es gibt einige wichtige Eigenschaften der z-Transformation, die es erlauben, die z-Transformierte einer Folge zu finden, wenn die Folge durch bestimmte Operationen aus anderen Folgen mit bekannten z-Transformierten entstanden ist.

Im Folgenden seien x_1 und x_2 Folgen, deren z-Transformierte X_1 und X_2 auf den Konvergenzgebieten D_1 und D_2 definiert sind.

Linearität

Aus der Definition (2.20) folgt direkt, dass die z-Transformation linear ist, dass also gilt:

$$\forall a \in \mathbb{C} : \mathcal{Z}\{a\,x_1 + x_2\}(z) = a\,X_1(z) + X_2(z) \tag{2.23}$$

Das Konvergenzgebiet von $\mathcal{Z}\{a\,x_1 + x_2\}$ umfasst mindestens die Schnittmenge der beiden ursprünglichen Konvergenzgebiete D_1 und D_2, kann aber unter Umständen noch erweitert werden [5, S. 208].

Verzögerung

Die Verzögerung einer Folge um k bewirkt die Skalierung ihrer z-Transformierten mit dem Faktor z^{-k}. Das kann man auch aus der Definition herleiten: Sei $x_2 = \mathscr{D}_k x_1$. Dann gilt:

$$\begin{aligned} X_2(z) &= \sum_{n=-\infty}^{\infty} x_1[n-k]\, z^{-n} \\ &= \sum_{m=-\infty}^{\infty} x_1[m]\, z^{-(m+k)} \quad \text{mit } m = n-k \\ &= z^{-k} \sum_{m=-\infty}^{\infty} x_1[m]\, z^{-m} \\ &= z^{-k}\, X_1(z) \end{aligned} \tag{2.24}$$

Das Konvergenzgebiet von X_2 ist das gleiche wie das von X_1, aber unter Umständen kommt wegen des Vorfaktors der Punkt $z = 0$ hinzu oder fällt weg [5, S. 209].

Faltung

Die z-Transformierte der Faltung zweier Folgen ist das Produkt der ursprünglichen z-Transformierten. Dieser Zusammenhang ist von Bedeutung und hat daher den eigenen Namen *Faltungstheorem*.

Man kann das Faltungstheorem so herleiten:

$$\begin{aligned} \mathscr{Z}\{x_1 * x_2\}(z) &= \sum_{n=-\infty}^{\infty} (x_1 * x_2)[n]\, z^{-n} \\ &= \sum_{n=-\infty}^{\infty} \left(\sum_{k=-\infty}^{\infty} x_1[k]\, x_2[n-k] \right) z^{-n} \\ &= \sum_{k=-\infty}^{\infty} x_1[k] \left(\sum_{n=-\infty}^{\infty} x_2[n-k]\, z^{-n} \right) \\ &= \sum_{k=-\infty}^{\infty} x_1[k] \left(\sum_{m=-\infty}^{\infty} x_2[m]\, z^{-(m+k)} \right) \quad \text{mit } m = n-k \\ &= \left(\sum_{k=-\infty}^{\infty} x_1[k]\, z^{-k} \right) \left(\sum_{m=-\infty}^{\infty} x_2[m]\, z^{-m} \right) \\ &= X_1(z)\, X_2(z) \end{aligned} \tag{2.25}$$

Das Konvergenzgebiet von $\mathscr{Z}\{x_1 * x_2\}$ besteht mindestens aus der Schnittmenge von D_1 und D_2 [5, S. 214].

2.2.4 Rücktransformation

Um bei einer gegebenen z-Transformierten $X : D \to \mathbb{C}$ (d. h. bei Angabe von $X(z)$ *und* D) auf die zugrundeliegende Folge x mit $\mathcal{Z}\{x\} = X$ zu schließen, gibt es verschiedene Möglichkeiten, die sich grob in zwei Kategorien einordnen lassen:

- formelle Berechnung mit Methoden aus der komplexen Analysis (z. B. mit Umlaufintegralen [5, S. 216])
- Zurückführen des Funktionsterms $X(z)$ auf bereits bekannte z-Transformierte unter Anwendung der genannten Rechenregeln („Tabellenverfahren“ [5, S. 200])

In der Praxis ist die zweite Möglichkeit von größerer Bedeutung, da die vorkommenden z-Transformierten oft eine Form haben, die das Umwandeln in Grundterme relativ einfach macht.

2.2.5 z-Transformation einiger Folgen

Um das „Tabellenverfahren“ anwenden zu können, müssen die z-Transformierten der grundlegenden Folgen bekannt sein, weswegen sie jetzt hergeleitet werden.

Einheitsimpuls

Die z-Transformierte des Einheitsimpulses δ ist konstant Eins und für alle komplexen Zahlen definiert:

$$\mathcal{Z}\{\delta\}(z) = \sum_{n=-\infty}^{\infty} \delta[n]\, z^{-n} = \delta[n]\, z^{-n}\Big|_{n=0} = 1 \tag{2.26}$$

Einheitssprung

Zur Berechnung der z-Transformierten des Einheitssprungs u benötigt man die Summenformel für die geometrische Reihe [9]:

$$\sum_{n=0}^{\infty} q^n = \frac{1}{1-q} \quad \text{falls } |q| < 1 \tag{2.27}$$

Damit lautet die Formel für $\mathcal{Z}\{u\}$:

$$\mathcal{Z}\{u\}(z) = \sum_{n=0}^{\infty} z^{-n} = \frac{1}{1-z^{-1}} \quad \text{falls } |z^{-1}| < 1 \tag{2.28}$$

Das Konvergenzgebiet ist also die Menge $\{z \in \mathbb{C} \mid |z| > 1\}$.

Einseitige Exponentialfolgen

Eine Folge x mit $x[n] = a^n\, u[n]$, also einer Exponentialfolge, die für $n < 0$ „abgeschnitten“ und damit rechtsseitig ist, hat eine z-Transformierte, die sich ähnlich berechnen lässt:

$$\mathcal{Z}\{x\}(z) = \sum_{n=0}^{\infty} a^n z^{-n} = \sum_{n=0}^{\infty} \left(az^{-1}\right)^n = \frac{1}{1-az^{-1}} \quad \text{falls } |az^{-1}| < 1 \tag{2.29a}$$

Aus der Bedingung für die Konvergenz der geometrischen Reihe ergibt sich das Konvergenzgebiet $\{x \in \mathbb{C} \mid |z| > |a|\}$.

Die Folgen der Form $x[n] = -a^n u[-n-1]$, die die linksseitig gemachten und invertierten Exponentialfolgen sind, haben diese z-Transformierte:

$$\begin{aligned}
\mathcal{Z}\{x\}(z) &= -\sum_{n=-\infty}^{-1} a^n\, z^{-n} = -\sum_{m=1}^{\infty} a^{-m}\, z^m \quad \text{mit } m = -n \\
&= 1 - \sum_{m=0}^{\infty} \left(a^{-1} z\right)^m = 1 - \frac{1}{1-a^{-1}z} \quad \text{falls } |a^{-1}z| < 1 \\
&= \frac{1}{1-az^{-1}}
\end{aligned} \tag{2.29b}$$

Der Funktionsterm $\mathcal{Z}\{x\}(z)$ ist also der gleiche wie bei der rechtsseitigen Exponentialfolge, aber der Definitionsbereich unterscheidet sich: Es ist diesmal die Menge $\{z \in \mathbb{C} \mid |z| < |a|\}$.

Dieses Beispiel zeigt, dass die Angabe des Funktionsterms einer z-Transformierten allein nicht ausreicht, um die zugehörige Folge eindeutig zu bestimmen, sondern dass auch das Konvergenzgebiet zur vollständigen Beschreibung einer Folge in der z-Transformationsdarstellung gehört.

2.2.6 Rationale Funktionen

In vielen Anwendungsfällen ist der Ausdruck $X(z)$ für die z-Transformierte eine rationale Funktion, lässt sich also als Quotient

$$X(z) = \frac{P(z)}{Q(z)} \tag{2.30a}$$

von zwei Polynomen P und Q in z schreiben. Nützlicher ist allerdings oft die Darstellung mit Polynomen in z^{-1}: Die Funktion hat dann die Form

$$X(z) = \frac{\sum_{k=0}^{M} b_k\, z^{-k}}{\sum_{k=0}^{N} a_k\, z^{-k}} \tag{2.30b}$$

mit $N+1$ komplexen Zahlen $\{a_0, \ldots, a_N\}$ und $M+1$ komplexen Zahlen $\{b_0, \ldots, b_M\}$. Die im vorangegangenen Abschnitt gezeigten z-Transformierten haben beispielsweise diese Form.

Man kann auch N komplexe Zahlen $\{d_1, \ldots, d_N\}$ und M komplexe Zahlen $\{c_1, \ldots, c_M\}$ finden, so dass gilt:

$$X(z) = \frac{b_0}{a_0} \frac{\prod_{k=1}^{M} \left(1 - d_k z^{-1}\right)}{\prod_{k=1}^{N} \left(1 - c_k z^{-1}\right)} \tag{2.30c}$$

Dies ist die *faktorisierte* Form der rationalen Funktion [5, S. 250].

Die Punkte $z \in \mathbb{C}$ mit $P(z) = 0$ nennt man *Nullstellen* und die Punkte mit $Q(z) = 0$ nennt man *Pole*. An der faktorisierten Form kann man ablesen, dass die Pole mit den Zahlen c_k und die Nullstellen mit den Zahlen d_k übereinstimmen.

Ein wichtiger Spezialfall tritt ein, wenn alle Koeffizienten a_k und b_k reelle Zahlen sind. Dann treten *alle nicht reellen* Pole und Nullstellen als *komplex konjugierte Paare* auf. Das folgt aus dieser Eigenschaft von Polynomen mit reellen Koeffizienten: Sei $z_0 \in \mathbb{C}$ mit $f(z_0) = \sum_{k=0}^{N} \lambda_k z_0^{-k} = 0$ mit $\lambda_k \in \mathbb{R}$. Dann gilt:

$$f(\overline{z_0}) = \sum_{k=0}^{N} \lambda_k \overline{z_0}^{-k} = \sum_{k=0}^{N} \lambda_k \overline{z_0^{-k}} = \sum_{k=0}^{N} \overline{\lambda_k z_0^{-k}} = \overline{\sum_{k=0}^{N} \lambda_k z_0^{-k}} = \overline{f(z_0)} = 0$$

Das heißt, wenn z_0 eine Nullstelle von f ist, dann auch das komplex konjugierte $\overline{z_0}$.

Damit nimmt die faktorisierte Darstellung (2.30c) der rationalen Funktion die Form

$$X(z) = \frac{b_0}{a_0} \frac{\prod_{k=1}^{M_\mathrm{r}} \left(1 - d_k z^{-1}\right)}{\prod_{k=1}^{N_\mathrm{r}} \left(1 - c_k z^{-1}\right)} \frac{\prod_{k=1}^{M_\mathrm{cc}} \left(1 - d'_k z^{-1}\right)\left(1 - \overline{d'}_k z^{-1}\right)}{\prod_{k=1}^{N_\mathrm{cc}} \left(1 - c'_k z^{-1}\right)\left(1 - \overline{c'}_k z^{-1}\right)} \tag{2.30d}$$

an, wobei hier c_k und d_k die reellen Pole und Nullstellen sind sowie $(c'_k, \overline{c'_k})$ und $(d'_k, \overline{d'_k})$ die komplex konjugierten Paare von Polen und Nullstellen. Die Gesamtzahl der Pole und Nullstellen bleibt natürlich erhalten, d. h. $N_\mathrm{r} + 2N_\mathrm{cc} = N$ und $M_\mathrm{r} + 2M_\mathrm{cc} = M$.

Multipliziert man die Terme der komplex konjugierten Paare aus, erhält man die Faktorisierung einer rationalen Funktion mit reellen Koeffizienten a_k und b_k, die selbst wieder nur *reelle* Koeffizienten hat, im Gegensatz zu (2.30c), wo die Koeffizienten nicht unbedingt reell sind [5, S. 364].

$$X(z) = \frac{b_0}{a_0} \frac{\prod_{k=1}^{M_\mathrm{r}} \left(1 - d_k z^{-1}\right)}{\prod_{k=1}^{N_\mathrm{r}} \left(1 - c_k z^{-1}\right)} \frac{\prod_{k=1}^{M_\mathrm{cc}} \left(1 - (d'_k + \overline{d'_k})\, z^{-1} + (d'_k \overline{d'_k})\, z^{-2}\right)}{\prod_{k=1}^{N_\mathrm{cc}} \left(1 - (c'_k + \overline{c'_k})\, z^{-1} + (c'_k \overline{c'_k})\, z^{-2}\right)} \tag{2.30e}$$

Der Funktionsterm $X(z)$ ist überall außer an den Polen definiert. Wegen der Eigen-

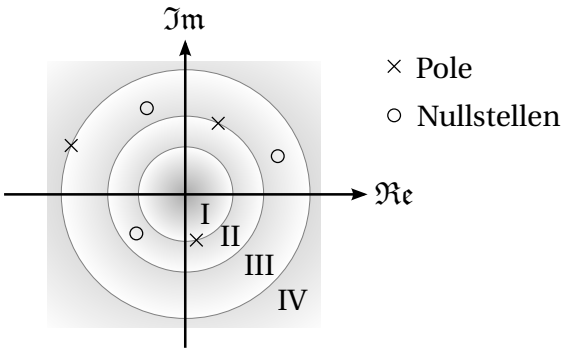

Abbildung 2.4: Pol-/Nullstellendiagramm einer rationalen Funktion und die sich daraus ergebenden Möglichkeiten I–IV für das Konvergenzgebiet einer z-Transformierten

schaft (2.21) des Konvergenzgebiets jeder z-Transformierten muss deshalb gelten:

Das Konvergenzgebiet einer z-Transformierten, die als rationale Funktion dargestellt werden kann, ist

- *entweder eine durch den Pol mit dem kleinsten Betrag nach außen begrenzte Kreisscheibe*
- *oder ein durch zwei Pole begrenzter Kreisring, der keinen Pol enthält*
- *oder das durch den Pol mit dem größten Betrag nach innen begrenzte Gebiet*

in der komplexen Zahlenebene.

Diese Erkenntnis kann man anhand eines *Pol-/Nullstellendiagramms* veranschaulichen. In Abbildung 2.4 ist ein solches Diagramm für eine rationale Funktion mit jeweils drei Polen und Nullstellen gezeigt. Sie kann der Funktionsterm von vier verschiedenen z-Transformierten sein.

2.2.7 Fouriertransformation

Die zeitdiskrete Fouriertransformation $\mathscr{F}$ kann als Spezialfall der z-Transformation aufgefasst werden. Sie bildet eine Folge x in eine komplexwertige Funktion über den reellen Zahlen ab [5, S. 51]:

$$\mathscr{F} : x \mapsto (\mathscr{F}\{x\} : \mathbb{R} \to \mathbb{C}) \tag{2.31}$$

Die Berechnungsvorschrift für den Funktionswert der Fouriertransformierten an der Stelle

$\omega \in \mathbb{R}$ lautet:

$$\mathscr{F}\{x\}(\omega) = \sum_{n=-\infty}^{\infty} x[n]\, e^{-i\omega n} \tag{2.32}$$

Die Fouriertransformierte einer Folge ist nur definiert, falls diese unendliche Reihe konvergiert.

Sie ist periodisch ($\mathscr{F}\{x\}(\omega + 2\pi) = \mathscr{F}\{x\}(\omega)$), weshalb sie durch die Funktionswerte im Intervall $0 \leq \omega < 2\pi$ bereits vollständig charakterisiert ist. Dies stellt den Zusammenhang mit der z-Transformation her, denn die Funktionswerte der Fouriertransformierten in diesem Intervall stimmen genau mit den Funktionswerten der z-Transformierten auf dem Einheitskreis in der komplexen Zahlenebene überein:

$$\mathscr{F}\{x\}(\omega) = \sum_{n=-\infty}^{\infty} x[n]\, e^{-i\omega n} = \mathscr{Z}\{x\}(e^{i\omega}) \tag{2.33}$$

Das bedeutet auch, dass die Fouriertransformierte einer Folge genau dann existiert, wenn der Einheitskreis $|z| = 1$ im Konvergenzgebiet der z-Transformierten enthalten ist.

Wegen (2.21) konvergiert die Reihe (2.32) dann absolut, das heißt:

$$\sum_{n=-\infty}^{\infty} \left| x[n]\, e^{-i\omega n} \right| = \sum_{n=-\infty}^{\infty} \left| x[n] \right| < \infty \tag{2.34}$$

Also sind die folgenden drei Aussagen über eine Folge gleichbedeutend:

- *Das Konvergenzgebiet der z-Transformierten enthält den Einheitskreis der komplexen Zahlenebene.*
- *Die Fouriertransformierte existiert.*
- *Die Folge ist absolut summierbar.*

2.3 Zeitdiskrete Systeme und Filter

Nachdem nun das Konzept der Folgen und ihrer Darstellung als z-Transformierte mitsamt der Eigenschaften und Rechenregeln eingeführt sind, können die zeitdiskreten Systeme betrachtet werden.

Ein zeitdiskretes System ist mathematisch gesehen nichts anderes als eine Abbildung T, die einer *Eingangsfolge* x eine *Ausgangsfolge* $y = T\{x\}$ zuordnet. $T\{x\}$ wird auch die *Antwort* des Systems auf die Folge x genannt. Die grundlegenden Rechenoperationen mit Folgen aus Abschnitt 2.1.2 können somit auch als zeitdiskrete Systeme interpretiert werden [5, S. 17].

Ein „Filter“ ist in diesem Sinn das Gleiche wie ein „System“, jedoch deutet die Benennung eines zeitdiskreten Systems als Filter darauf hin, dass es *gezielt* eingesetzt wird, um Signale, die durch Folgen repräsentiert werden, in ihren Eigenschaften zu verändern.

2.3.1 Eigenschaften

Zeitdiskrete Systeme können grundsätzlich einen beliebigen (oder gar keinen) Zusammenhang zwischen der Eingangs- und der Ausgangsfolge haben. Für die Anwendung als Filter sind aber natürlich solche Systeme von Interesse, deren Verhalten eine gewisse Vorhersehbarkeit hat, so dass man z. B. eine gegebene Eingangsfolge auf gewünschte Weise manipulieren kann oder von einer beobachteten Ausgangsfolge auf die zugrundeliegende Eingangsfolge schließen kann. Um das Verhalten von Systemen zu charakterisieren, sind einige Merkmale definiert, die nun vorgestellt werden.

Kausalität

Ein System T heißt genau dann *kausal*, wenn für jedes beliebige Paar von Eingangsfolgen x_1 und x_2 die folgende Aussage wahr ist [5, S. 22]:

$$\forall n_0 \in \mathbb{Z} : \Big(\big(\forall n \leq n_0 : x_1[n] = x_2[n] \big) \Rightarrow \big(\forall n \leq n_0 : T\{x_1\}[n] = T\{x_2\}[n] \big) \Big) \tag{2.35}$$

Das bedeutet in Worten: Wenn zwei Eingangsfolgen x_1 und x_2 bis zu einem Index n_0 übereinstimmen, müssen bei einem kausalen System auch die jeweiligen Ausgangsfolgen $T\{x_1\}$ und $T\{x_2\}$ bis zu demselben Index übereinstimmen.[2] Oder umgekehrt und in der Interpretation der Folgen als zeitliche Signale: Es kann bei einem kausalen System nicht sein, dass sich zwei Ausgangsfolgen zu einem Zeitpunkt unterscheiden, bis zu dem die entsprechenden Eingangsfolgen identisch waren.

Um diese Definition zu verdeutlichen, ist hier ein konstruiertes Beispiel für ein System, das nicht kausal ist:

$$T\{x\} = \begin{cases} \mathscr{D}_1 \delta, & x = u \\ x, & \text{sonst} \end{cases}$$

Das System lässt alle Folgen unverändert, bis auf den Einheitssprung, der in den um Eins verzögerten Einheitsimpuls abgebildet wird. Nimmt man dann z. B. $x_1 = \delta$, $x_2 = u$ und $n_0 = 0$,

[2] Die Aussage „$A \Rightarrow B$“ gilt als wahr, wenn A falsch ist. Falls zwei Eingangsfolgen also für ein n_0 *nicht* in allen Folgengliedern mit $n \leq n_0$ übereinstimmen, kann „$\forall n_0 \in \mathbb{Z} : \ldots$“ trotzdem erfüllt werden.

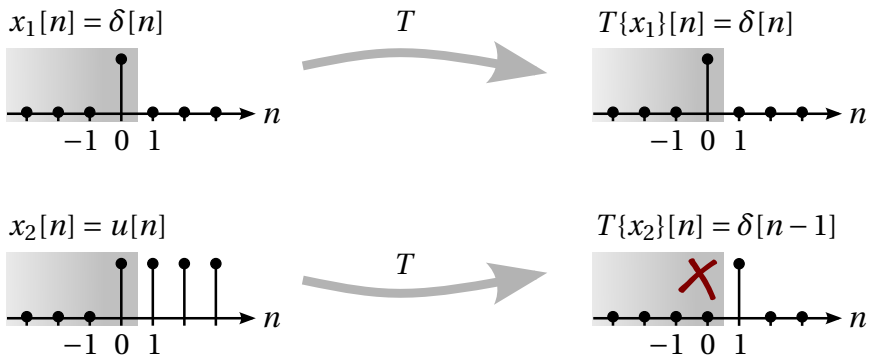

Abbildung 2.5: Beispiel für ein nicht-kausales System

gilt:

$$\forall\, n \le n_0 : x_1[n] = x_2[n],$$

aber

$$T\{x_1\}[0] = \delta[0] = 1$$
$$T\{x_2\}[0] = (\mathscr{D}_1\delta)[0] = \delta[-1] = 0.$$

Die Eingangsfolgen stimmen also bis zur Stelle Null überein, während sich die Ausgangsfolgen zwar für alle $n \le -1$ gleichen, aber für $n = 0$ unterschiedlich sind, wie in Abbildung 2.5 dargestellt. Somit ist das beschriebene System nicht kausal.

Für Systeme mit gewissen anderen Eigenschaften, die noch eingeführt werden, ist die Kausalität einfacher zu definieren und zu interpretieren.

Stabilität

Systeme T sind *stabil*, wenn sie jede beschränkte Eingangsfolge auf eine beschränkte Ausgangsfolge abbilden, d. h. die folgende Bedingung erfüllen [5, S. 23]:

$$\forall\, x : \big(x \text{ beschränkt} \Rightarrow T\{x\} \text{ beschränkt}\big) \tag{2.36}$$

Ein System, das jede Eingangsfolge mit einem konstanten Faktor multipliziert ist beispielsweise stabil, aber das folgende System ist nicht stabil:

$$T\{x\} = \begin{cases} y : y[n] = n, & x = \delta \\ x, & \text{sonst} \end{cases}$$

Es bildet nämlich den beschränkten Einheitsimpuls auf die Folge y mit $y[n] = n$ ab, die nicht beschränkt ist.

Zeitinvarianz

Ein System heißt *zeitinvariant*, wenn sich die Verzögerung der Eingangsfolge nur durch entsprechende Verzögerung der Ausgangsfolge bemerkbar macht. Das heißt, dass für alle Folgen x gelten muss [5, S. 21]:

$$\forall k \in \mathbb{Z} : T\{\mathscr{D}_k x\} = \mathscr{D}_k T\{x\} \tag{2.37}$$

Das oben genannte nicht-stabile System ist auch nicht zeitinvariant: Sei $x = \delta$, dann ist $T\{x\} = y$. Aber $T\{\mathscr{D}_1 x\} = T\{\mathscr{D}_1 \delta\} = \mathscr{D}_1 \delta \neq \mathscr{D}_1 T\{x\}$.

Linearität

Systeme, die dem *Superpositionsprinzip* genügen, sind *linear*. Für beliebige Folgen x_1 und x_2 muss dann gelten [5, S. 20]:

$$\forall a \in \mathbb{C} : T\{a\,x_1 + x_2\} = a\,T\{x_1\} + T\{x_2\} \tag{2.38}$$

Bei Überlagerung von Folgen am Eingang bewirkt also die Linearität, dass die Ausgangsfolge aus der entsprechenden Überlagerung der einzelnen Ausgangsfolgen besteht. Insbesondere bilden lineare Systeme die Nullfolge auf sich selbst ab:

$$\begin{aligned} T\{\mathscr{O}\} &= T\{0 \cdot x\} \quad \text{für eine beliebige Folge } x \\ &= 0 \cdot T\{x\} \\ &= \mathscr{O} \end{aligned} \tag{2.39}$$

Dasselbe System wie zuvor dient hier ebenfalls als Gegenbeispiel: Ist z. B. $x = \delta$, dann ist $T\{x\} = y$, aber $T\{a\,x\} = a\,\delta \neq a\,T\{x\}$.

2.3.2 Lineare und zeitinvariante Systeme

Wie in den Beispielen des vorangegangenen Abschnitts gezeigt, können Systeme grundsätzlich ein willkürlich gewähltes Verhalten haben. Es gibt jedoch eine Klasse von Systemen, die sich durch eine gewisse Vorhersehbarkeit auszeichnen, nämlich diejenigen, die die Eigenschaften der *Linearität* und der *Zeitinvarianz* besitzen. Sie werden auch als *LTI-Systeme* bezeichnet, abgekürzt von den englischen Begriffen *linear, time invariant* [5, S. 24].

Diese beiden Eigenschaften liefern alleine jeweils die Rechenregeln (2.37) und (2.38) für die Berechnung der Ausgangsfolgen bei Verzögerung und Überlagerung von Eingangsfolgen. Da bei LTI-Systemen beide Eigenschaften gleichzeitig erfüllt sind, lässt sich zusätzlich noch eine Rechenregel für die Faltung als Kombination von Verzögerung und Überlagerung finden: Seien x_1 und x_2 beliebige Folgen. Dann gilt:

$$\begin{aligned} T\{x_1 * x_2\} &= T\left\{\sum_{k=-\infty}^{\infty} a_k \mathscr{D}_k x_2\right\} \quad \text{mit } a_k = x_1[k] && \text{(nach 2.5b)} \\ &= \sum_{k=-\infty}^{\infty} a_k \, T\{\mathscr{D}_k x_2\} && \text{(Linearität)} \\ &= \sum_{k=-\infty}^{\infty} a_k \mathscr{D}_k T\{x_2\} && \text{(Zeitinvarianz)} \\ &= x_1 * T\{x_2\} && (2.40) \end{aligned}$$

Impulsantwort

Um mithilfe dieser Regel die erwähnte Vorhersehbarkeit zu begründen, muss noch die *Impulsantwort* definiert werden: Es handelt sich um die Ausgangsfolge h eines Systems T mit dem Einheitsimpuls δ als Eingangsfolge:

$$h = T\{\delta\} \tag{2.41}$$

Die Impulsantwort ist im Prinzip für jedes zeitdiskrete System definiert, aber nur bei LTI-Systemen gelangt sie zu ihrer besonderen Bedeutung: Sei nämlich x eine beliebige Eingangsfolge eines linearen, zeitinvarianten Systems T. Dann gilt für die Ausgangsfolge:

$$T\{x\} = T\{x * \delta\} = x * T\{\delta\} = x * h \tag{2.42}$$

Das heißt also: *Die Ausgangsfolge eines linearen und zeitinvarianten Systems ergibt sich durch Faltung der Eingangsfolge mit der Impulsantwort.*

Das Verhalten eines solchen Systems kann dann in dem Sinn nicht mehr willkürlich sein, dass man allein aus der Kenntnis der Impulsantwort die Ausgangsfolge des Systems für jede beliebige Eingangsfolge ableiten kann.

Außerdem stehen die Stabilität und die Kausalität eines LTI-Systems in Beziehung zu den Eigenschaften seiner Impulsantwort:

Stabilität

Ein LTI-System ist genau dann stabil, wenn die Impulsantwort absolut summierbar ist. [5, S. 32 f.]

Beweis: Sei die Eingangsfolge x beschränkt, d. h. es gebe eine positive reelle Zahl B, so dass $|x[n]| \leq B$ für alle n. Sei die Impulsantwort h absolut summierbar, also $\sum_{n=-\infty}^{\infty} |h[n]| = S < \infty$. Dann ist die Ausgangsfolge y auch beschränkt:

$$\begin{aligned} \big|y[n]\big| &= \left| \sum_{k=-\infty}^{\infty} x[k]\, h[n-k] \right| \\ &\leq \left| \sum_{k=-\infty}^{\infty} h[n-k] \right| \cdot B \\ &\leq \sum_{k=-\infty}^{\infty} \big|h[n-k]\big| \cdot B = S \cdot B \end{aligned}$$

Also ist das System stabil.

Sei hingegen h nicht absolut summierbar. Dann kann man mithilfe der komplex konjugierten Folge $\overline{h}$ eine beschränkte Eingangsfolge x konstruieren, so dass die Ausgangsfolge y nicht beschränkt ist:

$$x[n] = \begin{cases} \frac{\overline{h[-n]}}{|h[-n]|}, & h[-n] \neq 0 \\ 0, & \text{sonst} \end{cases}$$

Also ist $|x[n]| \leq 1$ für alle n. Die Ausgangsfolge an der Stelle Null ist dann

$$y[0] = \sum_{k=-\infty}^{\infty} h[k]\, x[-k] = \sum_{k=-\infty}^{\infty} h[k]\, \frac{\overline{h[k]}}{|h[k]|} = \sum_{k=-\infty}^{\infty} \frac{|h[k]|^2}{|h[k]|} = \sum_{k=-\infty}^{\infty} |h[k]|,$$

was nach Voraussetzung keine endliche Zahl ist. Deshalb ist ein System, dessen Impulsantwort nicht absolut summierbar ist, nicht stabil.

Kausalität

Bei jedem linearen und zeitinvarianten System gilt: *Das System ist genau dann kausal, wenn die Impulsantwort für alle negativen Indizes verschwindet.*

Beweis: Es gilt $\delta[n] = \mathcal{O}[n]$ für alle $n < 0$. Sei h die Impulsantwort eines kausalen LTI-

Systems und sei $n < 0$. Dann gilt:

$$\begin{aligned} h[n] &= T\{\delta\}[n] \\ &= T\{\mathcal{O}\}[n] && \text{(Kausalität)} \\ &= \mathcal{O}[n] = 0 && \text{(Linearität)} \end{aligned} \tag{2.43}$$

Damit ist zunächst gezeigt, dass wegen der Kausalität die Impulsantwort für alle negativen Indizes Null ist. Dies gilt sogar für alle linearen Systeme, die nicht zeitinvariant sind. Für den Beweis der ganzen Aussage muss noch die umgekehrte Implikation hergeleitet werden, dass also aus dem Verschwinden der Impulsantwort bei negativen Indizes die Kausalität folgt:

Seien x_1 und x_2 beliebige Folgen mit $x_1[n] = x_2[n]$ für alle $n < n_0$. Sei h die Impulsantwort eines LTI-Systems und $h[n] = 0$ für alle $n < 0$. Für alle $n < n_0$ gilt dann:

$$\begin{aligned} T\{x_1\}[n] &= (h * x_1)[n] \\ &= \sum_{k=-\infty}^{\infty} h[k]\, x_1[n-k] \\ &= \sum_{k=0}^{\infty} h[k]\, x_1[n-k] && \text{(wegen } h[k] = 0 \text{ für } n < 0) \\ &= \sum_{k=0}^{\infty} h[k]\, x_2[n-k] && \text{(wegen } n - k \le n < n_0) \\ &= \sum_{k=-\infty}^{\infty} h[k]\, x_2[n-k] \\ &= (h * x_2)[n] = T\{x_2\}[n] \quad \square \end{aligned}$$

Aus dem Ausdruck für die Berechnung eines Werts der Ausgangsfolge eines kausalen LTI-Systems ergibt sich wegen dieser Eigenschaft der Impulsantwort eine neue Interpretation der Kausalität, die anschaulicher ist als die ursprüngliche Definition (2.35):

$$T\{x\}[n] = \sum_{k=0}^{\infty} h[k]\, x[n-k]$$

Wenn man die Indizes n als Zeit auffasst, bedeutet das: *Bei einem kausalen LTI-System hängt ein Wert der Ausgangsfolge nicht von zukünftigen Werten der Eingangsfolge ab.* [5, S. 22]

Serienschaltung

Wenn man mehrere zeitdiskrete Systeme hintereinanderschaltet, ist im Allgemeinen die Reihenfolge dafür entscheidend, wie sich das Gesamtsystem verhält. Sind jedoch alle betei-

ligten Systeme linear und zeitinvariant, dann lassen sie sich beliebig vertauschen, ohne dass sich das Gesamtsystem dabei ändert. Dies ergibt sich aus den Rechenregeln für die Faltung von Folgen:

Seien T_1 und T_2 zwei in Reihe geschaltete LTI-Systeme mit den Impulsantworten h_1 bzw. h_2 und x eine beliebige Eingangsfolge. Dann ist die Ausgangsfolge:

$$\begin{aligned} T_1\{T_2\{x\}\} &= h_1 * (h_2 * x) \\ &= (h_1 * h_2) * x \\ &= h_2 * (h_1 * x) = T_2\{T_1\{x\}\} \end{aligned} \tag{2.44}$$

Ebenso gut kann man die beiden einzelnen Systeme durch ein Gesamtsystem T mit der Impulsantwort $h_1 * h_2$ auffassen [5, S. 31].

Systemfunktion

Die Besonderheit von linearen und zeitinvarianten Systemen ist, wie gezeigt, dass sie in Form einer Folge – ihrer Impulsantwort – vollständig beschrieben werden können. Mithilfe der z-Transformation kann man nun eine weitere äquivalente Darstellungsform einführen: Sei h die Impulsantwort eines LTI-Systems, dann heißt die z-Transformierte $\mathcal{Z}\{h\} = H$ die *Systemfunktion* [5, S. 245].

Mit der Systemfunktion wird der Zusammenhang zwischen Eingangs- und Ausgangsfolge wegen des Faltungstheorems (2.25) zu einer Multiplikation vereinfacht: Sei X die z-Transformierte der Eingangsfolge x und Y die z-Transformierte der Ausgangsfolge y. Dann gilt für alle z in dem gemeinsamen Konvergenzgebiet von X und H:

$$Y(z) = \mathcal{Z}\{y\}(z) = \mathcal{Z}\{h * x\}(z) = \mathcal{Z}\{h\}(z)\,\mathcal{Z}\{x\}(z) = H(z)\,X(z) \tag{2.45}$$

Die Eigenschaften des Konvergenzgebiets der Systemfunktion hängen mit den Eigenschaften der Stabilität und Kausalität des Systems, das durch sie beschrieben wird, zusammen:

Das Ergebnis des Abschnitts 2.2.7 war, dass eine Folge genau dann absolut summierbar ist, wenn das Konvergenzgebiet ihrer z-Transformierten den Einheitskreis der komplexen Zahlenebene enthält. Weiterhin wurde gezeigt, dass ein LTI-System genau dann stabil ist, wenn die Impulsantwort absolut summierbar ist. Daraus folgt:

Ein LTI-System ist genau dann stabil, wenn das Konvergenzgebiet der Systemfunktion den Einheitskreis der komplexen Zahlenebene enthält.

Für ein kausales System muss gelten:

Das Konvergenzgebiet der Systemfunktion eines kausalen LTI-Systems hat die Form $\{z \mid |z| > r\}$. Dies folgt daraus, dass die Impulsantwort bei negativen Indizes den Wert Null hat

und somit rechtsseitig ist.

2.3.3 Differenzengleichungen

Eine weitere Klasse von Systemen sind diejenigen, deren Eingangs- und Ausgangsfolge durch eine *lineare Differenzengleichung mit konstanten Koeffizienten* verknüpft sind. Das heißt, dass es komplexe Zahlen $\{a_0, \ldots, a_N\}$ und $\{b_0, \ldots, b_M\}$ gibt, so dass bei einem solchen System T für jede beliebige Folge x gilt [5, S. 37]:

$$\forall n \in \mathbb{Z} : \sum_{k=0}^{N} a_k \, T\{x\}[n-k] = \sum_{k=0}^{M} b_k \, x[n-k] \tag{2.46a}$$

$$\text{oder} \quad \sum_{k=0}^{N} a_k \mathscr{D}_k T\{x\} = \sum_{k=0}^{M} b_k \mathscr{D}_k x \tag{2.46b}$$

Es handelt sich um eine Bestimmungsgleichung für die Ausgangsfolge $T\{x\}$ des Systems bei gegebener Eingangsfolge x. Die Zahl N heißt *Ordnung* der Differenzengleichung.

Für den Fall, dass x die Nullfolge $\mathscr{O}$ ist, spricht man von der *homogenen Gleichung* mit der *homogenen Lösung* y_{hom} [5, S. 40]:

$$\forall n \in \mathbb{Z} : \sum_{k=0}^{N} a_k \, y_{\text{hom}}[n-k] = 0 \tag{2.47a}$$

$$\text{oder} \quad \sum_{k=0}^{N} a_k \mathscr{D}_k y_{\text{hom}} = \mathscr{O} \tag{2.47b}$$

Die Nullfolge $\mathscr{O}$ ist auf jeden Fall eine Lösung der homogenen Gleichung. Im Allgemeinen gibt es mehr als diese „triviale“ Lösung:

Sei zum Beispiel $a_0 = 1$, $a_1 = -q$ und $a_k = 0$ sonst, dann sind alle Folgen y mit $y[n] = c\,q^n$ und einer beliebigen Konstante c homogene Lösungen.

Betrachtet man die Differenzengleichung wieder im allgemeinen Fall einer beliebigen Eingangsfolge x, dann gilt: Wenn y eine Lösung der Gleichung ist, ist auch $y + y_{\text{hom}}$ eine Lösung:

$$\sum_{k=0}^{N} a_k \mathscr{D}_k \big(y + y_{\text{hom}}\big) = \sum_{k=0}^{N} a_k \mathscr{D}_k y + \underbrace{\sum_{k=0}^{N} a_k \mathscr{D}_k y_{\text{hom}}}_{\mathscr{O}} = \sum_{k=0}^{N} a_k \mathscr{D}_k y \tag{2.48}$$

Das alles bedeutet, dass die Differenzengleichung alleine nicht ausreicht, um für eine gegebene Eingangsfolge die Ausgangsfolge und damit das System eindeutig zu bestimmen. Erst zusammen mit *Randbedingungen* kann die Lösungsmenge so eingeschränkt werden,

dass die Differenzengleichung eine vollständige Beschreibung eines zeitdiskreten Systems ist.

Linearität und Zeitinvarianz

Eine Randbedingung kann z. B. sein, dass man $y_{\text{hom}} = T\{\mathcal{O}\} = \mathcal{O}$ fordert. Das ist gleichbedeutend damit, dass das System linear und zeitinvariant ist: Wenn $T\{\mathcal{O}\} \neq \mathcal{O}$ ist, kann T nicht linear sein (wegen 2.39) und ist im Allgemeinen auch nicht zeitinvariant (denn $T\{\mathcal{D}_k\mathcal{O}\} = T\{\mathcal{O}\} \neq \mathcal{D}_k T\{\mathcal{O}\}$). Wenn die Randbedingung gilt, *muss* hingegen T ein LTI-System sein.

Beweis: Seien x_1 und x_2 beliebige Folgen, sei $y_1 = T\{x_1\}$ und $y_2 = T\{x_2\}$. Dann erfüllen die Paare (x_1, y_1) und (x_2, y_2) jeweils notwendigerweise die Differenzengleichung. Sei a eine beliebige komplexe Zahl, $x = a\,x_1 + x_2$, $y = a\,y_1 + y_2$ und $y' = T\{x\}$. Um die Linearität nachzuweisen, muss man also $y' = y$ zeigen. Wegen (2.7) und (2.8) erfüllt das Paar (x, y) die Differenzengleichung und nach Voraussetzung notwendigerweise auch (x, y') (weil $y' = T\{x\}$ ist und T gerade durch die Differenzengleichung definiert ist). Also muss $y' = y + y_{\text{hom}}$ sein. Aber die Forderung ist $y_{\text{hom}} = \mathcal{O}$, also ist $y' = y$.

Um die Zeitinvarianz nachzuweisen, konstruiert man $x = \mathcal{D}_k x_1$ und $y = \mathcal{D}_k y_1$, womit die gleiche Argumentation gilt.

Wenn man also diese Randbedingung einführt, beschreiben Differenzengleichungen LTI-Systeme. Für diese ist es sinnvoll, die Impulsantwort oder deren z-Transformierte, die Systemfunktion, zu betrachten.

Indem man die z-Transformation auf die Differenzengleichung anwendet, erhält man eine Bedingung für die Funktionswerte der Systemfunktion H. Sei x eine beliebige Eingangsfolge, $y = T\{x\}$ die zugehörige Ausgangsfolge und X bzw. Y die jeweiligen z-Transformierten. Dann gilt für alle z, die sowohl im Konvergenzgebiet von X als auch im Konvergenzgebiet von Y liegen:

$$\begin{aligned}
&\mathcal{Z}\left\{\sum_{k=0}^{N} a_k \mathcal{D}_k y\right\}(z) = \mathcal{Z}\left\{\sum_{k=0}^{M} b_k \mathcal{D}_k x\right\}(z) \\
\Leftrightarrow\quad &\sum_{k=0}^{N} a_k \mathcal{Z}\{\mathcal{D}_k y\}(z) = \sum_{k=0}^{M} b_k \mathcal{Z}\{\mathcal{D}_k x\}(z) && \text{(wegen 2.23)} \\
\Leftrightarrow\quad &\sum_{k=0}^{N} a_k z^{-k} Y(z) = \sum_{k=0}^{M} b_k z^{-k} X(z) && \text{(wegen 2.24)}
\end{aligned}$$

Es ist aber auch $Y(z) = H(z)X(z)$ nach (2.45). Deshalb muss für die Funktionswerte der

Systemfunktion gelten:

$$H(z) = \frac{\sum_{k=0}^{M} b_k \, z^{-l}}{\sum_{k=0}^{N} a_k \, z^{-k}} \tag{2.49}$$

Das heißt: *Ein LTI-System, das eine lineare Differenzengleichung mit konstanten Koeffizienten erfüllt, hat eine rationale Systemfunktion.* Damit ist das System aber immer noch nicht eindeutig festgelegt:

Kausalität

Wie in Abschnitt 2.2.6 gezeigt, gibt es bei gegebenem Funktionsterm mehrere Möglichkeiten für den Definitionsbereich der Systemfunktion. Eine davon zeichnet sich dadurch aus, dass das System *kausal* ist: Das ist genau dann der Fall, wenn die Impulsantwort h für alle negativen Indizes Null ist (2.43) und somit *rechtsseitig*. Das bedeutet, dass das Konvergenzgebiet von $H = \mathcal{Z}\{h\}$ der Bereich *außerhalb* des Pols mit dem größten Betrag sein muss.

Setzt man also Linearität, Zeitinvarianz und Kausalität voraus, beschreibt eine Differenzengleichung der Form (2.46) ein eindeutig bestimmtes zeitdiskretes System mit der Systemfunktion (2.49). Die Randbedingungen, die neben der Linearität und Zeitinvarianz auch die Kausalität implizieren, nennt man den *Anfangsruhezustand*. Damit ist gemeint, dass nicht nur die Systemantwort auf die Nullfolge festgelegt ist ($T\{\mathcal{O}\} = \mathcal{O}$), sondern für alle Eingangsfolgen x gilt [5, S. 42]:

$$\forall n_0 \in \mathbb{Z} : \big(\forall n < n_0 : x[n] = 0 \quad \Rightarrow \quad \forall n < n_0 : T\{x\}[n] = 0\big) \tag{2.50}$$

Wenn die Randbedingungen festgelegt sind, ergibt sich durch Umformen der Differenzengleichung bei gegebener Eingangsfolge x eine eindeutige *rekursive* Berechnungsvorschrift für die Ausgangsfolge y:

$$y[n] = \frac{1}{a_0} \left(\sum_{k=0}^{M} b_k \, x[n-k] - \sum_{k=1}^{N} a_k \, y[n-k] \right) \tag{2.51}$$

Für die Impulsantwort, also $x = \delta$ und $y = h$ heißt das:

$$h[n] = \frac{1}{a_0} \left(b_n - \sum_{k=1}^{N} a_k \, h[n-k] \right) \tag{2.52}$$

Anhand der Koeffizienten a_k kann man alle durch Differenzengleichungen beschriebenen Systeme in zwei Kategorien einteilen:

- Wenn alle $\{a_1, \ldots, a_k\}$ Null sind, bzw. $N = 0$ gilt, ist $h[n] = b_n / a_0$ direkt gegeben. Da es nur endlich viele (nämlich für $0 \leq n \leq M$) von Null verschiedene Koeffizienten b_n gibt, ist die Impulsantwort dann eine endliche Folge. Die entsprechenden Systeme nennt man deswegen *FIR-Systeme* (engl. *finite impulse response*) [5, S. 43].

- Ansonsten ist die Impulsantwort wegen der Rekursion im Allgemeinen für alle $n \geq 0$ von Null verschieden, weshalb diese Systeme *IIR-Systeme (infinite impulse response)* genannt werden.

Die Kausalität ist eine Eigenschaft, die für jedes System, das irgendwie – etwa als elektronische Schaltung – realisiert wird, gelten muss, damit es ein vorhersehbares Verhalten aufweist. Deshalb wird ab sofort, wenn von einer rationalen Funktion als z-Transformierte die Rede ist, das zugehörige Konvergenzgebiet nicht mehr mit angegeben, weil durch die Kausalitätsbedingung automatisch das passende Konvergenzgebiet impliziert wird. Genauso wird bei der Angabe einer Differenzengleichung davon ausgegangen, dass die Randbedingungen des Anfangsruhezustands gelten, so dass die Differenzengleichung mit einem kausalen LTI-System identifiziert werden kann. Somit sind die Begriffe „rationale Systemfunktion“ und „Differenzengleichung“ als gleichwertige Beschreibungen zeitdiskreter Systeme anzusehen und sind über die Beziehung von (2.46) zu (2.49) miteinander verknüpft.

Stabilität

Aus der Lage der Pole einer rationalen Systemfunktion kann man ermitteln, ob das zugehörige kausales LTI-System stabil ist:

Da bei einem kausalen System das Konvergenzgebiet der Systemfunktion außerhalb des Pols mit dem größten Betrag liegt, und bei einem stabilen System den Einheitskreis der komplexen Zahlenebene enthält, muss gelten:

Ein kausales LTI-System, das durch eine Differenzengleichung beschrieben wird, ist genau dann stabil, wenn alle Pole innerhalb des Einheitskreises der komplexen Zahlenebene liegen. [5, S. 252] Dies ist in Abbildung 2.6 veranschaulicht.

Blockdiagramme

Eine lineare Differenzengleichung mit konstanten Koeffizienten und damit ein zeitdiskretes System kann grafisch in Form eines *Blockdiagramms* dargestellt werden. Dazu werden für die in der Differenzengleichung verwendeten Operationen *Skalierung*, *Addition* und *Verzögerung* jeweils Symbole eingeführt, die in Abbildung 2.7 gezeigt sind [5, S. 350].

Wenn man die Koeffizienten $\lambda \in \{a_1, \ldots, a_N\}$ nach dem Schema $\lambda \mapsto -\lambda / a_0$ und die Koeffizienten $\lambda \in \{a_0, b_0, \ldots, b_M\}$ nach $\lambda \mapsto \lambda / a_0$ umbenennt, bringt man die Differenzengleichung

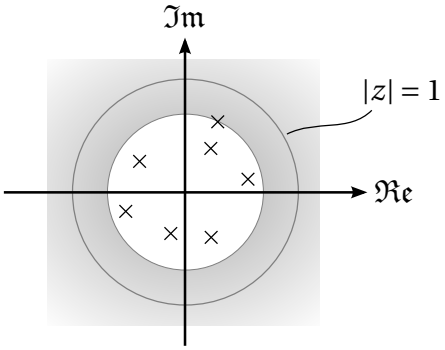

Abbildung 2.6: Bei einem stabilen und kausalen LTI-System mit rationaler Systemfunktion müssen alle Pole innerhalb des Einheitskreises liegen.

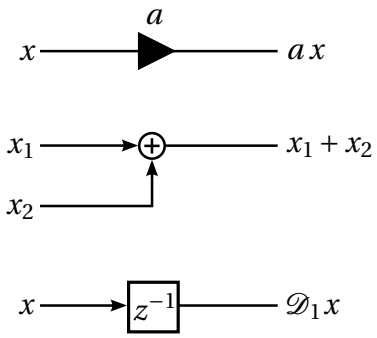

Abbildung 2.7: grafische Symbole für die Skalierung, Addition und Verzögerung von Folgen

in diese Form:

$$y = \sum_{k=1}^{N} a_k \mathscr{D}_k y + \sum_{k=0}^{M} b_k \mathscr{D}_k x \tag{2.53}$$

Eine mögliche Darstellung dieser Gleichung als Blockdiagramm ist in Abbildung 2.8 gezeigt. Diese Struktur lässt sich direkt aus der Gleichung ableiten: Die Folgen x und y werden jeweils durch eine Reihe von Verzögerungselementen geführt, wodurch die Folgen $\mathscr{D}_k x$ bzw. $\mathscr{D}_k y$ entstehen. Diese werden dann mit den entsprechenden Koeffizienten a_k und b_k skaliert und wieder aufaddiert. Man nennt die Struktur die *Direktform I* [5, S. 356].

Dabei sieht man, wie das Blockdiagramm sich in zwei Abschnitte teilen lässt: Der linke Teil, in dem die Folge x verarbeitet wird und der rechte Teil für die Folge y entsprechen zwei in Reihe geschalteten Systemen T_1 und T_2, die zusammen das Gesamtsystem T ergeben. Mit v als Ausgangsfolge des ersten Teilsystems gilt:

$$T\{x\} = T_2\big\{T_1\{x\}\big\} = T_2\{v\} = y \tag{2.54}$$

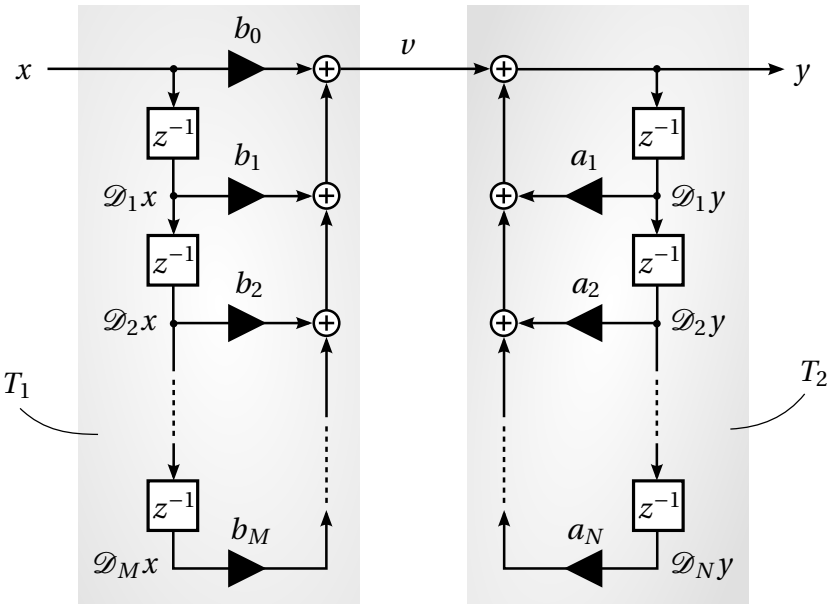

Abbildung 2.8: Blockdiagrammdarstellung einer linearen Differenzengleichung mit konstanten Koeffizienten in der Direktform I

Aus Abschnitt 2.3.2 ist bekannt, dass die Reihenfolge der Systeme T_1 und T_2 für die Gesamtfunktion keine Rolle spielt. Man kann daher ebenso gut das gleiche System T mit dem Blockdiagramm in Abbildung 2.9 beschreiben, bei dem der linke und der rechte Teil vertauscht sind. Als Zwischenergebnis entsteht dabei die Folge w:

$$T\{x\} = T_1\big\{T_2\{x\}\big\} = T_1\{w\} = y \tag{2.55}$$

In der Abbildung ist $M = N$ angenommen, was ohne Beschränkung der Allgemeinheit möglich ist. Im Vergleich zum allgemeinen Fall $M \neq N$ haben dann nur einige der Koeffizienten den Wert Null. Durch diese Maßnahme wird aber deutlich, dass die Folge w jeweils um die Werte $1, \ldots, N$ verzögert sowohl im linken als auch im rechten Teil der Struktur verwendet wird. Das bedeutet, dass man jeweils zwei der Verzögerungselemente zusammenfassen kann und dadurch das Blockdiagramm wie in der in Abbildung 2.10 gezeigten Weise vereinfacht wird. Das ist neben der Direktform I eine weitere Möglichkeit, die Gleichung (2.53) unmittelbar in ein Blockdiagramm zu überführen und wird *Direktform II* genannt [5, S. 356].

Betrachtet man T_1 und T_2 getrennt, kann man aus den jeweiligen Blockdiagrammen

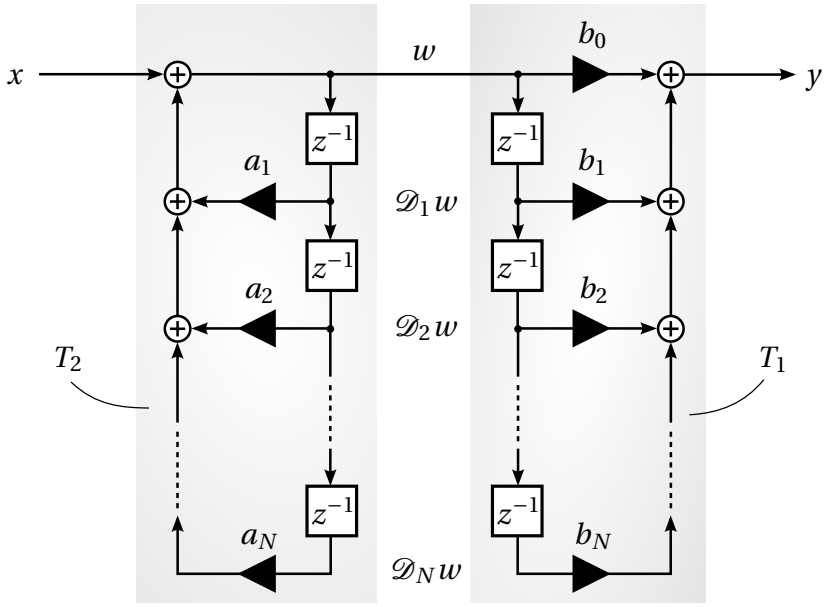

Abbildung 2.9: Die Vertauschung der beiden Teilsysteme T_1 und T_2 führt zum gleichen Gesamtsystem.

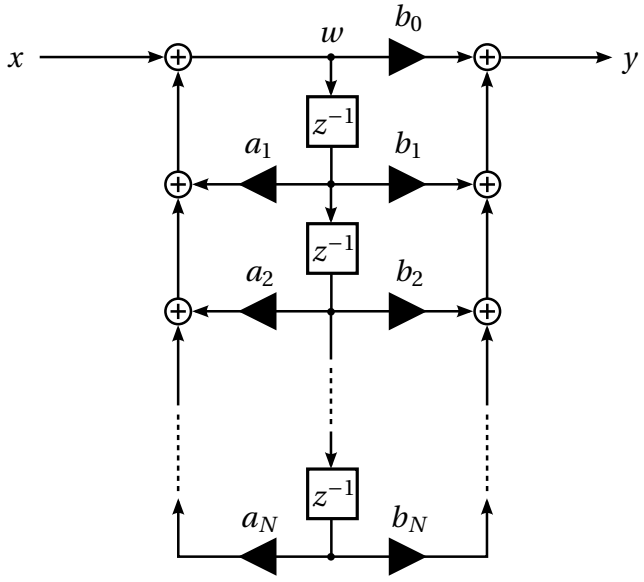

Abbildung 2.10: Darstellung des Systems in der Direktform II

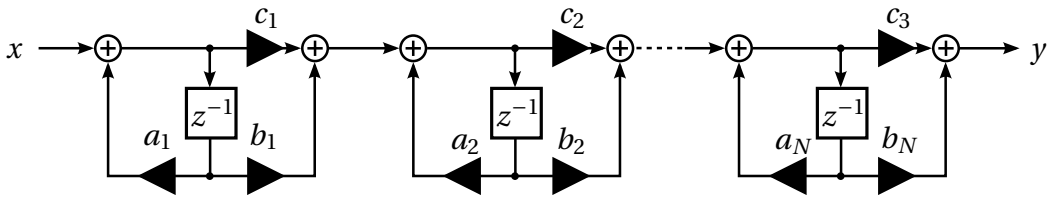

Abbildung 2.11: Reihenschaltung aus Teilsystemen erster Ordnung

wieder auf die Differenzengleichungen bzw. die Systemfunktionen H_1 und H_2 schließen:

$$v = \sum_{k=0}^{N} b_k \mathscr{D}_k x \qquad \text{bzw.} \qquad y = \sum_{k=0}^{N} b_k \mathscr{D}_k w \qquad \Leftrightarrow \qquad H_1(z) = \sum_{k=0}^{N} b_k z^{-k} \tag{2.56a}$$

$$y = v + \sum_{k=1}^{N} a_k \mathscr{D}_k y \qquad \text{bzw.} \qquad w = x + \sum_{k=1}^{N} a_k \mathscr{D}_k w \qquad \Leftrightarrow \qquad H_2(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{2.56b}$$

Die beiden Systemfunktionen entsprechen also dem Zähler und dem Nenner der rationalen Systemfunktion von T aus Gleichung (2.49), wobei daran erinnert sei, dass zugunsten der Darstellung als Blockdiagramm die Koeffizienten a_k mit negativem Vorzeichen versehen und generell alle Koeffizienten auf a_0 normiert wurden.

Die Aufteilung des Gesamtsystems in eine Reihenschaltung von Teilsystemen bedeutet, dass die Systemfunktion faktorisiert wird. Da es sich bei der Systemfunktion in den hier betrachteten Fällen um rationale Funktionen handelt, die man nach (2.30c) vollständig in Faktoren erster Ordnung (von denen jeder einen Pol oder eine Nullstelle darstellt) zerlegen kann, kann man umgekehrt für eine gegebene Differenzengleichung so viele verschiedene Blockdiagramme finden, wie es unterschiedliche Gruppierungen dieser Einzelfaktoren gibt. Die bisher gezeigten Möglichkeiten waren die, bei denen jeweils alle Pole und alle Nullstellen zusammengefasst wurden.

Der Extremfall dieser Zerlegung, der in Abbildung 2.11 zu sehen ist, besteht aus einer Reihenschaltung von Systemen erster Ordnung.

Bisher waren als Koeffizienten a_k und b_k alle beliebigen komplexen Zahlen erlaubt. Bei der Realisierung eines zeitdiskreten Systems sind aber in der Regel nur reelle Zahlen als Skalierungsfaktoren möglich. Mit der Reihenschaltung aus Systemen erster Ordnung kann man die komplex konjugierten Paare nicht umsetzen, wenn man auf reelle Koeffizienten beschränkt ist. Man muss deshalb auch Teilsysteme zweiter Ordnung einbeziehen. Diese Umsetzung der Gleichung (2.30e) als Blockdiagramm wird *Reihenschaltungsstruktur* genannt und ist in Abbildung 2.12 gezeigt [5, S.365].

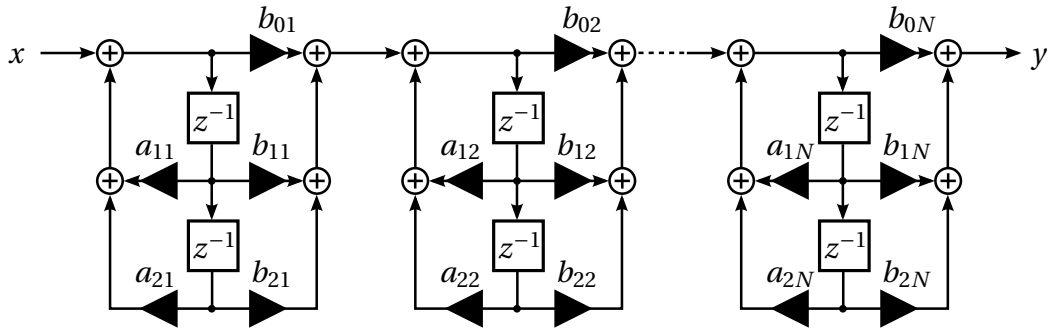

Abbildung 2.12: Reihenschaltung aus Teilsystemen zweiter Ordnung. Auch mit reellen Koeffizienten können komplexe Pole und Nullstellen realisiert werden.

2.4 Pulsfolgen

Im Abschnitt 2.1.4 wurde schon eine Klasse von Folgen genannt, die eine Verallgemeinerung der Exponentialfolgen sein sollte und auch den Einheitsimpuls einschließt. Diese Klasse der *Pulsfolgen* wird nun definiert und die wesentlichen Eigenschaften hergeleitet.

Zunächst einmal sei kurz erläutert, warum das Einführen dieses neuen Begriffs seine Berechtigung hat:

Das Hauptziel dieser Arbeit ist ja, eine Filterstruktur zu entwerfen, die ein als Folge dargestelltes Eingangssignal in seinen Eigenschaften in gewünschter Weise verändert. Die Beschreibung eines Filters geschieht dabei wie in den vorangegangenen Abschnitten gezeigt durch eine Differenzengleichung oder eine rationale Systemfunktion. Um nun den Einfluss eines Filters auf die Eingangsfolge zu untersuchen, ist es mit den bisher eingeführten Methoden am einfachsten, die z-Transformierte der Eingangsfolge zu bilden, mit der Systemfunktion zu multiplizieren und aus dem Ergebnis auf die Eigenschaften der Ausgangsfolge zu schließen, indem man die z-Transformation versucht umzukehren.

Die Pulsfolgen dienen nun erstens dazu, diese Schritte zu vereinfachen, indem sowohl die Ein- und Ausgangssignale mit ihnen modelliert werden, als auch die Filterstrukturen durch sie in Form der Impulsantwort beschrieben werden. Mithilfe von Rechenregeln für die Faltung von Pulsfolgen, kann dann ohne Umwege aus einer gegebenen Eingangsfolge die Ausgangsfolge genannt werden. Zweitens werden mit den Pulsfolgen den Objekten Namen gegeben, die sonst sprachlich nur umständlich zu umschreiben sind.

2.4.1 Definition

Die Pulsfolge σ_q^k ist für eine komplexe Zahl q und eine ganze Zahl k als Impulsantwort des kausalen LTI-Systems mit der Systemfunktion

$$\mathcal{Z}\left\{\sigma_q^k\right\}(z) = \left(\frac{1}{1-qz^{-1}}\right)^k \tag{2.57}$$

definiert. Die Zahl q heißt die *Basis* und die Zahl k der *Grad* der Pulsfolge.

2.4.2 Eigenschaften

Aufgrund der Kausalität müssen für jede Pulsfolge die Folgenglieder mit negativem Index verschwinden:

$$\forall\, n < 0 : \sigma_q^k[n] = 0 \tag{2.58}$$

Wegen des Anfangswerttheorems (2.22) gilt dann:

$$\sigma_q^k[0] = \lim_{z\to\infty}\left(\frac{1}{1-qz^{-1}}\right)^k = 1 \tag{2.59}$$

Die Pulsfolgen sind also grundsätzlich rechtsseitig und unterscheiden sich nur in den Folgengliedern mit $n > 0$. Die geschlossenen Ausdrücke für die Werte $\sigma_q^k[n]$ findet man am einfachsten, indem man zunächst einfache Spezialfälle betrachtet und dann zum allgemeinen Fall übergeht.

Einfache Fälle

Zuallererst stellt man fest: Pulsfolgen vom Grad Null oder der Basis Null sind der *Einheitsimpuls*. Das folgt aus (2.26):

$$\mathcal{Z}\left\{\sigma_q^0\right\}(z) = \mathcal{Z}\left\{\sigma_0^k\right\}(z) = 1 \quad \Rightarrow \quad \sigma_q^0 = \sigma_0^k = \delta \tag{2.60}$$

Die z-Transformierte einer Pulsfolge vom Grad Eins lautet:

$$\mathcal{Z}\left\{\sigma_q^1\right\}(z) = \frac{1}{1-qz^{-1}}$$

Diese z-Transformierte ist aus Gleichung (2.29a) ebenfalls schon bekannt. Es handelt sich

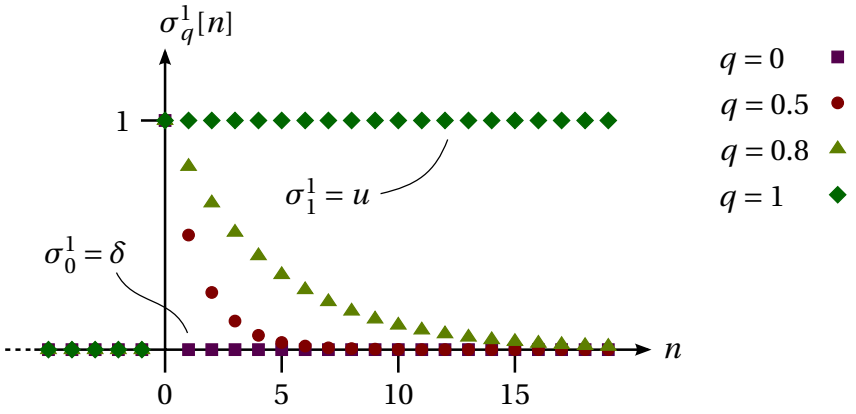

Abbildung 2.13: Einige Pulsfolgen vom Grad Eins

um die *rechtsseitigen Exponentialfolgen*:

$$\sigma_q^1[n] = u[n]\, q^n \tag{2.61}$$

Somit können der Einheitsimpuls, die für $n < 0$ ausgeblendeten Exponentialfolgen und der Einheitssprung als Pulsfolgen vom Grad Eins aufgefasst werden, die sich nur in ihrer Basis unterscheiden. Dies ist in Abbildung 2.13 veranschaulicht.

Höhere positive Grade

Aus der Definition der Pulsfolgen ergibt sich die folgende Beziehung zwischen aufeinanderfolgenden Graden:

$$\mathcal{Z}\left\{\sigma_q^k\right\}(z) = \frac{1}{1 - q z^{-1}}\, \mathcal{Z}\left\{\sigma_q^{k-1}\right\}(z) \quad \Rightarrow \quad \mathcal{Z}\left\{\sigma_q^k\right\} = \mathcal{Z}\left\{\sigma_q^{k-1} + q\, \mathscr{D}_1 \sigma_q^k\right\} \tag{2.62a}$$

Das korrespondiert mit einer Differenzengleichung, die es erlaubt, die Folgenglieder einer Pulsfolge mit Grad k zu berechnen, wenn die Pulsfolge mit Grad $k-1$ bekannt ist:

$$\sigma_q^k[n] = \sigma_q^{k-1}[n] + q\, \sigma_q^k[n-1] \tag{2.63}$$

Um einen geschlossenen Ausdruck für $\sigma_q^k[n]$ mit $k > 0$ zu gewinnen, also die Rekursionsformel aufzulösen, wird der Ansatz

$$\sigma_q^k[n] = f^k[n]\, u[n]\, q^n \tag{2.64}$$

gemacht. Damit wird q aus der Differenzengleichung eliminiert und es folgt eine Bestim-

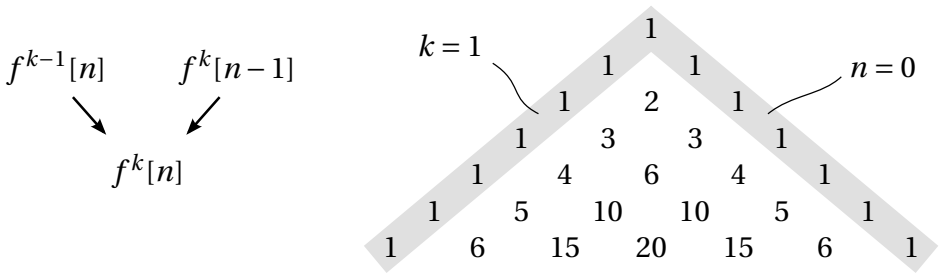

Abbildung 2.14: Zusammenhang zwischen den Vorfaktoren $f^k[n]$ und den Einträgen des Pascalschen Dreiecks

mungsgleichung für die Vorfaktoren $f^k[n]$, die für alle $n > 0$ gilt:

$$f^k[n] = f^{k-1}[n] + f^k[n-1] \tag{2.65}$$

Für $k = 1$ sind die Vorfaktoren schon bekannt:

$$\sigma_q^1[n] = u[n]\, q^n \quad \Rightarrow \quad \forall n : f^1[n] = 1 \tag{2.66}$$

Außerdem gilt bei $n = 0$ wegen (2.59):

$$\sigma_q^k[0] = f^k[0]\, u[0]\, q^0 = 1 \quad \Rightarrow \quad \forall k : f^k[0] = 1 \tag{2.67}$$

Damit sind die Startwerte für die Rekursion (2.65) festgelegt und man kann alle restlichen Vorfaktoren daraus berechnen. Die Rekursionsformel entspricht genau der Berechnungsvorschrift für das *Pascalsche Dreieck*, wie in Abbildung 2.14 illustriert.

Beginnt man die Zeilen im Pascalschen Dreieck mit Null zu nummerieren, entsprechen die Einträge mit $n + k = i + 1$ der i-ten Zeile, in der gerade die *Binomialkoeffizienten*

$$\binom{i}{j} = \frac{i!}{(i-j)!\, j!}, \quad 0 \le j \le i \tag{2.68}$$

stehen [10]. Damit hat man die Lösung für die Vorfaktoren $f^k[n]$ gefunden:

$$f^k[n] = \binom{n+k-1}{n} = \frac{(n+k-1)!}{(k-1)!\, n!} \tag{2.69}$$

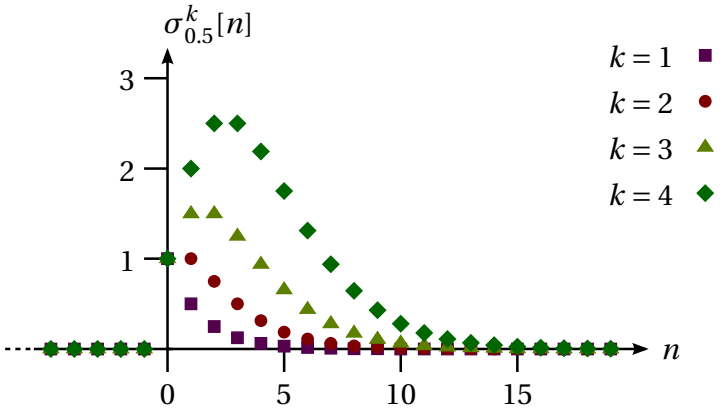

Abbildung 2.15: Pulsfolgen mit Basis $q = 0.5$ und positivem Grad

Die Pulsfolgen mit positivem Grad k sind also:

$$\sigma^k_q[n] = \binom{n+k-1}{n} u[n]\, q^n \tag{2.70}$$

Die Ausdrücke für die Folgen mit $k = 1,2,3,4$ lauten ausgeschrieben:

$$\sigma^1_q[n] = u[n]\, q^n \tag{2.71a}$$
$$\sigma^2_q[n] = (n+1)\, u[n]\, q^n \tag{2.71b}$$
$$\sigma^3_q[n] = \tfrac{1}{2}(n+2)(n+1)\, u[n]\, q^n \tag{2.71c}$$
$$\sigma^4_q[n] = \tfrac{1}{6}(n+3)(n+2)(n+1)\, u[n]\, q^n \tag{2.71d}$$

In Abbildung 2.15 sind einige dieser Folgen mit der Basis $q = 0.5$ gezeichnet.

Negativer Grad

Die Folgen $\sigma^k_q[n]$ mit negativem Grad kann man so finden: Sei der Grad der Folge $-l$ mit einer natürlichen Zahl l, dann ist die z-Transformierte laut Definition:

$$\mathcal{Z}\left\{\sigma^{-l}_q\right\}(z) = \left(1 - qz^{-1}\right)^l \tag{2.72a}$$

Mit dem *Binomischen Lehrsatz* [10] kann man dies so umformen:

$$\mathcal{Z}\left\{\sigma^{-l}_q\right\}(z) = \sum_{m=0}^{l} \binom{l}{m} \left(-qz^{-1}\right)^m = \mathcal{Z}\left\{\sum_{m=0}^{l} \binom{l}{m} (-q)^m \mathscr{D}_m \delta\right\}(z) \tag{2.72b}$$

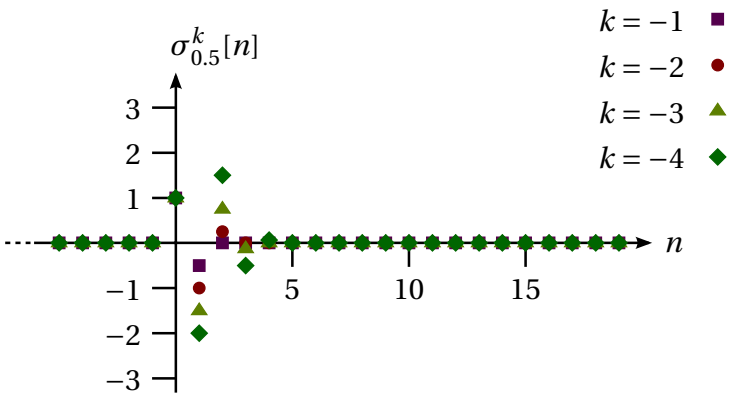

Abbildung 2.16: Pulsfolgen mit Basis $q = 0.5$ und negativem Grad

Es handelt sich also um eine Überlagerung von $l+1$ Einheitsimpulsen. Eine Pulsfolge vom Grad $-l$ hat damit nur $l+1$ von Null verschiedene Werte, die lauten:

$$\sigma_q^{-l}[n] = \begin{cases} \binom{l}{n}(-q)^n & 0 \le n \le l \\ 0 & \text{sonst} \end{cases} \tag{2.73}$$

Die ersten vier Pulsfolgen mit negativem Grad sind für $q = -0.5$ in Abbildung 2.16 gezeigt.

Das Ergebnis dieses Abschnitts ist, dass man nun bei beliebiger gegebener Basis q und gegebenem Grad k die Folgenglieder der Pulsfolge $\sigma_q^k[n]$ kennt.

2.4.3 Rechenregeln

Der eigentliche Nutzen der Pulsfolgen ergibt sich durch die Möglichkeit, die Eingangs- und Ausgangssignale eines Filters sowie das Filter selbst durch sie zu modellieren. Die Berechnung der Ausgangsfolge besteht dann in der Faltung zweier Pulsfolgen. Dafür lassen sich grundlegende Regeln finden, die dies ohne den Umweg der z-Transformierten ermöglichen:

- Die *Potenzregel* für zwei Pulsfolgen mit gleicher Basis,
- die *Aufspaltungsregel* für Pulsfolgen vom Grad Eins und beliebiger Basis,
- und die *Absteigeregel* für eine Pulsfolge vom Grad -1 mit einer beliebigen anderen.

Potenzregel

Für den Fall gleicher Basis kann man das Ergebnis direkt aus der Definition (2.57) und dem Faltungstheorem ableiten:

$$\mathcal{Z}\left\{\sigma_q^k * \sigma_q^l\right\}(z) = \mathcal{Z}\left\{\sigma_q^k\right\}(z) \cdot \mathcal{Z}\left\{\sigma_q^l\right\}(z) = \left(\tfrac{1}{1-qz^{-1}}\right)^{k+l}$$

Daraus folgt die Regel

$$\sigma_q^k * \sigma_q^l = \sigma_q^{k+l} \tag{2.74}$$

Aufspaltungsregel

Wenn die Basis der Folgen verschieden ist, kann man die Potenzregel nicht anwenden. Wenn die Folgen aber beide vom Grad Eins sind, kann man dafür wieder eine Regel angeben. Dazu muss man die Folge finden, deren z-Transformierte

$$\mathcal{Z}\left\{\sigma_q^1 * \sigma_p^1\right\}(z) = \frac{1}{1-qz^{-1}} \frac{1}{1-pz^{-1}} \tag{2.75a}$$

ist. Mithilfe der Partialbruchzerlegung kann man das in eine Summe von Termen umformen, für die man die z-Transformation leicht umkehren kann:

$$\mathcal{Z}\left\{\sigma_q^1 * \sigma_p^1\right\}(z) = \frac{A}{1-qz^{-1}} + \frac{B}{1-pz^{-1}} \tag{2.75b}$$

Indem man die Summanden auf den Hauptnenner bringt und anschließend mit (2.75a) vergleicht, erhält man ein lineares Gleichungssystem für A und B:

$$\begin{array}{l} A \quad + B \quad = 1 \\ Ap + Bq = 0 \end{array} \quad \Leftrightarrow \quad \begin{array}{l} A = \frac{q}{q-p} \\ B = \frac{p}{p-q} = 1 - A \end{array} \tag{2.75c}$$

Aus dem Ansatz (2.75b) und der Lösung (2.75c) ergibt sich nun die Regel

$$\sigma_q^1 * \sigma_p^1 = \tfrac{q}{q-p}\sigma_q^1 + \tfrac{p}{p-q}\sigma_p^1 \tag{2.76}$$

Mit der Aufspaltungsregel kann man also die Faltung durch eine einfache Addition ersetzen. Den Namen der Regel kann man so interpretieren: Von der Eingangsfolge σ_q^1 eines Systems mit der Impulsantwort σ_p^1 bleibt der Anteil $A = \frac{q}{q-p}$ übrig, während der Rest $1 - A = \frac{p}{p-q}$ mit der Basis p abgespalten wird.

Durch wiederholte Anwendung der Aufspaltungsregel kann man auch die Faltung einer

Pulsfolge vom Grad Eins mit einer Pulsfolge beliebigen Grades $k > 1$ berechnen:

$$\begin{aligned}\sigma_q^1 * \sigma_p^k &= \sigma_q^1 * \sigma_p^1 * \sigma_p^{k-1} \\ &= \left(A\,\sigma_q^1 + (1-A)\,\sigma_p^1\right) * \sigma_p^{k-1} \\ &= A\left(\sigma_q^1 * \sigma_p^{k-1}\right) + (1-A)\,\sigma_p^k \end{aligned} \tag{2.77}$$

Für $k = 2,3,4$ ergibt sich damit:

$$\sigma_q^1 * \sigma_p^2 = A^2\,\sigma_q^1 + A\ \ (1-A)\,\sigma_p^1 + \quad\ (1-A)\,\sigma_p^2 \tag{2.78a}$$

$$\sigma_q^1 * \sigma_p^3 = A^3\,\sigma_q^1 + A^2(1-A)\,\sigma_p^1 + A\ \ (1-A)\,\sigma_p^2 + \quad (1-A)\,\sigma_p^3 \tag{2.78b}$$

$$\sigma_q^1 * \sigma_p^4 = A^4\,\sigma_q^1 + A^3(1-A)\,\sigma_p^1 + A^2(1-A)\,\sigma_p^2 + A(1-A)\,\sigma_p^3 + (1-A)\,\sigma_p^4 \tag{2.78c}$$

Absteigeregel

Eine ähnliche Regel gibt es für den Fall, dass eine der beiden Pulsfolgen den Grad -1 hat. Von der Definition (2.57) ausgehend erhält man dann:

$$\mathcal{Z}\left\{\sigma_q^k * \sigma_p^{-1}\right\}(z) = \left(1 - pz^{-1}\right)\mathcal{Z}\left\{\sigma_q^k\right\}(z) = \mathcal{Z}\left\{\sigma_q^k - p\,\mathscr{D}_1\sigma_q^k\right\} \tag{2.79a}$$

Mithilfe der Beziehung (2.62a) kann man (falls $q \neq 0$ ist) $\mathscr{D}_1\sigma_q^k$ durch $\frac{1}{q}(\sigma_q^k - \sigma_q^{k-1})$ ersetzen und die z-Transformation umkehren:

$$\sigma_q^k * \sigma_p^{-1} = \sigma_q^k - \frac{p}{q}\left(\sigma_q^k - \sigma_q^{k-1}\right) \tag{2.79b}$$

Daraus folgt die Absteigeregel:

$$\sigma_q^k * \sigma_p^{-1} = \frac{q-p}{q}\,\sigma_q^k + \frac{p}{q}\,\sigma_q^{k-1} \tag{2.80}$$

Diese kann auf die gleiche Weise wie die Aufspaltungsregel interpretiert werden: Von der Folge σ_q^k bleibt nach Durchlaufen des Systems mit der Impulsantwort σ_p^{-1} der Anteil $\frac{q-p}{q}$ übrig, der Rest $\frac{p}{q}$ steigt im Grad von k auf $k-1$ ab.

3 Tail-Cancellation-Filter

In diesem Kapitel werden zunächst die Entstehung und die wesentlichen Eigenschaften der Eingangssignale des digitalen Filters untersucht. Das Phänomen des sogenannten *Ion Tails* wird mithilfe der in Kapitel 2 eingeführten Pulsfolgen modelliert. Schließlich wird eine Möglichkeit vorgestellt, wie man mit einer einfachen Filterstruktur die *Tail Cancellation* durchführen und damit die sich aus dem *Ion Tail* ergebenden Probleme lösen kann.

3.1 Signalentstehung

3.1.1 Vieldraht-Proportionalkammern

Das SPADIC-Auslesesystem ist im CBM-Experiment dafür vorgesehen, die Signale der Übergangsstrahlungsdetektoren (abgekürzt *TRD* für engl. *transition radiation detector*) aufzunehmen und zu verarbeiten. Die TRDs verwenden Vieldraht-Proportionalkammern (abgekürzt *MWPC* für engl. *multiwire proportional chamber*), um die eintreffenden Teilchen sowie die Übergangsstrahlung nachzuweisen. Die Eigenschaften der entstehenden Signale ergeben sich also aus der Funktionsweise einer MWPC.

Eine MWPC besteht aus einem Gasvolumen zwischen zwei planparallelen elektrisch leitenden Flächen, in dem dünne Drähte gespannt sind. Die Drähte werden auf einem gegenüber den Grenzflächen positiven Potential gehalten, sie bilden also die *Anode* und die Grenzflächen die *Kathode*.

Beim Durchgang von geladenen Teilchen oder elektromagnetischer Strahlung können bei geeigneter Wahl des Gasgemischs, der angelegten Spannung und der Geometrie der Kammer aufgrund der Ionisierung von Gasmolekülen freie Ladungen entstehen, deren Bewegung im elektrischen Feld der Kammer ein messbares Signal in den Elektroden erzeugt.

In Abbildung 3.1 ist der beispielhafte Querschnitt durch eine MWPC gezeigt. Die elektrische Feldstärke E und das Potential ϕ sind [11]:

$$E(x,y) = \frac{CV_0}{2\epsilon_0 s}\left(1+\tan^2\tfrac{\pi x}{s}\tanh^2\tfrac{\pi y}{s}\right)^{1/2}\left(\tan^2\tfrac{\pi x}{s}+\tanh^2\tfrac{\pi y}{s}\right)^{-1/2} \tag{3.1}$$

$$\phi(x,y) = \frac{CV_0}{4\pi\epsilon_0}\left(\frac{2\pi l}{s}-\ln\left(4\left(\sin^2\tfrac{\pi x}{s}+\sinh^2\tfrac{\pi y}{s}\right)\right)\right) \tag{3.2}$$

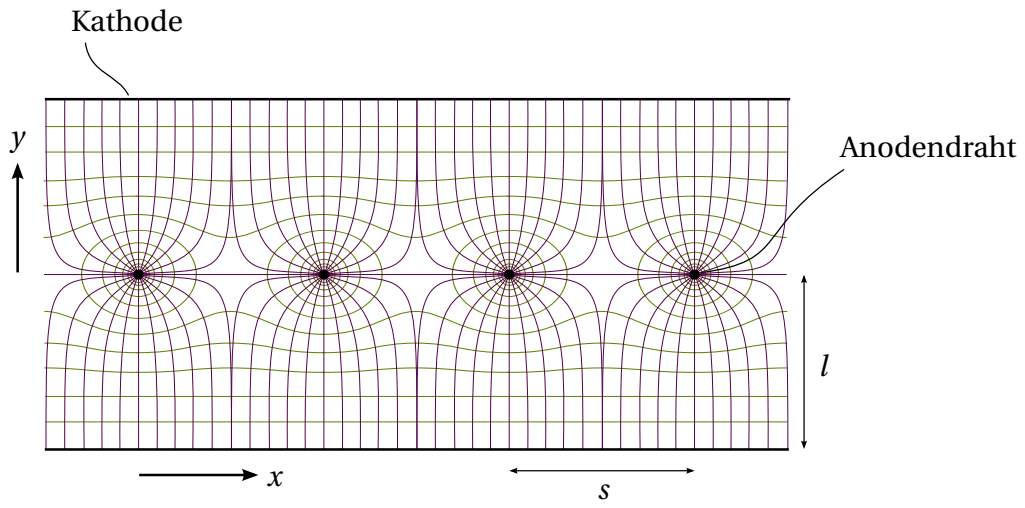

Abbildung 3.1: Querschnitt durch eine MWPC mit Verlauf der elektrischen Feldlinien und Äquipotentiallinien (nach [11])

Dabei ist C die Kapazität pro Längeneinheit in Richtung senkrecht zur (x, y)-Ebene:

$$C = \frac{2\pi\epsilon_0}{(\pi l/s) - \ln(2\pi a/s)} \tag{3.3}$$

Die übrigen verwendeten Formelzeichen sind:

a	Radius der Anodendrähte
s	Abstand der Drähte *(pitch)*
l	Abstand zwischen Anodendraht- und Kathodenebene
V_0	Anoden-Kathoden-Spannung
ϵ_0	Elektrische Feldkonstante

Einige typische Werte für die Kenngrößen der für CBM vorgeschlagenen MWPCs sind [2, 12–14] entnommen:

- Dicke des Gasvolumens: $2l$ = 9–12 mm
- Abstand zwischen den Drähten *(pitch)*: s = 2–5 mm
- Drahtdurchmesser: $2a$ = 20–25 µm
- Gasgemisch: Xe/CO_2 oder Ar/CO_2
- Spannung zwischen Anode und Kathode: V_0 = 1.6–1.8 kV

In der Nähe der Anodendrähte, also für $a < r \ll s$ mit $r^2 = x^2 + y^2$ kann die elektrische Feldstärke wie folgt näherungsweise angegeben werden [11]:

$$E(x,y) = \frac{CV_0}{2\pi\epsilon_0}\,\frac{1}{r} \tag{3.4}$$

Dies entspricht dem Feld eines einzelnen Drahtes, der von einer zylindrischen Kathode umhüllt ist, wie es in einem Proportionalzählrohr der Fall ist. Deshalb kann für das Verständnis der grundlegenden Arbeitsweise einer MWPC die Funktion eines Proportionalzählrohrs betrachtet werden. Das geschieht in [11, S. 40 ff.] ausführlich und wird im folgenden kurz zusammengefasst:

Aufgrund der $1/r$-Abhängigkeit ist die elektrische Feldstärke außerhalb der unmittelbaren Umgebung des Drahtes so gering, dass die durch Ionisation aufgrund einfallender Strahlung entstandenen Elektronen (Primärelektronen) nur in Richtung des Drahtes driften, ohne dabei genügend kinetische Energie zu gewinnen, um weitere Gasmoleküle zu ionisieren. Dies ändert sich aber, sobald die driftenden Elektronen einen gewissen Abstand zum Draht unterschreiten. Dann wird die Feldstärke so groß, dass die Primärelektronen durch Stöße weitere Gasmoleküle ionisieren und eine lawinenartige Ladungsverstärkung stattfindet. Das Resultat dieses Vorgangs ist eine Ladungswolke aus Elektronen und Molekülionen, die den Draht umhüllt.

Dies ist die Ausgangslage für die Entstehung des elektrischen Signals, das auf dem Draht oder der Kathode gemessen werden kann: Die entstandenen Ladungen driften nun weiter zum Draht (Elektronen) bzw. zur Kathode (positive Ionen). Bewegt sich eine Ladung Q im elektrischen Feld E der Kammer mit der Kapazität LC (Länge der Kammer multipliziert mit der Kapazität pro Längeneinheit) um eine Strecke $\mathrm{d}r$, bewirkt das eine Potentialänderung

$$\mathrm{d}u = \frac{Q}{LCV_0}\,E(r)\,\mathrm{d}r \tag{3.5}$$

auf den Elektroden.

Weil die Elektronen in sehr kurzer Zeit aufgrund des geringen Abstands und der hohen Geschwindigkeit bereits die Anode erreichen und von ihr gesammelt werden, tragen sie kaum oder sogar vernachlässigbar wenig [11, S. 44] zum Gesamtsignal bei. *Das Signal entsteht hauptsächlich durch die Driftbewegung der positiv geladenen Ionen vom Draht zur Kathode.*

Die Bewegung der Ionen im Gas mit dem Druck P ergibt sich unter Annahme konstanter Mobilität μ, d. h. Proportionalität von Geschwindigkeit und elektrischer Feldstärke,

$$\frac{\mathrm{d}r}{\mathrm{d}t} = \frac{\mu}{P}\,E, \tag{3.6}$$

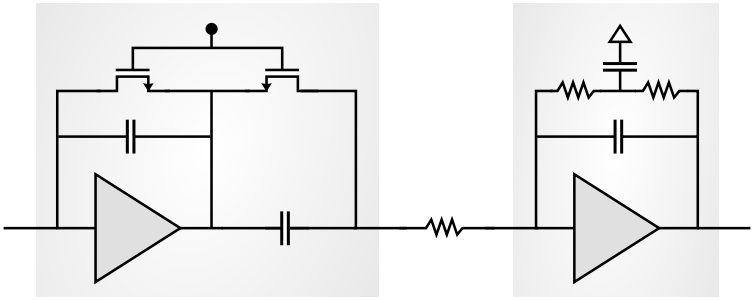

Abbildung 3.2: Verstärkerschaltung im SPADIC-Chip (nach [15])

zu

$$r(t) = a\left(1+\frac{t}{t_0}\right)^{1/2} \quad \text{mit} \quad t_0 = \frac{\pi\epsilon_0 P a^2}{\mu C V_0}. \tag{3.7}$$

Damit kann man dann unter Verwendung von (3.4) und (3.5) das auf der Elektrode induzierte Signal im zeitlichen Verlauf berechnen:

$$u(t) = -\int_0^t \mathrm{d}u = -\frac{Q}{2\pi\epsilon_0 L}\ln\frac{r(t)}{a} = -\frac{Q}{4\pi\epsilon_0 L}\ln\left(1+\frac{t}{t_0}\right) \tag{3.8}$$

Dies entspricht dem Fluss $i(t)$ einer influenzierten Ladung q auf die Elektrode:

$$i(t) = \frac{\mathrm{d}q}{\mathrm{d}t} = LC\frac{\mathrm{d}}{\mathrm{d}t}u(t) = -\frac{QC}{4\pi\epsilon_0}\frac{1}{t_0+t} \tag{3.9}$$

3.1.2 Ausleseelektronik

Im Fall der bei den CBM-TRDs verwendeten Kammern wird das induzierte Signal der Kathoden gemessen. Diese sind zum Zwecke der Ortsbestimmung des eintreffenden Teilchens in sogenannte *Pads* unterteilt [2], an die die Verstärkerkanäle des SPADIC-Chips angeschlossen sind.

Die Verstärkerschaltung besteht aus einem ladungsempfindlichen Vorverstärker und einem *Shaper* [15]. Sie ist in Abbildung 3.2 skizziert.

Im Vorverstärker wird der Ladungspuls $i(t)$ gesammelt und verstärkt. Die aufgenommene Ladung fließt kontinuierlich wieder ab *(continuous reset)*. Das dadurch entstehende Signal x_{pre} wird also durch eine Differenzialgleichung der Form

$$\frac{\mathrm{d}}{\mathrm{d}t}x_{\mathrm{pre}}(t) \propto i(t) - \frac{x_{\mathrm{pre}}(t)}{\tau} \tag{3.10}$$

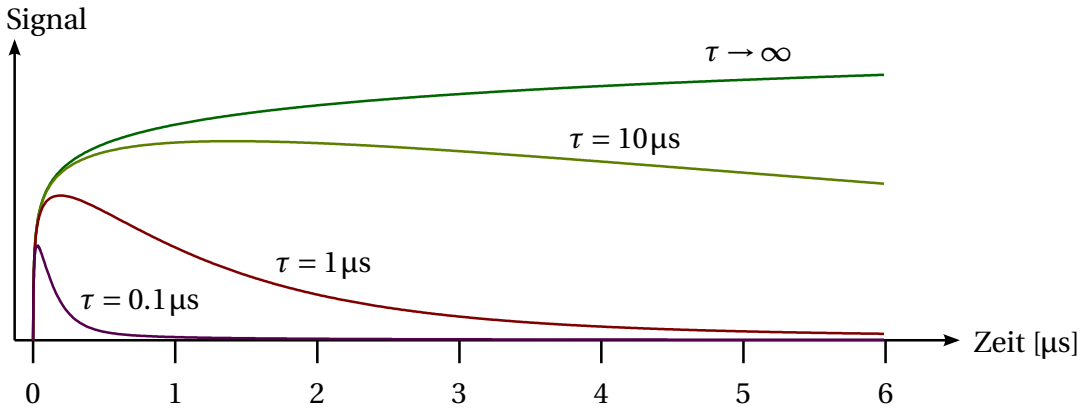

Abbildung 3.3: Ausgangssignal einer MWPC mit verschiedenen Zeitkonstanten der äußeren elektronischen Beschaltung (nach [11])

beschrieben. Dabei wird durch die Zeitkonstante τ das Abfließen der Ladung charakterisiert. In Abbildung 3.3 ist *beispielhaft* ein solches, numerisch berechnetes, Signal für verschiedene Zeitkonstanten gezeigt, wobei $i(t)$ aus Gleichung (3.9) und leicht abgewandelte Zahlenwerte von [11, S. 45] eingesetzt wurden. Die tatsächlichen Werte von t_0 und τ, die die Signalform bestimmen, hängen natürlich maßgeblich von den Parametern der Proportionalkammer und den Eigenschaften des Verstärkers ab, der hier nur stark vereinfacht dargestellt wurde. Es wird davon ausgegangen, dass das Signal ein kurzer Puls von weniger als 10 ns Dauer ist [15].

Unter dem Begriff *Ion Tail* wird nun verstanden, dass das Signal nach dem anfänglichen Puls nicht schnell wieder abklingt. Dies ist die Konsequenz aus der Gleichung (3.8), die besagt, dass die influenzierte Ladung nicht sprunghaft ansteigt und dann konstant bleibt, sondern nach dem schnellen Anstieg (beschrieben durch t_0) noch logarithmisch weiter wächst.

Die Bezeichnung ist insofern etwas irreführend, dass nicht nur der *Tail*, also das langsame Abklingen des Signals, sondern im Grunde das komplette Signal durch die Driftbewegung der Ionen im elektrischen Feld der Anodendrähte entsteht.

Nach dem Vorverstärker passiert das Signal den Shaper, der einen Tiefpass zweiter Ordnung darstellt. Er wird durch die *Impulsantwort* (vgl. Kapitel 2)

$$h_{\mathrm{sh}}(t) = \left(\tfrac{t}{\tau_{\mathrm{sh}}}\right)^2 e^{-t/\tau_{\mathrm{sh}}} \quad \text{für } t \geq 0 \tag{3.11}$$

beschrieben. Die *Shaping-Zeit* τ_{sh} beträgt ca. 80–100 ns [15–17]. In Abbildung 3.4 ist die Impulsantwort des Shapers gezeigt.

Hinter dem analogen Verstärkerteil wird das Signal mit 9 Bit Auflösung und 25 MHz Ab-

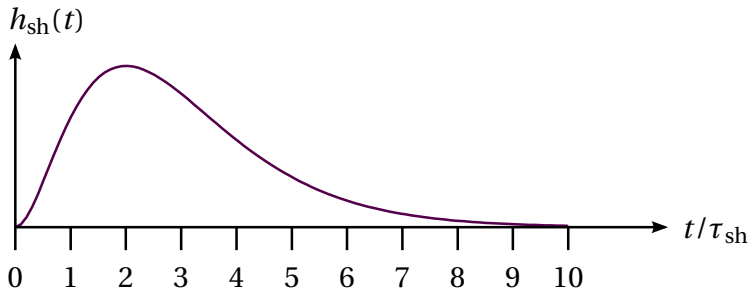

Abbildung 3.4: Impulsantwort $h_{\mathrm{sh}}(t)$ des Shapers

tastrate, d. h. in zeitlichen Abständen von $T = 40\,\mathrm{ns}$ für die weitere Verarbeitung digitalisiert [15]. Die Ausgangswerte des Analog-Digital-Wandlers, der dies durchführt, sind schließlich die Daten, die zum digitalen Filter gelangen, das die *Tail Cancellation* durchführen soll.

3.1.3 Hit Detection

Nachdem das Signal gefiltert wurde, gelangt es in die *Hit Detection*, die ein zentraler Bestandteil des SPADIC-Systems[1] ist: Dort werden aus dem fortlaufenden Signal die relevanten Bestandteile – nämlich die *Hits*; die durch das Eintreffen eines Teilchens im Detektor entstehenden Pulse – extrahiert und Triggersignale erzeugt, das die weiteren Aktivitäten der digitalen Schaltung zur Speicherung der Pulse auslöst.

Da die Erkennung der Pulse auf dem Überschreiten eines Schwellwerts beruht, ist der *Ion Tail* für das zuverlässige Funktionieren hinderlich, wenn mehrere *Hits* in kurzer Abfolge geschehen. Dieses Problem wird *Pile-Up* genannt und ist in Abbildung 3.5 illustriert. Daraus ergibt sich die eigentliche Notwendigkeit für die digitale Filterung der Detektorsignale.

3.2 Modellierung mit Pulsfolgen

3.2.1 Eingangssignal des Filters

Um das Ausgangssignal der Proportionalkammer in Kombination mit dem Vorverstärker, aber vor dem Shaper, zu beschreiben, kann der Ansatz einer Überlagerung mehrerer exponentiell abfallender Teilsignale gewählt werden [16, 18, 19]:

$$x_{\mathrm{pre}}(t) = \sum_{i=1}^{M} w_i \, e^{-t/\tau_i} \quad \text{für } t \geq 0 \tag{3.12}$$

[1] *Self Triggered* ...

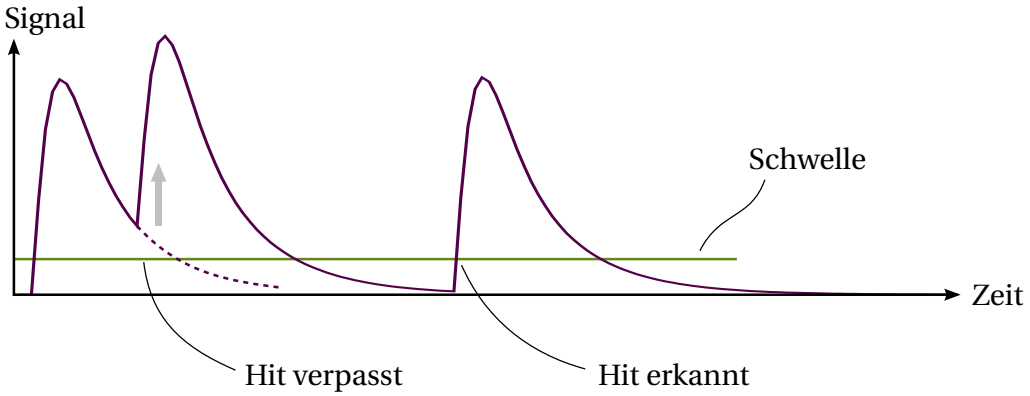

Abbildung 3.5: *Pile Up*: Das langsame Abklingen erschwert die Erkennung der Pulse durch Überschreiten einer Schwelle.

Dabei seien die einzelnen Zeitkonstanten τ_i der Größe nach sortiert, also $0 < \tau_1 < \cdots < \tau_M$, und die Gewichtungsfaktoren w_i der Einzelsignale positiv. Kleine Zeitkonstanten repräsentieren dabei schnell abfallende, pulsförmige Signalanteile und große Zeitkonstanten den langsam abfallenden *Tail*. Ziel der *Tail Cancellation* ist es nun, die langsamen Komponenten auszulöschen, oder zumindest zu unterdrücken, und die schnellen Komponenten hervorzuheben. Das bedeutet, dass in diesem Modell die zu großen Zeitkonstanten gehörenden Gewichtungsfaktoren relativ zu den zu kleinen Zeitkonstanten gehörenden verkleinert werden sollen. Das *ideale* Ergebnis der *Tail Cancellation* wäre ein Signal mit einer einzelnen Komponente mit sehr kleiner Zeitkonstante.

Nach der Zeitdiskretisierung kann das Signal als *Folge* im Sinne der in Kapitel 2 eingeführten Bezeichnung aufgefasst werden. Der Vergleich von (3.11) und (3.12) mit (2.71c) bzw. (2.71a) legt nahe, das Eingangssignal des Filters mit den Pulsfolgen zu modellieren:

$$x = x_{\text{pre}} * h_{\text{sh}} = \left(\sum_{i=1}^{M} w_i \, \sigma^1_{q_i} \right) * \sigma^3_{q_{\text{sh}}} \tag{3.13}$$

Die Basen q der Pulsfolgen hängen dabei folgendermaßen mit den Zeitkonstanten τ und der Abtastperiode T zusammen:

$$q = e^{-T/\tau} \quad \text{bzw.} \quad \tau = -\frac{T}{\ln q} \tag{3.14}$$

Da die Zeitkonstanten nicht negativ sind, liegen die Basen im Intervall $[0, 1]$ (sind also auch reell), wobei für $\tau \to 0$ auch $q \to 0$ und für immer größer werdende Zeitkonstanten $q \to 1$ gilt. Für eine angenommene Shaping-Zeit von $\tau_{\text{sh}} = 90\,\text{ns}$ ist beispielsweise die zugehörige Basis $q_{\text{sh}} \approx 0.64$.

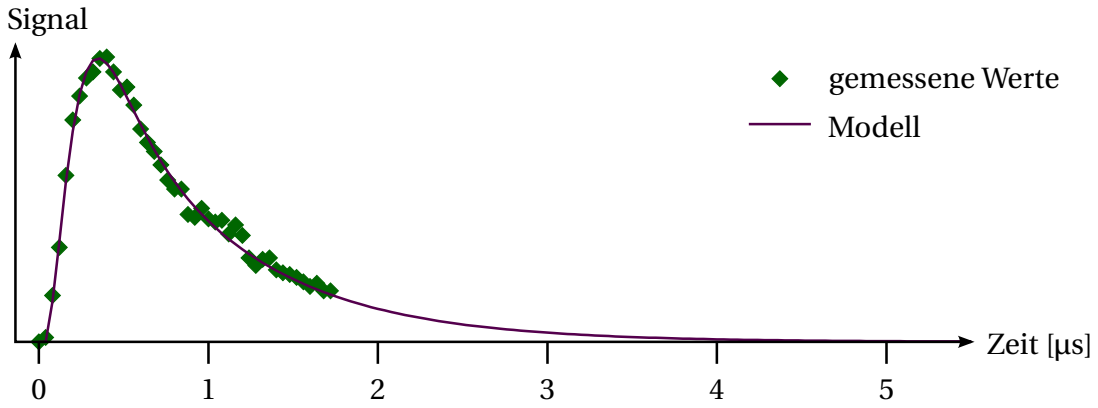

Abbildung 3.6: Modellierung der Eingangssignale des Filters mit Pulsfolgen

In Abbildung 3.6 ist ein mit einem Prototyp einer MWPC, die im Rahmen eines Strahltests im CERN im November 2010 an den SPADIC-Chip angeschlossen war, erzeugter und digitalisierter Puls zu sehen. Aufgrund der Architektur der digitalen Datenauslese konnten für jeden *Hit* 45 aufeinanderfolgend abgetastete Werte gespeichert und ausgegeben werden, weshalb die Messpunkte bei 1.8 µs enden. Zusätzlich zu den Messpunkten ist eine Kurve der Form (3.13) eingezeichnet, was zeigt, dass dieses mathematische Modell für das Signal gerechtfertigt ist. Es wurden für $M = 4$ die folgenden Modellparameter gewählt:

relative Gewichtung	Basis	Zeitkonstante
$w_1 = 5$	$q_1 = 0.10$	$\tau_1 = 17\,\text{ns}$
$w_2 = 1$	$q_2 = 0.30$	$\tau_2 = 33\,\text{ns}$
$w_3 = 4$	$q_3 = 0.70$	$\tau_3 = 112\,\text{ns}$
$w_4 = 5$	$q_4 = 0.95$	$\tau_4 = 780\,\text{ns}$

3.2.2 Filterstruktur

Eine der verschiedenen Möglichkeiten, ein zeitdiskretes Filter allgemein zu beschreiben ist eine Reihenschaltung von Systemen erster Ordnung, wie in Abbildung 3.7 gezeigt. Auch in der digitalen Signalverarbeitung für die *Time Projection Chamber* im ALICE-Experiment wird eine solche Filterstruktur eingesetzt [18]. Sie entspricht der faktorisierten Form der Systemfunktion:

$$H(z) = c \prod_{i=1}^{N} \frac{1 + b_i\, z^{-1}}{1 - a_i\, z^{-1}} \tag{3.15}$$

Der Vorteil dieser Darstellung ist, dass man die Impulsantwort direkt in Form von Pulsfolgen angeben kann:

$$h = c\left(\left(\sigma^1_{a_1} * \sigma^{-1}_{-b_1}\right) * \cdots * \left(\sigma^1_{a_N} * \sigma^{-1}_{-b_N}\right)\right) \tag{3.16}$$

Unter Annahme eines Eingangssignals x in der im vorherigen Abschnitt beschriebenen und begründeten Form lautet dann das Ausgangssignal y des Filters:

$$\begin{aligned} y &= x * h = x_{\text{pre}} * h_{\text{sh}} * h \\ &= \left(\sum_{i=1}^{M} w_i \sigma^1_{q_i}\right) * \sigma^3_{q_{\text{sh}}} * \left(c\left(\sigma^1_{a_1} * \sigma^{-1}_{-b_1}\right) * \cdots * \left(\sigma^1_{a_N} * \sigma^{-1}_{-b_N}\right)\right) \end{aligned} \tag{3.17a}$$

Prinzipiell kann nun mithilfe der Rechenregeln für die Pulsfolgen das Ausgangssignal bestimmt werden. Es ist aber ausreichend und bedeutend einfacher, die Wirkung eines einzelnen Teilsystems erster Ordnung auf eine einzelne Komponente des Eingangssignals zu betrachten, um die Wirkung des ganzen Filters zu verstehen, denn (3.17a) lässt sich so umformen:

$$y = c\left(\sum_{i=1}^{M} w_i \underbrace{\sigma^1_{q_i} * \left(\sigma^1_{a_1} * \sigma^{-1}_{-b_1}\right)}_{y_i}\right) * \left(\left(\sigma^1_{a_2} * \sigma^{-1}_{-b_2}\right) * \cdots * \left(\sigma^1_{a_N} * \sigma^{-1}_{-b_N}\right)\right) \tag{3.17b}$$

Der Ausdruck y_i ergibt (wobei $a_1 \to a$ und $b_1 \to b$ abgekürzt wird):

$$\begin{aligned} y_i &= \sigma^1_{q_i} * \left(\sigma^1_a * \sigma^{-1}_{-b}\right) \\ &= \sigma^1_{q_i} * \left(\tfrac{a+b}{a}\sigma^1_a + \tfrac{-b}{a}\delta\right) && \text{(Absteigeregel)} \\ &= \tfrac{a+b}{a}\left(\sigma^1_{q_i} * \sigma^1_a\right) + \tfrac{-b}{a}\sigma^1_{q_i} \\ &= \tfrac{a+b}{a}\left(\tfrac{q_i}{q_i-a}\sigma^1_{q_i} + \tfrac{a}{a-q_i}\sigma^1_a\right) + \tfrac{-b}{a}\sigma^1_{q_i} && \text{(Aufspaltungsregel)} \\ &= \tfrac{q_i+b}{q_i-a}\sigma^1_{q_i} + \tfrac{a+b}{a-q_i}\sigma^1_a \end{aligned} \tag{3.18}$$

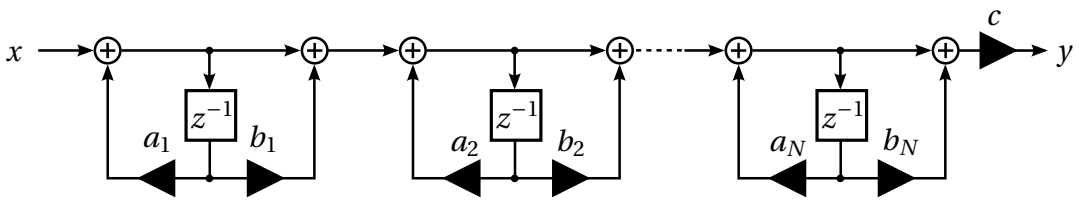

Abbildung 3.7: Reihenschaltung aus Teilsystemen erster Ordnung

Das bedeutet: Aus einer Komponente $w_i \sigma^1_{q_i}$ wird nach dem Passieren eines Filters erster Ordnung mit dem Pol a und der Nullstelle $-b$ die Folge $w'_i \sigma^1_{q_i} + (w_i - w'_i)\sigma^1_a$ mit der *relativen Änderung des Gewichtungsfaktors*

$$\frac{w'_i}{w_i} = \frac{q_i + b}{q_i - a}. \tag{3.19}$$

Jede der Komponenten erfährt so, je nach ihrer Basis q_i, eine Gewichtungsänderung, und die Gesamtänderung aller Gewichte w_i wird durch einen neuen Signalanteil mit der Basis a und dem Gewicht $w_a = \sum_{i=1}^{M}(w_i - w'_i)$ kompensiert. Die Wirkung des Filters besteht also darin, das Verhältnis der einzelnen Anteile untereinander umzuverteilen.

3.3 Wahl der Koeffizienten

3.3.1 Eingrenzung des Bereichs

Um die Suche nach geeigneten Koeffizienten a und b zu vereinfachen, kann man zunächst einmal betrachten, welcher Wertebereich überhaupt sinnvoll ist.

Da das Filter auf jeden Fall stabil sein soll, müssen alle Pole innerhalb des Einheitskreises in der komplexen Zahlenebene liegen. Der Parameter a stellt einen Pol dar, also muss $|a| < 1$ gelten.

Da wie gezeigt ein neuer Signalanteil σ^1_a entsteht, muss dessen Basis a so gewählt sein, dass dies keine unerwünschten Auswirkungen hat. Wäre a negativ oder nicht reell, also $\phi \neq 0$ in der polaren Darstellung $a = r\,e^{i\phi}$ mit $r \geq 0$, würde diese Folge ein oszillierendes Signal beschreiben:

$$\sigma^1_a[n] = r^n\, e^{in\phi} = r^n \left(\cos n\phi + i \sin n\phi\right) \tag{3.20}$$

Dies wäre für das Funktionieren der *Hit Detection* störend, also kann man a weiter einschränken:

$$a \in [0, 1) \subset \mathbb{R} \tag{3.21}$$

Für die Nullstelle $-b$ liegt es nahe, eine der Basen q_i zu wählen, um so die entsprechende Komponente ganz auszuschalten:

$$w'_i\Big|_{-b=q_i} = w_i \left.\frac{q_i + b}{q_i - a}\right|_{-b=q_i} = 0 \tag{3.22}$$

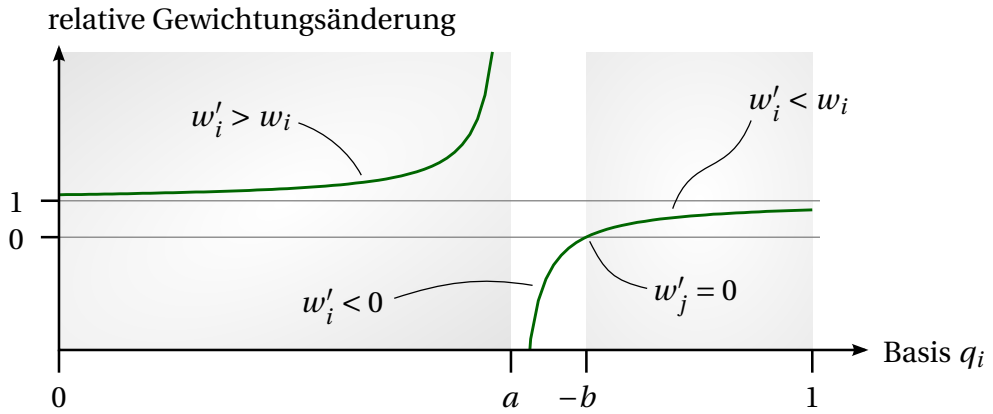

Abbildung 3.8: Relative Gewichtungsänderung w'_i / w_i der Signalkomponenten in Abhängigkeit der Basis bei gegebenen Koeffizienten $-b = q_j$ und $a < q_j$ nach (3.19)

Da alle Basen $0 \leq q_i \leq 1$ sind, gilt:

$$b \in [-1, 0] \subset \mathbb{R} \tag{3.23}$$

Sei also $-b = q_j$ für ein ausgesuchtes $j \in \{2, \ldots, M\}$. Dann hat man die j-te Signalkomponente aus (3.13) mit der Zeitkonstante $\tau_j = (-T / \ln q_j)$ ausgelöscht ($j = 1$ wird aus einem später ersichtlichen Grund zunächst ausgeschlossen, es wäre aber sowieso nicht vorteilhaft, ausgerechnet die schnellste Komponente auszulöschen). Gleichzeitig ist aber eine zusätzliche Komponente σ_a^1 mit der Zeitkonstante $\tau_a = (-T / \ln a)$ neu hinzugekommen. Natürlich ist es mit der Absicht der *Tail Cancellation* nicht hilfreich, einen langsamen Teil des Signals gegen einen noch langsameren auszutauschen. Deshalb gilt bei sinnvollen Werten für die Koeffizienten immer:

$$a < -b \quad \Rightarrow \quad a \in [0, q_j) \tag{3.24}$$

Mit dieser Vorgabe kann man betrachten, wie sich für ein gegebenes $-b = q_j$ und gegebenes $a < q_j$ die Gewichtungsfaktoren der übrigen Komponenten $i = 1, \ldots, j-1, j+1, \ldots, M$ verhalten. Der Verlauf der relativen Gewichtungsänderung (3.19) in Abhängigkeit der Basis einer Komponente ist in Abbildung 3.8 gezeigt. Man kann dabei drei Bereiche unterscheiden:

- Komponenten mit $q_i < a$ werden verstärkt,
- Komponenten mit $a < q_i < -b$ werden negativ,
- Komponenten mit $q_i > -b$ werden abgeschwächt.

Wählt man nun a so, dass keine der Basen q_i zwischen a und $-b$ liegt, d. h.

$$a \in (q_{j-1}, q_j) \tag{3.25}$$

hat man grundsätzlich eine gute Konfiguration gefunden, denn die langsamen Komponenten mit den Basen $q_{j+1}, \ldots, q_M$ verlieren gegenüber den schnellen Komponenten mit den Basen $q_1, \ldots, q_{j-1}$ an Gewicht, während alle Gewichte positiv bleiben und somit die grundsätzliche Signalform erhalten bleibt.

3.3.2 Ausschalten einzelner Komponenten

Nun kann man untersuchen, ob es gelingt, einen genauen Wert für a zwischen q_{j-1} und q_j zu finden, so dass die Gewichtung w_a des neuen Signalanteils mit der Basis a verschwindet. Dann hätte man nicht nur einen der M Anteile (nämlich den j-ten mit der Basis q_j) durch einen anderen (mit der Basis a) ersetzt, sondern tatsächlich nur noch $M-1$ Anteile.

Dazu muss man den Verlauf von w_a in Abhängigkeit von a bei gegebenem $-b = q_j$ betrachten (nach 3.18):

$$w_a\Big|_{-b=q_j} = \sum_{i=1}^{M} w_i \frac{a+b}{a-q_i}\Bigg|_{-b=q_j} = (a-q_j) \underbrace{\sum_{i=1}^{M} w_i \frac{1}{a-q_i}}_{f(a)} \tag{3.26}$$

Weil der Vorfaktor nach Voraussetzung $(a-q_j) \neq 0$ ist, muss man also eine Nullstelle der Funktion $f(a)$ finden, die zwischen q_{j-1} und q_j liegt. Das Schaubild der Funktion $f(a)$ ist für $M = 4$ beispielhaft in Abbildung 3.9 gezeigt. Es besteht aus der Überlagerung von um q_i verschobenen und mit w_i skalierten Kurven vom Typ $1/a$, hat also bei jedem der q_i eine senkrechte Asymptote. *Weil alle w_i positiv sind*, gibt es zwischen jeweils zwei aufeinanderfolgenden Asymptoten bei q_{i-1} und q_i eine Nullstelle α_i $(i = 2, \ldots, M)$, also insbesondere auch eine Nullstelle α_j zwischen q_{j-1} und q_j. Die genauen Werte der Nullstellen von $f(a) = \sum_{i=1}^{M} w_i/(a-q_i)$ kann man bei gegebenen w_i und q_i numerisch bestimmen, etwa mit dem *Newton-Verfahren*.

Das Ergebnis dieser Untersuchungen ist: Wenn das Eingangssignal des Filters durch eine Folge der Form (3.13) – d. h. aus M Komponenten mit positiven Gewichtungsfaktoren $w_1, \ldots, w_M$ und Basen $q_1, \ldots, q_M$ zwischen Null und Eins – gegeben ist, dann kann man mit einer Filterstufe erster Ordnung mit der Nullstelle $-b$ und dem Pol a eine der M Komponenten mit der Basis q_j, $j \in \{2, \ldots, M\}$ auslöschen, indem man

- $-b = q_j$ und
- $a = \alpha_j$, die Nullstelle der Funktion $f(a)$ zwischen q_{j-1} und q_j,

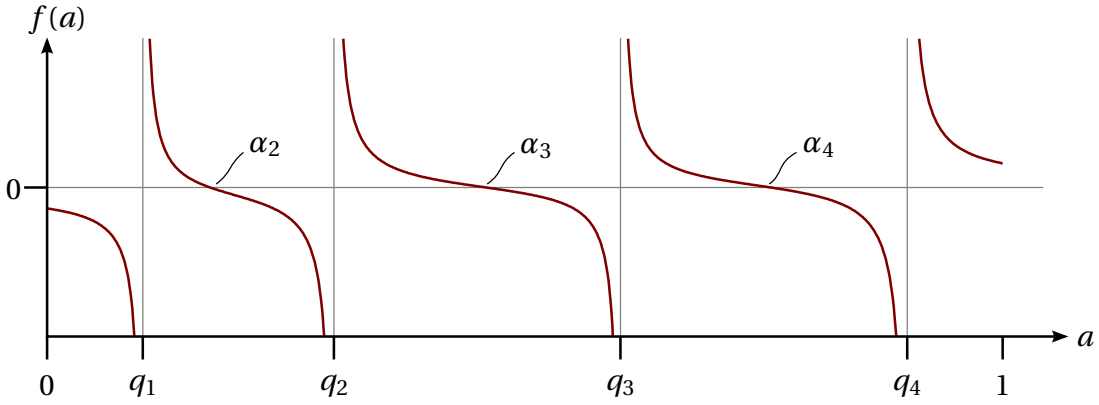

Abbildung 3.9: An den Nullstellen der Funktion $f(a)$ verschwindet der Gewichtungsfaktor w_a des Signalanteils σ_a^1.

setzt. Das Ausgangssignal dieser Filterstufe ist wieder eine Folge der Form (3.13), aber besteht aus nur $M-1$ Komponenten mit erneut positiven Gewichtungsfaktoren. Alle Signalanteile mit größeren Basen als q_j wurden abgeschwächt und alle Signalanteile mit kleineren Basen als q_j wurden verstärkt.

Diesen Vorgang kann man durch eine Aneinanderreihung von mehreren solcher Einzelfilter erster Ordnung wiederholen, um Schritt für Schritt die Anzahl der Komponenten, aus denen das Signal zusammengesetzt ist, zu verringern.

Ein Sonderfall tritt dabei auf, nachdem man den Vorgang $M-1$-mal durchgeführt hat und noch das aus der einzelnen Komponente mit der Basis q_1 bestehende Signal übrig ist. Dann sind alle Nullstellen $\alpha_2,\ldots,\alpha_M$ der Funktion $f(a)$ „verbraucht“ und es entsteht durch Anwenden einer weiteren, M-ten, Filterstufe mit Pol a zwangsläufig ein Signal σ_a^1, welches das Signal $\sigma_{q_1}^1$ ersetzt. Dann kann man aber $a=0$ wählen, so dass $\sigma_a^1 = \delta = x_{\text{pre}} * h$ übrig bleibt. Das Ausgangssignal von M Filterstufen erster Ordnung ist dann:

$$y = (x_{\text{pre}} * h) * h_{\text{sh}} = w\, h_{\text{sh}} \quad \text{mit} \quad w = \sum_{i=1}^{M} w_i \tag{3.27}$$

3.3.3 Anwendungsbeispiel

Sei die Eingangsfolge des Filters $x = \sum_{i=1}^{4} w_i \sigma_{q_i}^1 * \sigma_{q_{\text{sh}}}^3$ mit den Parametern, die das Signal aus Abbildung 3.6 beschreiben:

$$w_1 = 5 \qquad q_1 = 0.10$$
$$w_2 = 1 \qquad q_2 = 0.30$$
$$w_3 = 4 \qquad q_3 = 0.70$$
$$w_4 = 5 \qquad q_4 = 0.95$$

Es soll nun durch vier Filterstufen erster Ordnung jeweils eine Komponente ausgelöscht und dafür die Filterkoeffizienten $a_1, b_1, \ldots, a_4, b_4$ gefunden werden. Ein Paar (a_j, b_j) bezeichne hier die Koeffizienten derjenigen Filterstufe, die die j-te Komponente auslöscht. Die Nullstellen der Funktion $f(a) = \frac{5}{a-0.1} + \frac{1}{a-0.3} + \frac{4}{a-0.7} + \frac{5}{a-0.95}$ sind:

$$\alpha_2 \approx 0.246$$
$$\alpha_3 \approx 0.421$$
$$\alpha_4 \approx 0.826$$

Daraus ergeben sich die Koeffizienten:

$$a_1 = 0 \qquad b_1 = -q_1 - 0.10$$
$$a_2 = \alpha_2 \approx 0.246 \qquad b_2 = -q_2 - 0.30$$
$$a_3 = \alpha_3 \approx 0.421 \qquad b_3 = -q_3 - 0.70$$
$$a_4 = \alpha_4 \approx 0.826 \qquad b_4 = -q_4 - 0.95$$

Die Gewichtungsfaktoren ändern sich nach jeder Stufe wie in der folgenden Tabelle aufgeführt:

w_1	w_2	w_3	w_4
5	1	4	5
5.851	1.235	7.914	—
10.927	4.073	—	—
15	—	—	—

In Abbildung 3.10 sind das ursprüngliche Signal und die Ausgangssignale der einzelnen Filterstufen gezeigt. Man kann erkennen, dass die Auslöschung der langsamen Signalkomponenten mit $q_4 = 0.95$ und $q_3 = 0.7$ eine viel größere Auswirkung hat als die Auslöschung

der schnelleren Anteile.

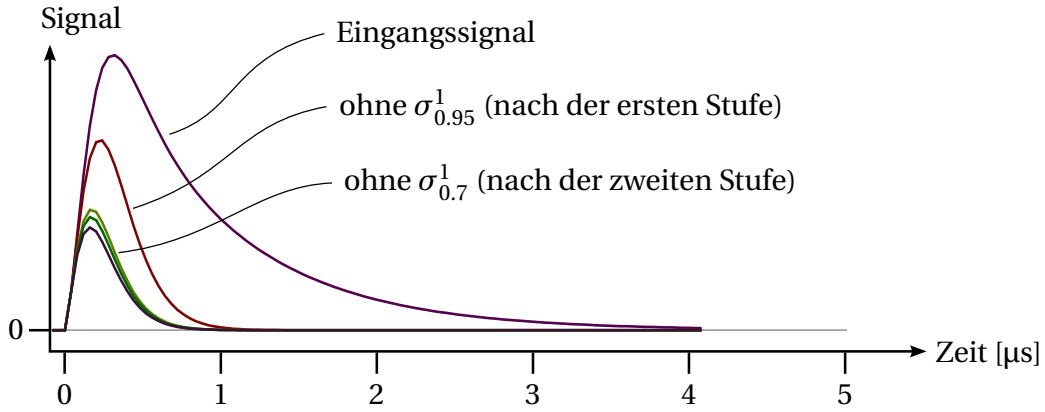

Abbildung 3.10: Aus dem Eingangssignal werden schrittweise die langsamen Anteile ausgelöscht.

3.3.4 Abweichung der Koeffizienten

Die zuvor vorgestellte Methode, die Koeffizienten a_i und b_i für das Tail-Cancellation-Filter zu wählen, beruht darauf, dass die Modellparameter des Eingangssignals genau bekannt sind und dass die Koeffizienten genau die ermittelten Werte annehmen können.

Nun wird untersucht, was die Abweichung der Koeffizienten von diesen Werten zur Folge hat. Eine Abweichung wird sich in der Praxis aus zwei Gründen nicht vermeiden lassen:

- Die Parameter q_i und w_i sind nur geschätzt und das tatsächliche Signal kann von der modellierten Form abweichen.
- Die Koeffizienten können keine beliebigen Werte annehmen, sondern müssen einer vorgegebenen Auswahl entnommen werden (Quantisierung).

Zunächst seien die Koeffizienten b_i, durch die die Nullstellen $-b_i$ der einzelnen Filterstufen dargestellt werden, ohne Abweichung von den ermittelten Werten, d. h. $-b_i = q_i$. Die Koeffizienten a_i, die die Pole der Systemfunktion des Filters sind, seien aber einmal größer (a'_i) als die berechneten Werte α_i und einmal kleiner (a''_i) – jedoch immer im Bereich zwischen Null und Eins, um Oszillationen und Instabilität auszuschließen:

$$
\begin{array}{lll}
a''_1 = 0 = a_1 & a_1 = 0 & a'_1 = 0.08 > a_1 \\
a''_2 = 0.12 < a_2 & a_2 = \alpha_2 \approx 0.246 & a'_2 = 0.26 > a_2 \\
a''_3 = 0.32 < a_3 & a_3 = \alpha_3 \approx 0.421 & a'_3 = 0.50 > a_3 \\
a''_4 = 0.72 < a_4 & a_4 = \alpha_4 \approx 0.826 & a'_4 = 0.88 > a_4
\end{array}
$$

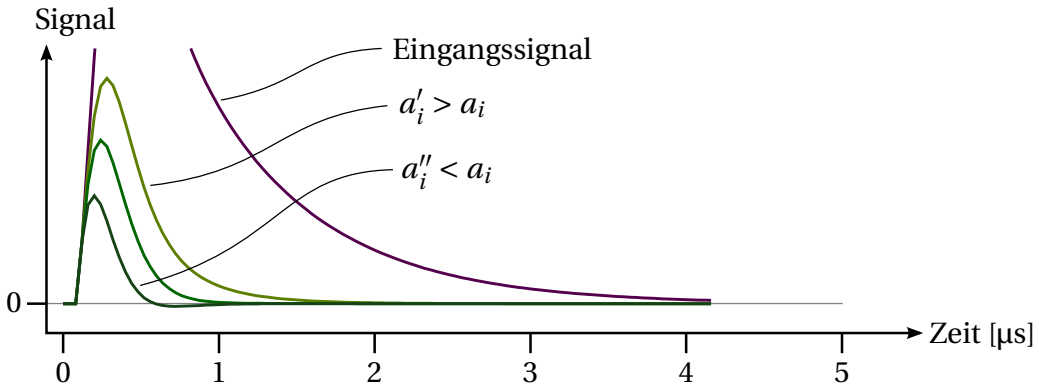

Abbildung 3.11: Abweichung der Koeffizienten a_i von den berechneten Werten

In Abbildung 3.11 ist das Ausgangssignal für die drei Fälle (keine Abweichung, Abweichung nach unten, Abweichung nach oben) gezeigt. Es wurde diesmal nicht wie im vorherigen Abschnitt analytisch mithilfe der Rechenregeln für die Pulsfolgen berechnet, sondern mit der Simulationssoftware `iirsim` (siehe Kapitel 5). Man kann sehen, dass die Wirkung des Filters abgeschwächt wird, wenn die Koeffizienten a_i größer werden, und die Wirkung verstärkt wird, wenn sie kleiner werden. Dabei nimmt auch die maximale Amplitude des Signals ab, und es kann dazu kommen, dass das Signal kurzzeitig negative Werte annimmt.

Nun seien die a_i auf den berechneten Werten α_i (bzw. $a_1 = 0$) festgehalten und stattdessen gebe es eine Abweichung der Koeffizienten b_i von den Werten $-q_i$:

$$
\begin{array}{lll}
b_1'' = -0.15 < b_1 & b_1 = -q_1 = -0.10 & b_1' = -0.08 > b_1 \\
b_2'' = -0.50 < b_2 & b_2 = -q_2 = -0.30 & b_2' = -0.22 > b_2 \\
b_3'' = -0.80 < b_3 & b_3 = -q_3 = -0.70 & b_3' = -0.60 > b_3 \\
b_4'' = -0.98 < b_4 & b_4 = -q_4 = -0.95 & b_4' = -0.91 > b_4
\end{array}
$$

Auch hier bewirkt eine Erhöhung der Koeffizienten b_i (d. h. in diesem Fall eine *Verringerung des Betrags* $-b_i$) die Abmilderung des Filtereffekts und eine Verringerung der Koeffizienten (also hier *Erhöhung des Betrags*) eine Intensivierung, die auch damit verbunden ist, dass das Signal unter die Nulllinie gerät, wie in Abbildung 3.12 zu sehen.

Man hat also für $-b_i = q_i$ und $a_i = \alpha_i < q_i$ eine Konfiguration, bei der – ohne, dass das Signal negativ wird – alle Komponenten $\sigma^1_{q_i}$ des Eingangssignals ausgelöscht werden und das Ausgangssignal nur noch aus der Impulsantwort h_{sh} des Shapers besteht. Wenn man die Koeffizienten zur positiven Seite hin verschiebt, hat man die Möglichkeiten des Filters bezüglich Tail Cancellation nicht ausgenutzt. Ob und wie weit man die Koeffizienten zur negativen Seite hin verschieben kann, um damit die Wirkung des Filters zu verstärken, hängt

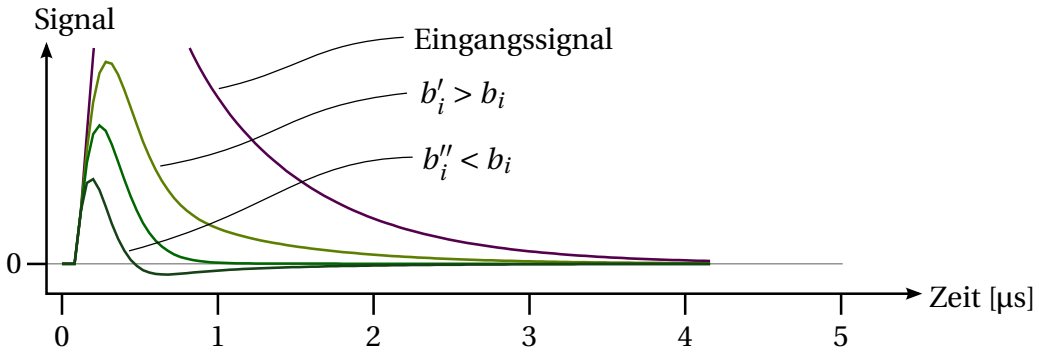

Abbildung 3.12: Abweichung der Koeffizienten b_i von den berechneten Werten

davon ab, in welchem Ausmaß negative Anteile in der Signalform eines *Hits* für die spätere Verarbeitung, d. h. zunächst die *Hit Detection*, unerwünscht sind.

3.3.5 Langsame Schwankungen des Signals

Bisher wurde immer angenommen, dass das Eingangssignal x des Filters konstant Null ist, solange keine Aktivität in der MWPC stattfindet. Aufgrund äußerer Störeinflüsse, der Temperaturabhängigkeit der Verstärkerschaltung oder Leckströmen sind dem Signal aber möglicherweise (im Verhältnis zur Dauer der Pulse) langsame Schwankungen überlagert [15]. In einem *einfachen* Modell ist das Eingangssignal dann

$$x = \left(x_{\text{pre}} * h_{\text{sh}}\right) + x_{\text{bas}} \quad \text{mit} \quad x_{\text{bas}}[n] = C\, e^{i(\omega n + \phi)}, \quad \omega \approx 0. \tag{3.28}$$

Dem bisher betrachteten Eingangssignal $x_{\text{pre}} * h_{\text{sh}}$ ist also eine sog. *Baseline* x_{bas} überlagert. Auch diese sollte für das Funktionieren der *Hit Detection* möglichst unterdrückt werden *(Baseline-Korrektur)* [15].

Wegen der Linearität des Filters T ist die neue Ausgangsfolge die Überlagerung des bisherigen Ausgangssignals aus (3.27) mit der Antwort des Filters auf die *Baseline*:

$$T\{x\} = T\{x_{\text{pre}} * h_{\text{sh}} + x_{\text{bas}}\} = T\{x_{\text{pre}} * h_{\text{sh}}\} + T\{x_{\text{bas}}\} \tag{3.29}$$

Die Wirkung eines LTI-Systems T mit der Systemfunktion H auf eine Folge der Form von x_{bas} ist die *Skalierung* mit dem Wert der Systemfunktion an der Stelle $e^{i\omega}$ [5, S. 44]:

$$T\{x_{\text{bas}}\} = H(e^{i\omega})\, x_{\text{bas}} \tag{3.30}$$

Mit der Systemfunktion aus (3.15) ergibt sich:

$$H(e^{i\omega})\Big|_{w\approx 0} = H(z)\Big|_{z\approx 1} \approx c \prod_{i=1}^{N} \frac{1+b_i}{1-a_i} \tag{3.31}$$

Die langsamen Schwankungen werden also relativ zu den Pulsen unterdrückt, wenn das Produkt aus den Brüchen $(1+b_i)/(1-a_i)$ kleiner als Eins ist (der Skalierungsfaktor c wirkt gleichermaßen auf die Pulse wie auf die Baseline und spielt deshalb keine Rolle). Durch die Forderung, dass für ein Paar (a_i, b_i) der Koeffizienten einer Filterstufe erster Ordnung in einer sinnvollen Konfiguration immer $a_i < -b_i$ gilt (siehe 3.24), ist dies automatisch erfüllt und die *Tail Cancellation* steht im Einklang mit der *Baseline-Korrektur*.

3.4 Reihenschaltung von Systemen zweiter Ordnung

Wenn die Koeffizienten auf reelle Zahlen beschränkt sind (wie es in einer üblichen Implementierung des Filters als elektronische Schaltung der Fall ist, siehe Kapitel 4), ist es mit einer Filterstruktur wie in Abbildung 3.7, das aus einer Reihenschaltung von Systemen erster Ordnung besteht, nicht möglich, ein System mit nicht-reellen Polen und Nullstellen zu realisieren. Nach (2.30e) kann man aber mit einer Reihenschaltung aus Systemen zweiter Ordnung trotz reeller Koeffizienten nicht-reelle Pole oder Nullstellen erzeugen, die paarweise komplex konjugiert sind.

Eine Filterstufe zweiter Ordnung mit der Systemfunktion

$$H(z) = \frac{1 + b_1\, z^{-1} + b_2\, z^{-2}}{1 - a_1\, z^{-1} - a_2\, z^{-2}} = \frac{\left(1 - d_1\, z^{-1}\right)\left(1 - d_2\, z^{-1}\right)}{\left(1 - c_1\, z^{-1}\right)\left(1 - c_2\, z^{-1}\right)} \tag{3.32}$$

hat beispielsweise die beiden Pole

$$c_{1,2} = \frac{a_1}{2} \pm \sqrt{\left(\frac{a_1}{2}\right)^2 + a_2}. \tag{3.33}$$

Falls $a_2 \geq -a_1^2/4$ ist, sind beide reell, andernfalls bilden sie ein komplex konjugiertes Paar mit dem Realteil $a_1/2$ und dem Imaginärteil $\pm\sqrt{-a_1^2/4 - a_2}$. Damit das System stabil ist, müssen beide Pole innerhalb des Einheitskreises der komplexen Zahlenebene liegen, d. h. $\max|c_{1,2}| < 1$. Eine genauere Untersuchung dieser Bedingung liefert das folgenden Stabilitätskriterium für die Koeffizienten a_1 und a_2:

$$-1 < a_2 < 1 - |a_1| \tag{3.34}$$

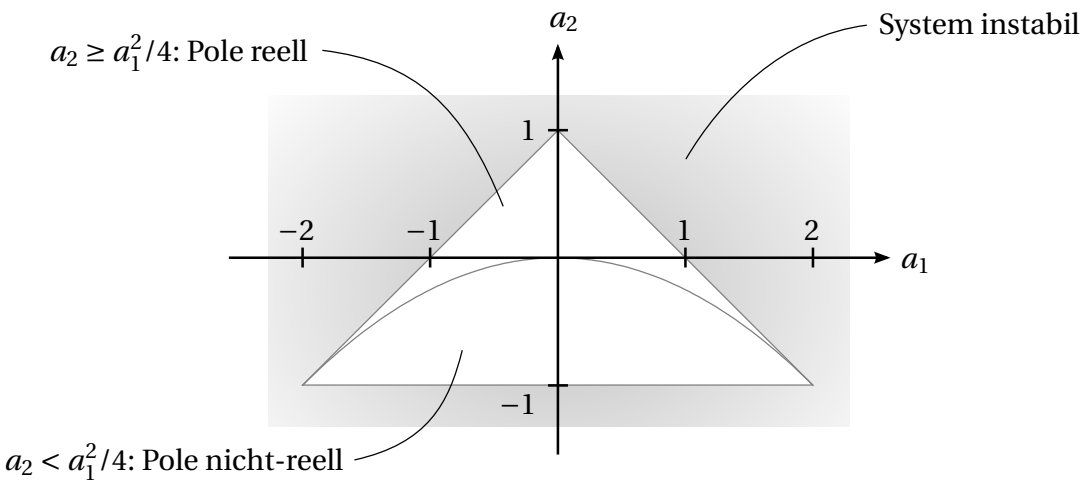

Abbildung 3.13: Anhand der Lage der Koeffizienten a_1 und a_2 eines Systems zweiter Ordnung in der (a_1, a_2)-Ebene kann entschieden werden, ob die Pole reell sind und ob das System stabil ist.

Dieser Wertebereich ist zusammen mit der Unterscheidung von reellen und nicht-reellen Polen in Abbildung 3.13 zusammengefasst.

Obwohl ein Filter auf diese Weise flexibler konfigurierbar ist, wird für den Einsatz als Tail-Cancellation-Filter im SPADIC-System von einer solchen Struktur abgesehen. Das hat die folgenden Gründe:

- Es reicht aus, wie im Abschnitt 3.3 dargelegt, reelle Pole und Nullstellen einstellen zu können.
- Bei einem System erster Ordnung entsprechen die Koeffizienten direkt den Polen und Nullstellen, so dass die Wirkung des Filters intuitiver vom Benutzer zu steuern ist, während bei einem System zweiter Ordnung die Pole und Nullstellen auf kompliziertere Weise durch die Koeffizienten bestimmt werden.
- Ein System erster Ordnung ist garantiert stabil, wenn der Bereich, innerhalb dessen die Koeffizienten einstellbar sind, zwischen -1 und 1 liegt, wohingegen ein System zweiter Ordnung immer durch ungeschickte Koeffizientenwahl instabil werden kann.

4 Umsetzung als digitale Schaltung

Um das Tail-Cancellation-Filter in das SPADIC-System zu integrieren, musste als Teil dieser Diplomarbeit eine Beschreibung in der Hardwarebeschreibungssprache *Verilog* erstellt werden. Daraus wird – zusammen mit den übrigen Bestandteilen des SPADIC-Digitalteils – unter Verwendung einer *Standardzellbibliothek*, d. h. einer Sammlung von vorgefertigten Schaltungselementen (z. B. einfache und gemischte Logikgatter, Bausteine für sequentielle Schaltungen wie *Flip-Flops*), mit geeigneten Softwarewerkzeugen eine digitale Schaltung *synthetisiert*.

Die grundlegende Vorgehensweise beim Übertragen der Filterstruktur in eine *Verilog*-Beschreibung bestand darin, von der Darstellung des Filters als Blockdiagramm (Abbildung 3.7) ausgehend, die einzelnen Bestandteile *Addierer*, *Multiplizierer* und *Verzögerer*, die im Blockdiagramm symbolisch für die entsprechenden Operationen mit Folgen stehen, durch digitale Schaltungselemente zu ersetzen.

Besonderes Augenmerk lag dabei auf der Umsetzung von Multiplizierern als digitale Schaltung, denn sie haben verglichen mit den Addierern und Verzögerern (diese werden aus Flip-Flops aufgebaut) eine größere Komplexität und beanspruchen deshalb mehr von der begrenzten Fläche des Chips. Weiterhin steht nur eine gewisse Zeitdauer für einen Multiplikationsvorgang zur Verfügung (bedingt durch die Abtastrate des ADCs), so dass die Multipliziererschaltung auch schnell genug sein muss. Der Entwurf von schnellen sowie leistungs-, und platzsparenden Multipliziererschaltungen ist seit den 1950er Jahren Gegenstand der Forschung [20–40]. Es wurde daher auch untersucht, ob es sich lohnt, aus den in der Literatur in großer Zahl dargestellten Methoden einen Nutzen zu ziehen, oder ob es ausreichend ist, durch die Synthesewerkzeuge allein durch Verwenden des Multiplikationsoperators im Verilog-Code automatisch eine entsprechende Schaltung erzeugen zu lassen.

4.1 Binärdarstellung von Zahlen

4.1.1 Grundlagen

Um zu untersuchen, wie sich geschickt Schaltungen entwerfen lassen, die arithmetische Funktionen (hier Addition und Multiplikation) möglichst schnell und mit möglichst gerin-

gem Aufwand durchführen, muss zunächst verstanden werden, wie Zahlen in einer digitalen Schaltung repräsentiert werden.

Die einzelnen Signale in einer Schaltung können üblicherweise zwei Zustände annehmen, die mit „0“ und „1“ bezeichnet werden (z. B. „Spannung kleiner als U_0“ = „0“ und „Spannung größer als U_1“ = „1“). Ein solches *binäres* Signal wird als *Bit* bezeichnet. Mit einer Vorschrift *(Code)*, die einem *Wort* „$b_{m-1}\, b_{m-2} \ldots b_1 b_0$“ aus m *Bits* (oder der *Breite* m) eine Zahl zuordnet, lassen sich 2^m verschiedene Zahlen darstellen. Die Zuordnung von Wörtern zu Zahlen kann im Grunde beliebig sein. Man kann etwa einen Code mit zwei Bits willkürlich so definieren:

„00“ $\mapsto 17$
„01“ $\mapsto -3$
„10“ $\mapsto 19.124$
„11“ $\mapsto 44$

Sinnvoller ist es allerdings, den Code so zu wählen, dass sich die einem Wort zugehörige Zahl aus den einzelnen Bits des Worts *berechnen* lässt, indem jedem Bit, abhängig von der Stelle $i \in \{0, \ldots, m-1\}$, an der es innerhalb eines Worts steht, ein bestimmter *Wert* $B \cdot 2^i$ beigemessen wird. Die dargestellte Zahl ist dann die Summe der Wertigkeiten aller Bits, die im Zustand „1“ sind, d. h. die *Zustände* „0“ und „1“ der Bits b_i werden auch mit den *Zahlen* 0 und 1 identifiziert:

$$„b_{m-1} \ldots b_1 b_0“ \mapsto B \sum_{i=0}^{m-1} b_i \cdot 2^i = B\left(b_{m-1} \cdot 2^{m-1} + \cdots + b_1 \cdot 2 + b_0\right) \tag{4.1}$$

Das Bit b_{m-1} an der ersten Stelle wird auch als *MSB (most significant bit)* bezeichnet und das Bit b_0 an der letzten Stelle als *LSB (least significant bit)*, weil sie jeweils von allen Bits den größen bzw. kleinsten Wert haben.

Mit zwei Bits lassen sich dann beispielsweise die folgenden vier Zahlen darstellen:

„00“ $\mapsto 0$
„01“ $\mapsto B$
„10“ $\mapsto 2B$
„11“ $\mapsto 3B$

Der Skalierungsfaktor B kann ignoriert werden, d. h. es ist $B = 1$, wenn die natürlichen Zahlen von 0 bis $2^m - 1$ dargestellt werden sollen. Ansonsten ist es sinnvoll, eine Zweierpotenz $B = 2^{-s}$ zu wählen, was bewirkt, dass die Werte aller Bits um s Stellen verschoben werden (das Bit mit dem Wert 1 steht nicht mehr an der letzten Stelle mit $i = 0$, sondern an der Stelle

$i = s$):

$$„b_{m-1}\dots b_1 b_0“ \mapsto 2^{-s} \sum_{i=0}^{m-1} b_i \cdot 2^i = \sum_{i=0}^{m-1} b_i \cdot 2^{i-s} \tag{4.2}$$

4.1.2 Zweierkomplement

Um sowohl negative als auch positive Zahlen zu repräsentieren, gibt es verschiedene Möglichkeiten, wie etwa die Verwendung des $(m-1)$-ten Bits als *Vorzeichenbit*, das bestimmt, ob die Zahl, die aus den Bits $m-2$ bis 0 gebildet wird, positiv oder negativ ist:

$$„b_{m-1}\dots b_1 b_0“ \mapsto (-1)^{b_{m-1}} B \sum_{i=0}^{m-2} b_i \cdot 2^i \tag{4.3}$$

Zählt man alle 2^m Wörter nach dem Schema „0…000“, „0…001“, „0…010“, „0…011“, „0…100“, „0…101“, …, „1…000“, …, „1…111“ ab und ignoriert den Skalierungsfaktor[1], durchläuft man dabei die Zahlen $0, 1, \dots, 2^{m-1}-1, 0, -1, \dots, -2^{m-1}+1$. Dabei stellt man fest, dass es erstens zwei Wörter für die Null gibt, und zweitens die positiven Zahlen in aufsteigender, die negativen Zahlen aber in absteigender Reihenfolge (also jeweils die Beträge in aufsteigender Reihenfolge) vorkommen.

Eine andere Möglichkeit zur Darstellung vorzeichenbehafteter Zahlen ist das *Zweierkomplement*, das häufig verwendet wird [5, S. 394], so auch im SPADIC-Chip für die Eingangswerte des Tail-Cancellation-Filters. Dabei wird das $(m-1)$-te Bit nicht zur *Multiplikation* der Zahl mit dem Vorzeichen verwendet und hat so einen Wert, der vom Zustand der restlichen Bits abhängig ist, sondern es hat, wie die anderen Bits auch, einen festen Wert, der *addiert* wird. Der Unterschied zu den Bits $m-2$ bis 0 ist, dass dieser Wert negativ ist:

$$„b_{m-1}\dots b_1 b_0“ \mapsto B\left(b_{m-1} \cdot \left(-2^{m-1}\right) + \sum_{i=0}^{m-2} b_i \cdot 2^i\right) \tag{4.4}$$

Dadurch sind ebenfalls alle Zahlen mit $b_{m-1} = 1$ negativ, aber beim Abzählen aller Wörter wie im Beispiel der Vorzeichenbit-Darstellung, kommen alle Zahlen von -2^{m-1} bis $2^{m-1}-1$ genau einmal vor (d. h. es gibt nur ein Wort für die Null) und außerdem durchläuft man sowohl die negativen als auch die positiven Zahlen in aufsteigender Reihenfolge. Der Unterschied ist in der nächsten Tabelle für $m = 3$ demonstriert:

[1] Soweit nicht anders dargestellt, wird künftig immer $B = 1$ angenommen.

	Vorzeichenbit	Zweierkomplement
„000“ $\mapsto$	0	0
„001“ $\mapsto$	1	1
„010“ $\mapsto$	2	2
„011“ $\mapsto$	3	3
„100“ $\mapsto$	0	-4
„101“ $\mapsto$	-1	-3
„110“ $\mapsto$	-2	-2
„111“ $\mapsto$	-3	-1

Im Zweierkomplement ist die Summe zweier Zahlen x und x', die durch Umkehren aller Bits zueinander in Beziehung stehen, immer -1: Bezeichnet man mit $\overline{b}$ die Umkehrung des Zustands eines Bits b (d. h. die Zustände „0“ und „1“ werden vertauscht; $\overline{b} = 1 - b$), dann gilt:

$$\begin{aligned}
x' &= \overline{b_{m-1}} \cdot \left(-2^{m-1}\right) + \sum_{i=0}^{m-2} \overline{b_i} \cdot 2^i \\
&= \left(1 - b_{m-1}\right) \cdot \left(-2^{m-1}\right) + \sum_{i=0}^{m-2} \left(1 - b_i\right) \cdot 2^i \\
&= \left(-2^{m-1} + \underbrace{\sum_{i=0}^{m-2} 2^i}_{2^{m-1}-1}\right) - \underbrace{\left(b_{m-1} \cdot \left(-2^{m-1}\right) + \sum_{i=0}^{m-2} b_i \cdot 2^i\right)}_{x} \\
&= -1 - x \quad \Rightarrow \quad x' + x = -1
\end{aligned} \tag{4.5}$$

Das heißt, dass man die Darstellung des Negativen einer Zahl findet, indem man alle Bits umkehrt und Eins addiert.

4.1.3 Ändern der Wortbreite

Mit einem Wort der Breite m lassen sich im Zweierkomplement die 2^m Zahlen von -2^{m-1} bis $2^{m-1} - 1$ darstellen. Ist dieser Bereich zu klein, kann man die Wortbreite um d erhöhen, um 2^d-mal so viele Zahlen darstellen zu können. Dabei gibt es zwei Möglichkeiten:

- Es werden d Bits an das LSB angehängt, und zwar im Zustand „0“. Damit werden zwischen zwei in der alten Darstellung aufeinanderfolgenden Zahlen neue Zwischenwerte möglich und die Zahlen der alten Darstellung werden beim Umwandeln in das neue Format um 2^d skaliert.
- Es werden d Bits vor dem MSB eingefügt, und zwar im selben Zustand wie das MSB. Dann wird der verfügbare Zahlenbereich zu größeren Beträgen hin erweitert und alle

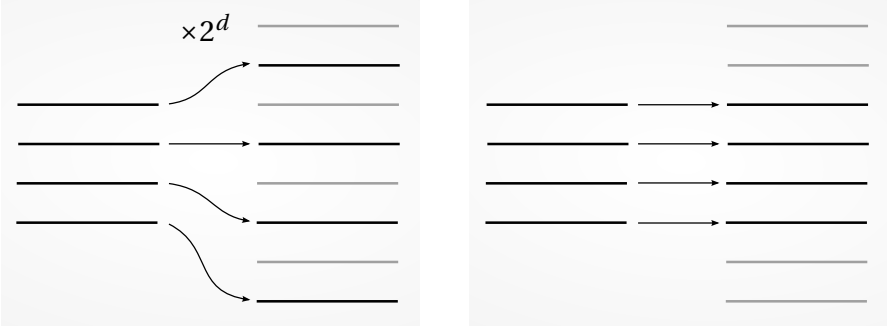

Abbildung 4.1: Erhöhen der Wortbreite durch Anhängen von d Bits nach dem LSB (links) oder durch Einfügen vor dem MSB (rechts), dargestellt für $d = 1$.

Zahlen der alten Darstellung mit m Bits haben in der neuen Darstellung mit $m + d$ Bits den gleichen Wert.

In der folgenden Tabelle ist das Erhöhen der Wortbreite für $m = 3$ und $d = 1$, sowie in Abbildung 4.1 für $m = 2$ und $d = 1$ als Beispiel gezeigt:

alte Darstellung	Einfügen vor MSB	Anhängen an LSB
„000“ ↦ 0	„0000“ ↦ 0	„0000“ ↦ 0
„001“ ↦ 1	„0001“ ↦ 1	„0010“ ↦ 2
„010“ ↦ 2	„0010“ ↦ 2	„0100“ ↦ 4
„011“ ↦ 3	„0011“ ↦ 3	„0110“ ↦ 6
„100“ ↦ −4	„1100“ ↦ −4	„1000“ ↦ −8
„101“ ↦ −3	„1101“ ↦ −3	„1010“ ↦ −6
„110“ ↦ −2	„1110“ ↦ −2	„1100“ ↦ −4
„111“ ↦ −1	„1111“ ↦ −1	„1110“ ↦ −2

Andersherum kann die Wortbreite verringert werden, wenn der Zahlenbereich zu groß ist, oder, was wahrscheinlicher ist, wenn die Komplexität der Schaltung reduziert werden soll. Wenn die Anzahl der Bits von $m + d$ auf m verringert wird, müssen zwangsläufig jeweils 2^d in der Darstellung mit $m + d$ Bit verschiedene Zahlen auf ein einzelnes Wort in der Darstellung mit m Bits abgebildet werden. Auch hier gibt es zwei Möglichkeiten:

- Es werden die d niederwertigsten Bits $d - 1, \ldots, 0$ verworfen und die m Bits $m + d - 1, \ldots, d$ behalten. Dadurch werden alle Zahlen durch Division mit 2^d auf den neuen Zahlenbereich abgebildet, wobei Zahlen, die kein Vielfaches von 2^d waren, abgerundet werden.

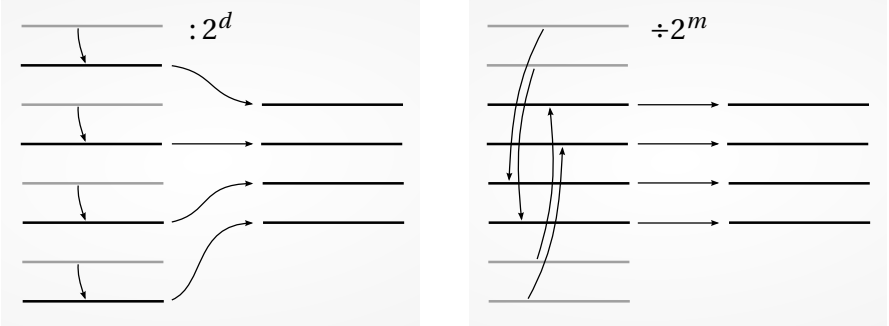

Abbildung 4.2: Verringern der Wortbreite durch Weglassen von d Bits nach dem LSB (*Abrunden*, links) oder vor dem MSB (*Überlauf*, rechts), dargestellt für $d = 1$.

- Es werden die d höchstwertigen Bits $m+d-1,\ldots,m$ verworfen und die m Bits $m-1\ldots,0$ behalten. Dann werden alle Zahlen x auf die Zahlen $x \div 2^m$ abgebildet (wobei $\div$ der Modulo-Operator sei), also bleiben alle Zahlen im Bereich von -2^{m-1} bis $2^{m-1}-1$ unverändert, aber alle Zahlen, die vorher außerhalb dieses Bereichs lagen, unterscheiden sich in der neuen Darstellung um ein Vielfaches von 2^d vom alten Wert. Dies nennt man *Überlauf*.

In der folgenden Tabelle ist für $m = 2$ und $d = 1$ Verringern der Wortbreite mit den beiden Möglichkeiten gezeigt:

alte Darstellung	Wegnahme der MSBs	Wegnahme der LSBs
„000“ $\mapsto$ 0	„00“ $\mapsto$ 0	„00“ $\mapsto$ 0
„001“ $\mapsto$ 1	„01“ $\mapsto$ 1	„00“ $\mapsto$ 0
„010“ $\mapsto$ 2	„10“ $\mapsto -2$	„01“ $\mapsto$ 1
„011“ $\mapsto$ 3	„11“ $\mapsto -1$	„01“ $\mapsto$ 1
„100“ $\mapsto -4$	„00“ $\mapsto$ 0	„10“ $\mapsto -2$
„101“ $\mapsto -3$	„01“ $\mapsto$ 1	„10“ $\mapsto -2$
„110“ $\mapsto -2$	„10“ $\mapsto -2$	„11“ $\mapsto -1$
„111“ $\mapsto -1$	„11“ $\mapsto -1$	„11“ $\mapsto -1$

Der Fall, dass sowohl auf der LSB- als auch auf der MSB-Seite Bits angehängt oder weggenommen werden, lässt sich als Kombination der beiden jeweils erläuterten Möglichkeiten zusammensetzen.

4.2 Addition

Sind zwei Zahlen a und b im Zweierkomplement durch die Wörter „$a_{m-1} \dots a_0$“ und „$b_{m-1} \dots b_0$“ gegeben, bedeutet „Addition“, dasjenige Wort zu finden, das die Zahl $a+b$ darstellt. Für die Summe $a+b$ gilt:

$$a+b = \left(a_{m-1}+b_{m-1}\right)\cdot\left(-2^{m-1}\right) + \sum_{i=0}^{m-2}\left(a_i+b_i\right)\cdot 2^i \tag{4.6}$$

Die Summe zweier Bits a_i und b_i kann dabei die Werte $0, 1, 2$ annehmen, ist also nicht durch ein einziges Bit darstellbar, so dass sich für die Stelle $i+1$ ein *Übertrag* c_{i+1} (engl. *carry*) ergibt. Wenn die Zahl $a+b$ durch m Bits darstellbar ist, also $-2^{m-1} \le a+b < 2^{m+1}$ gilt, dann kann man ihre Bits $s_{m-1}, \dots, s_0$ mit der Methode der „schriftlichen Addition“ berechnen, bei der man den Übertrag c_m verwirft:

$$\begin{array}{cccccc} & a_{m-1} & \dots & a_1 & a_0 \\ + & b_{m-1} & \dots & b_1 & b_0 \\ + & c_{m-1} & \dots & c_1 & \\ \hline (c_m) & s_{m-1} & \dots & s_1 & s_0 \end{array}$$

Ist die Summe $a+b$ hingegen nicht durch m Bits darstellbar, weil sie kleiner als -2^{m-1} oder größer oder gleich 2^{m-1} ist, macht man bei dieser Methode einen Fehler durch *Überlauf*. Das lässt sich vermeiden, indem die Zahlen a und b vor der Addition durch Einfügen von zusätzlichen Bits vor dem MSB – wie im vorigen Abschnitt beschrieben – in ein Format mit höherer Wortbreite umgewandelt werden.

Eine Schaltung, die aus den Bits a_i und b_i die Bits s_i erzeugt, kann man aus *Halbaddierern* und *Volladdierern* zusammensetzen. Ein Halbaddierer erzeugt aus *zwei* Bits a und b das Summenbit s und das Übertragsbit c mit $a+b = 2c+s$. Die Werte der Ausgangsbits c und s sind in Abhängigkeit der Eingangsbits a und b in der folgenden *Wahrheitstabelle* aufgeführt:

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Ein Volladdierer hat einen zusätzlichen Übertragseingang, erzeugt also aus *drei* Eingangsbits a, b und c_{in} die Ausgangsbits c_{out} und s mit $a+b+c_{\text{in}} = 2c_{\text{out}}+s$. Die Wahrheitstabelle des Volladdierers ist wie folgt:

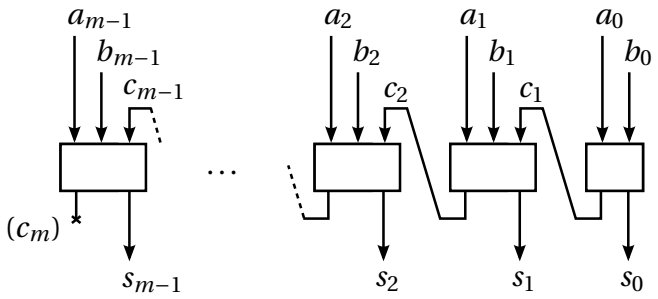

Abbildung 4.3: Schematische Darstellung eines *Ripple-Carry-Adders* aus einem Halbaddierer und $m-1$ Volladdierern

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Die einfachste Addiererschaltung, um die Summe aus zwei m-Bit-Zahlen zu bilden, ist der *Ripple-Carry-Adder*, bei dem genau wie in der handschriftlichen Methode die Überträge von der niederwertigsten zur höchstwertigen Stelle weitergereicht werden. Ein *Ripple-Carry-Adder* besteht aus einem Halbaddierer und $m-1$ Volladdierern, die wie in Abbildung 4.3 verbunden sind.

Der Vorteil dieses Addierertyps ist seine Einfachheit. Allerdings hat er den Nachteil, dass die Zeit, die vom Anlegen der Eingangsbits (in Form von elektrischen Signalen) bis zum Gültigwerden aller Ausgangsbits vergeht *(delay)*, linear mit der Wortbreite m zunimmt, weil die Ausgangsbits der i-ten Stelle erst ermittelt werden können, wenn der Übertrag von der $i-1$-ten Stelle feststeht, und sich so die Überträge von der LSB-Seite bis zur MSB-Seite schrittweise fortpflanzen müssen.

Es gibt zahlreiche Techniken, durch zusätzliche Logik und damit verbunden höhere Komplexität der Schaltung, die Ausbreitung der Übertrags-Information zu beschleunigen (u. A. *Carry Lookahead*, *Conditional Sum*, *Carry-Skip*, *Carry-Select* [27]), worauf hier nicht näher eingegangen wird.

Bei genauerer Betrachtung der Wahrheitstabellen kann man erkennen, dass die beiden Ausgangsbits von Voll- und Halbaddierer, als vorzeichenlose Binärzahl (nach Gleichung 4.1)

interpretiert, genau die Anzahl der Eingangsbits im Zustand „1“ ergeben. Man kann Halb- und Volladdierer daher auf das Konzept der (N,M)-*Zähler* [22, 24] verallgemeinern: Dabei handelt es sich um Schaltungen mit M Ausgängen und $N < 2^M$ Eingängen, die die Anzahl der Eingangsbits, die sich im Zustand „1“ befinden, als M-Bit-Wort ausgeben. In dieser Sichtweise ist ein Halbaddierer ein $(2,2)$- und ein Volladdierer ein $(3,2)$-Zähler.

4.3 Multiplikation

4.3.1 Vorzeichenlose Zahlen

Die Multiplikation zweier Binärzahlen lässt sich zunächst einfacher für den Fall von vorzeichenlosen Zahlen verstehen und danach auf vorzeichenbehaftete Zahlen (d. h. in der Darstellung des Zweierkomplements) erweitern.

Seien a und b zwei durch Wörter der Breite n bzw. m dargestellte natürliche Zahlen:

$$a = \sum_{i=0}^{n-1} a_i \cdot 2^i$$
$$b = \sum_{j=0}^{m-1} b_j \cdot 2^j$$

Für das Produkt der beiden Zahlen gilt dann:

$$a \cdot b = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_i b_j \cdot 2^{i+j} = \sum_{k=0}^{n+m-2} \left(\sum_{i+j=k} a_i b_j \right) 2^k \tag{4.7}$$

Das bedeutet, dass das Bit an der k-ten Stelle des Produkts entsteht, indem alle Teilprodukte *(partial products)* $p_{ij} = a_i b_j$ mit $i + j = k$, die den Wert 2^k haben, addiert werden. Dies entspricht wie bei der Addition der Vorgehensweise bei der handschriftlichen Berechnung, was für $n = 4$ und $m = 3$ hier verdeutlicht ist:

$$\begin{array}{cccccccc}
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
+ & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
+ & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
\hline
c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 &
\end{array}$$

Eine Schaltung, die „Multiplikation“ durchführt – also aus zwei Eingangswörtern ein Ausgangswort so erzeugt, dass es, als Zahl interpretiert, das Produkt der als Zahlen interpretierten Eingangswörter ist – besteht also im Kern wiederum aus einer Addiererschaltung, bei der mehr als zwei Zahlen addiert werden. Würde man diese einzelnen Additionen Zeile für

Zeile durchführen und dabei jeweils den Übertrag zur Addition der nächsten Zeile weiterreichen, wäre die benötigte Dauer proportional zur Anzahl der Zeilen, also zur Wortbreite m des Faktors b.

Wallace [21] erkannte, dass man die einzelnen Additionen auch parallel durchführen kann, wenn man die Überträge zunächst nicht weiterreicht *(carry save)*. Die Anzahl der Zeilen wird so schrittweise reduziert, bis zwei Zeilen übrig bleiben, die zum Schluss in einem Addierer zusammengefasst werden, wobei die „angesammelten" Überträge dann einfließen. Weil in jedem Schritt die Anzahl der Zeilen um einen gewissen *Faktor* kleiner wird, ist die Anzahl der notwendigen Schritte und somit die Multiplikationsdauer proportional zum Logarithmus der Wortbreite.

Bei solchen *Wallace-Trees* werden in jedem Schritt immer soviele Teilprodukte *(Partial Products)* wie möglich, d. h. unter Verwendung von Voll- und Halbaddierern jeweils drei, zusammengefasst. Dadda [22] hat diese Verfahrensweise so modifiziert, dass in einem Schritt nur so viele *Partial Products* zusammengefasst werden, dass die Reduktion auf zwei Zeilen in der minimalen Anzahl Schritte erfolgt. Die auf diese Weise entstehenden Schaltungen *(Dadda-Trees)* unterscheiden sich leicht von *Wallace-Trees* und benötigen die geringstmögliche Anzahl von Voll- und Halbaddierern [36].

Die gängigen Architekturen von Multiplizierern lassen sich oft auf dieses Grundprinzip zurückführen, das auch bei Wallace und Dadda Verwendung findet: In einer *Partial Product Generation Matrix* werden die Teilprodukte $p_{ij} = a_i b_j$ gebildet. Diese Schaltung besteht einfach aus $n \cdot m$ AND-Gattern. Die *Partial Products* werden in einem *Partial Product Reduction Tree (PPRT)* auf zwei Zahlen reduziert, die in einem *Final Adder*, der z. B. ein Ripple-Carry-Adder sein kann, addiert werden, was das Produkt der Eingangszahlen ergibt. Diese Architektur ist in Abbildung 4.4 gezeigt.

In [26] ist detailliert ein Algorithmus beschrieben, das sich *TDM (Three Dimensional Minimization)* nennt, mit dem sich PPRTs erzeugen lassen. Als Eingabe des Algorithmus dienen die Wortbreiten n und m der Faktoren und Angaben über das Zeitverhalten (d. h. die *Delays* für jedes Paar von Eingängen und Ausgängen) der verfügbaren Halb- und Volladdierer, aus denen der PPRT zusammengesetzt werden soll. Als Ausgabe des Algorithmus erhält man die Information, wie die Halb- und Volladdierer miteinander zu verbinden sind, so dass das Gesamt-Delay des PPRT möglichst gering ist. Dabei wird dieselbe Anzahl von Halb- und Volladdierern verwendet wie beim Verfahren von Dadda, also die minimale Anzahl.

In der Sichtweise des TDM-Verfahrens ist ein PPRT aus *Vertical Compressor Slices (VCS)* zusammengesetzt. In jeder VCS werden die *Partial Products* eines bestimmten Werts (also in der k-ten VCS diejenigen mit dem Wert 2^k) gezählt. Als Ausgänge der k-ten VCS gehen zwei Bits an den *Final Adder* sowie die Überträge mit dem Wert 2^{k+1} an die $k+1$-te VCS.

Trotz der unterschiedlichen Herangehensweisen entstehen beim Verfahren von Dadda und beim TDM-Verfahren ähnliche Schaltungen. In den Abbildungen 4.5 und 4.6 sind PPRTs

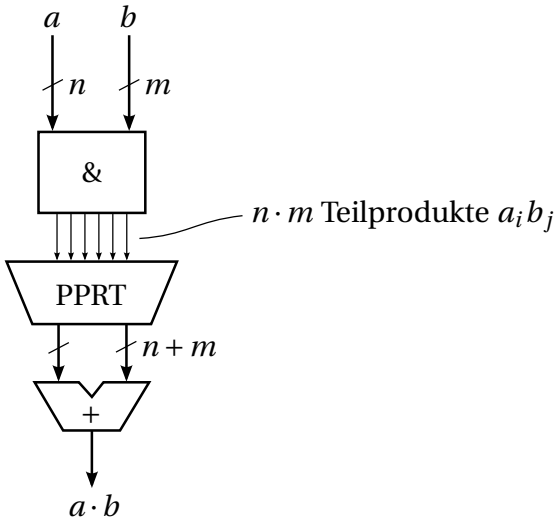

Abbildung 4.4: Aufbau eines Multiplizierers aus *Partial Product Generation Matrix*, *Partial Product Reduction Tree* und *Final Adder*

gezeigt, die für $n = m = 6$ mit jeweils einem der Verfahren erstellt wurden. Sie haben im Grunde den gleichen Aufbau, unterscheiden sich aber leicht darin, in welcher Reihenfolge die *Partial Products* verarbeitet werden.

Es gibt Ansätze, die Multiplikation zu beschleunigen, indem die Anzahl der Teilprodukte von vorneherein reduziert wird, z. B. *Booth Encoding* [20]. Allerdings wird angezweifelt, dass solche Techniken insgesamt einen Vorteil bringen, denn die Komplexität der Schaltung zur Erzeugung der Teilprodukte wird dadurch erhöht und bewirkt eine zusätzliche Verzögerung der Signallaufzeiten [25, 30, 32].

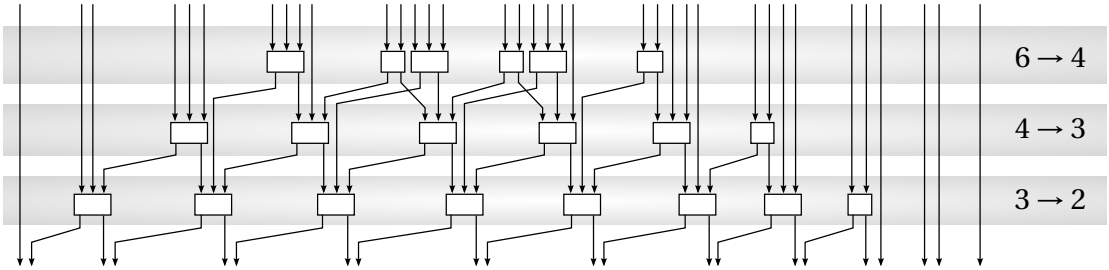

Abbildung 4.5: PPRT für $n = m = 6$ nach der Methode von Dadda: In drei Schritten werden die *Partial Products* von Halb- und Volladdierern zusammengefasst und so die Anzahl der zu addierenden Zahlen von sechs auf zwei reduziert, die zum *Final Adder* gehen.

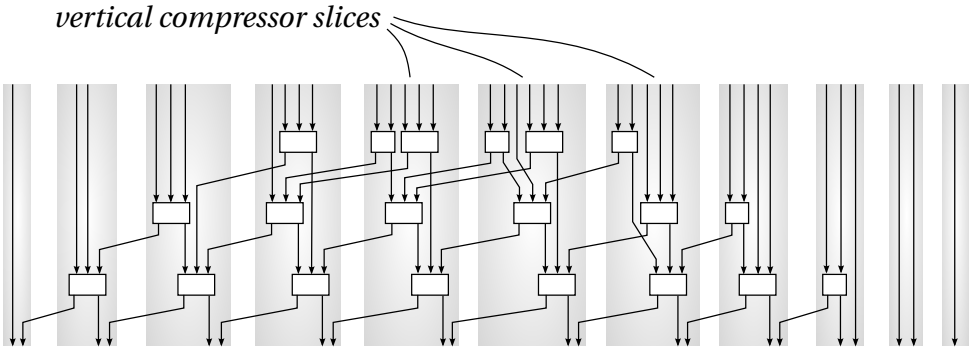

Abbildung 4.6: PPRT für $n = m = 6$ nach der TDM-Methode: Die *Partial Products* eines bestimmten Werts werden jeweils in *Vertical Compressor Slices (VCS)* gezählt und die Überträge zur nächsten VCS weitergegeben.

4.3.2 Vorzeichenbehaftete Zahlen

Zahlen in der Zweierkomplementdarstellung können mit einer Multipliziererarchitektur wie in Abbildung 4.4 gezeigt ebenfalls verarbeitet werden, wenn in der Erzeugung der Teilprodukte leichte Modifizierungen vorgenommen werden und dementsprechend der PPRT angepasst wird [36]. Die notwendigen Änderungen beruhen auf dem Algorithmus zur Multiplikation von Zahlen im Zweierkomplement von Baugh und Wooley [23, 39].

Als Beispiel für die Anwendung des Baugh-Wooley-Verfahrens wird nun die Multiplikation zweier Zahlen a und b mit vier bzw. drei Bits vorgeführt. Im Zweierkomplement ist der Wert des MSB (hier a_3 und b_2) im Vergleich zur vorzeichenlosen Interpretation des Binärworts negativ. Das heißt, dass in der Notation, in der in der k-ten Spalte (bei Null von rechts beginnend) die Teilprodukte mit dem Wert 2^k stehen, die Teilprodukte $a_3 b_j$ für $j = 0, 1$ sowie die Teilprodukte $a_i b_2$ für $i = 0, 1, 2$ subtrahiert anstatt addiert werden. Das Teilprodukt $a_3 b_2$ zählt wieder positiv, weil zweimal negiert wird:

$$
\begin{array}{cccccccc}
 & & & & -a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
+ & & & -a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
+ & & a_3b_2 & -a_2b_2 & -a_1b_2 & -a_0b_2 & & \\
\hline
 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

Man kann die Teilprodukte $a_3 b_j$ und $a_i b_2$ auch jeweils als eigene Zahlen auffassen, die als Ganzes subtrahiert werden:

					a_2b_0	a_1b_0	a_0b_0
+				a_2b_1	a_1b_1	a_0b_1	
+		a_3b_2					
−	0	0	a_3b_1	a_3b_0			
−	0	0	a_2b_2	a_1b_2	a_0b_2		
	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Die Subtraktion einer Zahl im Zweierkomplement ist nach (4.5) das Gleiche wie die Addition der Zahl, bei der alle Bits umgekehrt werden, und einer zusätzlichen Eins:

					a_2b_0	a_1b_0	a_0b_0
+				a_2b_1	a_1b_1	a_0b_1	
+		a_3b_2		1			
+	1	1	$\overline{a_3b_1}$	$\overline{a_3b_0}$	1		
+	1	1	$\overline{a_2b_2}$	$\overline{a_1b_2}$	$\overline{a_0b_2}$		
	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Die Addition der Einsen in der fünften und sechsten Spalte kann man sofort auflösen. Da alle Überträge nach der sechsten Spalte ignoriert werden, bleibt eine Eins in der sechsten Spalte übrig. Die Addition von Eins zu c_6 kann man wiederum durch Umkehren zu $\overline{c_6}$ ersetzen, denn es ist $c_6 + \text{„}1\text{“} = \text{„}01\text{“}$ für $c_6 = \text{„}0\text{“}$ und $c_6 + \text{„}1\text{“} = \text{„}10\text{“}$ für $c_6 = \text{„}1\text{“}$, und das Übertragsbit wird ignoriert. Das heißt, dass man zwei Zahlen im Zweierkomplement multiplizieren kann, indem man einige der Teilprodukte umkehrt, in den Spalten $n-1$ und $m-1$ jeweils eine zusätzliche Eins addiert, sowie nach dem Addieren aller Teilprodukte das MSB des Ergebnisses umkehrt:

				$\overline{a_3b_0}$	a_2b_0	a_1b_0	a_0b_0
+			$\overline{a_3b_1}$	a_2b_1	a_1b_1	a_0b_1	
+		a_3b_2	$\overline{a_2b_2}$	$\overline{a_1b_2}$	$\overline{a_0b_2}$		
+				1	1		
	$\overline{c_6}$	c_5	c_4	c_3	c_2	c_1	c_0

Um dies schaltungstechnisch umzusetzen, muss man lediglich in der *Partial Product Generation Matrix* einige AND-Gatter durch NAND-Gatter ersetzen, den PPRT so anpassen, dass in der $n-1$-ten und $m-1$-ten Spalte (die bei der TDM-Methode der $n-1$-ten und $m-1$-ten VCS entsprechen) jeweils beim Zählen der *Partial Products* eine Eins hinzukommt, sowie einen Inverter am Ausgang des *Final Adders* einfügen.

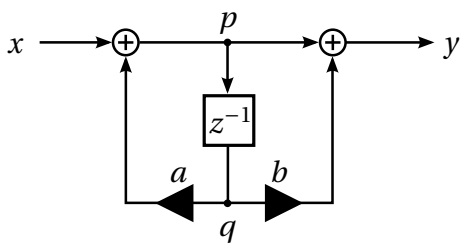

Abbildung 4.7: Eine Filterstufe erster Ordnung, aus denen das Tail-Cancellation-Filter zusammengesetzt ist.

4.3.3 Zusammenfassen von Multiplikation und Addition

In der Filterstruktur aus einer Reihenschaltung von Systemen erster Ordnung, die jeweils in der Direktform II vorliegen (siehe Abbildung 4.7), werden die einzelnen Systeme durch die folgenden Gleichungen beschrieben:

$$p = x + a \cdot q \tag{4.8a}$$

$$y = p + b \cdot q \tag{4.8b}$$

Es muss also stets (die Gleichung $y = p + b \cdot q$ diene als Beispiel) ein durch ein n-Bit-Wort dargestellter Wert q der Signalfolge mit einem m Bit breiten Koeffizienten b multipliziert und zu einem weiteren n-Bit-Wort p des Signals addiert werden (Diese Operation kommt aufgrund des Aufbaus von Filterstrukturen in der digitalen Signalverarbeitung häufig vor und trägt die Bezeichnung *Multiply-Accumulate* oder abgekürzt *MAC* [29, 31]). Die zusätzliche Addition kann man durch eine weitere Zeile im Additionsschema der Teilprodukte der Multiplikation darstellen. Für $n = 4$ und $m = 3$ würde es etwa lauten:

$$\begin{array}{cccccccc}
 & & & & \overline{q_3 b_0} & q_2 b_0 & q_1 b_0 & q_0 b_0 \\
+ & & & \overline{q_3 b_1} & q_2 b_1 & q_1 b_1 & q_0 b_1 & \\
+ & & q_3 b_2 & \overline{q_2 b_2} & \overline{q_1 b_2} & \overline{q_0 b_2} & & \\
+ & & & & 1 & 1 & & \\
+ & & & & p_3 & p_2 & p_1 & p_0 \\
\hline
 & \overline{y_6} & y_5 & y_4 & y_3 & y_2 & y_1 & y_0
\end{array}$$

Nun möchte man aber nicht mit ganzzahligen Koeffizienten multiplizieren, sondern wie in Abschnitt 3.3 beschrieben, mit reellen Zahlen mit Betrag kleiner als Eins. Dies kann man erreichen, indem man die Zahl p um s Stellen nach links (d. h. zu höherwertigen Stellen hin) verschiebt. Das entspricht gleichermaßen einem Skalierungsfaktor $B = 2^{-s}$ nach (4.3), mit dem der Faktor b versehen wird. Für $s = 2$ und $m = 3$ hat man beispielsweise mögliche Faktoren im Bereich von $-4/4 = -1$ bis $+3/4$. Das entsprechende Schema sieht so aus:

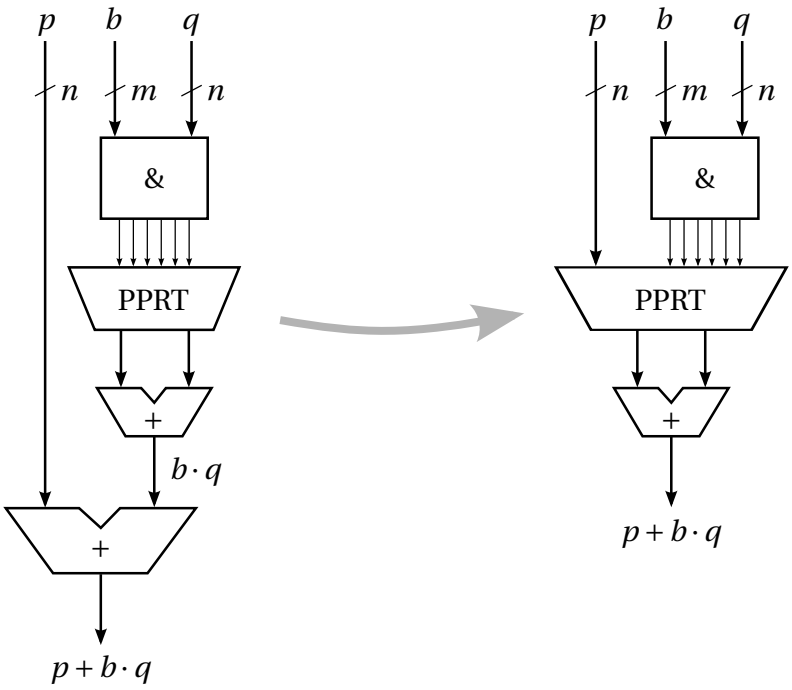

Abbildung 4.8: Multiplikation und Addition können in einer gemeinsamen Schaltung ausgeführt werden.

			$\overline{q_3 b_0}$	$q_2 b_0$	$q_1 b_0$	$q_0 b_0$
+		$\overline{q_3 b_1}$	$q_2 b_1$	$q_1 b_1$	$q_0 b_1$	
+		$q_3 b_2$	$\overline{q_2 b_2}$	$\overline{q_1 b_2}$	$\overline{q_0 b_2}$		
+				1	1		
+		p_3	p_2	p_1	p_0		
	$\overline{y_6}$	y_5	y_4	y_3	y_2	y_1	y_0

Um wieder ein Datenwort y der Breite n zu erhalten, das im gleichen Zahlenbereich wie p und q liegt, muss man die Bits y_{i+s} behalten, die sich im Additionsschema in den gleichen Spalten wie die Bits p_i befinden. Das bedeutet, dass man für $s > 0$ einen Quantisierungsfehler durch Abrunden macht, sowie einen Fehler durch Überlauf, wenn es Überträge in die Spalten $i \geq n + s$ gibt.

Da die Teilprodukte der Multiplikation $b \cdot q$ und die Bits des Summanden p im Additionsschema die gleiche Bedeutung haben, kann die Rechenoperation $p + b \cdot q$ in einer Schaltung zusammengefasst werden, indem der PPRT so erweitert wird, dass nicht nur die Ausgangsbits der *Partial Product Generation Matrix*, sondern auch die Bits des Datenworts p darin verarbeitet werden. Dies ist in Abbildung 4.8 illustriert.

4.4 Synthese

4.4.1 TDM-Algorithmus

Der TDM-Algorithmus wurde im Rahmen dieser Arbeit als Python-Programm [41] implementiert, das eine Verilog-Datei ausgibt, in der eine Multipliziererschaltung als Netzliste auf der Ebene von Voll- und Halbaddierern beschrieben ist. Diese „halbautomatische“ Synthese wurde mit der „vollautomatischen“ Synthese verglichen, bei der anstelle der konkreten Beschreibung des Multiplizierers als Schaltung aus Voll- und Halbaddierern im Verilog-Code nur die Multiplikationsanweisung angegeben ist, so dass das verwendete Synthesewerkzeug (*Encounter RTL Compiler*[2] von *Cadence* [42]) selbst über die genaue Implementierung entscheidet.

Als *Top-Level-Modul* für die Synthese wurde eine Filterstufe wie in Abbildung 4.7 verwendet, um gleich eine Einschätzung über die Größe und Geschwindigkeit des Filters für den Einsatz im SPADIC-Chip zu erhalten. Die Verilog-Beschreibung der Filterstufe ist wie folgt:

```
module filterstage #(parameter n=12, m=8) (
    input                 clk,
    input                 res_n,
    input  signed [n-1:0] X,
    output signed [n-1:0] Y,
    input  signed [m-1:0] a,
    input  signed [m-1:0] b);

    wire signed [n-1:0] P;
    reg  signed [n-1:0] Q;
    always @ (posedge clk or negedge res_n) begin
        if (~res_n)
            Q <= 0;
        else
            Q <= P;
        end
    mul_add #(n, m) madd1(Q, a, X, P);
    mul_add #(n, m) madd2(Q, b, P, Y);
endmodule
```

Die beiden Instanzen des Moduls `mul_add` sind die „vollautomatische“ Form der Multiplizierschaltung:

```
module mul_add #(parameter n=12, m=8) (
```

[2] Version RC10.1.301 – v10.10-s313_1 (64-bit)

```
    input  signed [n-1:0] A,
    input  signed [m-1:0] B,
    input  signed [n-1:0] C,
    output signed [n-1:0] P);

    parameter s = m-1;
    wire signed [n+m:0] R_full;
    assign R_full = ((A*B) >>> s) + C;
    assign P = R_full[n-1:0];
endmodule
```

Um sie mit der „halbautomatisch“ generierten Multipliziererschaltung zu vergleichen, werden sie durch Instanzen der durch das TDM-Programm erzeugten Module ersetzt (sie können nicht parametrisiert werden, sondern für verschiedene Wortbreiten müssen neue Verilog-Dateien mit ihrer Beschreibung erzeugt werden):

```
module filterstage #(parameter n=12, m=8) (...);
    ...
    muladd_signed_TDM_12x8 madd1(Q, a, X, P);
    muladd_signed_TDM_12x8 madd2(Q, b, P, Y);
endmodule
```

Die vom TDM-Algorithmus generierten Module haben in etwa den folgenden Aufbau:

```
module muladd_signed_TDM_12x8(
  input  [11:0] A,
  input  [ 7:0] B,
  input  [11:0] C,
  output [11:0] P
);

// Erzeugen der Teilprodukte
wire pp_1_0 = A[1] & B[0];
wire pp_2_0 = A[2] & B[0];
wire pp_3_0 = A[3] & B[0];
...
wire pp_8_7 = ~(A[8] & B[7]);
wire pp_9_7 = ~(A[9] & B[7]);
wire pp_10_7 = ~(A[10] & B[7]);

// Instanzen von Halb- und Volladdierern bilden den PPRT
// und den Final Adder (als Ripple-Carry-Adder)
```

```
wire HA_2_0_S;
wire HA_2_0_C;
HalfAdder HA_2_0(pp_0_2, pp_1_1, HA_2_0_S, HA_2_0_C);
wire HA_3_0_S;
wire HA_3_0_C;
HalfAdder HA_3_0(pp_2_1, pp_3_0, HA_3_0_S, HA_3_0_C);
wire FA_3_1_S;
wire FA_3_1_C;
FullAdder FA_3_1(pp_0_3, pp_1_2, HA_3_0_S, FA_3_1_S,
                                           FA_3_1_C);
...
wire FA_17_1_S;
wire FA_17_1_C;
FullAdder FA_17_1(FA_16_0_C, FA_17_0_S, FA_16_1_C, FA_17_1_S,
                                                   FA_17_1_C);

wire Ripple_HA_1_S;
wire Ripple_HA_1_C;
HalfAdder Ripple_HA_1(pp_1_0, pp_0_1, Ripple_HA_1_S,
                                      Ripple_HA_1_C);
...
wire Ripple_FA_18_S;
wire Ripple_FA_18_C;
FullAdder Ripple_FA_18(FA_17_1_C, FA_18_0_S, Ripple_FA_17_C,
                      Ripple_FA_18_S, Ripple_FA_18_C);
// Ausgabe des Ergebnisses
assign P[0]  = Ripple_FA_7_S;
assign P[1]  = Ripple_FA_8_S;
assign P[2]  = Ripple_FA_9_S;
...
assign P[10] = Ripple_FA_17_S;
assign P[11] = ~Ripple_FA_18_S;
endmodule
```

Um sicherzugehen, dass beide MAC-Schaltungen das gleiche Verhalten zeigen, wurde eine einfache *Testbench* erstellt, in der beide Varianten instanziiert und (vom Benutzer) willkürlich gewählte Eingangswerte angelegt werden. Es wurde keine Kombination von Eingangswerten `A`, `B`, `C` gefunden, bei denen sich der Ausgangswert `P_ref` des vom RTL Compiler erzeugten MACs vom Ausgangswert `P_TDM` des TDM-MACs unterscheidet. Ein Beispiel für die Ausgabe der Testbench ist das folgende:

```
A     = 000010101001 =  169
B     =     10001000 = -120
C     = 010011100101 = 1253
P_ref = 010001000110 = 1094
P_TDM = 010001000110 = 1094
```

Nachdem nun die Korrektheit des TDM-Programms zumindest naheliegend, wenn auch nicht sicher verifiziert, war, konnte überprüft werden, wie gut eine aus beiden MAC-Schaltungen aufgebaute Filterstufe erster Ordnung in Bezug auf die Anzahl der Gatter und die Geschwindigkeit synthetisiert werden kann.

Die Synthese mit dem Encounter RTL Compiler wurde mit dem Kommando

```
synthesize -to_mapped -effort high
```

aufgerufen und die am Lehrstuhl verfügbare `SUSLIB_UCL_tt` als Standardzellbibliothek verwendet. Als Taktfrequenz wurde mit

```
define_clock -period 40000 -name clk1 clk
```

der realistische Wert von 25 MHz angegeben. Sowohl für die „vollautomatische“ als auch für die „halbautomatische“ Variante wurde jeweils die Wortbreite der Filterkoeffizienten konstant auf $m = 8$ gesetzt und dafür die Wortbreite n der Werte der Signalfolge variiert. Die Geschwindigkeit wurde mit dem Befehl

```
report timing
```

und die Anzahl der verwendeten Zellen bzw. die Fläche mit

```
report gates
```

(wobei der `total`-Wert verwendet wurde) analysiert.

Beim Ablauf der Synthese (in der „vollautomatischen“ Variante) fällt auf, dass der RTL Compiler die arithmetischen Funktionen, die im Verilog-Code beschrieben sind, erkennt, und Optimierungen durchführt. Die Bezeichnung `carrysave` deutet darauf hin, dass dies ähnlich zu der in Abbildung 4.8 gezeigten Vereinfachung der Schaltung sein könnte:

```
Trying carrysave optimization (configuration 1 of 1) on module
'filterstage_csa_cluster'...
Info    : Done carrysave optimization. [RTLOPT-20]
        : There are 2 CSA groups in module
          'filterstage_csa_cluster' ... Accepted.
```

Das Ergebnis dieser Messreihe ist in der folgenden Tabelle aufgeführt:

	„vollautomatisch“			„halbautomatisch“ (TDM)		
n	Delay [ps]	Zellen	Fläche [µm^2]	Delay [ps]	Zellen	Fläche [µm^2]
8	3841	491	10094	3783	527	11691
12	4651	703	14747	4537	840	18638
16	5556	924	19471	5274	1129	25319
20	6310	1153	24191	6027	1441	32108
24	7258	1356	28757	6918	1746	38901

Man sieht, dass die mit TDM erzeugten Schaltungen zwar geringfügig schneller sind, aber dafür deutlich mehr Fläche belegen. Da die Durchlaufzeit selbst bei 24 Bit Wortbreite, was bei 9 Bit ADC-Auflösung mehr als genug sein dürfte (dies wird in Kapitel 6 noch genauer untersucht), weit unter der erlaubten Dauer von 40 ns liegt, ist die Fläche das ausschlaggebende Argument, die Synthese der Multipliziererschaltung vollständig dem RTL Compiler zu überlassen. Zudem ist dies leichter und sicherer handhabbar, weil sich die Wortbreite einfacher (durch Ändern eines Parameters) anpassen lässt, und Fehler in der Implementierung des TDM-Algorithmus nicht ausgeschlossen werden können.

4.4.2 Volladdiererzelle

In der verwendeten Standardzellbibliothek war noch keine Zelle vorhanden, die die Funktion eines Volladdierers erfüllt. Daher wurde eine solche Zelle in Form der geometrischen Beschreibung *(Layout)* anhand eines Schaltplans aus [43] entworfen und in die Bibliothek unter der Bezeichnung `UCL_FA` eingefügt.

Die Zelle hat drei Eingänge a, b, c_{in} und zwei Ausgänge c_{out} und s, die die folgende Wahrheitstabelle erfüllen:

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Diese logische Funktion wird durch die in Abbildung 4.9 gezeigte Schaltung aus 28 Transistoren in der CMOS-Technik erreicht. Das Layout der Zelle ist in Abbildung 4.10 zu sehen. Die

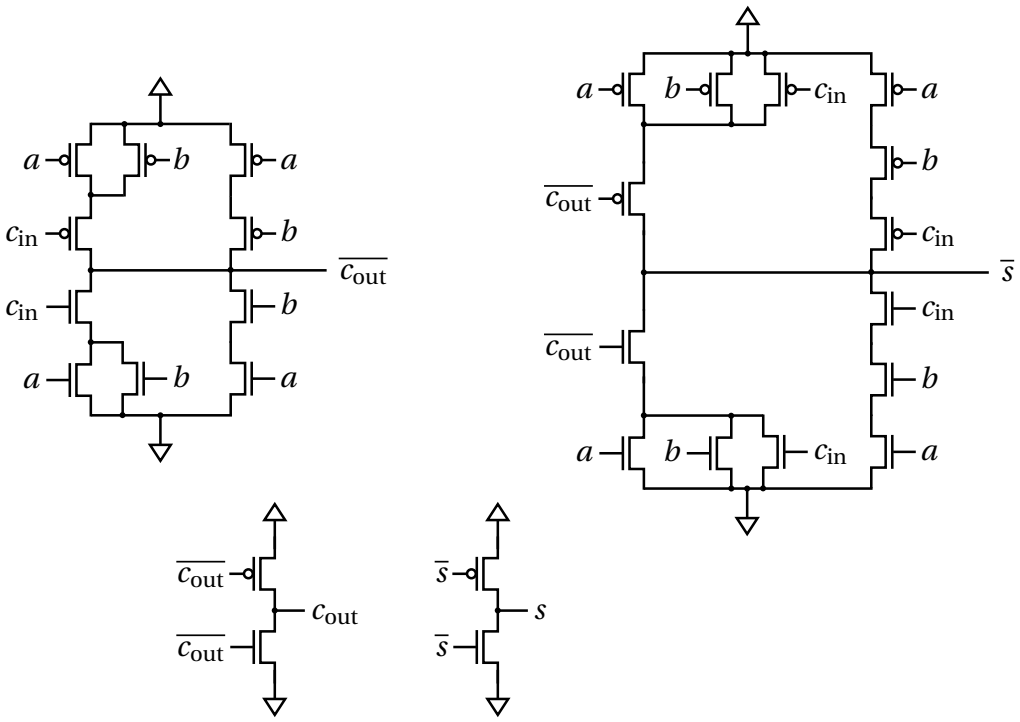

Abbildung 4.9: Schaltplan der Volladdiererzelle mit 28 Transistoren (nach [43])

Zelle ist ca. 12 µm lang und ca. 5 µm hoch (die Höhe ist für alle Standardzellen vorgegeben), die Fläche der Zelle ist also etwa 60 μm^2.

Mit dieser neuen Zelle in der Bibliothek `SUSLIB_UCL` wurde die „vollautomatische“ Synthese der Filterstufe erneut durchgeführt. Zunächst war interessant, ob der RTL Compiler den Volladdierer „anerkennt“ und in die Schaltung mit einbezieht.

Dies ist der Fall: Für die Wortbreite $n = 12$ sind hier die jeweils am häufigsten verwendeten Zellen aufgeführt, wobei einmal die Verwendung der Volladdiererzelle durch den Befehl

```
set_attribute avoid true UCL_FA
```

verboten wurde und einmal nicht:

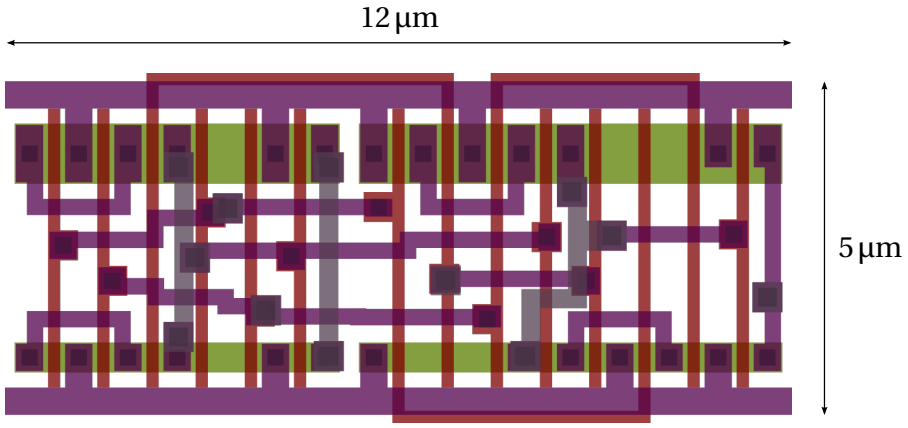

Abbildung 4.10: Layout der Volladdiererzelle UCL_FA

ohne UCL_FA			mit UCL_FA		
Zellname	Anzahl	Fläche [µm^2]	Zellname	Anzahl	Fläche [µm^2]
UCL_MUX2A	120	3484	UCL_OAI22	92	1908
UCL_OAI22	92	1908	UCL_FA	91	6038
UCL_CGI2	76	1891	UCL_AOI22	47	975
UCL_INV_LP2	59	489	UCL_AOI22_2	45	933
UCL_MUX2	58	1443	UCL_INV_LP2	32	265
UCL_AOI22	49	1016	UCL_INV	30	249
UCL_AOI22_2	46	954	UCL_NAND2	13	162
UCL_INV	42	348	UCL_MUX2	13	323
UCL_INV_LP	29	241	UCL_INV_LP	12	100

Man sieht, dass die Zelle UCL_FA tatsächlich verwendet wird und mit großem Abstand vor allen anderen Zellen den größten Flächenanteil der Schaltung bildet (6000 µm^2 von 13000 µm^2). Steht die Volladdiererzelle nicht zur Verfügung, sind stattdessen die Zellen UCL_MUX2A und UCL_CGI2 stark vertreten.

Die Messreihe mit Variation von n ist unter Verwendung der Volladdiererzelle noch einmal durchgeführt worden, um die Auswirkungen zu beobachten. Die Ergebnisse für das „vollautomatische“ Verfahren, einmal ohne Volladdiererzelle und einmal mit, sind:

	ohne `UCL_FA`			mit `UCL_FA`		
n	Delay [ps]	Zellen	Fläche [µm^2]	Delay [ps]	Zellen	Fläche [µm^2]
8	3841	491	10094	3931	314	8991
12	4651	703	14747	5022	437	13047
16	5556	924	19471	5865	539	16900
20	6310	1153	24191	7029	648	20831
24	7258	1356	28757	8005	746	24692

Die Geschwindigkeit ist mit der Volladdiererzelle etwas geringer, aber immer noch weit über der geforderten. Die Anzahl der Zellen nimmt natürlich stark ab, weil in einer Zelle die Funktion vereint ist, die zuvor aus mehreren Zellen kombiniert wurde. Die Gesamtfläche der Filterstufe nimmt ab. Die neue Volladdiererzelle trägt also dazu bei, das Tail-Cancellation-Filter auf dem SPADIC-Chip so kompakt wie möglich zu halten.

4.5 Verilog-Beschreibung des gesamten Filters

4.5.1 Übersicht

Mit den gewonnenen Erkenntnissen aus dem Vergleich der verschiedenen Synthesemethoden konnte nun eine Verilog-Beschreibung für die ganze Filterstruktur aus mehreren Filterstufen erstellt werden. Dabei wurden zusätzliche Funktionen eingebaut, die die Bedienung des Filters in der Praxis komfortabler machen:

- Nach der letzten Filterstufe kann das Signal noch einmal skaliert und um einen *Offset* verschoben werden, um es in den verfügbaren Wertebereich einzupassen.
- Vor dem Zurückwandeln von der filterinternen Wortbreite auf die ursprüngliche Wortbreite der Daten vom ADC wird das Signal auf den gültigen Bereich begrenzt, um Überläufe zu verhindern.
- Jede Filterstufe und die abschließende Skalierung können unabhängig voneinander einzeln ausgeschaltet werden. Dafür sind zwei Möglichkeiten vorgesehen:
  - *disable:* Das Taktsignal für die Register in den Filterstufen, in denen die vergangenen Werte der Signalfolge gespeichert werden, wird ausgeschaltet *(clock gating)*, so dass der zuletzt gespeicherte Wert bestehen bleibt. Dadurch wird zwar Energie gespart, aber die arithmetische Funktion des Filters ist nicht mehr zweckmäßig.
  - *bypass:* Im Fall der Filterstufen wird der in den Registern gespeicherte Wert durch Null ersetzt. Somit ändert die Stufe das Signal nicht: $y = x + a \cdot 0 + b \cdot 0 = x$. Um die

Skalierung und Verschiebung nach den eigentlichen Filterstufen zu deaktivieren, wird durch einen Multiplexer das Signal daran vorbeigeleitet.

Nach den Untersuchungen in Abschnitt 3.3 ist es sinnvoll, wenn immer einer der Koeffizienten *a* Null ist. Daher kann man eine entsprechende „halbe“ Filterstufe weglassen und somit den schaltungstechnischen Aufwand reduzieren. Die übriggebliebene Hälfte der Filterstufe wird gleich zu Beginn (als „nullte“ Filterstufe) der Signalkette eingesetzt, um die Datenwerte von der ADC-Auflösung `DATA_WIDTH` auf die höhere interne Wortbreite `FILTER_DATA_WIDTH` zu bringen. Dazu gibt es das Modul `SPADIC_FilterStageZero`, während die restlichen Filterstufen durch das Modul `SPADIC_FilterStage` gebildet werden, das im Aufbau dem Modul `filterstage` aus dem vorherigen Abschnitt ähnelt.

Am Ausgang jeder Filterstufe ist ein zusätzliches Register eingefügt. Dadurch wird die Signalfolge zwar um eine Taktperiode pro Filterstufe verzögert, aber ohne diese Zwischenspeicherung müsste sich eine Aktualisierung des Eingangswerts innerhalb einer Taktperiode durch sämtliche Filterstufen bis zum Ausgang fortpflanzen, was ab einer gewissen Anzahl Stufen nicht mehr machbar wäre. So kann aber die Anzahl Stufen bei Bedarf beliebig erhöht werden, ohne die Zeitvorgaben zu verletzen.

Die Beschreibung des Moduls `SPADIC_FilterStageZero` lautet:

```
module SPADIC_FilterStageZero
#(parameter DATA_WIDTH = 9, FILTER_DATA_WIDTH = 12,
                            COEFF_WIDTH = 8)
( input clk,
  input res_n,
  input bypass,
  input disabled,
  input  signed [DATA_WIDTH-1:0] X,
  output signed [FILTER_DATA_WIDTH-1:0] Y,
  input  signed [COEFF_WIDTH-1:0] b );

wire signed [DATA_WIDTH+COEFF_WIDTH:0] R_full;
wire signed [FILTER_DATA_WIDTH-1:0] R;

// internes Register
reg  signed [DATA_WIDTH-1:0] Q;
always @ (posedge clk or negedge res_n)
    if (~res_n)
        Q <= 0;
    else if(!disabled)
        if (bypass)
```

```
            Q <= 0;
        else
            Q <= X;
    else
        Q <= Q;
```

Es wird gleichzeitig der Wert $Q \cdot b + X$ berechnet und die Wortbreite von DATA_WIDTH auf FILTER_DATA_WIDTH erhöht, wobei der Koeffizient b mit einem Skalierungsfaktor 2^{-s} versehen wird, der so gewählt ist, dass effektiv Koeffizienten mit Betrag kleiner als Eins einstellbar sind. Die Erhöhung der Wortbreite ist so gemacht, dass eines der FILTER_DATA_WIDTH − DATA_WIDTH neuen Bits vor dem MSB eingefügt wird, um den verfügbaren Zahlenbereich zu verdoppeln. Dadurch wird die Gefahr von Überläufen verringert. Die restlichen d = FILTER_DATA_WIDTH − DATA_WIDTH − 1 Bits werden an der LSB-Seite angehängt und erhöhen somit die Anzahl der möglichen Zwischenwerte um den Faktor 2^d:

```
parameter d = FILTER_DATA_WIDTH-DATA_WIDTH-1;
parameter s = COEFF_WIDTH-1;
assign R_full = ((Q*b) >>> (s-d)) + (X <<< d);
assign R = R_full[FILTER_DATA_WIDTH-1:0];

// Ausgangsregister
reg signed [FILTER_DATA_WIDTH-1:0] Y_reg;
always @ (posedge clk or negedge res_n)
    if (~res_n)
        Y_reg <= 0;
    else if(!disabled)
        Y_reg <= R;
    else
        Y_reg <= Y_reg;
assign Y = Y_reg;
endmodule
```

Die übrigen „normalen“ Filterstufen sind folgendermaßen aufgebaut:

```
module SPADIC_FilterStage
#(parameter FILTER_DATA_WIDTH = 12, COEFF_WIDTH = 8)
( input clk,
  input res_n,
  input bypass,
  input disabled,
  input  signed [FILTER_DATA_WIDTH-1:0] X,
  output signed [FILTER_DATA_WIDTH-1:0] Y,
```

```
  input  signed [COEFF_WIDTH-1:0] a,
  input  signed [COEFF_WIDTH-1:0] b );

wire signed [FILTER_DATA_WIDTH-1:0] P;
wire signed [FILTER_DATA_WIDTH-1:0] R;

// internes Register
reg signed [FILTER_DATA_WIDTH-1:0] Q;
always @ (posedge clk or negedge res_n)
    if (~res_n)
        Q <= 0;
    else if(!disabled)
        if (bypass)
            Q <= 0;
        else
            Q <= P;
    else
      Q <= Q;

// Ausgangsregister
reg signed [FILTER_DATA_WIDTH-1:0] Y_reg;
always @ (posedge clk or negedge res_n)
    if (~res_n)
        Y_reg <= 0;
    else if(!disabled)
        Y_reg <= R;
    else
        Y_reg <= Y_reg;
assign Y = Y_reg;

// Instanzen von MAC-Modulen
SPADIC_FilterMultAdd #(FILTER_DATA_WIDTH, COEFF_WIDTH)
                     MultAdd_1(Q, X, a, P);
SPADIC_FilterMultAdd #(FILTER_DATA_WIDTH, COEFF_WIDTH)
                     MultAdd_2(Q, P, b, R);
endmodule
```

Die Module `SPADIC_FilterMultAdd` enthalten die eigentliche Multiplizierer-Addierer-Schaltung (MAC), die ja wie gezeigt ausreichend gut durch die arithmetischen Operatoren der Sprache Verilog beschrieben werden kann:

```
module SPADIC_FilterMultAdd
#(parameter FILTER_DATA_WIDTH = 12, COEFF_WIDTH = 8)
( input  signed [FILTER_DATA_WIDTH-1:0] M,
  input  signed [FILTER_DATA_WIDTH-1:0] A,
  input  signed [COEFF_WIDTH-1:0]       C,
  output signed [FILTER_DATA_WIDTH-1:0] R
);
parameter s = COEFF_WIDTH-1;

wire signed [FILTER_DATA_WIDTH+COEFF_WIDTH:0] R_full;
assign R_full = ((M*C) >>> s) + A;
assign R = R_full[FILTER_DATA_WIDTH-1:0];
endmodule
```

Schließlich wird aus diesen Bausteinen das Modul `SPADIC_Filter` zusammengesetzt, wobei die Anzahl der Stufen durch `NUM_STAGES` parametrisiert ist:

```
module SPADIC_Filter
#(parameter DATA_WIDTH = 9, NUM_STAGES = 4,
            FILTER_DATA_WIDTH = 12, COEFF_WIDTH = 8)
( input clk,
  input res_n,
  input  signed [DATA_WIDTH-1:0] X,
  output signed [DATA_WIDTH-1:0] Y,
  input [NUM_STAGES:0] REG_bypass,
  input REG_disable,
  input [ NUM_STAGES   *COEFF_WIDTH-1:0] REG_b,
  input [(NUM_STAGES-1)*COEFF_WIDTH-1:0] REG_a,
  input signed [COEFF_WIDTH-1:0] REG_scaling,
  input signed [DATA_WIDTH-1:0] REG_offset );

wire [FILTER_DATA_WIDTH-1:0] R [0:NUM_STAGES-1];
```

Zunächst wird eine Instanz von `SPADIC_FilterStageZero` erzeugt:

```
SPADIC_FilterStageZero
#(DATA_WIDTH, FILTER_DATA_WIDTH, COEFF_WIDTH)
StageZ(
    .clk(clk),
    .res_n(res_n),
    .bypass(REG_bypass[0]),
```

```
    .X(X),
    .Y(R[0]),
    .b(REG_b[COEFF_WIDTH-1:0]),
    .disabled(REG_disable)
);
```

Anschließend folgen `NUM_STAGES` – 1 Instanzen von `SPADIC_Filterstage`, deren Ein- und Ausgänge `X` und `Y` miteinander zu einer Reihenschaltung verbunden werden:

```
genvar i;
generate for (i=1; i<NUM_STAGES; i=i+1)
    begin: StageGen
        SPADIC_FilterStage
        #(FILTER_DATA_WIDTH, COEFF_WIDTH)
        Stage(
            .clk(clk),
            .res_n(res_n),
            .bypass(REG_bypass[i]),
            .X(R[i-1]),
            .Y(R[i]),
            .a(REG_a[i*COEFF_WIDTH-1:(i-1)*COEFF_WIDTH]),
            .b(REG_b[(i+1)*COEFF_WIDTH-1:i*COEFF_WIDTH]),
            .disabled(REG_disable)
        );
    end
endgenerate
```

Daran schließt sich die Logik für die (ausschaltbare) Skalierung und Verschiebung inklusive Begrenzung des Signals an. Im Gegensatz zu den Filterkoeffizienten kann der Skalierungsfaktor auf Werte zwischen −2 und 2 gestellt werden, indem der Parameter `s` um Eins kleiner gewählt ist:

```
parameter d = FILTER_DATA_WIDTH-DATA_WIDTH-1;
parameter s = COEFF_WIDTH-2;
wire signed [FILTER_DATA_WIDTH-1:0]           filter_out;
wire signed [FILTER_DATA_WIDTH+COEFF_WIDTH:0] Y_full;
wire signed [FILTER_DATA_WIDTH-1:0]           Y_noscaling;
wire signed [FILTER_DATA_WIDTH-d-1:0]         Y_preclip;
wire signed [DATA_WIDTH-1:0]                  Y_clipped;
assign filter_out = R[NUM_STAGES-1];
assign Y_full = ((filter_out * REG_scaling) >>> (s+d))
                + REG_offset;
```

```
assign Y_noscaling = filter_out >>> d;
assign Y_preclip = REG_bypass[NUM_STAGES] ?
                   Y_noscaling[FILTER_DATA_WIDTH-d-1:0] :
                   Y_full[FILTER_DATA_WIDTH-d-1:0];

parameter DATA_MAX =   2**(DATA_WIDTH-1)-1;
parameter DATA_MIN = -(2**(DATA_WIDTH-1));
assign Y_clipped = (Y_preclip > DATA_MAX) ?  DATA_MAX :
                   (Y_preclip < DATA_MIN) ?  DATA_MIN :
                   Y_preclip[DATA_WIDTH-1:0];
```

Zum Schluss wird ein Ausgangsregister erzeugt:

```
reg signed [DATA_WIDTH-1:0] Y_reg;
always @ (posedge clk or negedge res_n)
    if (~res_n)
        Y_reg <= 0;
    else if(REG_disable)
        Y_reg <= Y_reg;
    else
        Y_reg <= Y_clipped;
assign Y = Y_reg;
endmodule
```

4.5.2 Wahl der Parameter

Die Wortbreite der Filtereingangs- und Ausgangswerte, `DATA_WIDTH`, ist auf den Wert 9 festgelegt, was die Anzahl Bits der vom ADC kommenden Daten ist.

Nach den Ergebnissen von Abschnitt 3.2 können die Pulse, die als Eingangssignale das Filter erreichen durch eine Überlagerung von vier Pulsfolgen modelliert werden. Daher ist diese Zahl auch ein sinnvoller Wert für `NUM_STAGES`, denn in einer Stufe kann je eine der überlagerten Komponenten ausgelöscht werden.

Die Wortbreite der Filterkoeffizienten muss nicht sehr hoch sein. Ein geringerer Wert bedeutet keine Ungenauigkeit in der Berechnung des Ausgangssignals, sondern nur eine kleinere Anzahl von möglichen Einstellungen. Die Untersuchung in Abschnitt 3.3 hat gezeigt, dass eine Abweichung der Koeffizienten von den „optimalen“ Werten keine gravierende Verschlechterung der Tail Cancellation bewirkt. Ein minimaler Wert von `COEFF_WIDTH` = 5 scheint angebracht, damit jeder der Koeffizienten wenigstens je in 16 Stufen, d. h. in Schritten von $1/16 = 0.0625$, auf der positiven wie negativen Seite einstellbar ist. Mehr als `COEFF_WIDTH` = 8, also eine Abstufung von $1/128 \approx 0.008$, sollte nicht erforderlich sein.

Einzig die Wortbreite `FILTER_DATA_WIDTH` der intern gespeicherten Datenwerte muss noch ermittelt werden. Sie muss so gewählt werden, dass die Rundungsfehler beim Berechnen der Ausgangswerte nicht zu groß werden. Dies wird in Kapitel 6 untersucht, und zwar mithilfe der dafür notwendigen Filtersimulationssoftware `iirsim`, die im folgenden Kapitel vorgestellt wird.

5 Software zur Simulation des Filters

5.1 Zweck und Funktionsumfang

Schon früh im Verlauf dieser Arbeit wurde klar, dass es unabdingbar ist, ein Werkzeug in Form eines Computerprogramms in der Hand zu haben, mit dem sich das Verhalten eines digitalen Filters nachbilden lässt. Das bedeutet, dass ein Softwaremodell des Filters geschaffen werden sollte, welches die Rechenoperationen, die darin stattfinden, unter Berücksichtigung der Quantisierungseffekte simuliert.

Das Anwendungsgebiet des Softwaremodells ist einerseits von der rein mathematischen Beschreibung – durch die Systemfunktion oder eine Differenzengleichung – eines Filters abgegrenzt, bei der sich die Auswirkungen der Quantisierung nicht gut vorhersagen lassen. Andererseits werden nicht alle Details berücksichtigt, die bei der Simulation einer elektronischen Schaltung eine Rolle spielen, wie z. B. die zeitliche Verzögerung der Signale zwischen oder innerhalb von Bauteilen.

Vereinfacht ausgedrückt besteht die Funktion des Programms also darin, bei gegebener Eingangszahlenfolge und Filterstruktur die Ausgangszahlenfolge zu berechnen.

Während der Entwicklung des SPADIC-Filters hat das Programm wesentlich dazu beigetragen, das Verhalten digitaler Filter zu verstehen und die Auswirkung der verschiedenen Kenngrößen sowie die Möglichkeiten, die ein Filter bietet, einschätzen zu können. Außerdem diente es als Referenz, wenn es darum ging, die Korrektheit der Verilog-Beschreibung des Filters zu überprüfen.

Daher wurde viel Sorgfalt darauf verwandt, eine solche Software so zu erstellen, dass die Rechenoperationen im Hinblick auf die Quantisierung genau unter Kontrolle bleiben, d. h. dass zum Beispiel das Abrunden einer Zahl explizit im Programmcode festgehalten ist und nicht etwa durch Verwendung eines Datentyps oder eines Operators mit beschränkter Genauigkeit impliziert wird. Dadurch ist es auch möglich, die Quantisierungseffekte auszuschalten, um somit ein ideales Filter zu simulieren, dessen Verhalten man leicht mit dem nicht-idealen vergleichen kann.

Außerdem sollte die Möglichkeit bestehen, verschiedene Filterstrukturen in der Software abbilden zu können, ohne sie von vorneherein im Quellcode festzulegen. Deshalb kann ein Filter durch eine Konfigurationsdatei beschrieben werden, die vom Programm eingelesen und in eine interne Datenstruktur übertragen wird.

Als zusätzliche Funktion wurde noch eine grafische Benutzeroberfläche erstellt, die es möglich macht, mit verschiedenen Filterkoeffizienten zu experimentieren und unmittelbar die Auswirkung zu beobachten.

Die Software wurde unter dem Namen `iirsim` in der Sprache *Python* [41] programmiert. Die beschriebene Funktionalität ist auf drei Module aufgeteilt:

- `iirsim_lib`: Enthält die Grundfunktionen für die Simulation eines Filters.
- `iirsim_cfg`: Bietet die Möglichkeit, Filterstrukturen aus Konfigurationsdateien einzulesen.
- `iirsim_gui`: Macht die beiden zuvor genannten Module durch eine grafische Oberfläche zugänglich.

5.2 Beschreibung des Hauptmoduls

Die Grundidee des Programms besteht darin, durch einen Graphen, also eine durch Kanten verbundene Menge von Knoten, das Blockdiagramm nachzubilden, welches ein kausales LTI-System definiert.

Dabei bilden die Knoten Instanzen der Grundelemente Addierer, Multiplizierer und Verzögerer und die Kanten definieren die Verbindungen, über die die Daten (also Zahlenwerte) von einem Knoten zum nächsten gelangen. Einer der Knoten stellt den Eingang des Filters dar und ein Knoten (sinnvollerweise ein anderer, es kann aber theoretisch auch derselbe sein) den Ausgang des Filters.

In dieser abstrakten Sichtweise ist den verschiedenen Filterelementen gemein, dass sie einen Ausgangswert liefern, der eine durch eine gewisse Anzahl Bits dargestellte Zahl ist. Dieser Ausgangswert kann auf verschiedene Weisen von einem oder mehreren Eingangswerten abhängen, oder unabhängig davon sein. Die Eingangswerte stammen jeweils von anderen Knoten, zu denen eine Verbindung besteht.

Dabei ist zu beachten, dass eine gerichtete Kante $A \rightarrow B$ vom Knoten A zum Knoten B „B ist mit einem Eingang von A verbunden“ oder „A erhält Daten von B“ bedeutet, d. h. die Richtung des Datenflusses und die Richtung der Kanten sind einander entgegengesetzt.

In Abbildung 5.1 ist als Beispiel eine einfache Filterstruktur und der zugehörige Graph gezeigt. Man sieht wie die unterschiedlichen Filterelemente jeweils einem Knoten („Add“, „Mult“ und „Delay“) entsprechen.

Um zu einem Zeitschritt den Ausgangswert des gesamten Filters zu erhalten, muss man den Wert des als Ausgang definierten Knotens Y abfragen. Um diesen zu bekommen, muss man wiederum die Ausgangswerte derjenigen Knoten abfragen, die mit den Eingängen

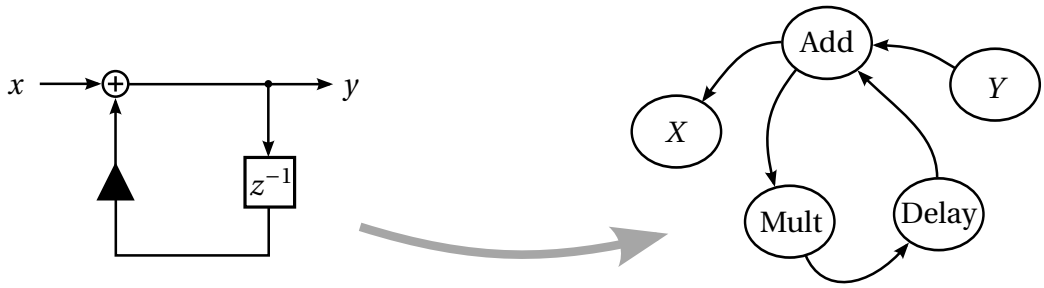

Abbildung 5.1: Darstellung eines Blockdiagramms als Graph

von Y verbunden sind. Das bedeutet, dass die Berechnung *rekursiv* stattfindet, indem man sich von Y ausgehend entlang der gerichteten Kanten durch den Graphen bewegt. Dieser Vorgang setzt sich so lange fort, bis man bei Knoten angelangt ist, deren Ausgangswerte *nicht von den Eingängen abhängig* sind. Man folgt also nicht zwingend jeder Kante, die es gibt. Beispielsweise gibt es zwar in dem Beispiel in der Abbildung eine Kante vom Knoten „Delay“ (der dem Verzögerungselement in der linken Bildhälfte entspricht) zum Knoten „Add“, aber der Ausgangswert von „Delay“ ist zu einem gegebenen Zeitschritt unabhängig vom Ausgangswert von „Add“.

Durch diese Rekursion wird also genau die Berechnungsvorschrift für die Ausgangsfolge eines kausalen LTI-Systems umgesetzt, die sich aus der zu dem Blockdiagramm gehörenden Differenzengleichung ergibt.

Es kann passieren, dass die Berechnung des Filterausgangswerts in einer Endlosschleife endet, wenn die Knoten und Kanten nicht richtig eingerichtet sind. Der Benutzer muss durch sinnvolles Konfigurieren der Struktur dafür sorgen, dass es nicht dazu kommt. Möglicherweise kann das Programm in Zukunft so weiterentwickelt werden, dass solche Fälle erkannt und automatisch vermieden werden. Nach dem derzeitigen Stand ist das aber nicht realisiert.

Die Umsetzung des Filters in einen Graphen geschieht durch eine Python-Klasse, deren Instanzen die Knoten darstellen. Die zugehörigen Attribute, die einen Knoten kennzeichnen, sind:

- die Anzahl der Eingänge
- eine Liste mit den verbundenen Knoten
- die Anzahl der Bits für die Zahlenwerte, die den Knoten durchlaufen

Weiterhin gibt es Methoden, die die oben beschriebene Rekursion realisieren, sowie andere, mit denen man beispielsweise die Verbindungen zu anderen Knoten oder die Anzahl der Bits ändern kann.

Die verschiedenen Typen von Filterelementen sind als Unterklassen von dieser Hauptklasse abgeleitet. Der wesentliche Unterschied zwischen den Unterklassen besteht darin, wie der Ausgangswert eines Knotens berechnet wird. Manche haben aber auch spezielle Methoden, die für die anderen Knotentypen nicht relevant sind.

Schließlich gibt es noch eine weitere Klasse, die ein ganzes Filter repräsentiert, indem als Attribut ein solcher Graph gespeichert wird und Methoden zur Ein- und Ausgabe von Daten zur Verfügung gestellt werden.

Alle diese Klassen sind im Modul `iirsim_lib` enthalten. Für das Rechnen mit Zahlen, die mit begrenzter Wortbreite dargestellt werden, gibt es darin auch noch einige interne Hilfsfunktionen.

Hätte man nur dieses Modul, müsste man zur Simulation eines Filters alle Knoten des zugehörigen Graphen instanziieren und auf die gewünschte Art und Weise miteinander verbinden. Dieses Vorgehen kann durch das Modul `iirsim_cfg` automatisiert werden. Es erlaubt dem Benutzer, eine Filterstruktur in einer Konfigurationsdatei zu beschreiben. Nach dem Einlesen einer solchen Datei werden die richtigen Instanzen der Knoten-Klasse erzeugt, miteinander verbunden und als eine Instanz der Filter-Klasse zurückgegeben.

In den folgenden Unterabschnitten wird zunächst näher auf die Implementierung der Module `iirsim_lib` und `iirsim_cfg` eingegangen. Die Beschreibung des Moduls `iirsim_gui` für die grafische Benutzeroberfläche folgt in einem separaten Abschnitt.

Bei der Erläuterung des Programms sind nicht immer alle Teile des Quelltextes angegeben, um die Übersichtlichkeit zu erhalten. Dies betrifft vor allem die Behandlung von Ausnahmefällen, Kommentare oder Fehlermeldungstexte, die hier mit " . . . " abgekürzt wiedergegeben werden können.

Manche Variablen-, Funktions-, oder Klassennamen beginnen mit einem einzelnen Unterstrich („`_Name`“). Dies entspricht der Konvention, nur intern (in Klassen oder Modulen) verwendete Variablen als solche zu kennzeichnen [44].

5.2.1 Umgang mit quantisierten Zahlenwerten

In dem Modul `iirsim_lib` sind einige Funktionen definiert, die dazu dienen, die durch eine gewisse Anzahl Bits dargestellten Zahlen zu behandeln.

Innerhalb dieses Programms wird davon ausgegangen, dass alle Zahlen im Zweierkomplement dargestellt werden. Das bedeutet, dass eine Zahl, die durch N Bits repräsentiert wird, einen der ganzzahligen Werte $\{-2^{N-1}, \ldots, 2^{N-1}-1\}$ annehmen kann. Um die Quantisierungseffekte zu simulieren, muss man überprüfen können, ob eine Zahl einen der erlaubten Werte hat.

Da in Python die Variablen keinen festen Datentyp besitzen, sondern ein Objekt eines beliebigen Typs bezeichnen können (*Dynamic Typing*), muss hier zunächst einmal explizit

geprüft werden, ob eine Variable überhaupt auf ein Objekt verweist, das eine ganze Zahl ist.[1] Das geschieht mit der Funktion `_test_int`:

```
def _test_int(x):
    return (isinstance(x, int) or isinstance(x, long))
```

In der Funktion `_test_overflow` wird abgefragt, ob eine Zahl `x`, die durch `N` Bits dargestellt werden soll, außerhalb des erlaubten Wertebereichs liegt. Dabei wird vorausgesetzt, dass es sich um eine ganze Zahl handelt, andernfalls wird eine Fehlermeldung erzeugt.

```
def _test_overflow(x, N):
    if not _test_int(x):
        raise TypeError("...")
    B = 2**(N-1)
    return (x < -B) or (x > B-1)
```

Eine Zahl, die keinen der erlaubten Werte hat, muss bei der Simulation eines nicht-idealen Filters durch eine gültige Zahl ersetzt werden. Dafür sind zwei Möglichkeiten vorgesehen: Entweder die Zahl „geht in Sättigung“, d. h. sie wird durch den kleinstmöglichen Wert ersetzt, wenn sie kleiner als dieser ist, oder durch den größtmöglichen Wert, wenn sie größer als dieser ist:

```
def _saturate(x, N):
    if not _test_int(x):
        raise TypeError("...")
    B = 2**(N-1)
    if x < -B:
        return -B
    elif x > B-1:
        return B-1
    else:
        return x
```

Oder es wird ein Überlauf simuliert, der sich bei der Darstellung im Zweierkomplement wie eine Modulo-Operation verhält:

```
def _wrap(x, N):
    if not _test_int(x):
        raise TypeError("...")
    B = 2**(N-1)
```

[1] In der Python-Version 2.7, für die das Programm geschrieben ist, gibt es zwei Datentypen, die ganze Zahlen darstellen: `int` und `long`. Ab der Version 3.0 werden die beiden leicht unterschiedlichen Typen zu einem einzigen mit dem Namen `int` vereinheitlicht [45].

```
    return ((x+B) % 2**N) - B
```

5.2.2 Grundklasse für die Knoten

In der Klasse `_FilterNode` ist die grundlegende Funktion der Knoten festgehalten.

Beim Erzeugen einer Instanz[2] werden die Anzahl der Bits für den Ausgangswert des Knotens und die Anzahl der Eingänge festgelegt. Gleichzeitig wird eine Liste der Eingangsknoten angelegt, die später Verweise auf andere Instanzen von `_FilterNode` enthält, wodurch die Kanten des Graphen definiert werden. Zu Beginn ist diese Liste noch mit Platzhalter-Objekten vom Typ `None` gefüllt, um kenntlich zu machen, dass die Eingänge noch nicht mit anderen Knoten verbunden sind.

```
class _FilterNode():
    def __init__(self, ninputs, bits):
        self.set_bits(bits)
        self._ninputs = ninputs
        self._input_nodes = [None for i in range(ninputs)]
```

Dabei wird schon die Methode `set_bits` verwendet, die verhindert, dass mit der Variable `bits` unsinnige Werte übergeben werden: Es müssen mindestens zwei Bits verwendet werden, damit überhaupt positive Zahlen dargestellt werden können (mit einem Bit lassen sich im Zweierkomplement nur die Zahlen 0 und -1 darstellen).

```
    def set_bits(self, bits):
        if not _test_int(bits):
            raise TypeError("...")
        elif bits < 2:
            raise ValueError("...")
        else:
            self._bits = bits
```

Mit der Methode `connect` wird eine Liste, die andere Instanzen der Klasse `FilterNode` enthält, an einen Knoten übergeben. Wenn die Länge der Liste mit der Anzahl der Eingänge des Knotens und die Wortbreite aller Knoten übereinstimmen, wird die Liste als die der am Eingang verbundenen Knoten übernommen.

```
    def connect(self, input_nodes):
        if not len(input_nodes) == self._ninputs:
            raise ValueError("...")
```

[2] Wenn eine Klasse eine Methode mit dem besonderen Namen `__init__` besitzt, wird diese beim Erzeugen einer Instanz automatisch ausgeführt.

```
        elif not all([node._bits == self._bits
                      for node in input_nodes]):
            raise ValueError("...")
        else:
            self._input_nodes = input_nodes
```

Sobald auf diese Weise alle Eingänge eines Knotens angeschlossen sind, kann man mit der folgenden Methode die Eingangswerte abfragen. An dieser Stelle ist die Möglichkeit eingebaut, zwischen einem idealen Filter ohne Quantisierungseffekte und einem nicht-idealen Filter zu unterscheiden, indem der Schalter `ideal`, der im Normalfall den Wert `False` hat, bei Bedarf auf `True` gesetzt wird. Dann wird die Überprüfung der Eingänge auf den gültigen Wertebereich umgangen, die bei einem nicht-idealen Filter dafür sorgt, dass nur Eingangswerte akzeptiert werden, die mit der gegebenen Anzahl Bits dargestellt werden können.

```
    def _get_input_values(self, ideal=False):
        if any([node is None
                for node in self._input_nodes]):
            raise RuntimeError("...")
        input_values = [node.get_output(ideal)
                        for node in self._input_nodes]
        if not ideal:
            if any([_test_overflow(value, self._bits)
                    for value in input_values]):
                raise ValueError("...")
        return input_values
```

Der Wert der Eingangsknoten wird durch den Aufruf deren Methode `get_output` geliefert. Diese ist je nach Typ des Knotens (Addierer, Multiplizierer, ...) unterschiedlich implementiert und ist in der Basisklasse nur als Platzhalter definiert:

```
    def get_output(self, ideal=False):
        raise NotImplementedError
```

5.2.3 Unterklassen für verschiedene Knotentypen

Die Grundelemente, aus denen ein Filter aufgebaut ist, sind Addierer, Multiplizierer und Verzögerer. Für jeden dieser Typen von Knoten gibt es eine von `_FilterNode` abgeleitete Unterklasse. Zusätzlich gibt es jedoch noch eine weitere Unterklasse. Diese hat eine spezielle Funktion als Eingabeknoten des gesamten Filters: Sie hat selbst keine Eingänge und

ihr Wert wird direkt vom Benutzer gesetzt, anstatt dass er von anderen Knoten abhängt. Somit gibt es vier verschiedene Knotentypen, deren Charakteristik in der folgenden Tabelle zusammengefasst ist:

Knotentyp	Eingänge	Ausgangswert
Addierer	2	Summe der Eingangswerte
Multiplizierer	1	Vielfaches des Eingangswerts
Verzögerer	1	Eingangswert des vorherigen Zeitschritts
Eingabeknoten	0	vom Benutzer gesetzter Wert

Addierer

Die Funktionalität des Addiererelements ist in der Klasse `Add` implementiert. Sie weicht nur in der Konstruktormethode `__init__`, in der die Anzahl der Eingänge festgelegt wird, und in der Methode `get_output`, in der der Ausgangswert berechnet wird, von der Grundklasse ab.

```
class Add(_FilterNode):
    def __init__(self, bits):
        _FilterNode.__init__(self, 2, bits)

    def get_output(self, ideal=False):
        input_values = self._get_input_values(ideal)
        S = sum(input_values)
        if not ideal:
            value = _wrap(S, self._bits)
        else:
            value = S
        return value
```

Durch Verwendung der Funktion `_wrap` wird ein Überlauf simuliert, wenn die Summe der Eingangswerte zu klein oder zu groß ist.

Multiplizierer

Die Unterklasse `Multiply` für den Multiplizierer-Knotentyp ist komplexer aufgebaut. Für jede Instanz dieser Klasse muss der Faktor gespeichert werden, mit dem der Eingangswert multipliziert wird. Der Faktor besteht aus dem Quotienten einer ganzen Zahl M (die auch durch eine Anzahl Bits m im Zweierkomplement dargestellt wird) und einer Normierungs-

konstante 2^s:

$$\text{Ausgangswert} = \frac{M}{2^s} \cdot \text{Eingangswert}$$

Auf diese Weise können Faktoren realisiert werden, die keine ganzen Zahlen sind, sondern je nach Normierung innerhalb eines Wertebereichs in 2^m Stufen einstellbar sind:

$$\text{minimaler Faktor} = \frac{-2^{m-1}}{2^s} = -2^{m-s-1}$$
$$\text{maximaler Faktor} = \frac{2^{m-1}-1}{2^s} = 2^{m-s-1} - \frac{1}{2^s}$$

Zum Beispiel kann für $m = 8$ und $s = 6$ der Faktor die 256 verschiedenen Werte in Abständen von $\frac{1}{64}$ zwischen $-\frac{128}{64} = -2$ und $\frac{127}{64} = 2 - \frac{1}{64}$ annehmen.

Um dieses Verhalten nachzubilden, gibt es in der Unterklasse `Multiply` folgende Attribute:

- `factor_bits`: die Anzahl der Bits m für die Darstellung von M
- `factor`: der Wert von M
- `norm_bits`: die Anzahl der Bits s für die Normierung (die Division mit 2^s entspricht in der Zweierkomplementdarstellung dem Verschieben um s Bits nach rechts)

Die Methode `__init__` ist so modifiziert, dass diese zusätzlichen Werte aufgenommen werden.

```
class Multiply(_FilterNode):
    def __init__(self, bits,
                 factor_bits, norm_bits, factor=0):
        _FilterNode.__init__(self, 1, bits)

        if not _test_int(factor_bits):
            raise TypeError("...")
        elif factor_bits < 2:
            raise ValueError("...")
        else:
            self._factor_bits = factor_bits

        if not _test_int(norm_bits):
            raise TypeError("...")
        elif norm_bits < 0:
```

```
            raise ValueError("...")
        else:
            self._norm_bits = norm_bits

        self.set_factor(factor)
```

Der Faktor wird mit der Methode `set_factor` eingestellt. Man kann den Faktor entweder in Form des ganzzahligen Zählers angeben oder inklusive der Normierung als Fließkommazahl, indem man den Schalter `norm` beim Funktionsaufruf auf `True` setzt. Dann wird `factor` (der Zähler) so gewählt, dass der angegebene Wert so nahe wie möglich erreicht wird.

```
    def set_factor(self, factor, norm=False):
        if norm:
            factor = int(round((1 << self._norm_bits)
                               * factor))
        if _test_overflow(factor, self._factor_bits):
            raise ValueError("...")
        else:
            self._factor = factor
```

Die Berechnung des Ausgangswerts geschieht wieder in der `get_output`-Methode. Der Eingangswert wird (als Liste mit einem Element) abgefragt und als Fließkommazahl mit dem eingestellten Faktor multipliziert, was den idealen Ausgangswert ergibt. Falls Quantisierungsfehler simuliert werden sollen (`ideal` hat den Wert `False`), wird dieser ideale Wert zunächst auf die nächst kleinere ganze Zahl abgerundet und dann mit der Funktion `_wrap` auf einen der für die eingestellte Anzahl Bits gültigen Ausgangswerte abgebildet.

```
    def get_output(self, ideal=False):
        [input_value] = self._get_input_values(ideal)
        idealvalue = input_value*(self._factor
                                  / 2.0**self._norm_bits)
        if not ideal:
            P = int(math.floor(idealvalue))
            value = _wrap(P, self._bits)
        else:
            value = idealvalue
        return value
```

Verzögerer

Die Funktion des Verzögerungselements ist im Grunde einfach, die korrekte Implementierung in diesem Programm in der Unterklasse `Delay` hat sich jedoch als etwas trickreicher erwiesen.

Ein Knoten dieses Typs hat einen Wert gespeichert, der zu einem früheren Zeitpunkt von dem am Eingang verbundenen Knoten übertragen wurde. Dieser Wert wird solange am Ausgang ausgegeben, bis er durch einen neuen Wert ersetzt wird. Der Ausgangswert ist also nicht direkt vom Eingangswert abhängig, wodurch ein solcher Knoten als Endpunkt der rekursiven Berechnung des Filterausgangswerts dient.

Zunächst sind hier wieder die Konstruktormethode und die Methode zur Ausgabe des gespeicherten Werts angegeben:

```
class Delay(_FilterNode):
    def __init__(self, bits):
        _FilterNode.__init__(self, 1, bits)
        self.reset()
```

Hier wird eine Methode `reset` aufgerufen, deren Bedeutung im Anschluss erläutert wird. Fürs Erste ist entscheidend, dass sie die interne Variable `_value` initialisiert, die als Ausgangswert dient:

```
    def get_output(self, ideal=False):
        value = self._value
        if not ideal:
            value = int(value)
        return value
```

Der problematische Teil besteht darin, wie das Ersetzen des gespeicherten Werts durch den Eingangswert ausgeführt wird. Die naheliegendste Lösung wäre eine Funktion dieser Art:

```
    def update_value(self):
        [input_value] = self._get_input_values()
        self._value = input_value
```

Dann käme es aber in dem folgenden Beispielszenario zu einem Problem:

Es seien zwei Verzögerungselemente D_1 und D_2 hintereinandergeschaltet, so dass der Ausgang von D_1 mit dem Eingang von D_2 verbunden ist (im zugehörigen Graphen gäbe es also eine gerichtete Kante $D_2 \to D_1$). Zu einem Zeitschritt liege am Eingang von D_1 der Wert a an und in D_1 selbst sei der Wert b gespeichert, liege also am Eingang von D_2 an. Diese Situation ist in Abbildung 5.2 gezeigt.

Abbildung 5.2: Beispielszenario mit zwei in Reihe geschalteten Verzögerungselementen und dem zugehörigen Graphen

Geht man zum nächsten Zeitschritt über, erwartet man daher das folgende Verhalten: Der Wert a wird von D_1 übernommen, während gleichzeitig der Wert b von D_2 übernommen wird.

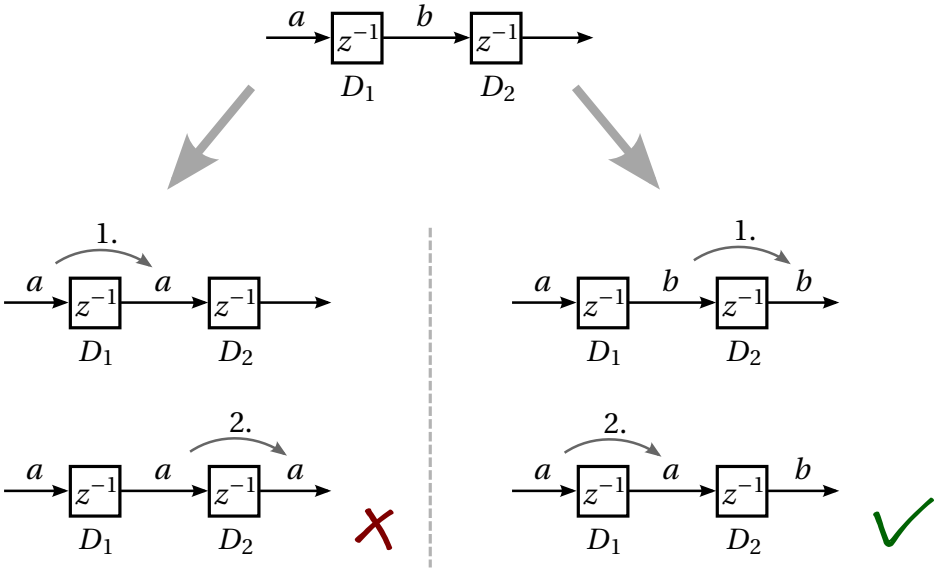

Abbildung 5.3: Bei falscher Implementierung hängt der Endzustand davon ab, in welcher Reihenfolge die Aktualisierungsvorgänge abgearbeitet werden.

Um in der Simulation den Übergang von einem Zeitschritt zum nächsten nachzubilden, muss man für alle Verzögerungselemente einmal den gespeicherten Wert aktualisieren. Mit der beschriebenen Funktion `update_value` gäbe es in dem Beispiel zwei Möglichkeiten (wie auch in Abbildung 5.3 illustriert):

- Zuerst wird der Wert von D_1 aktualisiert und dann der Wert von D_2. Dann würde in D_1 der gespeicherte Wert b durch a ersetzt werden. Danach würde in D_2 der am Eingang anliegende Wert, der jetzt a ist, übernommen. Es wäre also schließlich sowohl in D_1 als auch in D_2 der Wert a gespeichert und der Wert b wäre verlorengegangen. Das ist nicht das gewünschte Ergebnis.

- Es wird zuerst der Wert von D_2 und dann der Wert von D_1 aktualisiert. Dann würde b in D_2 und a in D_1 gespeichert sein, was korrekt ist.

Aus diesem Beispiel folgt: Wenn man die Methode `update_value` verwenden würde, hinge das Simulationsergebnis davon ab, in welcher Reihenfolge die in den Verzögerungselementen gespeicherten Werte aktualisiert werden. Um ein korrektes Ergebnis zu erhalten, müsste man für eine gegebene Filterstruktur die richtige Reihenfolge herausfinden, wobei zunächst noch nicht einmal klar ist, ob es diese überhaupt immer gibt.

In der tatsächlich vorliegenden Implementierung der Simulation ist diese Problematik daher so umgangen, dass das Ergebnis der Simulation *nicht* von der Reihenfolge der Aktualisierung abhängt. Statt der Methode `update_value` gibt es zwei Methoden `sample` und `clk`[3], auf die die beiden Schritte

- Abfragen des Eingangswerts
- Ersetzen des gespeicherten Werts

aufgeteilt sind. Außerdem gibt es noch eine weitere interne Variable `_next_value`, in der der schon gelesene, aber noch nicht übernommene Eingangswert zwischengespeichert wird.

Um nun den Übergang zum nächsten Zeitschritt zu simulieren, muss man zuerst für alle Instanzen von `Delay` in beliebiger Reihenfolge die Methode `sample` aufrufen, so dass der jeweilige Eingangswert schon gespeichert wird, ohne dass er am Ausgang sichtbar ist:

```
    def sample(self, ideal=False):
        [input_value] = self._get_input_values(ideal)
        self._next_value = input_value
```

Danach wird für alle Instanzen in beliebiger Reihenfolge die Methode `clk` aufgerufen und somit die zuvor zwischengespeicherten Werte als neue Ausgangswerte übernommen:

```
    def clk(self):
        self._value = self._next_value
```

Schließlich gibt es noch die schon zuvor erwähnte Methode `reset`, die die beiden internen Variablen `_value` und `_next_value` auf einen definierten Anfangswert setzt:

```
    def reset(self):
        self._value = 0
        self._next_value = 0
```

[3] „clock“, in Anlehnung an die übliche Bezeichnung des Taktsignals, das bei einer digitalen Schaltung den Übergang zwischen den Zeitschritten auslöst

Eingabeknoten

Die vierte Unterklasse mit dem Namen `Const` ist dagegen wieder sehr einfach, weil sie auch nur den Zweck hat, einen vom Benutzer gesetzten Wert auszugeben, welcher als Eingabewert des ganzen Filters dient. Somit sind Knoten dieses Typs (normalerweise gibt es nur einen davon) neben den Verzögerungsknoten diejenigen, an denen die Rekursion zur Berechnung des Filterausgangswerts aufgelöst wird.

In der Konstruktormethode `__init__` kann ein Anfangswert übergeben werden, falls er weggelassen wird, wird stattdessen der Wert 0 gespeichert.

```
class Const(_FilterNode):
    def __init__(self, bits, value=0):
        _FilterNode.__init__(self, 0, bits)
        self.set_value(value)
```

Das Setzen des Werts geschieht in der Methode `set_value`. Es werden nur Werte akzeptiert, die mit der eingestellten Anzahl Bits darstellbar sind.

```
    def set_value(self, value, ideal=False):
        if not ideal:
            if _test_overflow(value, self._bits):
                raise ValueError("...")
        self._value = value
```

Der Wert wird durch Aufrufen der Methode `get_output` wieder ausgegeben. Der Parameter `ideal` wird zwar nicht benötigt, ist aber dennoch als Argument enthalten, damit die Methode für alle Unterklassen die gleiche Schnittstelle hat.

```
    def get_output(self, ideal=False):
        value = self._value
        return value
```

5.2.4 Klasse für das gesamte Filter

In der Klasse `Filter` wird die Darstellung eines Filters als Graph, der aus Instanzen der verschiedenen Unterklassen von `_FilterNode` besteht, gekapselt und mit Benutzerschnittstellen versehen.

Um ein Objekt vom Typ `Filter` zu erstellen, wird eine Menge von `_FilterNode`-Instanzen sowie die Angabe über die Verbindungen zwischen ihnen an die Methode `__init__` übergeben. Das geschieht jeweils in Form eines *Dictionarys*, der Python-Implementierung eines assoziativen Arrays. Dabei ist einem Dictionary eine Menge von Objekten abgelegt, auf die

jeweils über einen eindeutigen Schlüssel zugegriffen werden kann. Das lässt diese Datenstruktur zunächst ähnlich zu einer Python-Liste erscheinen, aber im Gegensatz dazu sind die Schlüssel nicht die natürlichen Zahlen, sondern können komplexere Objekte sein. In diesem Fall sind es Zeichenketten (*Strings*), die die vom Benutzer festgelegten Namen der einzelnen Knoten darstellen.

In dem Dictionary `node_dict` befindet sich zu einem Namen eines Knotens eine Instanz einer Unterklasse von `_FilterNode`, im Dictionary `adjacency_dict` zu jedem Namen eines Knotens eine Liste mit Namen anderer Knoten, zu denen es eine gerichtete Kante gibt. Dazu kommen als Argumente von `__init__` die Namen der beiden Knoten, die als Eingang und Ausgang des Filters fungieren.

Zunächst wird überprüft, ob es sich bei allen Objekten in `node_dict` (die mit der Dictionary-Methode `values` als Liste zurückgegeben werden) um Objekte vom Typ `_FilterNode` bzw. einer davon abgeleiteten Klasse handelt. Außerdem muss der Knoten, der als Eingang gekennzeichnet ist, vom Typ `Const` sein. Falls diese beiden Tests bestanden sind, werden ersteinmal alle übergebenen Objekte als interne Variablen gespeichert.

```
class Filter():
    def __init__(self, node_dict, adjacency_dict,
                 in_node, out_node):
        if not all([isinstance(node, _FilterNode)
                    for node in node_dict.values()]):
            raise TypeError("...")
        if not isinstance(node_dict[in_node], Const):
            raise TypeError("...")
        self._nodes = node_dict
        self._adjacency = adjacency_dict
        self._in_node = node_dict[in_node]
        self._out_node = node_dict[out_node]
```

Dann wird eine Liste mit allen Knoten vom Typ `Delay` sowie eine Liste mit den *Namen* aller Knoten vom Typ `Multiply` angelegt (die Dictionary-Methode `keys` gibt eine Liste mit allen Schlüsseln, also allen Namen der Knoten, zurück):

```
        self._delay_nodes = \
            [node for node in self._nodes.values()
             if isinstance(node, Delay)]
        self._mul_node_names = \
            [name for name in self._nodes.keys()
             if isinstance(self._nodes[name], Multiply)]
```

Schließlich geschieht der eigentliche Aufbau des Graphen: Für jeden Knoten (d. h. für jeden Eintrag im Dictionary `_nodes`) wird die Methode `connect` aufgerufen, wobei aus dem Dictionary `_adjacency`, welches die Kanten definiert, die Liste der verbundenen Knoten erzeugt wird:

```
        for (name, node) in self._nodes.iteritems():
            input_nodes = [self._nodes[n]
                           for n in self._adjacency[name]]
            node.connect(input_nodes)
```

Nach dem Aufruf von `__init__` ist also eine Menge von Knoten gespeichert, zwischen denen die gewünschten Verbindungen hergestellt sind. Nun bietet die Klasse `Filter` noch einige Methoden, mit denen sich die Eigenschaften der einzelnen Knoten einstellen oder ausgeben lassen. Hier sind als Beispiele nur drei davon genannt, die am wichtigsten sind:

Die Methode `set_bits` setzt für jeden Knoten die Anzahl der Bits für die Darstellung der Ein- und Ausgangswerte.

```
    def set_bits(self, bits):
        for node in self._nodes.itervalues():
            node.set_bits(bits)
```

Mit der Methode `set_factor` kann der Faktor eines `Multiply`-Knotens, der durch seinen Namen identifiziert wird, eingestellt werden.

```
    def set_factor(self, name, factor, norm=False):
        mul_node = self._nodes[name]
        if not isinstance(mul_node, Multiply):
            raise TypeError("...")
        else:
            mul_node.set_factor(factor, norm)
```

Die eingestellten Faktoren aller Multiplizierer können mit der Methode `factors` ausgegeben werden. Dafür wird die gespeicherte Liste der Namen aller Multiplizierer verwendet. Das Objekt, das ausgegeben wird, ist wieder ein Dictionary, in dem unter dem Namen eines Multiplizerers dessen Faktor eingetragen ist. Der Faktor wird entweder als der ganzzahlige Zähler M (siehe Seite 104) oder als Fließkommazahl angegeben.

```
    def factors(self, norm=False):
        return dict((name, self._nodes[name].factor(norm))
                    for name in self._mul_node_names)
```

Als nächstes gibt es zwei Methoden, die die `Delay`-Knoten des Filters steuert: Die Methode `reset` setzt das Filter in den Anfangszustand zurück, in dem in allen Verzögerungselementen

sowie dem Eingabeknoten Nullen gespeichert sind.

```
    def reset(self):
        self._in_node.reset()
        for node in self._delay_nodes:
            node.reset()
```

Die Methode `_update` setzt das auf Seite 108 beschriebene Verfahren um, mit dem der Übergang zum nächsten Zeitschritt simuliert wird:

```
    def _update(self, ideal=False):
        for node in self._delay_nodes:
            node.sample(ideal)
        for node in self._delay_nodes:
            node.clk()
```

Um alle Vorgänge, die innerhalb eines Zeitschritts ablaufen, zu simulieren, gibt es die Methode `feed`. Darin wird

- einmal `_update` aufgerufen,
- ein neuer Wert am Eingang eingegeben und
- der Ausgangswert zurückgegeben.

```
    def feed(self, input_value, norm=False, ideal=False):
        self._update(ideal)
        if norm:
            input_value = (input_value *
                           (1 << self._in_node._bits-1))
        if not ideal:
            input_value = int(input_value)
        self._in_node.set_value(input_value, ideal)
        output_value = self._out_node.get_output(ideal)
        if norm:
            output_value = (float(output_value) /
                            (1 << self._out_node._bits-1))
        return output_value
```

Aus der Wortbreite des Eingangsknotens ergibt sich der erlaubte Bereich für den Wert `input_value`, den man mit `feed` übergibt. Möchte man aber beispielsweise das Verhalten des Filters untersuchen, wenn man die interne Wortbreite variiert, ist es sinnvoll, dafür gleichwertige Eingangsdaten zu verwenden. Das bedeutet, dass man die Eingangsdaten

mitskalieren muss, wenn man die Wortbreite ändert. Zum Beispiel entspricht der dezimale Wert „100“ bei 10 Bit ungefähr einem Fünftel des Maximalwerts, bei 11 Bit jedoch nur ungefähr einem Zehntel, weswegen man stattdessen eine „200“ eingeben muss.

Um die Skalierung automatisch durchzuführen, gibt es daher ähnlich wie beim Einstellen des Faktors von Multiplizierern den Parameter `norm`. Wenn dieser `True` ist, werden die Eingabewerte als Fließkommazahlen, die den relativen Wert in Bezug auf den Maximalwert angeben, interpretiert. Umgekehrt wird auch der Ausgabewert wieder in eine Fließkommazahl umgerechnet. In dem obigen Beispiel würde man also „0.1953125“ anstatt „100“ bzw. „200“ als `input_value` übergeben.

Schließlich gibt es noch eine weitere Methode `response`, die es dem Benutzer erleichtert, eine ganze Folge von Eingangswerten an das Filter zu übergeben und die Folge der Ausgangswerte zurückzubekommen, indem wiederholt `feed` aufgerufen wird. Dabei kann noch angegeben werden, wieviele Ausgangswerte ausgegeben werden sollen. Falls diese Zahl kleiner als die Länge der Liste mit den Eingangswerten ist, wird die Ausgabe einfach nach der gewünschten Anzahl Wiederholungen abgebrochen. Falls aber mehr Werte ausgegeben werden sollen, als am Eingang zur Verfügung stehen, wird `feed` mit einer Null als `input_value` aufgerufen, nachdem die eigentlich Eingangswerte „verbraucht“ sind. Dies ist mit einem *Generator*-Objekt [46] `gen_response` gelöst, das beim Aufruf von `response` erstellt wird.

```
    def response(self, data, length,
                 norm=False, ideal=False):
        self.reset()
        def gen_response():
            if length > len(data):
                for x in data:
                    yield self.feed(x, norm, ideal)
                for i in range(length-len(data)):
                    yield self.feed(0, norm, ideal)
            else:
                for i in range(length):
                    yield self.feed(data[i], norm, ideal)
        return [x for x in gen_response()]
```

In der letzten Zeile des folgenden Codeabschnitts wird über diesen Generator iteriert, um die Liste mit den Ausgangswerten zu erzeugen und zurückzugeben.

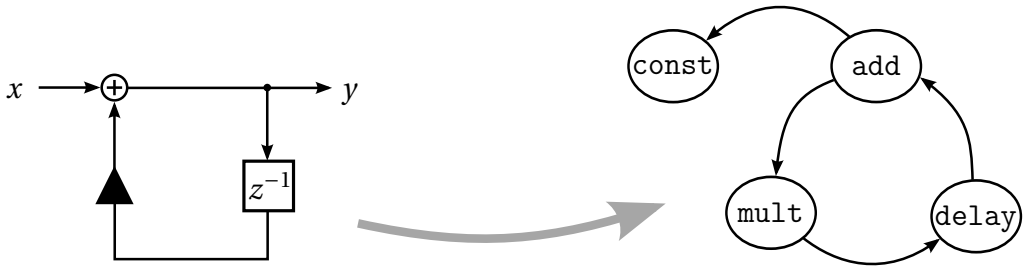

Abbildung 5.4: Filterstruktur aus dem Anwendungsbeispiel

5.2.5 Anwendungsbeispiel

Um die Funktionsweise des Moduls `iirsim_lib` zu verdeutlichen, ist nun die Bildschirmausgabe einer interaktiven Python-Sitzung[4] dargestellt, in der die Filterstruktur aus Abbildung 5.4 erzeugt und anschließend ihre Impulsantwort berechnet wird. Die Eingaben des Benutzers beginnen mit „>>>“.

Zunächst wird das Modul `iirsim_lib` geladen.

```
>>> import iirsim_lib
```

Nun können die notwendigen Filterknoten als Instanzen der von `_FilterNode` abgeleiteten Unterklassen erzeugt werden, wobei die Wortbreite auf 10 Bit und die Parameter m und s des Multiplizierers auf 8 bzw. 7 Bit gesetzt werden.

```
>>> const = iirsim_lib.Const(10)
>>> add = iirsim_lib.Add(10)
>>> mult = iirsim_lib.Multiply(10, 8, 7)
>>> delay = iirsim_lib.Delay(10)
```

Nun werden die beiden Dictionarys erzeugt, die für die Instanziierung von `Filter` benötigt werden. Im Dictionary `nodes` sind die Instanzen selbst enthalten, während in `adjacency` zu jedem Namen eines Knotens eine Liste mit den Namen der Knoten, zu denen eine Kante besteht, enthalten ist. Vom Knoten `const` geht keine Kante aus, daher ist unter seinem Namen nur eine leere Liste `[]` angegeben.

```
>>> nodes = {'Const': const, 'Add': add,
             'Mult': mult, 'Delay': delay}
>>> adjacency = {'Const': [], 'Add': ['Const', 'Mult'],
                 'Mult': ['Delay'], 'Delay': ['Add']}
```

[4] Man gelangt in eine interaktive Sitzung, indem man von einer Shell aus einfach das Programm `python` ohne weitere Angaben startet.

Jetzt kann ein `Filter`-Objekt erzeugt werden. Als Eingangs- und Ausgangsknoten sind `'Const'` bzw. `'Add'` angegeben.

```
>>> F = iirsim_lib.Filter(nodes, adjacency,
                          'Const', 'Add')
```

Um zu prüfen, ob das Verbinden der Knoten untereinander funktioniert hat, kann man sich z. B. das Attribut `_input_nodes` von `add` ausgeben lassen. Es sollte eine Liste mit einem `Const`- und einem `Multiply`-Objekt sein.

```
>>> add._input_nodes
[<iirsim_lib.Const instance at 0x7f89750eff38>,
 <iirsim_lib.Multiply instance at 0x7f897509e1b8>]
```

Der Faktor des Multipliziers wird nun auf einen Wert gestellt, der möglichst nahe bei 0.6 liegen soll. Da der Parameter $s = 7$ ist, wird der Faktor als ganzzahliges Vielfaches von $\frac{1}{128}$ gespeichert.

```
>>> F.set_factor('Mult', 0.6, norm=True)
```

Der tatsächlich gespeicherte Faktor ist jetzt $\frac{77}{128}$ und weicht etwas von 0.6 ab:

```
>>> F.factors()
{'Mult': 77}
>>> F.factors(norm=True)
{'Mult': 0.6015625}
```

Als Eingangsdaten wird eine Liste vorbereitet. Da die Impulsantwort gesucht ist, muss nur ein beliebiger Wert in der Liste enthalten sein.

```
>>> data_in = [100]
```

Trotzdem kann mithilfe der Methode `response` die Folge der Ausgangswerte in beliebiger Länge berechnet werden:

```
>>> F.response(data_in, 10)
[100, 60, 36, 21, 12, 7, 4, 2, 1, 0]
```

Die Ausgangswerte sind jetzt auf ganze Zahlen beschränkt, weil ja die Quantisierungseffekte simuliert werden sollen. Man kann im Vergleich dazu die Ausgangswerte betrachten, bei denen tatsächlich (im Rahmen der Genauigkeit des Rechners, auf dem das Programm ausgeführt wird) jeder Wert $\frac{77}{128}$ des Vorgängers beträgt.

```
>>> F.response(data_in, 10, ideal=True)
[100.0, 60.15625, 36.187744140625, 21.769189834594727,
```

```
13.09552825987339, 7.8777787188300863, 4.7389762605462238,
2.8507904067348377, 1.7149286040514258, 1.0316367383746858]
```

5.3 Einlesen der Filterstruktur

Die Schritte, die im vorherigen Beispiel zu dem Objekt `F`, einer Instanz der Klasse `Filter`, geführt haben, sind bei größeren Filterstrukturen mühsam durchzuführen und führen leicht zu Fehlern. Deshalb gibt es das Modul `iirsim_cfg`, mit dem aus der Beschreibung einer Filterstuktur durch eine Textdatei automatisiert ein `Filter`-Objekt erzeugt wird.

Die Beschreibung eines Filters erfolgt auf die Weise, dass es für jeden Knoten des zugehörigen Graphen eine Zeile gibt, in der durch Kommata getrennte Ausdrücke stehen:

```
<Ausdruck 1>, <Ausdruck 2>, <Ausdruck 3>
```

Jeder Ausdruck wiederum besteht aus mindestens einem durch Leerzeichen getrennten Zeichenketten (im einfachsten Fall Wörtern oder Zahlen). Falls eine der Zeichenketten auch ein Leerzeichen enthalten soll, muss es von Anführungszeichen umschlossen sein. Die erste der Zeichenketten eines Ausdrucks dient als Schlüsselwort, die restlichen als Parameter, die zu diesem Schlüsselwort gehören. Ein Ausdruck mit einem Schlüsselwort und zwei Parametern hätte z. B. die folgende Form:

```
<Schluesselwort> <Parameter 1> <Parameter 2>
```

In der folgenden Tabelle sind die gültigen Schlüsselwörter genannt. Jedes Schlüsselwort bezeichnet eine Eigenschaft eines Knotens und die zugehörigen Parameter legen den Wert fest, den diese Eigenschaft annimmt.

Schlüsselwort	Bedeutung	Parameter
`node`	Knotentyp	Name der Klasse
`name`	Name des Knotens	beliebiger eindeutiger Wert
`connect`	ausgehende Kanten	Namen anderer Knoten
`input`	Eingangsknoten	—
`output`	Ausgangsknoten	—
`factor`	Faktor eines Multiplizierers	Fließkommazahl

Die Schlüsselwörter `node` und `name` kommen für alle Knoten vor, die anderen nur bei manchen Knoten. `input` und `output` dürfen etwa nur jeweils einmal vorkommen, `factor` nur bei Multiplizierern.

Die Anzahl Bits für die Darstellung der Ein- und Ausgangswerte sowie die Parameter m und s der Multiplizierer werden nicht für jeden Knoten einzeln festgelegt, da diese Einstellungen im Normalfall für alle Knoten gleich sind.

Stattdessen sind dafür eigene Schlüsselwörter vorgesehen, die in einer Konfigurationsdatei jeweils genau einmal in einer eigenen Zeile vorkommen müssen:

- `bits_global`
- `factor_bits_global`
- `norm_bits_global`

Die Beschreibung des Filters aus dem vorherigen Beispiel in Abschnitt 5.2.5 sieht dann folgendermaßen aus:

```
bits_global 10
factor_bits_global 8
norm_bits_global 7

node Const, name "Const", input
node Multiply, name "Mult", connect "Delay", factor 0.6
node Delay, name "Delay", connect "Add"
node Add, name "Add", connect "Const" "Mult", output
```

Das Einlesen einer solchen Beschreibung erfolgt mithilfe des Moduls `shlex` (*Simple Lexical Analysis*) [47], das in der *Python Standard Library* enthalten ist. Es vereinfacht es, die in der Konfigurationsdatei enthaltenen Ausdrücke in Python-Datenstrukturen zu übertragen. Das Ziel dieser Übersetzung sind wieder die beiden Dictionarys, die beim Aufruf des Konstruktors von `Filter` benötigt werden.

Der Quelltext des Moduls `iirsim_cfg` ist hier nur skizziert. Den Hauptteil nimmt die Funktion `load_config` ein, die einen Dateinamen als Argument erhält und ein `Filter`-Objekt zurückgibt, das der Beschreibung in der angegebenen Datei entspricht.

Nachdem die benötigten Module `iirsim_lib` und `shlex` geladen sind, folgt in der Definition der Funktion `load_config` das Öffnen der Datei, in der die Konfiguration des Filters gespeichert ist. Die Datei wird zeilenweise gelesen und für jede Zeile wird ein Dictionary `cfg_item` angelegt, das am Schluss unter den in der Zeile vorkommenden Schlüsselwörtern die zugehörigen Parameter enthält. Diese Dictionaries werden in einer Liste `cfg_items` gesammelt.

```
import iirsim_lib
import shlex

```

```
def load_config(filename):
    f = open(filename)
    cfg_items = []
    for line in f:
        cfg_item = {}
```

Um für eine Zeile das Dictionary `cfg_item` mit Werten zu füllen, wird aus der Zeile mit einer Funktion aus `shlex` ein Objekt erzeugt, das es ermöglicht, nach einigen Zwischenschritten über die einzelnen Ausdrücke, die in der Zeile vorkommen, zu iterieren. Dabei werden dann die Schlüsselwörter und zugehörigen Parameter aufgesammelt und in `cfg_item` abgelegt. Schließlich wird die Liste `cfg_items` um den neuen Eintrag erweitert.

```
        shlex_object = shlex.shlex(line)
        ...          # Zwischenschritte
        for ... :    # aus shlex_object abgeleitete Iteration
            ...      # Zwischenschritte
            key = ...    # Schlüsselwort
            value = ... # Parameter
            cfg_item[key] = value
        cfg_items.append(cfg_item)
```

Würde man den Code bis hierher ausführen, wobei man die Beschreibung des Filters aus dem Anwendungsbeispiel verwenden würde, wäre der Inhalt von `cfg_items` folgender:

```
>>> cfg_items
[{'bits_global': ['10']},
 {'factor_bits_global': ['8']},
 {'norm_bits_global': ['7']},
 {'node': ['Const'], 'input': [], 'name': ['Const']},
 {'node': ['Multiply'], 'factor': ['0.6'],
  'name': ['Mult'], 'connect': ['Delay']},
 {'node': ['Delay'], 'name': ['Delay'],
  'connect': ['Add']},
 {'node': ['Add'], 'output': [], 'name': ['Add'],
  'connect': ['Const', 'Mult']}]
```

Stattdessen werden aber aus dieser Datenstruktur im nächsten Schritt die Objekte erzeugt, die für den Aufruf von `Filter` benötigt werden. Die beiden Dictionaries mit den `_FilterNode`-Instanzen und den Kanten des Graphen werden angelegt und gefüllt, während über die einzelnen `cfg_items` iteriert wird.

```
    filter_nodes = {}
```

```
    adjacency    = {}
    input_node   = None
    output_node  = None
    for cfg_item in cfg_items:
        [node] = cfg_item['node']
        [name] = cfg_item['name']
        if 'input' in cfg_item:
            input_node = name
        if 'output' in cfg_item:
            output_node = name
        ... # weitere Schritte
        if node == 'Const':
            filter_nodes['name'] = iirsim_lib.Const(bits)
        elif node == 'Add':
            filter_nodes['name'] = iirsim_lib.Add(bits)
        elif ... # etc.
```

Schließlich wird eine Instanz von `Filter` erzeugt und zurückgegeben:

```
    return iirsim_lib.Filter(filter_nodes, adjacency,
                             input_node, output_node)
```

Mit dieser Funktion vereinfacht sich das Anwendungsbeispiel von Seite 114. Angenommen, man hat die oben genannte Beschreibung der Filterstruktur (Seite 117) in einer Datei `filter.txt` gespeichert, dann genügen die folgenden Befehle, um an das `Filter`-Objekt `F` zu gelangen:

```
import iirsim_cfg
F = iirsim_cfg.read_config('filter.txt')
```

5.4 Grafische Benutzeroberfläche

Um die bisher beschriebene Funktion des Simulationsprogramms, also der Module `iirsim_lib` und `iirsim_cfg`, noch zugänglicher zu machen, gibt es eine grafische Benutzeroberfläche im Modul `iirsim_gui`. Sie ist besonders geeignet, um direkt zu beobachten, wie sich die Eigenschaften eines Filters ändern, wenn man beispielsweise die Koeffizienten variiert oder verschiedene Eingangssignale einspeist.

Das Modul `iirsim_gui` ist unter Verwendung von *PyQt* [48], das die Programmbibliothek *Qt* [49] für Python verfügbar macht, erstellt.

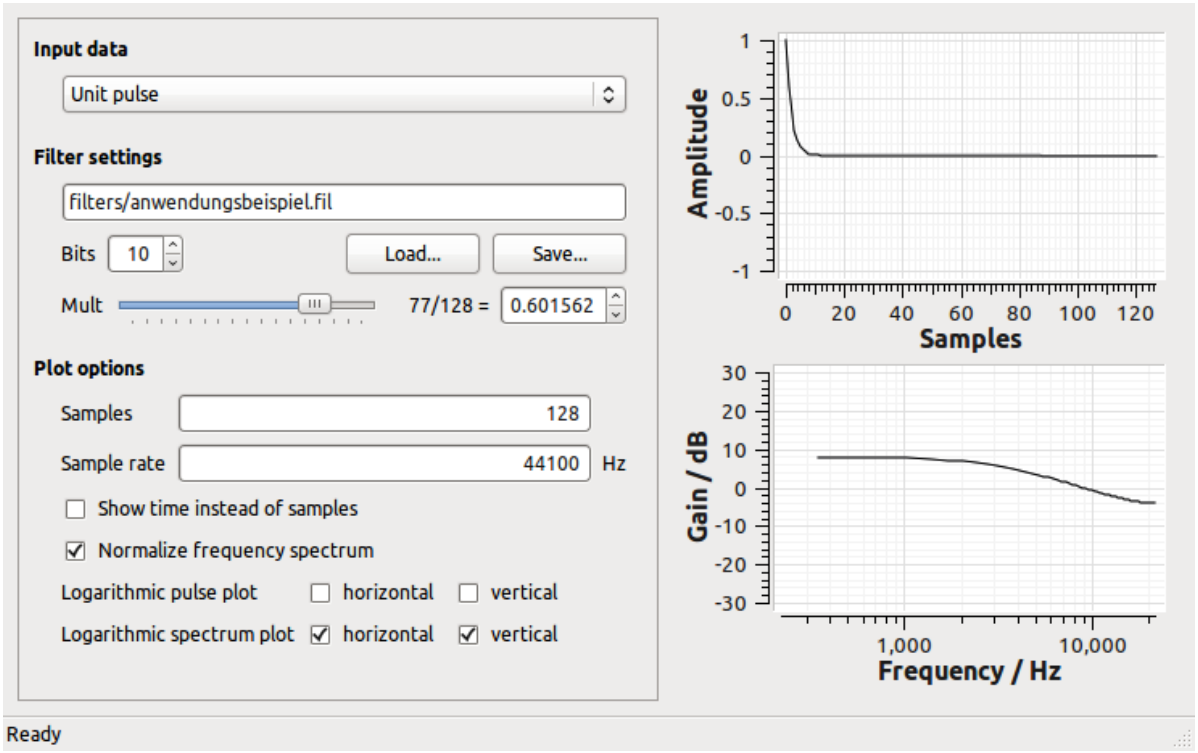

Abbildung 5.5: Programmfenster von `iirsim`

In Abbildung 5.5 ist das Programmfenster zu sehen. Auf der linken Seite sind die Bedienfelder für alle Einstellungen, die man machen kann. Auf der rechten Seite sind oben die Eingangs- und Ausgangsdaten des Filters und unten der Frequenzgang grafisch dargestellt.

Man kann eine Filter-Konfigurationsdatei angeben, die daraufhin mit der Funktion `load_config` aus `iirsim_cfg` eingelesen wird. Es gibt eine Eingabemöglichkeit für die Anzahl Bits für die Darstellung der Zahlenwerte und zu jedem der enthaltenen Multiplizierer einen Schieberegler, mit dem man den jeweiligen Faktor einstellen kann. Die Faktoren werden in ihrer Darstellung als Bruch und als Kommazahl angezeigt.

Zwei Beispiele für die Schieberegler sind in Abbildung 5.6 gezeigt. Im linken Bildausschnitt ist das Filter aus dem in den vorherigen Abschnitten verwendeten Beispiel, das nur einen Multiplizierer hat, geladen. Im rechten Teil ist ein größeres Filter mit sieben Multiplizierern geladen und es sind automatisch eine entsprechende Anzahl Schieberegler vorhanden, die mit den Namen der zugehörigen Multiplizierknoten beschriftet sind. Um dies zu erreichen, wird die Methode `factors` von `Filter` verwendet, nachdem die Filterstruktur eingelesen ist.

Als Eingangsdaten können entweder ein Einheitsimpuls oder benutzerdefinierte Daten, die aus einer Textdatei eingelesen werden, ausgewählt werden. Sobald die Filterstruktur,

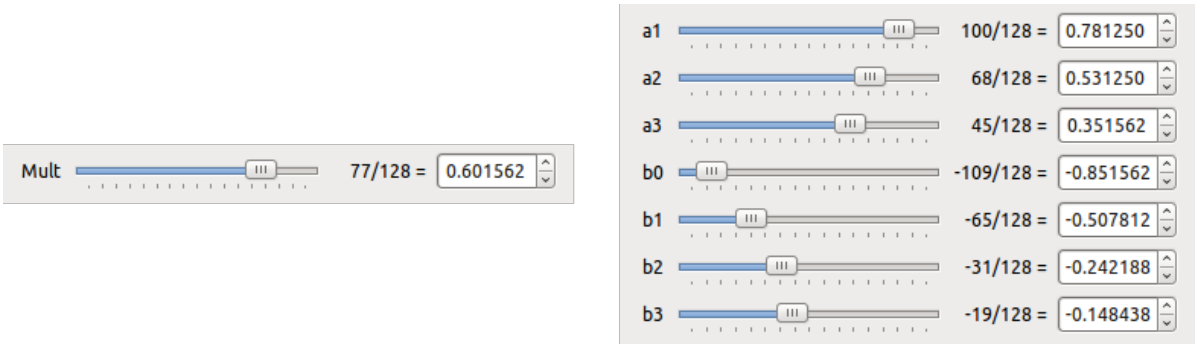

Abbildung 5.6: Schieberegler für die Faktoren verschiedener Filter

die Eingangsdaten oder die Einstellungen des geladenen Filters (Bits, Faktoren) geändert werden, wird automatisch die Folge der Ausgangswerte (mit der Methode `response` von `Filter`) neu berechnet und die grafische Anzeige im rechten Teil des Programmfensters auf den neuesten Stand gebracht. Somit lässt sich die Auswirkung einer Änderung direkt beobachten.

6 Messung des Quantisierungsfehlers

Eine wichtige Erkenntnis für die tatsächliche Umsetzung des Filters als Teil des SPADIC-Chips muss noch gewonnen werden: Wie groß muss die interne Wortbreite des Filters sein, damit die Rundungsfehler beim Berechnen der Filterausgangswerte nicht zu groß sind? Um dies herauszufinden, werden Messungen des Quantisierungsfehlers durchgeführt, die in diesem Kapitel beschrieben sind. Die Ergebnisse dieser Messungen dienen nicht nur dazu, eine Entscheidung über die Wortbreite der filterinternen Daten zu treffen, sondern auch dazu, bei gegebener Wortbreite in der Anwendung des Filters eine Einschätzung über die Fehler, mit denen die Ausgangsdaten behaftet sind, zu erhalten.

6.1 Vorgehensweise

Mithilfe der Filtersimulation `iirsim` wird ein Modell des in Abschnitt 4.5 beschriebenen Filters erstellt. Auf dieser Grundlage wird für eine Folge x von Eingangswerten des Filters die Ausgangsfolge y simuliert, die unter Berücksichtigung der Quantisierung entsteht. Außerdem wird die *ideale* Ausgangsfolge y_{id} berechnet, bei der die Simulation der Quantisierungseffekte ausgeschaltet ist. Unter dem *Quantisierungsfehler* Δy wird nun die Folge

$$\Delta y = y - y_{\mathrm{id}} \quad \Leftrightarrow \quad y = y_{\mathrm{id}} + \Delta y \tag{6.1}$$

verstanden, d. h. die Ausgangsfolge y mit Quantisierung kann so aufgefasst werden, dass sie durch Addition des Quantisierungsfehlers zur idealen Ausgangsfolge y_{id} entsteht. Als Einheit, in der die Größe Δy gemessen wird, dient der kleinstmögliche Abstand *(1 LSB)* zwischen zwei Werten der Eingangs- oder Ausgangsfolge des Filters, die jeweils in der 9-Bit Auflösung des SPADIC-ADCs dargestellt werden.

6.2 Qualitative Untersuchung

Zunächst ist interessant, wie sich ein realistisches, d. h. quantisiertes, Ausgangssignal y des Filters überhaupt von einem idealen Ausgangssignal y_{id} unterscheidet. Um dies rein qualitativ zu betrachten, wurde als Eingangsfolge x ein Signal erstellt, das aus Pulsen der

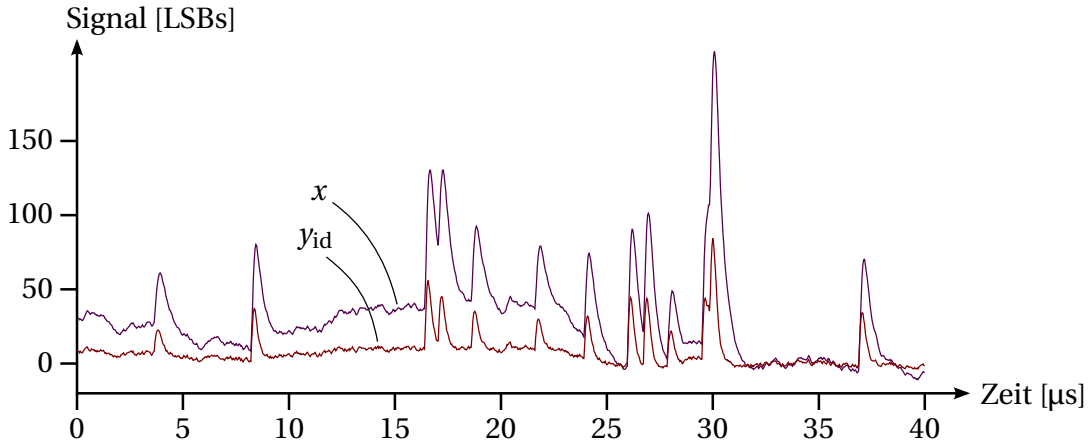

Abbildung 6.1: Künstlich erzeugtes Eingangssignal x und ideales Ausgangssignal y_{id} des Tail-Cancellation-Filters

Form 3.13 besteht, die einem Rauschsignal als *Baseline* überlagert wurden. In Abbildung 6.1 ist sind das Eingangssignal x und das Ausgangssignal y_{id} ohne Quantisierungsfehler gezeigt, wobei als Filterkoeffizienten die nach der in Abschnitt 3.3 erläuterten Methode gewählten Werte eingestellt wurden. Ein vergrößerter Ausschnitt dieser Grafik ist in Abbildung 6.2 gezeigt, wo auch die mit 10 bzw. 12 Bit berechneten Ausgangssignale enthalten sind. Auf den ersten Blick ist zu sehen, dass die quantisierten Signale nach unten verschoben sind, was daher kommt, dass die Zahlenwerte nach jeder Multiplikation *abgerundet* werden. Nach der Definition 6.1 bedeutet das, dass der Quantisierungsfehler *negativ* ist. Dies ist auch in Abbildung 6.3 zu sehen, in der der Quantisierungsfehler Δy bei 10 bzw. 12 Bit für den gleichen Ausschnitt wie in Abbildung 6.2 dargestellt ist. Man kann zwei Beobachtungen über den Unterschied des Quantisierungsfehlers für die beiden gewählten Filterauflösungen machen:

- Bei der kleineren Auflösung ist der Quantisierungsfehler weiter nach unten verschoben, d. h. sein Betrag ist im Mittel größer.
- Bei der kleineren Auflösung schwankt der Quantisierungsfehler stärker.

Zunächst könnte man meinen, es sei grundsätzlich schlechter, wenn der Quantisierungsfehler betragsmäßig größer ist. Aber: Wenn man *weiß*, dass der Quantisierungsfehler immer um einen gewissen *Mittelwert* μ herum schwankt, kann man dies nutzen, um aus dem quantisierten Filterausgangssignal y das ideale Ausgangssignal y_{id} zu rekonstruieren, wobei es egal ist, wie groß dieser Mittelwert ist. Das, was dabei als Unsicherheit bestehen bleibt, ist die *Schwankung* σ. Die Bedeutung von μ und σ ist ebenfalls in Abbildung 6.3 für den Fall des Quantisierungsfehlers bei 12 Bit Wortbreite illustriert. Bei bekannten Werten μ

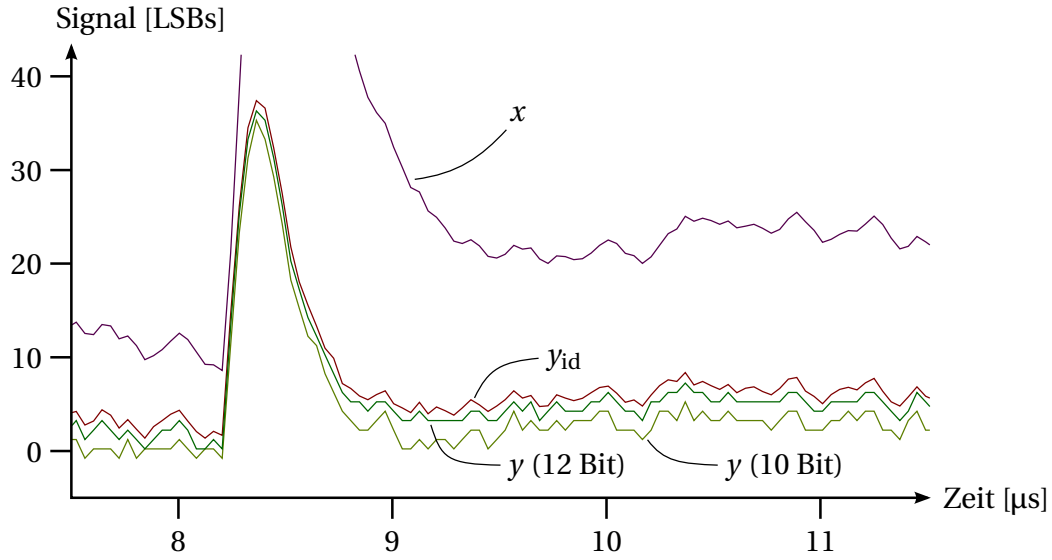

Abbildung 6.2: Detailansicht des idealen Ausgangssignals sowie des mit einer Wortbreite von 10 Bit bzw. 12 Bit berechneten

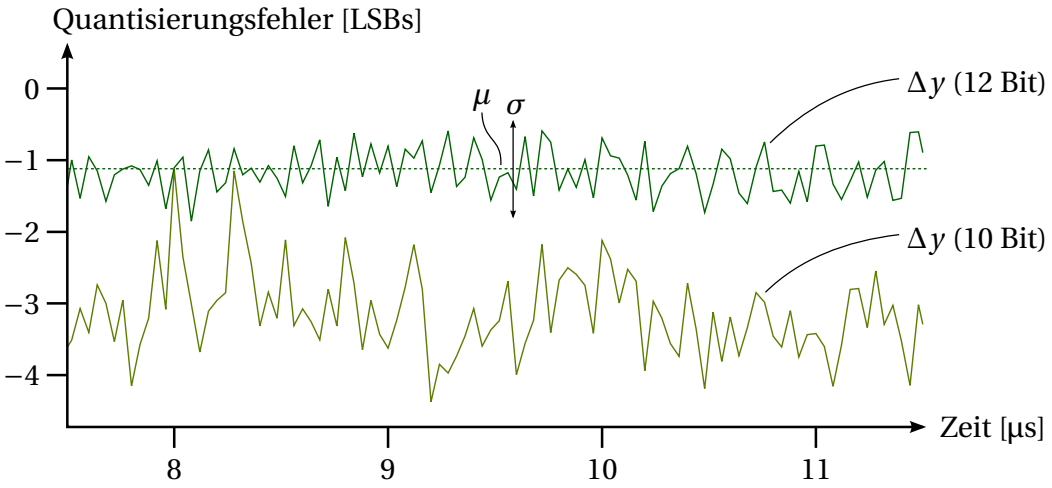

Abbildung 6.3: Quantisierungsfehler bei 10 Bit bzw. 12 Bit interner Wortbreite mit dem Mittelwert μ und der Schwankung σ

und σ kann man das ideale Ausgangssignal $y_{\text{id, est}}$ aus dem gemessenen Ausgangssignal y folgendermaßen *schätzen*:

$$y_{\text{id, est}} = (y - \mu) \pm \sigma \tag{6.2}$$

Die Schwankung σ des Quantisierungsfehlers sollte also so klein wie möglich sein, während der mittlere Fehler μ nicht entscheidend für die Genauigkeit der Rekonstruktion von y_{id} ist.

6.3 Statistische Auswertung

6.3.1 Momente

Hat man für ein gegebenes Eingangssignal x mit N bekannten Werten *(Samples)*, eine gegebene interne Wortbreite und einen bestimmten Satz von Filterkoeffizienten die Folge der Quantisierungsfehler Δy durch Simulation berechnet, kann man den Mittelwert und die Schwankung (bzw. deren Quadrat, die *Varianz*) daraus ermitteln:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} \Delta y[i] \tag{6.3a}$$

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (\Delta y[i] - \mu)^2 \tag{6.3b}$$

Der Mittelwert μ und die Varianz σ^2 sind dabei zwei der *Momente*, mit der die Verteilung der Quantisierungsfehler beschrieben werden kann [50]. Die Momente μ'_k einer Verteilung X sind:

$$\mu'_k = \frac{1}{N} \sum_{i=1}^{N} X[i]^k \tag{6.4a}$$

Der Mittelwert ist also das erste Moment $\mu'_1 = \mu$. Wird die Verteilung X um den Mittelwert zentriert, erhält man die *zentralen Momente* μ_k:

$$\mu_k = \frac{1}{N} \sum_{i=1}^{N} (X[i] - \mu)^k \tag{6.4b}$$

Die Varianz ist demnach das zweite zentrale Moment $\mu_2 = \sigma^2$.

Das dritte zentrale Moment beschreibt die *Schiefe* γ_1 einer Verteilung d. h. es weicht von

Null ab, wenn die Verteilung nicht symmetrisch um den Mittelwert ist:

$$\gamma_1 = \frac{\mu_3}{\sigma^3} = \frac{1}{\sigma^3} \cdot \frac{1}{N} \sum_{i=1}^{N} \big(X[i] - \mu\big)^3 \tag{6.5}$$

Vom vierten zentralen Moment ist die Größe „Kurtosis“ abgeleitet, die beschreibt, wieviele Werte der Verteilung bei konstanter Varianz weit entfernt vom Mittelwert liegen. Da die Normalverteilung eine Kurtosis von 3 hat, wird als *Exzess* die Differenz der Kurtosis zu 3 definiert, um eine Verteilung mit der Normalverteilung, die nach dieser Definition den Exzess Null hat, vergleichbar zu machen. Der Exzess γ_2 einer Verteilung X berechnet sich zu

$$\gamma_2 = \frac{\mu_4}{\sigma^4} - 3 = \frac{1}{\sigma^4} \cdot \frac{1}{N} \sum_{i=1}^{N} \big(X[i] - \mu\big)^4 - 3. \tag{6.6}$$

Für die Analyse der Verteilung der Quantisierungsfehler ist es hilfreich, zu wissen, was die Varianz und der Exzess einer Gleichverteilung sind. Sei f die normierte und zentrierte Gleichverteilung der Breite A, d. h. $f(x) = 1/A$, falls $-A/2 \le x \le A/2$, und $f(x) = 0$ sonst, dann gilt für ihre Varianz

$$\sigma^2_{\text{uni}} = \frac{1}{A} \int_{-A/2}^{A/2} x^2 \mathrm{d}x = \frac{1}{3A} \cdot \frac{A^3}{4} = \frac{A^2}{12} \quad \Rightarrow \quad \sigma_{\text{uni}} = \frac{A}{\sqrt{12}} \approx 0.289\, A \tag{6.7a}$$

sowie für ihren Exzess

$$\gamma_{2,\text{uni}} = \frac{1}{\sigma^4_{\text{uni}}} \cdot \frac{1}{A} \int_{-A/2}^{A/2} x^4 \mathrm{d}x - 3 = \frac{12^2}{A^5} \cdot \frac{1}{5} \cdot \frac{A^5}{16} - 3 = \frac{9}{5} - 3 = -\frac{6}{5}. \tag{6.7b}$$

6.3.2 Unendliche interne Auflösung

Bei einem gedachten Filter, in dem die Werte der Signalfolge mit unendlich hoher Wortbreite verarbeitet werden, entstehen bei der Berechnung der Ausgangswerte keine Rundungsfehler. Aber trotzdem werden die Daten nach dem Verlassen des Filters wieder durch Wörter der Breite 9 Bit repräsentiert. Das bedeutet, dass der bestmögliche Quantisierungsfehler derjenige ist, der durch einmaliges Abrunden des idealen Filterausgangswerts auf ein Vielfaches des LSB entsteht. Die so entstandenen Fehler wären also im besten Fall Null und im schlimmsten Fall (fast) −1 LSB. Da es keinen Grund gibt, warum die idealen Ausgangswerte einen bestimmten Wert zwischen zwei Quantisierungsstufen häufiger annehmen sollten als einen anderen, wäre der Quantisierungsfehler im Fall unendlicher interner Auflösung im Intervall $(-1, 0]$ gleichverteilt. Der Mittelwert müsste dann $-1/2$, die Varianz $1/12$, die Schiefe 0 und der Exzess $-6/5$ betragen.

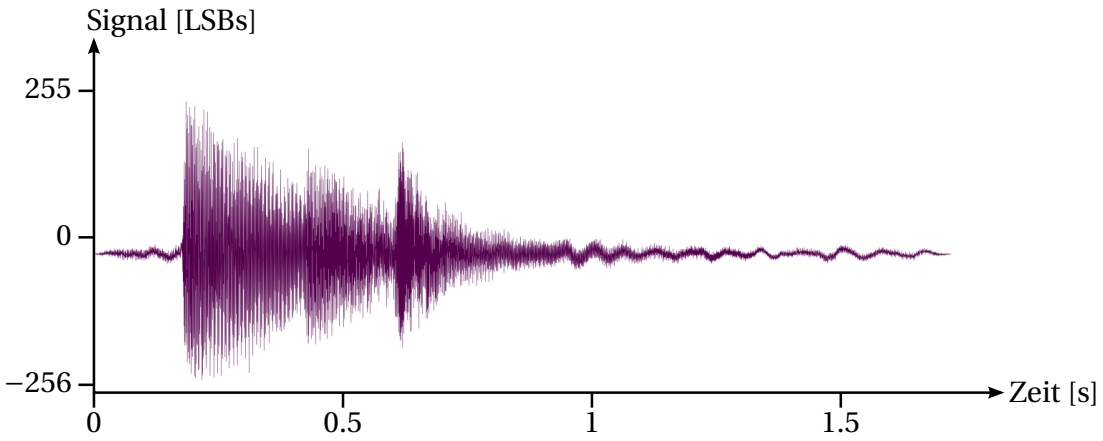

Abbildung 6.4: Als Eingangssignal für die Messung des Quantisierungsfehlers dient ein kurzer Ausschnitt eines mit 44.1 kHz abgetasteten Audiosignals.

6.3.3 Messung

Die Messung des Quantisierungsfehlers und die Berechnung der Momente seiner Verteilung erfolgte für verschiedene interne Wortbreiten des Filters. Um ein Signal mit vielen Werten ($N \gg 1000$) zu erhalten, ohne es künstlich erzeugen zu müssen, wurde ein kurzer Ausschnitt (ca. 1.7 Sekunden) eines mit 44.1 kHz abgetasteten und mit 9 Bit quantisierten Audiosignals verwendet, wodurch $N \approx 76\,000$ erreicht wird. Das Eingangssignal ist in Abbildung 6.4 zu sehen.

Um die eventuelle Abhängigkeit des Quantisierungsfehlers von den Filterkoeffizienten (b_1 bis b_4 und a_1 bis a_3; a_4 fällt durch die halbe erste Filterstufe weg) mit einzubeziehen, wurde die Messung für viele zufällig ausgewählte Sätze von Koeffizienten wiederholt (insgesamt 2807-mal), wobei immer die Bedingung

$$0 \leq -b_4 < a_3 < -b_3 < a_2 < -b_2 < a_1 < -b_1 < 1 \tag{6.8}$$

eingehalten wurde, d. h. die Koeffizienten mit dem größten Betrag werden in den ersten Filterstufen eingesetzt und für jedes Paar (a_i, b_i) gilt die Voraussetzung 3.24 für sinnvolle Koeffizientenwahl, $a_i < -b_i$.

In den folgenden Abbildungen 6.5, 6.6, 6.7 und 6.8 sind die Ergebnisse dieser Messreihe grafisch dargestellt. Es ist für jede der untersuchten Größen (Mittelwert μ, Schwankung σ, Schiefe γ_1, Exzess γ_2) für jede Messung mit einem Satz von Koeffizienten eine feine Linie gezeichnet, die den Wert der jeweiligen Größe für verschiedene Wortbreiten repräsentiert. Durch Überlagerung von jeweils 2807 dieser Linien wird einerseits die Abhängigkeit des Quantisierungsfehlers von der Auflösung sichtbar und andererseits die Stärke, mit der die

Verteilung des Quantisierungsfehlers ihrerseits schwankt, wenn verschiedene Koeffizienten eingestellt werden.

Anhand dieser Messungen kann man feststellen:

- Für filterinterne Wortbreiten bis zu 13 Bit hängen die Eigenschaften des Quantisierungsfehlers auch von den gewählten Filterkoeffizienten ab. Je größer die Auflösung ist, desto einheitlicher ist das Verhalten des Quantisierungsfehlers.
- Für größer werdende interne Wortbreiten nähern sich die untersuchten Eigenschaften des Quantisierungsfehlers (Mittelwert, Schwankung, Schiefe, Exzess) denen einer Gleichverteilung im Intervall $(-1,0]$ an, was bei einem Filter, das mit unendlich hoher Wortbreite arbeitet, zu erwarten ist. Ab einer Auflösung von 13–14 Bit ist der Wert der für die Rekonstruktion des idealen Ausgangssignals wichtigen Größe σ nicht mehr weit vom minimalen Wert der Gleichverteilung entfernt. Bei 16 Bit scheint schon keine Verbesserung mehr möglich zu sein.

Damit das Tail-Cancellation-Filter also mit einer nahezu optimalen Genauigkeit funktioniert, sollte die interne Wortbreite mindestens 14 Bit betragen. Mehr als 16 Bit sind nach den Ergebnissen dieser Untersuchung nicht notwendig.

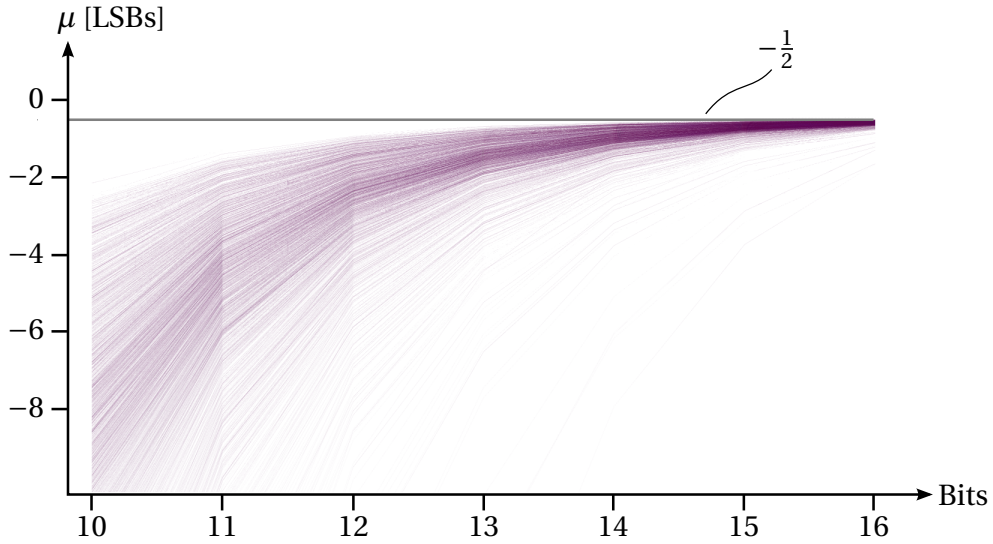

Abbildung 6.5: Mittelwert μ des Quantisierungsfehlers Δy für verschiedene Sätze von Koeffizienten in Abhängigkeit der internen Wortbreite

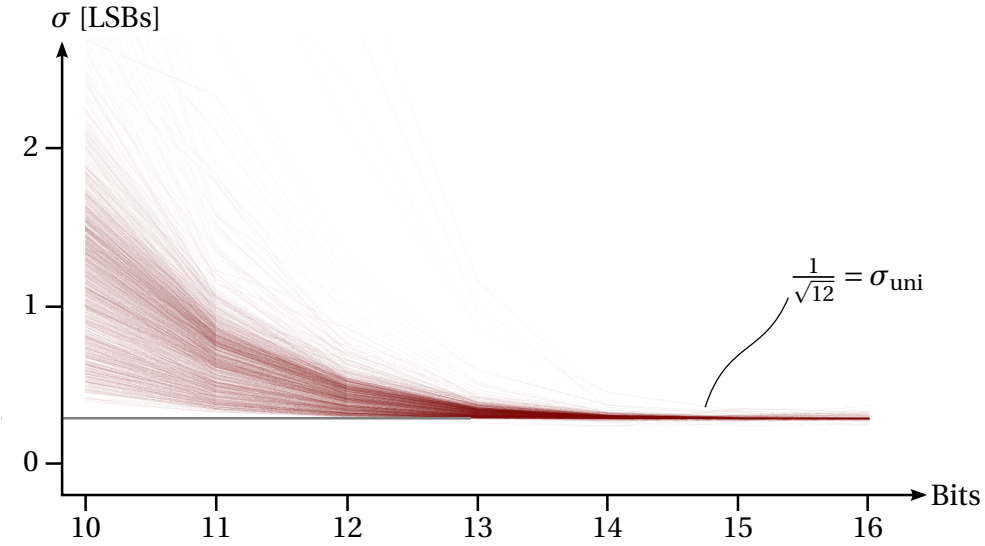

Abbildung 6.6: Schwankung σ des Quantisierungsfehlers Δy für verschiedene Sätze von Koeffizienten in Abhängigkeit der internen Wortbreite

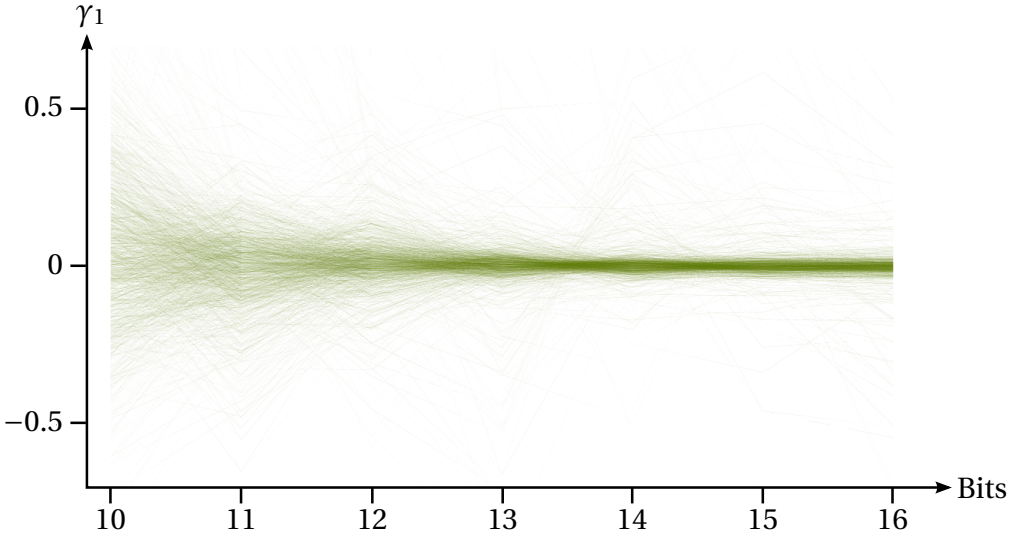

Abbildung 6.7: Schiefe γ_1 der Verteilung des Quantisierungsfehlers Δy für verschiedene Sätze von Koeffizienten in Abhängigkeit der internen Wortbreite

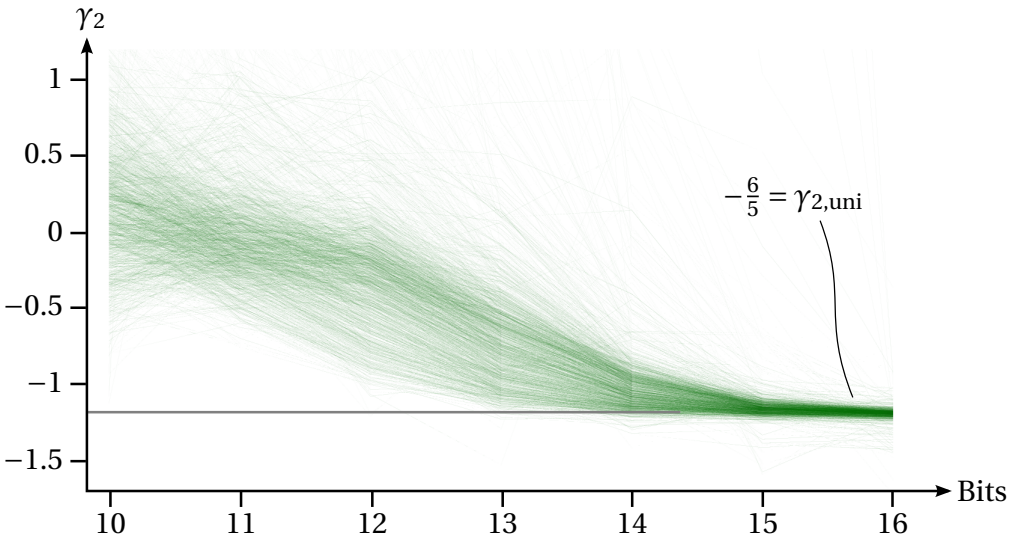

Abbildung 6.8: Exzess γ_2 der Verteilung des Quantisierungsfehlers Δy für verschiedene Sätze von Koeffizienten in Abhängigkeit der internen Wortbreite

7 Schluss

Mithilfe der Beschreibung der Signale durch die Pulsfolgen wurde gezeigt, dass mit einer Reihenschaltung von Filterstufen erster Ordnung der *Ion Tail* ausgelöscht werden kann. Dabei wird die Stabilität des Filters durch die Einschränkung der Koeffizienten auf einen sinnvollen Wertebereich garantiert. Es steht eine Simulationssoftware zur Verfügung, die es erlaubt, die Auswirkung des Filters auf ein gegebenes Eingangssignal zu beobachten. Dabei werden die Quantisierungseffekte berücksichtigt, so dass sich Schlüsse über die Genauigkeit der Signalverarbeitung im Filter ziehen lassen.

Die erarbeitete Filterstruktur liegt als Verilog-Beschreibung vor, so dass sie in den Digitalteil des SPADIC-Chips integriert werden kann. Durch Parametrisierung lassen sich die Anzahl der Filterstufen und die Wortbreite, sowohl für die interne Repräsentation der Signalfolge, als auch für die Filterkoeffizienten, an neue Gegebenheiten anpassen. Durch Vergleich mit einem der Literatur entnommenen Verfahren zum Entwurf von Multipliziererschaltungen wurde gezeigt, dass die Leistungsfähigkeit der am Lehrstuhl verwendeten Synthesewerkzeuge diesbezüglich mehr als ausreichend gut ist.

Es bleibt abzuwarten, wie nützlich der neue Teil des SPADIC-Systems in der praktischen Anwendung sein wird, und ob die vorgeschlagenen Methoden zur *Tail Cancellation* sich so einsetzen lassen, wie dies hier angedacht wurde. Mögliche Verbesserungen des Filters könnten die Multiplizierschaltung oder die Filterstruktur im Ganzen betreffen, die die Geschwindigkeit oder die Genauigkeit im Verhältnis zum schaltungstechnischen Aufwand günstig beeinflussen.

Literaturverzeichnis

[1] GSI Helmholtzzentrum für Schwerionenforschung: *The Compressed Baryonic Matter Experiment – Introduction.* Website. http://www.gsi.de/forschung/fair_experiments/CBM/1intro_e.html, besucht: 30.08.2011.

[2] Cyrano Bergmann: *Development and Test of a Transition Radiation Detector Prototype for CBM at FAIR.* Diplomarbeit, Westfälische Wilhelms-Universität Münster, 2009.

[3] Tim Armbruster: *The SPADIC Project Website.* Website. http://spadic.uni-hd.de.

[4] David Emschermann: *CBM TRD in Münster.* Website. http://cbm.uni-muenster.de/daq/, besucht: 29.08.2011.

[5] Alan V. Oppenheim und Ronald W. Schafer: *Zeitdiskrete Signalverarbeitung.* R. Oldenbourg Verlag, 3. Auflage, 1999, ISBN 3-486-24145-1.

[6] Norbert Fliege und Markus Gaida: *Signale und Systeme.* J. Schlembach Fachverlag, 2008, ISBN 978-3-935340-42-7.

[7] Karl Dirk Kammeyer und Kristian Kroschel: *Digitale Signalverarbeitung.* B. G. Teubner Verlag, 5. Auflage, 2002, ISBN 3-519-46122-6.

[8] Alexander Babenko: *Komplexe Analysis,* 2006. http://www.math.tu-berlin.de/~bobenko/Lehre/Skripte/FT.pdf.

[9] Eberhard Freitag: *Vorlesungen über Analysis.* http://www.rzuser.uni-heidelberg.de/~t91/skripten/analysis/a1.pdf.

[10] V. E. Hoggatt, Jr.: *Fibonacci Numbers and Generalized Binomial Coefficients.* The Fibonaccy Quarterly, 5(4):383–400, 1967. http://www.fq.math.ca/Scanned/5-4/hoggatt.pdf.

[11] Fabio Sauli: *Principles of Operation of Multiwire Proportional and Drift Chambers.* CERN Academic Training Lecture. CERN, 1977. http://cdsweb.cern.ch/record/117989/files/CERN-77-09.pdf.

[12] C. Bergmann, M. Klein-Bösing, D. Emschermann, J. P. Wessels, M. Petris, V. Simion, M. Petrovici und C. Höhne: *Development and Test of a Real-Size Prototype for the CBM TRD*. CBM Progress Report, 2009.

[13] Patrick Reichelt: *Simulationsstudien zur Entwicklung des Übergangsstrahlungszählers für das CBM-Experiment*. Diplomarbeit, Institut für Kernphysik Frankfurt, 2011.

[14] S. Chernenko, G. Cheremukhina, S. Chepurnov, S. Razin, Yu. Zanevsky und V. Zryuev: *Research and Development of Fast Position Sensitive Gas Detectors for CBM*. CBM Progress Report, 2009.

[15] Tim Armbruster: *SPADIC for RICH*, 2011. http://spadic.uni-hd.de/publications/talks/2011/04_2.pdf.

[16] Tim Armbruster: *Ion Tail Cancellation Filter – Small Introduction*, 2010. http://spadic.uni-hd.de/publications/talks/2010/06.pdf.

[17] Tim Armbruster: *Design Aspects of typical Preamplifier-Shaper Topologies in Detector Readout Systems*, 2008. http://spadic.uni-hd.de/publications/talks/2008/09.pdf.

[18] B. Mota, L. Musa, R. Esteve und A. Jimenez de Parga: *Digital Implementation of a Tail Cancellation Filter for the Time Projection Chamber of the ALICE Experiment*. 6th Workshop on Electronics for LHC Experiments, Seiten 164–168, 2000.

[19] Elke Svenja Wulff: *Position Resolution and Zero Suppression of the ALICE TRD*. Diplomarbeit, Westfälische Wilhelms-Universität Münster, 2009.

[20] Andrew D. Booth: *A Signed Binary Multiplication Technique*, 1950.

[21] C. S. Wallace: *A Suggestion for a Fast Multiplier*. IEEE Trans. Electron. Comput., 13:14–17, 1964.

[22] Luigi Dadda: *Some Schemes for Parallel Multipliers*. Colloque sur l'Algèbre de Boole, Grenoble, 1965.

[23] Charles R. Baugh und Bruce A. Wooley: *A Two's Complement Parallel Array Multiplication Algorithm*. IEEE Transactions on Computers, C-22(12), 1973.

[24] Luigi Dadda: *On Parallel Digital Multipliers*. Alta Freq., 45:574–580, 1976.

[25] David Villeger und Vojin G. Oklobdzija: *Evaluation of Booth Encoding Techniques for Parallel Multiplier Implementation*. Electronic Letters, 29(23), 1993.

[26] Vojin G. Oklobdzija, David Villeger und Simon S. Liu: *A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach.* IEEE Transactions on Computers, 45, 1996.

[27] Paul F. Stelling und Vojin G. Oklobdzija: *Design Strategies for Optimal Hybrid Final Adders in a Parallel Multiplier.* Journal of VLSI Signal Processing, 14:321–331, 1996.

[28] Paul F. Stelling und Vojin G. Oklobdzija: *Design Strategies for the Final Adder in a Parallel Multiplier*, 1996.

[29] Paul F. Stelling und Vojin G. Oklobdzija: *Optimal Designs for Multipliers and Multiply-Accumulators*, 1996.

[30] Pascal Bonatto und Vojin G. Oklobdzija: *Evaluation of Booth's Algorithm for Implementation in Parallel Multipliers*, 1996.

[31] Paul F. Stelling und Vojin G. Oklobdzija: *Implementing Multiply-Accumulate Operation in Multiplication Time*, 1997.

[32] David Villeger und Vojin G. Oklobdzija: *Analysis of Booth Encoding Efficiency in Parallel Multipliers Using Compressors for Reduction of Partial Products*, 1993.

[33] Paul F. Stelling, Charles U. Martel, Vojin G. Oklobdzija und R. Ravi: *Optimal Circuits for Parallel Multipliers.* IEEE Transactions on Computers, 47(3), 1998.

[34] Wen Chang Yeh und Chein Wei Jen: *High-Speed Booth Encoded Parallel Multiplier Design.* IEEE Transactions on Computers, 49(7), 2000.

[35] Fayez Elguibaly: *A Fast Parallel Multiplier-Accumulator Using the Modified Booth Algorithm.* IEEE Transactions on Circuits and Systems, 47(9), 2000.

[36] Whitney J. Townsend, Earl E. Swartzlander, Jr. und Jacob A. Abraham: *A Comparison of Dadda and Wallace Multiplier Delays*, 2003.

[37] Patrik Kimfors, Niklas Broman, Andreas Haraldsson, Kasyab P. Subramaniyan, Magnus Själander, Henrik Eriksson und Per Larsson-Edefors: *Custom Layout Strategy for Rectangle-Shaped Log-Depth Multiplier Reduction Tree*, 2006.

[38] Henrik Eriksson, P. Larsson-Edefors, M. Sheeran, M. Själander, D. Johansson und M. Schölin: *Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity*, 2006.

[39] Magnus Själander und Per Larsson-Edefors: *The Case for HPM-Based Baugh-Wooley Multipliers*, 2008.

[40] Dursun Baran, Mustafa Aktan und Vojin G. Oklobdzija: *Energy Efficient Implementation of Parallel CMOS Multipliers with Improved Compressors*, 2010.

[41] *Python Programming Language – Official Website*. http://www.python.org/.

[42] *Cadence Encounter RTL Compiler*. http://www.cadence.com/products/ld/rtl_compiler/pages/default.aspx, besucht: 27.08.2011.

[43] Peter Fischer: *Digitale Schaltungstechnik*. Vorlesung, 2010. http://sus/Lehre/DSTVorlesung1011.

[44] Guido van Rossum und Barry Warsaw: *Style Guide for Python Code*. http://www.python.org/dev/peps/pep-0008/, besucht: 12.07.2011.

[45] Guido van Rossum: *What's New in Python 3.0*. http://docs.python.org/release/3.0.1/whatsnew/3.0.html, besucht: 12.07.2011.

[46] *Python Tutorial – Generators*. http://docs.python.org/tutorial/classes.html#generators, besucht: 19.07.2011.

[47] *Python Documentation – shlex*. http://docs.python.org/library/shlex.html, besucht: 19.07.2011.

[48] *Riverbank Computing – PyQt*. http://www.riverbankcomputing.co.uk/software/pyqt/intro, besucht: 20.07.2011.

[49] *Nokia – Qt*. http://qt.nokia.com/products, besucht: 20.07.2011.

[50] Fred Hamprecht: *Introduction to Statistics*. Vorlesung, 2005. http://hci.iwr.uni-heidelberg.de/MIP/Teaching/stat/lec-2up.pdf.