Verfahren zur direkten Implementierung von Algorithmen auf Gatterebene

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.)

von Dipl.-Ing. Thomas Reinemann

geb. am 8. Juni 1965 in Magdeburg

genehmigt durch die Fakultät für Maschinenbau

der Otto-von-Guericke-Universität Magdeburg

Gutachter:

Prof. Dr.-Ing. Roland Kasper

Prof. Dr.-Ing. Rolf Ernst

Promotionskolloquium am 30. Oktober 2003

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Mechatronik und Antriebstechnik der Otto-von-Guericke-Universität Magdeburg.

Mein besonderer Dank gilt dem Leiter des Instituts, Herrn Prof. Dr.-Ing. Roland Kasper. Er gab mir Gelegenheit, diese Arbeit am Lehrstuhl für Mechatronik durchzuführen und hat sie stets durch anregende Diskussionen und wertvolle Ratschläge begleitet.

Ich danke dem Gutachter der vorliegenden Dissertation, Herrn Professor Dr.-Ing. Rolf Ernst, für sein Engagement und seinen Einsatz, insbesondere hinsichtlich der Betrachtungen aus Sicht eines Experten für den Entwurf digitaler Systeme.

Weiterhin bedanke ich mich bei meiner Lebensgefährtin und meinen Kindern für die Unterstützung und das aufgebrachte Verständnis.

Zusammenfassung

Die digitale Signalverarbeitung dominiert immer mehr die Produkteigenschaften. Gewöhnlich wird diese als Software in Prozessoren implementiert. Steigende Anforderungen werden meist durch steigende Taktfrequenzen, verbunden mit einem höheren Energieverbrauch, befriedigt. Eine Alternative stellt die Informationsverarbeitung in PLDs und ASICs dar. Da es hier keine exklusiven Betriebsmittel wie eine Arithmetik-Logik-Einheit gibt, kann die Verarbeitung parallel erfolgen. Gegenwärtig ist die breite Anwendung jedoch noch problematisch, da es keine einheitlichen Bibliotheken und Entwicklungswerkzeuge für den fachfremden Ingenieur gibt.

Die Verwendung eines Blockdiagrammeditors vereinfacht die Spezifikation des Algorithmus, jedoch fehlt immer noch eine Bibliothek mit vorgefertigten Operatoren. In dieser Arbeit wurde eine solche Bibliothek auf bitserieller Basis entwickelt. Der bitserielle Ansatz erlaubt die Implementierung einfacher und schneller Operatoren, die sich leicht auf eine beliebige Wortbreite skalieren lassen. Durch die Übertragung der Operanden im Zeitmultiplex entfällt die implizite Kennzeichnung der Wertigkeit der Bits durch die Signalleitung. Durch Einführung eines Synchronisationsbusses konnte eine alternative Kennzeichnung geschaffen werden. Aufgrund der parallelen Informationsverarbeitung durchlaufen die Operanden unterschiedliche Signalketten und treffen u.U. mit unterschiedlichen Verzögerungen an einem Operator zusammen. Diese Verzögerungen sind auszugleichen, jedoch müssen sie dazu dem Operator bekannt sein. Dazu wurde ein automatisches Synchronisationsverfahren implementiert, welches dem Anwender die aufwändige und fehleranfällige Spezifikation der Operandenverzgögerungen abnimmt.

Zur Vervollständigung wurden die Bibliothek und das Synchronisationsverfahren in einen Blockdiagrammeditor eingebunden, so dass eine einfache Anwendung möglich ist.

Summary

Today, digital signal processing more and more dominates product properties. Usually it is implemented in processors in kind of software. Increasing performance demands are almost satisfied by increasing clock rates, linked with higher energy consumption. Information processing in PLDs and ASICs represents an alternative. Since they have no exclusive resources, as e.g. an arithmetic-logic-unit, they process information in a parallel manner. But presently, a broadly application is handicapped by missing of common libraries and development tools for non-electrical engineers.

The usage of a block diagram editor simplifies the specification, but a library containing prebuild operators is still missing. In this work such a library basing on bit serial arithmetic has been developed. The bit serial approach enables the implementation of simple and fast operators, which are well scaleable in steps of one bit. The implicit marking of operand bits by the transmission line is not possible by the transmission in time division. This can be compensated by introduction of a synchronization bus.

Formelverzeichnis

Δ	Anstieg innerhalb eines Intervalls
ΔT	Tastperiode
Δy	Differenz zwischen zwei Stützstellen
a_i, b_i	Koeffizienten der Übertragungsfunktion bzw. Differenzengleichung
$\overline{a}_{ij}$, $\overline{b}_{ij}$, $\overline{c}_i$	Matrix- bzw. Vektorelemente im ZRM
$\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, d	Matrix, Vektoren und Durchgriff im ZRM
$\hat{\mathbf{A}}$, $\hat{\mathbf{b}}$, $\hat{\mathbf{c}}$, $\hat{d}$	transformierte Größe des ZRM
$\mathbf{C_0}$	Matrix zur Abbildung der Ausgangssignalverzögerungen auf die der Komponentenausgänge
$\mathbf{C_u}$	Matrix zur Abbildung der Eingangssignalverzögerungen auf die Ausgangssignalverzögerungen
$\mathbf{C_v}$	Matrix zur Abbildung der variablen Verzögerungen auf die Ausgangssignalverzögerungen
d_x	allgemeine Signalverzögerungen
$\mathbf{D}$	Matrix zur Abbildung des Durchgriffs bei Signalverzögerungen
i, j, k	allgemeine Laufvariable
K_x, K_D, K_I, K_P	allgemeiner, D-, I- und P-Verstärkungsfaktor
m	Systemordnung
n	Verarbeitungswortbreite
$\mathbf{P}$	Matrix zur Abbildung der Ausgangsverzögerungen auf die Eingangsverzögerungen
$\mathbf{P_0}$	Matrix zur Abbildung der Komponenteneingangsverzögerungen auf die Eingangsverzögerungen
r_x	Länge der Register zum Verzögerungsausgleich
S_u, $\mathbf{S}_x$, S_y	Skalierungsfaktoren
T_N	Nachstellzeit des I-Anteils
T_V	Vorstellzeit des D-Anteils
u, y	Eingangs- bzw. Ausgangssignal eines Systems
$\hat{u}$, $\hat{y}$	skalierte Eingangs- bzw. Ausgangssignale eines Systems
$\mathbf{u}_0$, $\mathbf{y}_0$	Eingangs- bzw. Ausgangssignalverzögerung einer Komponente
$\mathbf{v}$, $\mathbf{v_k}$	variable bzw. konstante Verzögerungen einer verarbeitenden Komponente
$\mathbf{x}$	Zustandsvektor im ZRM
u_{max}, u_{min}, x_{max}, x_{min}, y_{max}, y_{min}	gemessene Grenzwerte
$\hat{u}_{max}$, $\hat{u}_{min}$, $\hat{x}_{max}$, $\hat{x}_{min}$, $\hat{y}_{max}$, $\hat{y}_{min}$	Vorgaben für den Wertebereich der skalierten Größen

Inhaltsverzeichnis

Kapitel 1

Einleitung / Motivation

1.1 Signalverarbeitung heute / bestehende Probleme

Die Anforderungen an moderne Elektronikkomponenten und die daraus aufgebauten informationsverarbeitenden Komponenten für mechatronische Systeme nehmen permanent zu. Dies bezieht sich gleichermaßen auf die Leistungsfähigkeit ihrer Prozessschnittstellen, die Komplexität und Verarbeitungsgeschwindigkeit der realisierten Steuer-, Regler- und Signalverarbeitungsfunktionen sowie die Flexibilität einer integrierten Netz- oder Systemschnittstelle. Die Ursache liegt darin, dass bei vielen technisch hochwertigen Produkten eine Verdrängung mechanischer, elektrischer oder fluidischer Lösungen durch elektronische/informationsverarbeitende Systeme erfolgt. Das bedeutet umgekehrt, dass viele mechanische, elektrische oder fluidische Funktionen, die von ihren physikalischen Phänomenen meistens als zeit- und amplitudenkontinuierliche Größen zu beschreiben sind, in informationsverarbeitenden Komponenten möglichst wirklichkeitsgetreu abgebildet werden. Insbesondere an den Schnittstellen zwischen mechanischen und informationsverarbeitenden Systemteilen, realisiert durch Sensoren und Aktuatoren, treten Probleme der Zeitdiskretisierung und der Amplitudenquantisierung bei der Konversion physikalischer in informationstechnische Größen und umgekehrt häufig auf. Angekoppelt an die Schnittstellen zu realen physikalischen Signalen entfallen wesentliche Teile der Informationsverarbeitung mechatronischer Systeme auf die Verarbeitung dieser gewandelten und daraus abgeleiteten Größen. Typischerweise werden hierzu Steuer-, Regler- und Signalverarbeitungsfunktionen ausgeführt. Die Implementierung dieser Funktionen unter den speziellen Randbedingungen, wie Quantisierungs- oder Diskretisierungsfehlern bzw. dem exakten Einhalten von Echtzeitbedingungen, gehört zu den Grundaufgaben der Informationsverarbeitung mechatronischer Systeme. Hinzu kommen weitere Randbedingungen wie die Äquidistanz der Abtastung oder das Abtasttheorem von Shannon [Föl93]. Letzteres besagt, dass ein Signal mindestens mit dem Doppelten der höchsten auftretenden Frequenz abzutasten ist und ist somit der Gradmesser für die Einhaltung der Echtzeitbedingungen. Diese resultieren ausschließlich daraus.

Jedoch bereitet gerade deren Einhaltung bei hochdynamischen Signalen Probleme und zwingt zur ständigen Erhöhung der Rechenleistung.

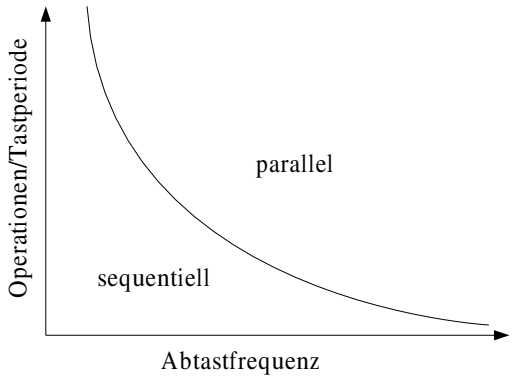

Abbildung 1.1: Abhängig zwischen Abtastfrequenz und der Anzahl der erreichbaren Operationen [And03a]

Grundsätzlich besteht die Möglichkeit, Informationen sequentiell oder parallel zu verarbeiten. Ein Prozessor macht dies sequentiell, da nur eine begrenzte Anzahl von Verarbeitungselementen zur Verfügung steht. Somit ist die Anzahl der pro Tastperiode durchführbaren Operationen immer nach oben begrenzt und nimmt mit steigender Abtastfrequenz ab. Bild 1.1 stellt dies qualitativ dar. In parallel arbeitenden Technologien, wie PLDs und ASICs, können jedoch theoretisch beliebig viele Verarbeitungselemente implementiert werden. Daher ist dort die Anzahl der pro Tastperiode durchführbaren Operationen unbegrenzt.

Gängige PC-Prozessoren (Athlon, Pentium) verfügen intern zwar auch über mehrere Verarbeitungseinheiten (z.B. parallel arbeitende Addierer), jedoch werden diese im wesentlichen zur spekulativen Berechnung genutzt und die Resultate zum größten Teil verworfen. Dadurch sinkt die Effizienz bezüglich Energie und Chipfläche drastisch (siehe dazu auch Abschnitt 1.2).

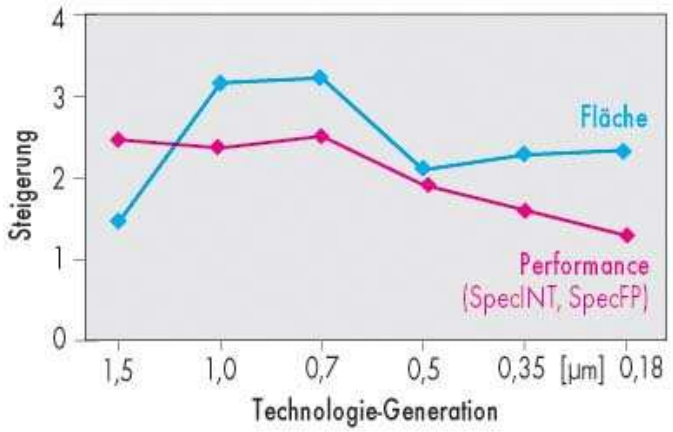

Abbildung 1.2: Vergleich Chipfläche-Performance [Sti01]

Damit stellt sich die Frage, welche Möglichkeiten bei der Signalverarbeitung zur Erhöhung der

Rechenleistung bei hoher Effizienz bezüglich Energie und Chipfläche bestehen.

1.2 Auswege / Lösungsmöglichkeiten

In einer Studie von [Blu02] wurden typische Signalverarbeitungsalgorithmen ausgewählt und auf verschiedenen Plattformen wie digitale Signalprozessoren (DSP), Field Programmable Gate Arrays (FPGA), anwendungsspezifischen Chips (ASIC) und optimierten ICs (Full Customized IC) implementiert. Anschließend erfolgte ein Vergleich bezüglich Flexibilität (Änderungen pro Stunde) und Produkt aus Ausführungszeit, Energieverbrauch und Chipfläche. Demzufolge verbraucht ein DSP eine Millionen Mal mehr als ein maßgeschneiderter Chip. Da dessen Herstellung aber auch die Fertigung einschließt, dauert hier ein Designdurchlauf tausendmal länger (siehe Grafik in Bild 1.3).

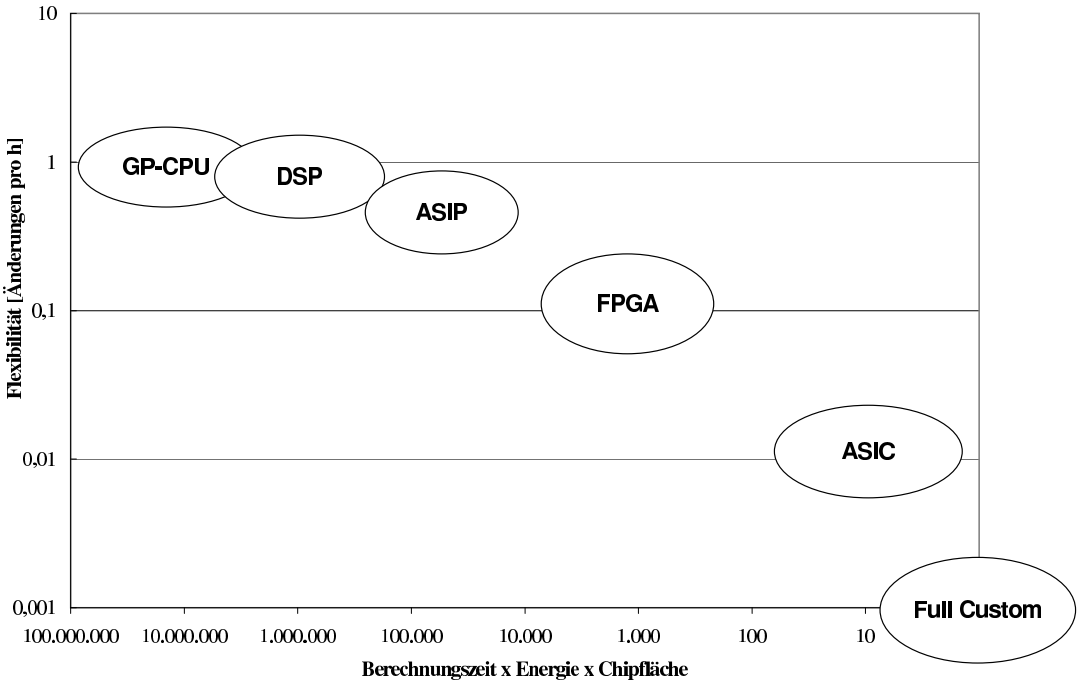

Abbildung 1.3: Vergleich Flexibilität im Verhältnis zu den Kosten aus Energieverbrauch, Chipoberfläche und Berechnungszeit [Sie02a]

Die Wiederverwendung ausgetesteter Lösungen ist vor dem Hintergrund solch langer Iterationszyklen enorm wichtig, stellt weiterhin einen wesentlichen Marktvorteil dar und verkürzt die

"Time to Market" drastisch. Durch die im Einsatz geforderte Effektivität kommen in vielen Fällen nur Hardwarelösungen in Frage, jedoch fehlen hier standardisierte Funktionsbibliotheken wie bei Programmiersprachen und damit die ausgetesteten Lösungen. Für den breiten Einsatz Hardware-basierter Lösungen stellt sich somit die Forderung nach einer universellen, hersteller- und technologie-unabhängigen Funktionsbibliothek, die die Signalverarbeitung unterstützt und damit die existierenden Hardwarebeschreibungssprachen, die auf Logik- oder Verhaltensbeschreibung beschränkt sind, ergänzt.

Das Design einer solchen Bibliothek ist nicht trivial. Folgende Anforderungen sind zu erfüllen:

- Die Verarbeitungswortbreite muss effizient in Schritten von einem Bit skalierbar sein.
- Die Architektur muss verschiedenen Geschwindigkeitsanforderungen genügen.

Die Forderung nach Skalierbarkeit der Geschwindigkeitsanforderungen bedeutet, dass dieselbe Komponente in einem weiten Frequenzbereich nutzbar ist. Übliche Architekturen erfüllen diese Bedingung nicht. Entweder sind sie einfach aufgebaut, aber nicht für hohe Taktfrequenzen geeignet oder sie sind schnell und behindern die Skalierbarkeit durch eine hohe Komplexität. Denn hohe Performance wird oft durch eine gekoppelte Verarbeitung mehrerer Bits erkauft, was die Skalierung, in Schritten kleiner als die Blockgröße, verhindert. Eine entsprechende Bibliothek hilft auch die Entwicklung von Hardware einem breiteren Anwenderkreis zugänglich zu machen.

Bei der parallelen Informationsverarbeitung, wie sie in Schaltkreisen geschieht, werden alle Signalpfade gleichzeitig durchlaufen. Damit resultiert die Gesamtzeit nicht mehr aus der Summe aller Pfade, sondern hängt nur noch vom längsten Pfad ab. Oft sind jedoch verschiedene Signalpfade zusammenzuführen. An solchen Stellen ist dafür Sorge zu tragen, dass Werte aus den richtigen Abtastzeitpunkten zusammentreffen. Dies wird an dem im Bild 1.4 dargestellten PID-Regler deutlich, wenn man die Pfade für den P- und D-Anteil vergleicht. Während im P-Pfad nur der Verstärker von den Operanden zu durchlaufen ist, kommt im D-Pfad noch der Addierer hinzu, d.h. die Operanden sind zu synchronisieren. Die Synchronisation hängt auch von der gewählten Implementierungsstrategie ab.

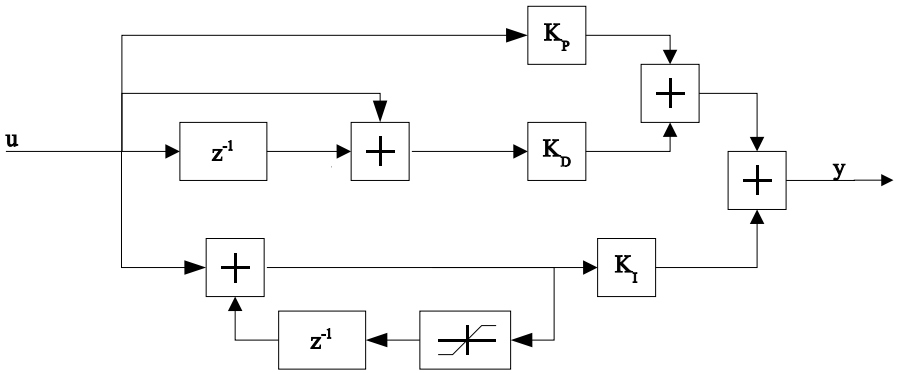

Abbildung 1.4: modaler PID-Regler

Für einen allgemeingültigen Ansatz sind folgende Forderungen zu erfüllen:

- für die effektive Energie- und Chipausnutzung ist eine überwiegend parallele Informationsverarbeitung erforderlich
- zur einfachen Implementierung muss eine universelle Bibliothek vorhanden sein
- ein Verfahren zur automatischen Synchronisation der Operanden wird benötigt

Es gibt verschiedene Ansätze, diese Forderungen zu erfüllen. Einige werden im folgenden vorgestellt.

1.3 Hardware

Weit verbreitet ist die Kombination eines Standardprozessors mit konfigurierbarer Logik. Diese wird genutzt, um die parallele Informationsverarbeitung zu realisieren und je nach Technologie auch zur Implementierung eines Prozessors als Softcore. Unter einem Softcore ist ein Makro zu verstehen, das die Funktionalität eines Prozessors beinhaltet und bei Bedarf genutzt werden kann. Sogenannte Hardcores sind schon auf dem Die (siehe Abschnitt 1.6) vorhanden. Als Quellen für die Softcores treten die Logikhersteller selber auf, jedoch sind diese Cores dann nicht portabel. Weiterhin werden sie von Drittherstellern verkauft oder sind als freie Cores unter verschiedenen Lizenzen im Internet zu finden.

Altera bietet mit dem Nios 16- oder 32-Bit RISC-Softcores für Apex PLDs an und auch bereits integrierte ARM und MIPS Prozessoren als Hardcores.
Altera: www.altera.com

Atmel bietet mit seiner FPSLIC-Technologie (Field Programmable System Level Integrated Circuits) einen AVR 8-Bit RISC-Prozessor und FPGA in einem Gehäuse an.
Atmel: www.atmel.com

Leon Von der ESA ein unter LGPL stehender in VHDL geschriebener Prozessor, der zum SPARC Embeded Prozessor kompatibel ist.
Leon: www.gaisler.com/leonmain.html

Triscend bietet eine Kombination verschiedener ARM-RISC-Prozessoren mit programmierbarer Logik auf einem Chip an.
Triscend: www.triscend.com

Xilinx bietet im Rahmen seiner Virtex II Pro Reihe bereits integrierte PowerPC 405 Hardcores an.
Xilinx: www.xilinx.com/xlnx/xil_prodcat_product.jsp?iLanguageID=1&i

All diese Lösungen stellen jedoch nur einen zusätzlichen Prozessorkern zur sequentiellen Informationsverarbeitung zur Verfügung und erfüllen damit nicht alle oben genannten Bedingungen.

1.4 Software

Durch Entwicklung neuer Sprachen oder die Anpassung existierender Programmiersprachen bzw. Entwicklungsumgebungen soll die Spezifikation von Systemen vereinfacht werden. Dieser Abschnitt gibt einen Überblick über einige Ansätze.

1.4.1 Matlab/Simulink

Simulink kann dahingehend erweitert werden, dass man für ein darin entworfenes System VHDL-Code erzeugt. Dieser Code bindet Xilinx-Softcores ein, welche verschiedenste arithmetische Funktionen implementieren. Dabei finden bitparallele Algorithmen Anwendung.
Simulink: www.mathworks.com/products/connections/product_main.shtml.

1.4.2 Superlog

Bei Superlog (www.superlog.org) von Co-Design Automation (www.co-design.com) handelt es sich um eine Hardwarebeschreibungssprache, die eine Übermenge von Verilog darstellt. Die

Sprache erlaubt Entwicklern die Modellierung von digitalen Systemen auf einem höheren Abstraktionsniveau. Mit den vorhandenen Synthesewerkzeugen ist eine durchgängige Implementierung von einer hohen Abstraktionsebene bis auf Gatterniveau möglich. Superlog unterstützt neue Funktionen und Konstrukte, welche die Methode für den Entwurf auf Architekturniveau verbessern - sowohl hinsichtlich der Implementation als auch der Verifikation.

Unter den hier genannten digitalen Systemen sind jedoch die klassischen Anwendungsfälle der Elektronik, wie Buscontroller, zu verstehen. Daher unterstützt Superlog die Signalverarbeitung nicht besser als bekannte HDLs, wie VHDL oder Verilog.

1.4.3 SystemC

Bei SystemC (www.systemc.org) handelt es sich um eine Erweiterung von C/C++, um die Möglichkeit der gesamtheitlichen Spezifikation von Systemen zu schaffen. Oft bestehen Systeme aus einem Teil der die Algorithmen in Software abarbeitet, während in einem weiteren Teil die Algorithmen in Hardware realisiert sind. SystemC erlaubt es, ein System komplett in *einer einzigen* Hochsprache zu beschreiben. Die oft erforderliche Übertragung des Hardwareteils auf eine HDL soll entfallen. Der Software-Teil kann nach dem Kompilieren auf einem Prozessor ablaufen, während der Hardware-beschreibende-Teil nach der Synthese in einen IC gebracht wird. Die oben angebrachten Probleme hinsichtlich einer Funktionsbibliothek und der Synchronisation der Operanden bleiben weiterhin bestehen.

Gegenwärtig befindet sich SystemC noch in der Standardisierung, so dass noch kein Sprachumfang für die Synthese festgelegt ist. Daher ist es für praktische Anwendungen noch nicht geeignet.

1.4.4 Handel-C

Die Entwicklung von Handel-C [Pag96] begann Mitte der 90er Jahre an der Oxford-University. Einige Jahre später wurde die Firma Embedded Solutions Ltd. gegründet, die Produkte um Handel-C anbietet. Es ist eine von C abgeleitete Programmiersprache zur Spezifikation von Signalverarbeitungsroutinen. Sie richtet sich an Programmierer und nicht an Hardwareentwickler und versteckt daher aller Hardware-relevanten Probleme vor dem Anwender. Es stehen Werkzeuge zur Implementation in FPGAs und ASICs zur Verfügung.

Jedoch erfolgt die Implementierung der Operatoren bitparallel und es verbleiben die damit verbundenen Probleme.

1.5 Alternative Ansätze

Während die oben vorgestellten Lösungen im wesentlichen auf bekannten Technologien aufbauen (so wurden Programmiersprachen erweitert parallele und sequentielle Informationsverarbeitung kombiniert) werden in diesem Abschnitt Lösungen vorgestellt, die schon im Ansatz bekannte Pfade verlassen.

1.5.1 Xputer

Viele Anwendungen erfordern die iterative Manipulation großer Datenmengen. Die neue Architekturklasse Xputer wurde insbesondere entworfen, um den von Neumann-Flaschenhals der wiederholten Dekodierung, Adresseninterpretation und Datenmanipulation zu reduzieren. Dieser Flaschenhals steuert einen großen Anteil (bis zu 90% bei der Bildverarbeitung, 58% bei der digitalen Signalverarbeitung) zur Verarbeitungszeit der genannten Algorithmentypen bei.

Der Xputer ist ein nicht von Neumann-Maschinen Paradigma, da er keine Befehlssequenzer hat, sondern anstelle dessen einen Datensequenzer. Die Basisstruktur eines Xputermoduls besteht aus drei Hauptteilen:

- zweidimensional organisierter Datenspeicher
- rekonfigurierbare Arithmetik und Logikeinheit (rALU), eingeschlossen verschiedene rALU Teilnetze mit mehrfachen Abtastfenstern
- rekonfigurierbarer Datensequenzer (DS) verschiedene generische Adressgeneratoren (GAGs) umfassend

Der Hauptunterschied zum Computer ist, dass der Datensequenzer und die rekonfigurierbare ALU den Programmspeicher, den Befehlssequenzer und die hartverdrahtete ALU ersetzen. Für die Operatorenauswahl wird anstelle des Sequenzers eine andere Einheit, ein so genannter Residuencontroller, genutzt. Die Operatoren Aktivierung ist Transport getriggert, im Gegensatz zur Steuerungsfluss getriggerten Aktivierung beim von Neumann-Computer. Genauere Informationen sind unter Xputer: `xputers.informatik.uni-kl.de/xputer/index_xputer.html` und [Sie01b] zu finden.

1.5.2 Universal Configurable Machine

Die Idee der Universal Configurable Machine verfolgt den Ansatz, einen Prozessor aus mehreren universellen konfigurierbaren Blöcken aufzubauen (Universal Configurable Block UCB). Während FPGAs aus einem Array mit einfachen Elementen bestehen, die jedes für sich nur logische Funktionen realisieren kann, realisiert ein UCB arithmetische Funktionen. Zur Implementierung komplexer Funktionen werden mehrere davon in geeigneter Weise verschaltet (siehe[Sie01a]).

1.5.3 PACT/XPP

Die XPP-Architektur ist eine Realisierung des UCM-Ansatzes aus Abschnitt 1.5.2. Hierbei handelt es sich um ein Array von 32-Bit-Prozessoren, das mit 50 oder 100 MHz getaktet wird und daher über eine enorme Rechenleistung verfügt. Dort liegt auch der Anwendungsbereich.

Ausschnitt aus [Sie02b]:

In eine völlig andere Bausteinklasse, die noch keinen eigenen Namen besitzt, fällt die XPP-Architektur der Firma PACT (www.pactcorp.com), München. Hier sind es relativ große Blöcke, bestehend z.B. aus einer ALU mit Registern, die konfigurierbar miteinander verbunden sind. Aus diesem Grund heißt diese Klasse UCB (Universal Configurable Blocks). In der XPP-Architektur findet man vier Ebenen der Hierarchie (Hardware-Objekte, Processing Array Elements (PAE), Processing Array Cluster (PAC), Devices [Sie01b]). Das Ziel dieses hierarchischen Aufbaus ist eine hohe Skalierbarkeit für Applikationen. Die Processing Array Cluster dieser Architektur (Bild 1.5) zeigen die für die UCB-Klasse typische Struktur: Sie sind aufgebaut aus kleinen Blöcken, welche die rechnenden Elemente (ALU, Register, kleine programmierbare Gate Arrays) beinhalten und die miteinander konfigurierbar verbunden werden.

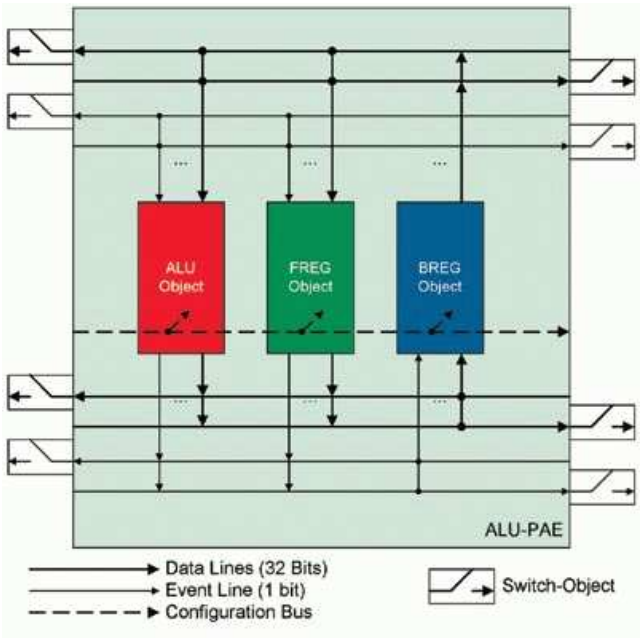

Abbildung 1.5: Aufbau des Processing Array Clusters (PAC) der XPP-Architektur [Sie02b]

1.5.4 FLYSIG-Prozessor

Hierbei handelt es sich um die Idee eines datengetriebenen Prozessors. Eine Beschreibung ist z.B. in [Har98] oder Flysig-Idee: www.uni-paderborn.de/sfb376/projects/b1/PS/HaKl98.ps.gz zu finden. Zum Verständnis ist ein Auszug hier abgedruckt. Bild 1.7 illustriert den Datenfluss

im FLYSIG-Prozessor, welcher aus den drei dort zu sehenden Komponenten aufgebaut ist. Zusätzlich gibt es noch eine Verbindung zur Umgebung. Der FLYSIG-Prozessor wird durch die Konfigurations- und Zustandssteuerungskomponente initialisiert. Die Speicher- und Routingkomponente ist mit der Operationskomponente zu einer zyklischen Struktur verbunden. Innerhalb dieser Struktur ist das Multiring-Konzept eingebettet. Die Ringstruktur verfügt über Ein- und Ausgänge, so dass leicht weitere FLYSIG-Prozessoren angeschlossen werden können. Damit ist der Aufbau von Prozessornetzwerken sehr einfach.

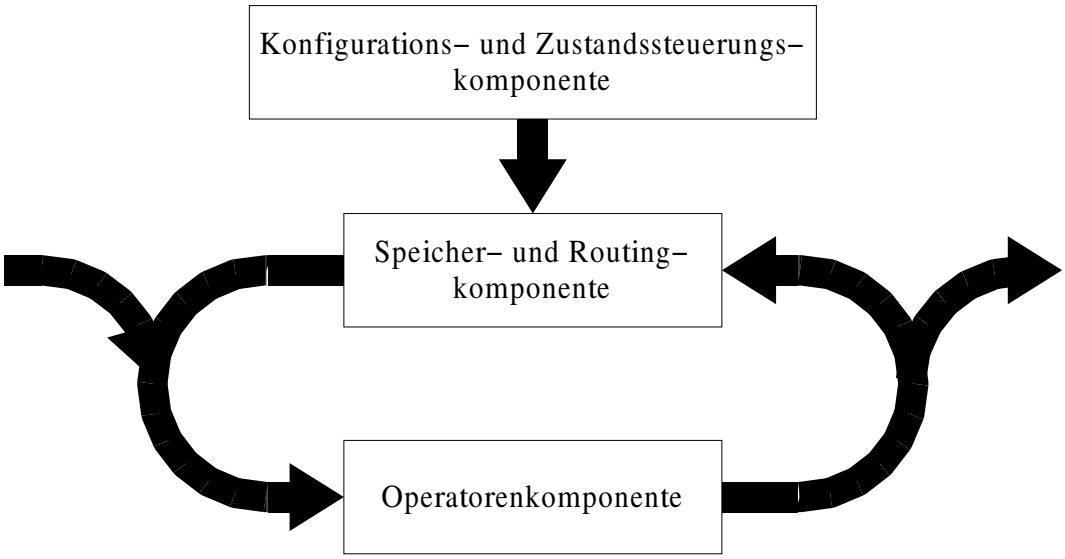

Abbildung 1.6: Der Datenfluss im FLYSIG-Prozessor [Har98]

Eine im FLYSIG-Prozessor zu implementierende Anwendung wird als Steuerungs- und Datenflussgraph spezifiziert. Jede Operation wird durch einen Knoten im Datenflussgraphen repräsentiert. Bei der Operationsaufteilung (operation scheduling) wird jedem Datenflussknoten ein Operator der Operatorenkomponente zugewiesen. Während des Verbindungstasks werden die Routen im Prozessor festgelegt. Bild 1.6 zeigt einige Details.

(a) Die Konfigurations- und Zustandssteuerungskomponente dient der Konfiguration des Prozessors. Die Aufteilungsinformationen werden in den lokalen Speicher und in die Routingkomponente für die anfängliche Operanden- und Ergebnisweiterleitung eingespeist. Zusätzlich werden die Anfangsoperanden in den Speicher geschrieben.

(b) Die Speicher- und Routingkomponente verarbeitet die Operanden und die Berechnungsresultate. Ein Datum im Speicher wird als Token betrachtet und es wird nicht unterschieden, ob es sich um einen Operand oder um ein Ergebnis handelt. Die Token fließen vom Speicher in die Operatorenkomponente über die Tokenevaluierung. Dieser Block bestimmt, ob eine Speicherzelle einen gültigen Token enthält. In diesem Fall leitet der Routingblock den Token zum entsprechenden Operator. Jeder

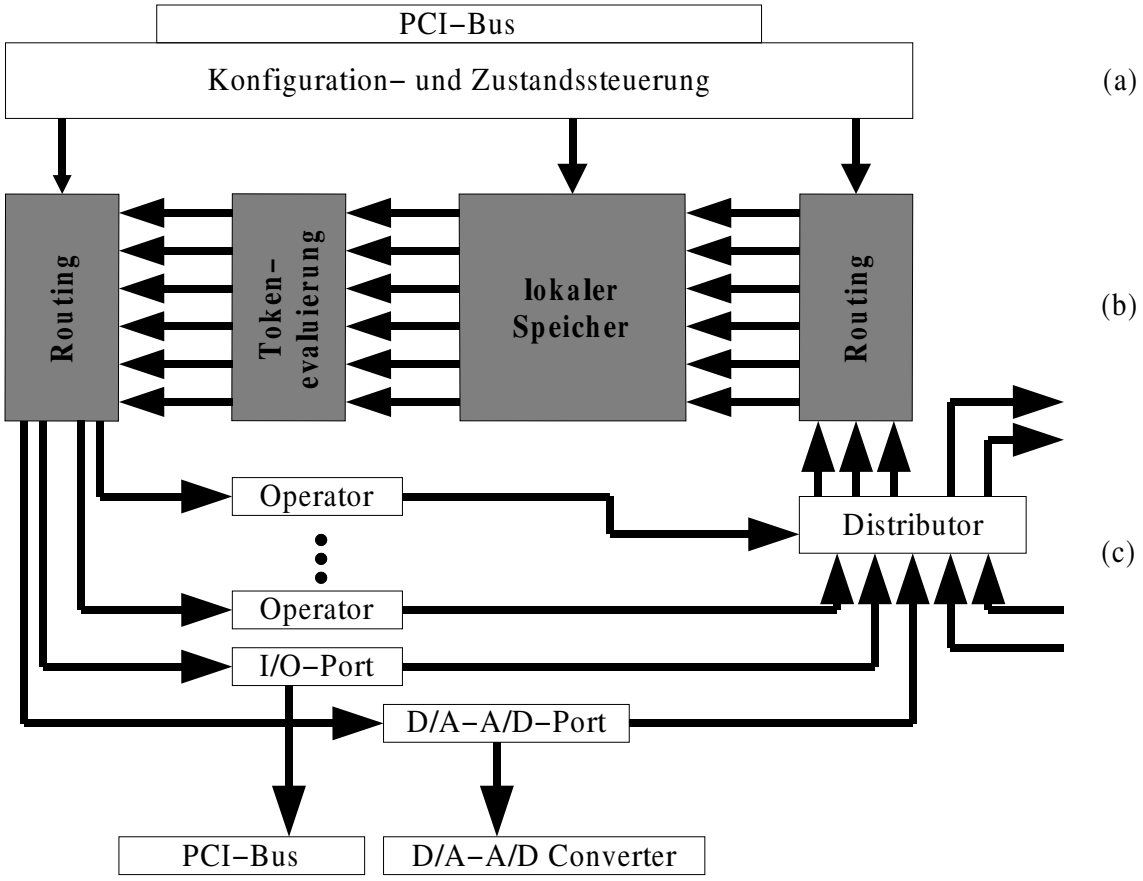

Abbildung 1.7: Das Konzept des FLYSIG-Prozessors mit (a) Konfigurations- und Steuerungskomponente (b) Speicher- und Routingkomponente und (c) der Operationskomponente [Har98]

Token besteht aus einer Operations-ID (identifiziert die Operation), einem Valid-Flag-Segment (zeigt die Verfügbarkeit von Operanden an) und einem Guard-Flag (bestimmt, wo die Ergebniswerte benötigt werden).

(c) Die Operationskomponente implementiert die Operatoren für alle möglichen Berechnungen. Die Operatoren lesen die Token vom Speicher oder vorherigen Operatoren. Damit ist ein Operatoren-Pipelining möglich. Durch die bitserielle Implementierung führt dies zu problemarmen Pipelines und damit zu einem hohen Durchsatz. Die Berechnungsresultate werden in den lokalen Speicher oder direkt in die Register weiterer FLYSIG-Prozessoren geschrieben. Das ist ein wichtiges Merkmal, welches die Verteilung eines einzelnen Algorithmus über mehrere FLYSIG-Prozessoren erlaubt.

1.5.5 Bitserieller Ansatz

Heutzutage wird die digitale Signalverarbeitung in Produkten von Mikro-Controllern und digitalen Signalprozessoren dominiert. Durch Interaktion mit der Umwelt sind in fast allen Fällen Echtzeitanforderungen zu erfüllen. Da einige Prozessorressourcen nur einmal vorhanden sind,

müssen sie verschiedenen Tasks quasi parallel zugeordnet werden. Diese Verwaltung wird üblicherweise von einem Echtzeitbetriebssystem vorgenommen [Gra99, Kop98] und kostet zusätzliche Rechenleistung. Deren Anteil steigt bei hochdynamischen Systemen dramatisch an, so dass nicht selten ein leistungsfähigerer und teurer Prozessor benötigt wird.

Bei sehr hohen Abtastfrequenzen kommen FPGAs [Aue95] und ASICs (anwendungsspezifische ICs) zum Einsatz. Sie bieten die Möglichkeit der parallelen Informationsverarbeitung, d.h. alle Signalpfade werden parallel in mehrfach vorhandenen Hardware-Elementen berechnet. Damit resultiert die Laufzeit eines Signals nur noch aus der längsten Signalkette, die zu durchlaufen ist und nicht mehr aus der Summe aller Signalketten wie bei Software-Lösungen. Jedoch behindern einige Probleme die Anwendung dieser Technologie:

- Die Implementierung direkt auf Hardware-Ebene verlangt anderes Wissen als die Erstellung von Software.
- Hardwarebeschreibungssprachen (HDL) wurden für Schaltungsentwickler entworfen, jedoch nicht für Mechatroniker, Regelungstechniker und Systemingenieure.
- Es fehlen allgemeine standardisierte, herstellerunabhängige Funktionsbibliotheken. Nur sehr einfache Funktionen (Addition, Subtraktion) sind verfügbar. Bei Bedarf sind komplexere Komponenten vom Anwender selbst zu entwickeln.

Um den Entwurf zu vereinfachen, kann die HDL mit Hilfe eines Blockdiagrammeditors in Blockdiagrammen versteckt werden. Aus diesen wird dann Code generiert. Jedoch muss eine geeignete Bibliothek vorhanden sein, welche die Operatoren zur Verfügung stellt, in denen die eigentliche Signalverarbeitung abläuft. Die Bibliothek muss den in Abschnitt 1.2 genannten Forderungen genügen.

Der hier präsentierte Ansatz basiert auf der bitseriellen Übertragung und Verarbeitung der Operanden. Das heißt, eine Komponente verarbeitet nur ein Bit während eines Taktes. Das ermöglicht sehr einfache und sehr schnelle Verarbeitungselemente.

Ähnliche Arbeiten In [And03b] sind Beispiele zur Implementierung bitserieller Operatoren auf Basis der binären Zahlenrepräsentation zu finden, während [Hla92] die Möglichkeiten zur bitseriellen Implementierung von Gleitkomma-Operatoren aufzeigt. In beiden Fällen erfolgt jedoch die Übertragung der Operanden bitparallel.

Es gibt auch Anwendungsfälle mit bitserieller Übertragung der Operanden. Das sind z.B.:

- Implementierung eines Filters für Telefonnetze in bitserieller Technik [Nil97]
- Implementierung des IDEA Kryptoalgorithmus mittels bitserieller Verarbeitung [Leo00]

- Implementierung eines Verfahrens zur genetischen Optimierung [Bla97]

Diese Beispiele sind jedoch Lösungen für ein besonderes Problem und stellen dem Ingenieur kein Entwicklungswerkzeug zur Verfügung.

In der binären Zahlendarstellung werden vier Bit zu einer Stelle im Hexadezimalcode zusammengefasst. Üblicherweise besteht ein Operand aus mehreren Stellen. Diese Stellen können auch seriell übertragen werden. Dies wird dann als digit-seriell bezeichnet. In [Erc02] sind einige grundlegende digit-serielle Algorithmen zu sehen. Anwendungsbeispiele zur Implementierung in FPGAs sind in [Lee97] und [Val98] zu finden.

1.6 Herstellungsprozess von digitalen integrierten Schaltkreisen

Bild 1.8 zeigt in groben Schritten den Ablauf zur Herstellung eines digitalen Schalkreises, ohne auf die Überprüfung der logischen Funktion einzugehen. Bis zum Tapeout ist dieser Prozess rein virtuell. Das Tapeout bezeichnet die Übergabe der Fertigungsdaten an die Fabrik, die früher eben auf Magnetband erfolgte. Da bei der Verwendung von PLDs dieser schon als Bauelement vorliegt, müssen vom Anwender nur noch die Konfigurationsdaten generiert werden.

Unter Spezifikation ist die Schaltungseingabe zu verstehen, die hier mit einer Hochsprache (HDL Hardware Description Language) erfolgt. Daraus wird mit einem Synthesewerkzeug eine Netzliste generiert, welche nur noch eine Hierarchieebene umfasst und ausschließlich logische Elemente (RTL, Register-Transfer-Logik) enthält. Das Mapping bezeichnet deren Abbildung auf die tatsächlich in der jeweiligen Technologie vorhandenen Komponenten. Danach werden diese optimal platziert und verbunden. Anschließend erfolgt dann bei PLDs die Ausgabe der Konfigurationsdaten, während bei ASICs die Daten auf Einhaltung von Fertigungsparametern (z.B. Isolationsabständen) überprüft werden. Diese Daten gehen dann an die Fabrik, in welcher die Masken für den Herstellungsprozess gefertigt werden. Damit erfolgt die eigentliche Herstellung des Dies. Dieses wird dann mit den Anschlussbeinen verbunden (Bonden) und verpackt. (siehe [Rei98])

Während der Spezifikation erfolgt schon die rein funktionale Simulation der Schaltung. Dies geschieht mittels Simulatoren für die verwendete Beschreibungssprache. Dazu wird der Quelltext kompiliert (seltener interpretiert) und ein komplettes Abbild der Schaltung mit allen Instanzen auf eine Ebene heruntergebrochen im Speicher des Computers angelegt. Dieser Prozess nennt sich Elaboration (engl. Ausarbeitung, ausführliche Darstellung). Weitere Simulationen erfolgen nach dem Platzieren und Routen, dann als Timing-Simulation und seltener nach der Synthese als funktionale Simulation auf Register-Transfer-Ebene. Während der Produktion finden Tests vor Zerschneiden des Wafers und nach dem Verpacken statt.

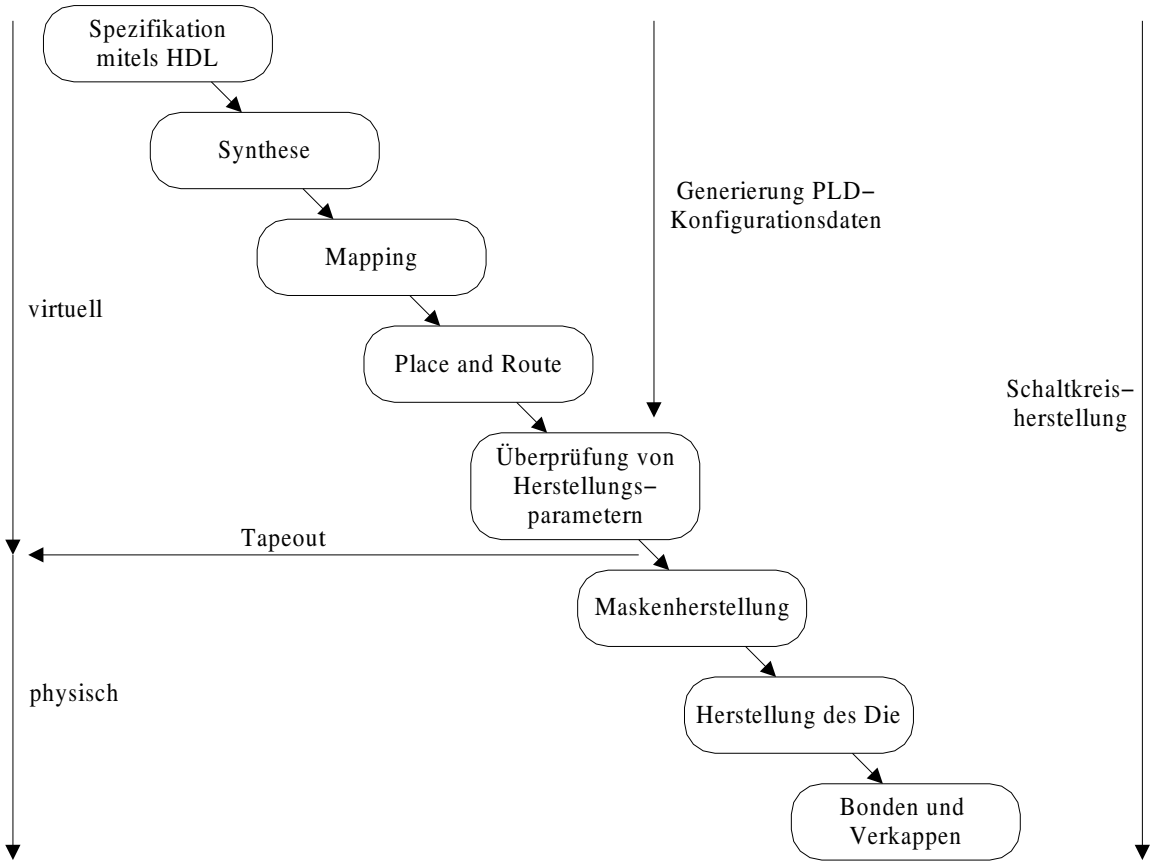

Abbildung 1.8: Schaltkreisherstellungsprozess

Kapitel 2

Grundlagen der bitseriellen Verarbeitung[1]

Diese Kapitel stellt Grundlagen der bitseriellen Verarbeitung und einige Basiskomponenten vor. Die digitale Signalverarbeitung ist bisher durch die bitparallele Übertragung und Verarbeitung der Operanden gekennzeichnet, verbunden mit komplizierten und logikintensiven Operatoren [Kor93].

Diese Nachteile sollen durch einen bitseriellen Ansatz überwunden werden. Im Unterschied zu einigen in [And03b] dargestellten Algorithmen, erfolgt hier nicht nur die Verarbeitung sondern auch die Übertragung der Operanden bitseriell. Die Übertragung beginnt mit dem niederwertigsten Bit (LSB), dem dann die höherwertigen Bits folgen.

Anders als bei der bitparallelen Übertragung, wo jedes Signal über eine eigene Leitung übertragen wird, teilen sich im bitseriellen Fall alle Bits eine Leitung im Zeitmultiplex. Bild 2.1 stellt beide Prinzipien gegenüber, während Bild 2.2 den Operandenframe zeigt.

2.1 Randbedingungen

Hier zu verwendende arithmetische Algorithmen müssen, bedingt durch den festen Rahmen zur Übertragung eines Operanden, bestimmten Bedingungen genügen. Diese Bedingungen entstehen dadurch, dass nach einer gewissen Anzahl von Takten immer ein neues Ergebnisbit auszugeben ist. Die Berechnung eines Ergebnisbits kann zwar mehrere Takte in Anspruch nehmen, jedoch muss mit jedem Takt ein Bit ausgegeben werden.

- Die zur Berechnung eines Ergebnisbits notwendige Anzahl der Takte darf nicht größer sein, als die zur Übertragung vorgesehene Anzahl der Takte, d.h. es darf nicht zu einer zunehmenden Warteschlagenlänge kommen.

[1] Anmerkungen zur Formelnotation sind im Anhang A zu finden.

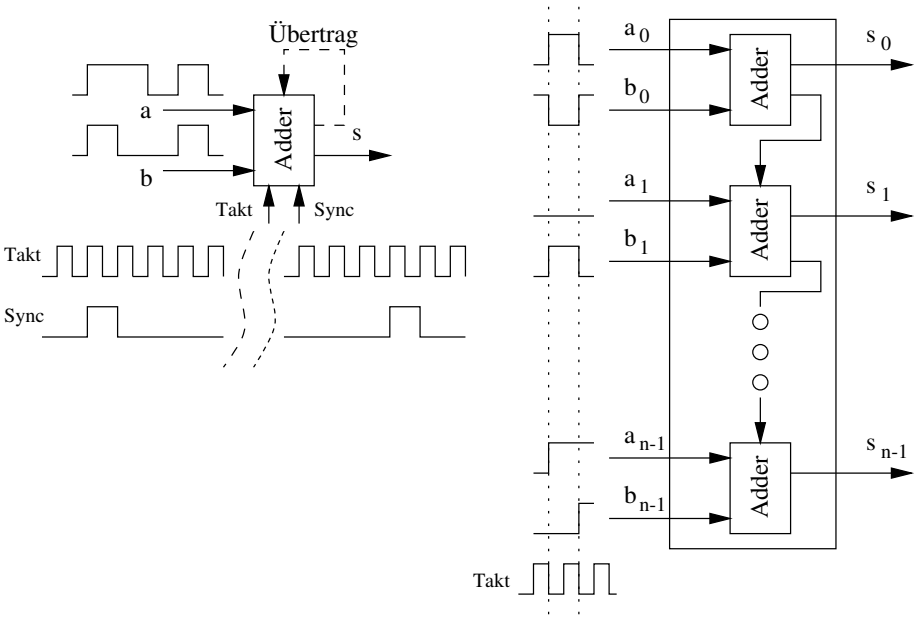

Abbildung 2.1: Bitserielle und bitparallele Verarbeitung

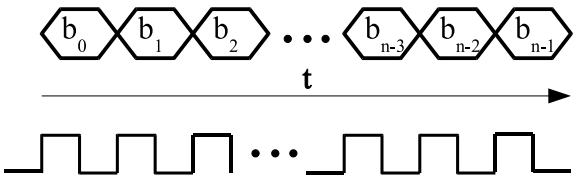

Abbildung 2.2: Übertragungsrahmen

- Die maximale Anzahl von Iterationszyklen muss bekannt sein, da für jeden Operator eine konstante Verzögerung anzugeben ist. Wenn einmal weniger Takte benötigt werden, ist das Ergebnis im Operator entsprechend zu verzögern.

2.2 Zahlenrepräsentation

Prinzipiell muss die unabhängige Verarbeitung einzelner Operandenbits gewährleistet sein, d.h. durch die Auswahl einer Zahlenrepräsentation darf es nicht schon bei einfachen Operationen wechselseitige Abhängigkeiten zwischen den zuerst und den zuletzt übertragenen Operandenbits geben.

2.2.1 Festkomma-Zahlensysteme

Eine wechselseitige Abhängigkeit zwischen LSB und MSB ist z.B. bei der binären ganzzahligen Zahlendarstellung nicht gegeben. Eine Zahl wird folgendermaßen repräsentiert:

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i \tag{2.1}$$

Um auch negative Zahlen einfach verarbeiten zu können, wird diese Darstellung zum Zweierkomplement erweitert:

$$X = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \tag{2.2}$$

2.2.2 Gleitkomma-Zahlensysteme

Weit verbreitet ist die Gleitkomma-Darstellung zum Beispiel im IEEE 64 Bit Format

$$F = (-1)^S 1, f \cdot 2^{E-1023}$$

$$\text{mit } S = 0;\ 1 \text{ und } f = \sum_{i=0}^{51} \overline{f}_i \cdot 2^i \text{ und } E = \sum_{i=0}^{10} e_i \cdot 2^i \tag{2.3}$$

Durch den zusätzlichen Exponenten vereinfacht die Gleitkomma-Darstellung die Skalierung. Jedoch sind die Mantisse f und der Exponent E voneinander abhängig. Denn der gebrochen rationale Teil der Mantisse ist in den Bereich $0 \leq f < 1$ zu skalieren, davon ist auch der Exponent betroffen. Somit können die beiden Operandenteile nicht isoliert verarbeitet und übertragen werden. Das verhindert die direkte Verwendung des hier vorgeschlagenen bitseriellen Ansatzes.

2.2.3 Logarithmische Zahlensysteme

Logarithmische Zahlensysteme sind im wesentlichen Gleitkomma-Zahlensysteme mit konstanter Mantisse. Eine Zahl wird wie folgt repräsentiert:

$$A = (-1)^{S_A} \cdot 2^{E_A}$$

Durch die logarithmische Darstellung ist die Realisierung der Multiplikation und ähnlicher Operationen sehr einfach. Dort liegt auch der Anwendungsbereich.

2.2.4 Residuen Zahlensystem

Das residuen Zahlensystem ist ein ganzzahliges Zahlensystem, dessen wichtigste Eigenschaft die übertragsfreie Addition, Subtraktion und Multiplikation ist. Daraus folgt, dass man addieren, subtrahieren und multiplizieren kann, ohne auf die Länge der Operanden achten zu müssen. Leider sind dafür andere Operationen wie Division, Vergleich und Vorzeichenerkennung sehr komplex und langsam. Ein weiteres Problem ist die Darstellung von nicht ganzzahligen Werten. Aus all diesen Gründen sind residuen Zahlensysteme für die allgemeine Verwendung nicht geeignet, sondern auf Spezialfälle beschränkt, bei denen die Anzahl der Additionen, Subtraktionen und Multiplikationen deutlich größer ist, als die Anzahl der schwer zu realisierenden Operationen. Näheres zu residuen Zahlensystemen ist u.a. in [Kor93] zu finden.

2.3 Addition

Es werden die Operanden a und b in binärer Zahlendarstellung verwendet. c_i stellt den Übertrag der i-ten Stelle dar.

$$s_i = (a_i + b_i + c_{i-1}) \, mod \, 2 \tag{2.4}$$

$$c_i = (a_i + b_i + c_{i-1}) \, div \, 2 \tag{2.5}$$

$$s_0 = (a_0 + b_0) \, mod \, 2 \tag{2.6}$$

$$c_0 = (a_0 + b_0) \, div \, 2 \tag{2.7}$$

Daraus lässt sich die Logiktabelle für die Addition ableiten.

c_{i-1}	a	b	sum	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabelle 2.1: Logiktabelle für Addition

Diese Logiktabelle ist im Bild 2.3 in den Blöcken “Berechnung Summenbit” und “Berechnung Übertragsbit” implementiert.

Der Multiplexer dient zur Realisierung der Gleichungen 2.6 und 2.7, während der Berechnung des höchstwertigen Bits (MSB) wird hier eine Null ausgeben und so ein Übertrag verhindert.

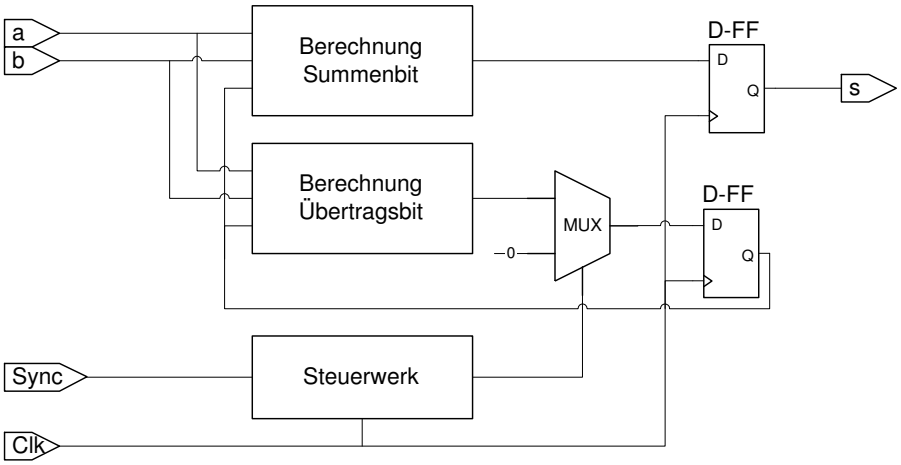

Abbildung 2.3: Implementierung Addierer

Der Vorteil der Verwendung des Zweierkomplements für vorzeichenbehaftete Operanden liegt darin, dass die Operationen genauso ausgeführt werden wie bei vorzeichenlosen Operanden. Lediglich das Ergebnis wird anders interpretiert.

2.4 Subtraktion

Es werden die Operanden a und b in binärer Zahlendarstellung verwendet. c_i stellt das Borgen der i-ten Stelle dar.

$$d_i = (a_i - b_i - c_{i-1}) \, mod \, 2 \tag{2.8}$$

$$c_i = (a_i - b_i - c_{i-1}) \, div \, 2 \tag{2.9}$$

$$d_0 = (a_0 - b_0) \, mod \, 2 \tag{2.10}$$

$$c_0 = (a_0 - b_0) \, div \, 2 \tag{2.11}$$

Daraus lässt sich die Logiktabelle für die Subtraktion ableiten.

Der Subtrahierer ist genauso aufgebaut wie der Addierer in Bild 2.3, nur enthalten die Blöcke zur Berechnung Logikfunktionen entsprechend Tabelle 2.2. Die Darstellung der Tabellen für die Fehlerbehandlung ist im Abschnitt 2.5 zu finden.

2.5 Fehlerbehandlung bei Addition und Subtraktion

Eine Fehlerbehandlung ist nötig, wenn das Ergebnis nicht in den vorhandenen Rahmen passt, also mehr Bit zur Repräsentation notwendig als vorhanden sind. Der auszugebende Wert hängt von der Operation und dem Vorzeichen des aktuellen Operanden ab. Da sich die Operation

c_{i-1}	a	b	diff	c_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Tabelle 2.2: Logiktabelle für Subtraktion

eines Rechenelements nicht ändert, wird der Ersatzwert effektiv vom Vorzeichen beeinflusst. Gleichung 2.5 zeigt die Berechnungsvorschrift für ein Summenbit bei der Addition. Ob ein Ergebnis richtig oder falsch ist, kann allein anhand der drei verwendeten Operatoren (a, b und c) entschieden werden. D.h., bei Auftreten bestimmter Operatorenkombinationen kann das Ergebnis nur falsch werden. Darum ist es nicht sinnvoll, dies auch noch zu berechnen und dafür Zeit zu verschenken. Zumal ein Übertrag in die Stelle $n+1$ beim Zweierkomplement kein eindeutiges Kriterium ist.

2.5.1 Fehlerbehandlung über n Bit

Das Auftreten eines Fehlers kann anhand der MSBs der Operanden und des Übertrags in das MSB erkannt werden (siehe Gleichung 2.4). Zur Bewertung wurden Beispiele konstruiert, welche die erforderlichen Fälle abdecken. Sie sind in den Tabellen 2.3 und 2.4 abgebildet. Da zur Generierung eines Übertrags/des Borgens unterschiedliche Operanden bei Addition und Subtraktion erforderlich sind, werden diese Fälle auch unterschiedlich betrachtet. Die linke Seite der Tabellen (x_{n-1}, x_{n-2}) zeigt die Summanden bzw. Subtrahend und Minuend, sowie das Ergebnis der Operation. Eine klein vorangestellte 1 bei b_{n-1} bedeutet, dass hier ein Übertrag oder Borgen stattgefunden hat. Auf der rechten Seite ist in Abhängigkeit der Operation und der Zahlenrepräsentation (vorzeichenlos + oder vorzeichenbehaftet +signed) abzulesen, ob das Ergebnis richtig (1) oder falsch (0) ist.

Die MSBs (a_{n-1}, b_{n-1}) sowie der Übertrag der Tabellen 2.3 und 2.4 wurden mit der Bewertung der Richtigkeit (die rechte Seite der Tabellen) in Tabelle 2.5 zusammengefasst, um eine Fehlerfunktion für jede Operation bei den jeweils möglichen Zahlenrepräsentationen bilden zu können.

	x_{n-1}	x_{n-2}	+	+signed
a	0	0		
b	0	0		
+	0	0	1	1

	x_{n-1}	x_{n-2}	+	+signed
a	1	1		
b	$_1$0	1		
+	0	0	0	1

	x_{n-1}	x_{n-2}	+	+signed
a	0	1		
b	$_1$0	1		
+	1	0	1	0

	x_{n-1}	x_{n-2}	+	+signed
a	1	0		
b	0	0		
+	1	0	1	1

	x_{n-1}	x_{n-2}	+	+signed
a	0	1		
b	$_1$1	1		
+	0	0	0	1

	x_{n-1}	x_{n-2}	+	+signed
a	1	0		
b	1	0		
+	0	0	0	0

	x_{n-1}	x_{n-2}	+	+signed
a	0	0		
b	1	0		
+	1	0	1	1

	x_{n-1}	x_{n-2}	+	+signed
a	1	1		
b	$_1$1	1		
+	1	0	0	1

Tabelle 2.3: Bestimmung der Fehlerfunktion der Addition über n Bit

	x_{n-1}	x_{n-2}	-	-signed
a	0	0		
b	0	0		
-	0	0	1	1

	x_{n-1}	x_{n-2}	-	-signed
a	1	0		
b	0	0		
-	1	0	1	1

	x_{n-1}	x_{n-2}	-	-signed
a	0	0		
b	$_1$0	1		
-	1	1	0	1

	x_{n-1}	x_{n-2}	-	-signed
a	1	0		
b	$_1$0	1		
-	0	1	1	0

	x_{n-1}	x_{n-2}	-	-signed
a	0	0		
b	$_1$1	1		
-	0	1	0	1

	x_{n-1}	x_{n-2}	-	-signed
a	1	0		
b	1	0		
-	0	0	1	1

	x_{n-1}	x_{n-2}	-	-signed
a	0	0		
b	1	0		
-	1	0	0	0

	x_{n-1}	x_{n-2}	-	-signed
a	1	0		
b	$_1$1	1		
-	1	1	0	1

Tabelle 2.4: Bestimmung der Fehlerfunktion der Subtraktion über n Bit

			Addition		Subtraktion	
a_{n-1}	b_{n-1}	c_{n-2}	unsigned	signed	unsigned	signed
0	0	0	1	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	1	0	1
1	0	0	1	1	1	1
1	0	1	0	1	1	0
1	1	0	0	0	1	1
1	1	1	0	1	0	1

Tabelle 2.5: Fehlerfunktion über n Bit

Aus Tabelle 2.5 wurden als Beispiel für den Addierer die Logikfunktionen zur Implementierung der Fehlerfunktion herausgelesen.

$$error_{Add_{signed}} = (a \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee c)$$

$$error_{Add_{unsigned}} = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (\overline{a} \vee \overline{b} \vee \overline{c})$$

2.5.2 Fehlerbehandlung über n-1 Bit

Eine Fehlerbehandlung über $n-1$ Bit ist erforderlich, um den Wertebereich auf die Hälfte einzuschränken. Damit kann man Vorarbeit zur Fehlervermeidung in nachfolgenden Operatoren leisten. Ob ein Ergebnis richtig oder falsch ist, wird nur durch die beiden höchstwertigen Bits und einen Übertrag hierein beeinflusst. Die Entscheidung wird nicht nur durch einen eventuellen Überlauf/Borgen beeinflusst, sondern es ist auch der eingeschränkte Wertebereich einzuhalten. Dadurch sind bestimmte Kombinationen der beiden MSBs im Resultat von vornherein falsch. Eine Verschiebung der Betrachtung aus der Behandlung über n Bit um ein Bit ist nicht möglich, da dabei die höchstwertigen Bits nicht berücksichtigt werden.

Die Wertebereiche sind:

- vorzeichenlos $0 \ldots 2^{n-1}-1$, das entspricht binär $00xxx \ldots 01xxx$. Dadurch sind alle Resultate mit 11 und 10 in den MSBs zu behandeln.
- Zweierkomplement $-2^{n-2} \ldots 0 \ldots 2^{n-2}-1$, das entspricht binär $11xxx \ldots 00xxx \ldots 001xx$. Hier sind alle Resultate mit einer 10 und 01 in den MSBs zu behandeln.

Die Tabellen 2.6 und 2.8 zeigen die Ergebnisse der Addition und Subtraktion für alle möglichen Werte der Operanden einschließlich eines Übertrags in die Stelle $n-2$. Da die Stellen 0 bis

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	0	0	0				
+	0	0	-	1	-	1	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	0	1	0				
+	1	0	-	1	-	0	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	0	0	1				
+	0	1	-	1	-	0	-
-	1	1	-	-	0	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	0	1	1				
+	1	1	-	0	-	1	-
-	1	1	-	-	0	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	0	1	0				
+	0	1	-	0	-	0	-
-	1	1	-	-	0	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	1	0	0				
+	1	0	-	0	-	0	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	0	1	1				
+	1	0	-	1	-	0	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	1	0	1				
+	1	1	-	0	-	1	-
-	0	1	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	0	0	0				
+	0	1	-	1	-	0	-
-	0	1	-	-	1	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	1	1	0				
+	1	1	-	0	-	1	-
-	0	1	-	-	0	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	0	0	1				
+	1	0	-	1	-	0	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	0	-				
b	1	1	1				
+	0	0	-	0	-	1	-
-	0	0	-	-	0	-	1

Tabelle 2.6: Bestimmung der Fehlerfunktion (1)

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	1	0	0				
+	1	1	-	0	-	1	-
-	1	1	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	0	1	0				
+	1	1	-	0	-	1	-
-	0	1	-	-	1	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	1	0	1				
+	0	0	-	0	-	0	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	0	1	1				
+	0	0	-	0	-	1	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	1	1	0				
+	0	0	-	0	-	0	-
-	0	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	0	0	0				
+	1	1	-	0	-	1	-
-	1	1	-	-	0	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	0	1	-				
b	1	1	1				
+	0	1	-	0	-	0	-
-	0	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	0	0	1				
+	0	0	-	0	-	1	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	0	0	0				
+	1	0	-	0	-	0	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	0	1	0				
+	0	0	-	0	-	1	-
-	1	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	0	0	1				
+	1	1	-	0	-	1	-
-	0	1	-	-	1	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	0	1	1				
+	0	1	-	0	-	0	-
-	0	1	-	-	1	-	0

Tabelle 2.7: Bestimmung der Fehlerfunktion (2)

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	1	0	0				
+	0	0	-	0	-	0	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	1	0	0				
+	0	1	-	0	-	0	-
-	0	1	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	1	0	1				
+	0	1	-	0	-	0	-
-	0	1	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	1	0	1				
+	1	0	-	0	-	0	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	1	1	0				
+	0	1	-	0	-	0	-
-	0	1	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	1	1	0				
+	1	0	-	0	-	0	-
-	0	0	-	-	1	-	1

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	0	-				
b	1	1	1				
+	1	0	-	0	-	0	-
-	0	0	-	-	0	-	0

				unsigned		signed	
	x_{n-1}	x_{n-2}	c_{n-3}	+	-	+	-
a	1	1	-				
b	1	1	1				
+	1	1	-	0	-	1	-
-	1	1	-	-	0	-	1

Tabelle 2.8: Bestimmung der Fehlerfunktion (3)

$n-3$ das Ergebnis nur über den von ihnen generierten Übertrag beeinflussen, sind sie nicht zu betrachten.

In Tabelle 2.9 ist die Fehlerfunktion noch einmal zusammengefasst. Die Summenbits sind nicht aufgeführt, da sie nicht direkt zur Bewertung des Ergebnisses beitragen. Sie sind selber Resultat der Eingangssignale.

Die Anzahl der gültigen Resultate weicht bei der vorzeichenlosen Addition und Subtraktion voneinander ab. Die geringere Anzahl von gültigen Resultaten bei der Addition hängt mit dem nach oben reduzierten Wertebereich zusammen. Da hier das Resultat grundsätzlich größer wird, wirkt sich die nach unten verschobene Grenze öfter aus.

Der Aufbau der Tabellen 2.6 bis 2.8 ist analog zu denen im Abschnitt 2.5.1. Die linke Seite enthält wieder die zwei höchstwertigen Bits der Operatoren sowie den Übertrag und die rechte

					Addition		Subtraktion	
a_{n-1}	b_{n-1}	a_{n-2}	b_{n-2}	c_{n-3}	unsigned	signed	unsigned	signed
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	0	0	1
0	0	0	1	0	0	0	0	1
0	0	0	1	1	1	0	0	0
0	0	1	0	0	1	0	1	0
0	0	1	0	1	1	0	1	1
0	0	1	1	0	1	0	1	1
0	0	1	1	1	0	1	0	1
0	1	0	0	0	0	0	0	0
0	1	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	0	1	1	0	1	0	0
0	1	1	0	0	0	1	0	0
0	1	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0	0
0	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	1	0
1	0	0	1	0	0	1	1	0
1	0	0	1	1	0	1	1	1
1	0	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0	0
1	0	1	1	0	0	1	0	0
1	0	1	1	1	0	0	1	0
1	1	0	0	0	0	0	1	1
1	1	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	0	1	0	0	1	1
1	1	1	1	0	0	0	1	1
1	1	1	1	1	0	1	0	1

Tabelle 2.9: Fehlerfunktion über n-1 Bit

Seite die Bewertung der Richtigkeit des Ergebnisses in Abhängigkeit der Operation (Addition + oder Subtraktion-) und der Zahlenrepräsentation (vorzeichenlos oder vorzeichenbehaftet).

Aus den Tabellen 2.6 bis 2.8 wurden dann die Operanden und die Bewertungen in Tabelle 2.9 zusammengefasst, um dass Ablesen einer Fehlerfunktion zu ermöglichen.

2.5.3 Operatoren mit mehr als zwei Operanden

Operatoren mit mehr als zwei Operanden wurden bisher nicht implementiert. Denn diese sind nur dann erforderlich, wenn alle Operanden mit der gleichen Verzögerung eintreffen. Das tritt in den seltensten Fällen auf. Daher können z.B. bei der Addition zwei Addierer verwendet werden, ohne dass sich die Verzögerung vergrößert.

2.6 Multiplikation

Bei der Multiplikation sind zwei Fälle zu unterscheiden. Einerseits die lineare Operation "Verstärkung", also Multiplikation einer Variablen mit einer Konstanten und andererseits die Multiplikation von zwei Variablen. Da der zweite Fall auf dem Verstärker aufbaut, wird dieser zuerst vorgestellt.

2.6.1 Verstärker

Ziel ist die Multiplikation des variablen Faktors X mit dem konstanten Faktor K. Dies wird durch das Aufsummieren eines Partialprodukts erreicht.

$$P^{(i+1)} = (P^{(i)} + x_i \cdot K)\, div\, 2 \tag{2.12}$$

Dabei wird von

$$P^{(0)} = 0 \tag{2.13}$$

ausgegangen. Die Ergebnisbits ergeben sich wie folgt:

$$p_i = (P^{(i)} + x_i \cdot K)\, mod\, 2 \tag{2.14}$$

Die Multiplikation in Gleichung 2.12 kann aufgelöst werden in:

$$x_i \cdot K = \sum_{j=0}^{n-1} x_i \cdot k_j \cdot 2^j \tag{2.15}$$

Da x_i und k_j binäre Zahlen repräsentieren, kann deren Produkt durch eine Konjunktion ersetzt werden.

$$x_i \cdot K = \sum_{j=0}^{n-1} x_i \wedge k_j \cdot 2^j \tag{2.16}$$

Bild 2.4 zeigt die Implementation eines seriellen Verstärkers. Die UND-Gatter zwischen den Addierern dienen zur Sicherstellung von Gleichung 2.13 während die Gatter zwischen Eingang und Addierer Gleichung 2.16 realisieren.

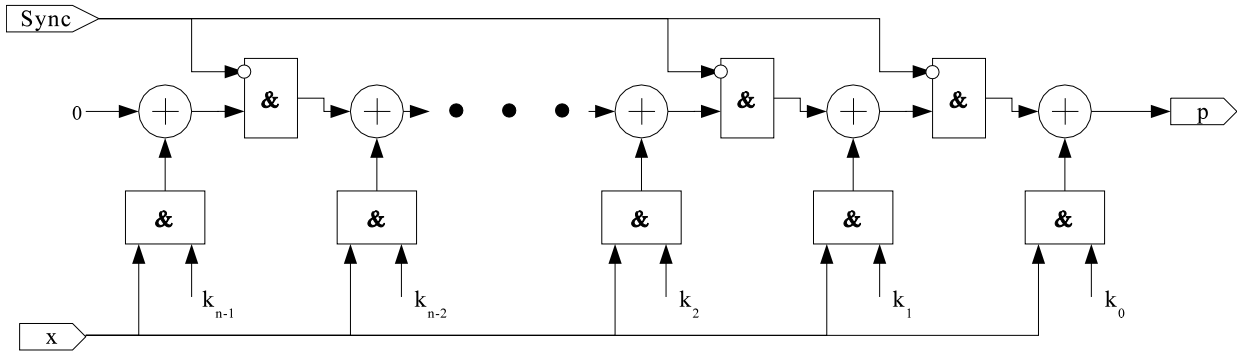

Abbildung 2.4: Serieller Verstärker

Das vorgestellte Verfahren erlaubt nur die Verwendung eines ganzzahligen Verstärkungsfaktors. Durch Weglassen von m-niederwertigen Bits des Ergebnisses P ist die Implementierung von Verstärkungsfaktoren $K = k/2^m$ möglich. Da nun nicht nur n, sondern n+m Bits zu verarbeiten sind, werden zwei Verstärker benötigt, die, wie in Bild 2.5 gezeigt, jeweils abwechselnd einen Operanden verarbeiten. Dazu werden an den Wortgrenzen des Eingangssignals x jeweils die linken Schalter umgelegt. Der rechte Schalter dient zur Auswahl des Rechenergebnisses und wird immer so geschaltet, dass die n-niederwertigen Bits nicht ausgegeben werden.

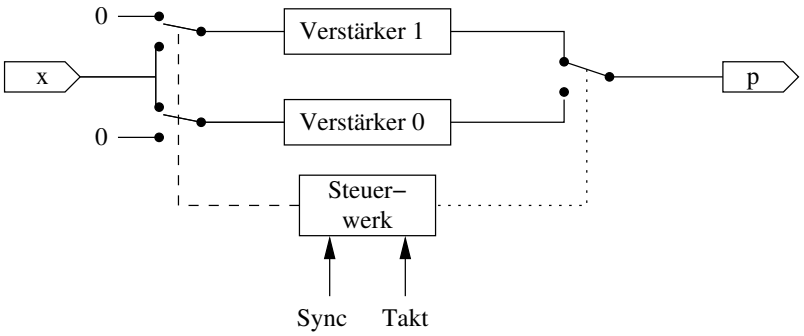

Abbildung 2.5: Interne Skalierung

Der skalierende Verstärker kann auch zur Vermeidung von Überläufen verwendet werden, dann ist jedoch der Skalierungsfaktor 2^m im weiteren Verarbeitungsverlauf zu berücksichtigen.

2.6.2 Div 2^k

In einigen Fällen reicht es nur die unteren k-Bit wegzulassen. Dafür wurde der Div 2^k-Operator implementiert.

2.6.3 Modulo 2^k

Das Modulo ist der ganzzahlige Rest der Division. Der Modulo 2^k-Operator lässt nur die k-niederwertigen Bits durch und schneidet die höherwertigen ab.

2.6.4 Multiplizierer

Bei der Verwendung von n Bit langen Operanden, kann durch die Multiplikation prinzipiell ein 2n Bit langes Ergebnis entstehen. Da dies nicht in den Übertragungsrahmen passen würde, ist eine Skalierung vorzusehen. Dazu gibt es zwei Möglichkeiten. Einerseits kann das komplette Ergebnis berechnet werden, um dann einige niederwertige Bits wegzulassen, siehe skalierender Verstärker im Abschnitt 2.6.1. Andererseits kann gleich auf die Berechnung der niederwertigen Bits verzichtet werden. Durch das Weglassen der niederwertigen Bits entsteht auch kein Übertrag, so dass die Genauigkeit leidet.

Bild 2.6 zeigt den prinzipiellen Aufbau eines Multiplizierers. Zur Implementation wurden die Gleichungen aus Abschnitt 2.6.1 verwendet, jedoch kommt anstelle der Konstanten K ein zweiter variabler Operand y zum Einsatz. Dieser wird mit Hilfe eines Schieberegisters seriell-parallel gewandelt und während der Verarbeitung in einem Register zwischengespeichert. Dadurch kommt es zur zeitlichen Verschiebung dieses Faktors um eine Tastperiode, und ist u.U. durch Einfügen eines Schieberegisters in den Weg des anderen Faktors auszugleichen.

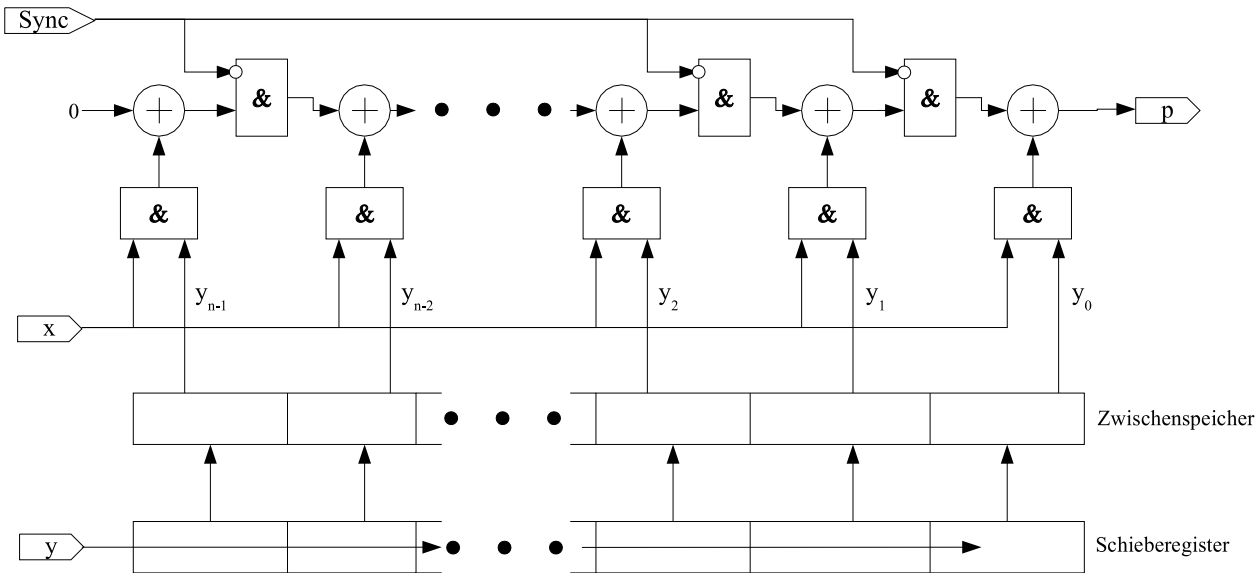

Abbildung 2.6: Multiplizierer

Zur Skalierung in den Wertebereich wird dasselbe Prinzip wie bei dem Verstärker im Bild 2.5 angewendet.

2.7 Division

Die Division von zwei Variablen konnte nicht realisiert werden, da bekannte Algorithmen mehr als n Takte zur Berechnung eines jeden Ergebnisbits benötigen. Die Verarbeitung eines Operanden würde damit n^2 Takte benötigen und somit zu einer zunehmenden Warteschlagenlänge führen (siehe Abschnitt 2.1). Alternativ kommt die Multiplikation mit dem Reziproken des Divisors in Frage. Der reziproke Wert ist dafür mit einer Kennlinie zu interpolieren.

2.8 Sonstige Funktionen

Zur Berechnung der Funktionswerte von transzendenten Funktionen gibt es verschiedene Möglichkeiten:

- Taylor-Reihe
- Iteration mit der Konvergenz zu einem bestimmten Wert, dieser ist oft 0 oder 1. D.h., die Algorithmen enthalten zwei oder mehr rekursive Formeln derart, dass eine Formel sich einem bestimmten konstanten Wert annähert, während dann die anderen das Ergebnis liefern.
- Approximation mit gebrochenen rationalen Funktionen.

Anhand der Exponentialfunktion wird auf die Probleme der unterschiedlichen Verfahren eingegangen. Die Taylor-Reihe für die e-Funktion lautet:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Diese Reihe konvergiert gut für gebrochene Argumente, benötigt jedoch eine große Anzahl von Schritten für x nahe 1 [Kor93, Kap. 9].

Eine weitere Möglichkeit besteht im Ersetzen von:

$$e^x = 2^{x log_2 e}$$

und teilen des Exponenten in einen ganzzahligen Teil I und einen gebrochenen Teil f z.B. $x\log_2 e = I + f$. Damit wird:

$$e^x = 2^I + 2^f$$

und die Realisierung von 2^I sehr einfach. Die Berechnung von 2^f kann durch eine gebrochen rationale Approximation erfolgen. Was jedoch wieder eine Division erfordert und daher nicht

im Rahmen des bitseriellen Ansatzes realisierbar ist. Alternativ kann die Berechnung mittels eines Konvergenzverfahrens erfolgen. Für ein gebrochenes x_0 kann $y = e^{x_0}$ wie folgt berechnet werden:

$$x_{i+1} = x_i - ln\, b_i \tag{2.17}$$

$$y_{i+1} = y_i \cdot b_i \tag{2.18}$$

Die b_i werden derart gewählt, dass:

$$x_{i+1} = x_0 - \sum_{l=0}^{i} ln\, b_l \rightarrow 0$$

Unabhängig davon, dass in Gleichung 2.17 ein Logarithmus zur Anwendung kommt, verhindert hier die variable Länge der Rekursion in Verbindung mit dem starren Übertragungsrahmen (siehe Bild 2.2) die Benutzung des Verfahrens. Bei den anderen transzendenten Funktionen kommen ähnlich gelagerte Algorithmen zur Anwendung, so dass deren Implementierung hier auch nicht möglich ist.

2.9 Kennlinienapproximation

Viele Funktionen in Steuerungen und Regelungen werden durch Kennlinien approximiert. Dabei erfolgt die Annäherung der nichtlinearen Funktionen mit linearen Funktionsabschnitten folgendermaßen:

$$y = \frac{(y_{i+1} - y_i) \cdot (x - x_i)}{x_{i+1} - x_i} + y_i$$

Diese Gleichung kann vereinfacht werden, indem man von einer äquidistanten Stützstellenweite ausgeht, die als Potenz von zwei $x_{i+1} - x_i = 2^k$ gewählt wird. Damit reduziert sich die Berechnung des Anstiegs auf eine Subtraktion mit anschließendem Abschneiden der unteren k Bit (div 2^k) und die Berechnung der Position im Intervall auf das Abschneiden der n-k höherwertigen Bit, also einem Modulo 2^k.

$$y = [(y_{i+1} - y_i) \cdot x \bmod 2^k] \operatorname{div} 2^k + y_i$$

Zur Berechnung von $\Delta y_i = y_{i+1} - y_i$ müssten gleichzeitig y_i und y_{i+1} aus der Tabelle mit den Stützstellen ausgelesen werden, damit es bei der Subtraktion nicht zu einer zusätzlichen Verzögerung um einem Takt kommt. Das erfordert jedoch den lesenden Zugriff auf zwei Speicherplätze gleichzeitig, was die Halbierung des Verarbeitungstaktes erzwingt. Denn nur so sind zwei Zugriffe auf den Stützstellenspeicher innerhalb eines Verarbeitungstaktes möglich sind. Allerdings halbiert sich so auch der Datendurchsatz. Eine andere Variante ist das doppelte Abspeichern der Stützstellen in zwei Tabellen. Dann ist es jedoch sinnvoller, in der zweiten Tabelle gleich

die Anstiege Δy_i abzulegen. Deren Berechnung erfolgt dann schon während der Elaboration (siehe Abschnitt 1.6 auf Seite 13). Das zusätzliche Abspeichern geht jedoch mit einem erhöhten Logikaufwand einher. Gleichung 2.19 stellt die tatsächlich implementierte Formel dar.

$$y = [\Delta y_i \cdot x \bmod 2^k] \operatorname{div} 2^k + y_i \tag{2.19}$$

Für den Anstieg gilt $\Delta = \frac{\Delta y}{2^k} = \tan\alpha$ mit α als Winkel für den Anstieg. Da grundsätzlich mit ganzen Zahlen gerechnet wird, wäre z.B. im Bereich $0° \leq \alpha < 45°$ $\Delta = 0$, was die Genauigkeit verschlechtert. Durch die Verlagerung der Division durch 2^k im Anschluss an die Multiplikation wird die Genauigkeit erhöht. Denn für die Multiplikation ergibt sich: $\Delta y = 2^k \Delta = 2^k \tan\alpha$ und für $\Delta y = 1$ reicht $1 = 2^k \tan\alpha_{min} = \frac{2^k}{2^k}$ mit $\tan\alpha_{min} = \frac{1}{2^k}$ oder $\alpha_{min} = \arctan\frac{1}{2^k}$. Mit anderen Worten, ein größeres k (Stützstellenweite) erhöht die Genauigkeit. Dies klingt widersprüchlich, da eine große Stützstellenweite mit größeren Fehlern verbunden wird. Aber da $\Delta y \sim \Delta$ ist und es durch die späte Division einen größeren Wertebereich ausnutzt, erklärt sich die Erhöhung der Genauigkeit. Damit stellt sich die Frage, wie k zu bestimmen ist. Mit x_{max} als größtem Wert, der am Eingang einer Kennlinienapproximation mit l-Stützstellen eintreffen kann, gilt $x_{max} \leq (l-1)\, 2^k - 1$. Da sowohl ein großes l als auch ein großes k die Genauigkeit erhöhen, muss x möglichst so skaliert werden, dass es einen großen Wertebereich umfasst.

Bild 2.7 zeigt den Signalfluss für die Kennlinienapproximation. Die Bestimmung des Intervalls i wird durch Abschneiden der unteren k Bit realisiert. Das erfolgt bei der seriell-parallel-Wandlung des Operanden, welche zur Adressierung der Speicherplätze von y_i und Δy_i erforderlich ist. Die abgeschnittenen k Bit entsprechen der Stützstellenweite. Um den Zuwachs im Intervall zu berechnen, wird der Anstieg mit der Position im Intervall multipliziert. Dabei wird Δy_i parallel weiterverarbeitet. Dazu wurde ein spezieller Multiplizierer entwickelt, der einen parallelen Operanden erwartet. Weiterhin nimmt er auch noch die Division durch 2^k vor. Anschließend erfolgt die Addition von y_i.

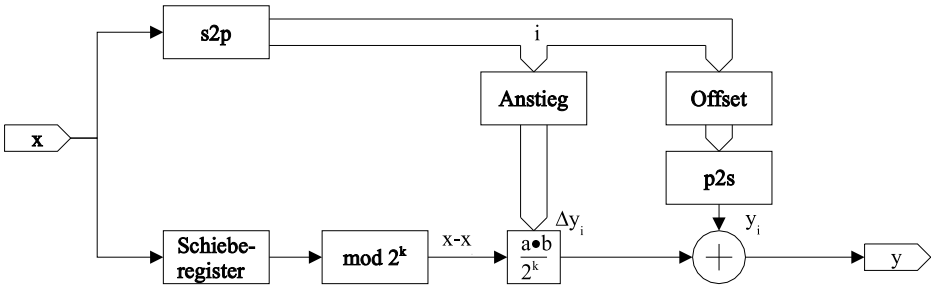

Abbildung 2.7: Signalflussplan Kennlinienapproximation

Ablage der Stützstellen

Im Package `klmemtable` sind die Stützpunkte in einem Feld (`rom`) abgelegt. Das Feld kann alle Kennlinien, die im Design Verwendung finden, aufnehmen. Jede Kennlinie belegt dort eine Zeile.

Diese wird über einen Index ausgewählt. Dieser Index ist bei der Instanzierung zu übergeben. Alle Zeilen des Feldes haben die gleiche Länge, die sich aus der Kennlinie mit der größten Anzahl Stützstellen ergibt. Die tatsächliche Anzahl der Stützstellen l ist in einem weiteren Feld (`rom_length`) abgespeichert. Bei der Elaboration werden aus dem Feld `rom` die über den Index angegebenen Stützstellen ausgelesen und in eine eigene Tabelle übernommen. Diese hat dann die Länge $l-1$. Zur Berechnung der Anstiege sind l-Stützstellen notwendig, jedoch zur Interpolation nur noch $l-1$. Desweiteren wird bei der Elaboration noch ein Feld mit $l-1$ Elementen angelegt, welches die Anstiege zu jedem Stützpunkt enthält. Ein Beispielpackage ist im Anhang B.1 zu finden.

2.10 Vergleicher, Komparator

Im Komparator werden alle möglichen Vergleichsergebnisse zwischen zwei entweder vorzeichenlosen oder vorzeichenorientierten Operanden berechnet. Dazu wird deren Differenz berechnet und ausgewertet. In die Auswertung gehen die Differenz der MSBs (Diff), ein eventuell geborgtes Bit (Borrow ①) und ob beide Zahlen gleich sind (Zero), ein. Tabelle 2.10 zeigt alle möglichen Kombinationen von je zwei MSBs beider Operanden und die sich daraus ergebenen Vergleichsergebnisse. Im vorzeichenlosen Fall gilt für alle Felder unterhalb der Hauptdiagonalen $a < b$ und über der Hauptdiagonalen $a > b$. Diese Eindeutigkeit ist leider im vorzeichenorientierten Fall für die Felder auf der Nebendiagonalen direkt neben der Hauptdiagonalen nicht gegeben.

Da die Operanden nicht in die Logikfunktion eingehen, wurde zu deren Herleitung noch Tabelle 2.11 aufgestellt. Dort findet sich auch die Mehrdeutigkeit im vorzeichenorientierten Fall wieder. Diese lässt sich durch Einbeziehung der Operanden MSBs auflösen. Für den Fall $a_{MSB} \neq b_{MSB}$ ergibt sich $y_{a>b} = b_{MSB}$ und $y_{a<b} = a_{MSB}$.

Der Komparator benötigt keine Fehlerbehandlung, da ein Überlauf lediglich auf die Größenverhältnisse zwischen den Operanden verweist, aber keinen Fehler darstellt.

2.11 Begrenzer

Der Begrenzer dient zur Begrenzung der Werte auf den halben Wertebereich $-2^{n-2} \leq u < 2^{n-2}$. Damit kann eine Fehlervermeidung, z.B. in einem Integrator, vorgenommen werden. Der Begrenzer überprüft anhand einer Fehlerfunktion (siehe Tabelle 2.12) ob ein Wert den zulässigen Wertebereich überschreitet und gibt im Fehlerfall einen Ersatzwert aus.

Abbildung 2.8 zeigt den prinzipiellen Aufbau des Begrenzers und Abbildung 2.9 die Anwendung im Integrator. Durch die Verwendung des Begrenzers nutzt die Rückkopplung nur den halben Wertebereich aus. Wenn nun auch u auf den halben Wertebereich skaliert wurde, kann es am

a	0 0	0 1	1 0	1 1
b	-0 0	-0 0	-0 0	-0 0
	0 0	0 1	1 0	1 1
a>b	0	1	1	1
a<b	0	0	0	0
z=0	1	0	1	0

a	0 0	0 1	1 0	1 1
b	-0 1	-0 1	-0 1	-0 1
	① 1 1	0 0	0 1	1 0
a>b	0	0	1	1
a<b	1	0	0	0
z=0	0	1	0	1

a	0 0	0 1	1 0	1 1
b	-1 0	-1 0	-1 0	-1 0
	① 1 0	① 1 1	0 0	0 1
a>b	0	0	0	1
a<b	1	1	0	0
z=0	1	0	1	0

a	0 0	0 1	1 0	1 1
b	-1 1	-1 1	-1 1	-1 1
	① 0 1	① 1 0	① 1 1	0 0
a>b	0	0	0	0
a<b	1	1	1	0
z=0	0	1	0	1

a) vorzeichenlos

a	0 0	0 1	1 0	1 1
b	-0 0	-0 0	-0 0	-0 0
	0 0	0 1	1 0	1 1
a>b	0	1	0	0
a<b	0	0	1	1
z=0	1	0	1	0

a	0 0	0 1	1 0	1 1
b	-0 1	-0 1	-0 1	-0 1
	① 1 1	0 0	0 1	1 0
a>b	0	0	0	0
a<b	1	0	1	1
z=0	0	1	0	1

a	0 0	0 1	1 0	1 1
b	-1 0	-1 0	-1 0	-1 0
	① 1 0	① 1 1	0 0	0 1
a>b	1	1	0	1
a<b	0	0	0	0
z=0	1	0	1	0

a	0 0	0 1	1 0	1 1
b	-1 1	-1 1	-1 1	-1 1
	① 0 1	① 1 0	① 1 1	0 0
a>b	1	1	0	0
a<b	0	0	1	0
z=0	0	1	0	1

b) vorzeichenorientiert

Tabelle 2.10: Zusammenhang Rechenergebnis und Vergleichsergebnis

			vorzeichenlos			vorzeichenorientiert $a_{MSB} = b_{MSB}$		
$Zero_n$	Diff	Borrow	a>b	a<b	$Zero_{n+1}$	a>b	a<b	$Zero_{n+1}$
0	0	0	1	0	0	1	0	0
0	0	1	0	1	0	1	0	0
0	1	0	1	0	0	0	1	0
0	1	1	0	1	0	0	1	0
1	0	0	0	0	1	0	0	1
1	0	1	-	-	-	-	-	-
1	1	0	1	0	0	0	1	0
1	1	1	0	1	0	1	0	0

Tabelle 2.11: Logiktabelle für die Vergleichsergebnisse am Komparator

Addierer nie zu einem Fehler kommen. Da es sich bei der Begrenzung um eine nichtlineare Operation handelt, ergibt sich immer eine Zeitverschiebung um eine Tastperiode zwischen Eingangs-

u 2^{n-1}	u 2^{n-2}	Begrenzen
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 2.12: Fehlerfunktion für vorzeichenbehaftete Werte im Begrenzer

und Ausgangssignal. Da die niederwertigen Bits erst ausgegeben werden können, nachdem alle Bits eingelesen wurden. Diese Verschiebung dient als Ersatz für die Operation z^{-1}.

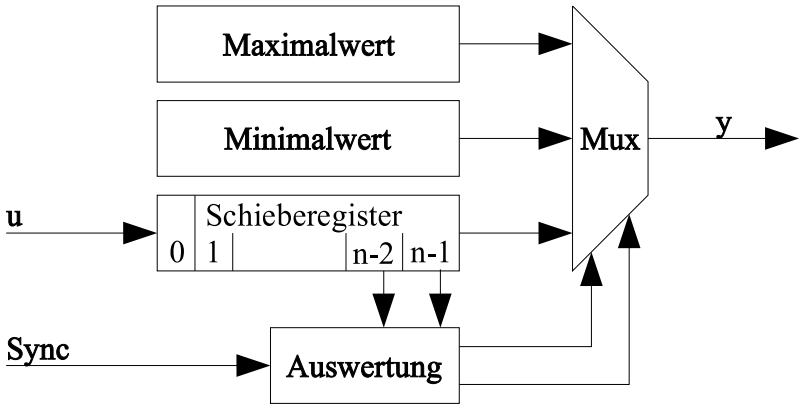

Abbildung 2.8: Aufbau Begrenzer

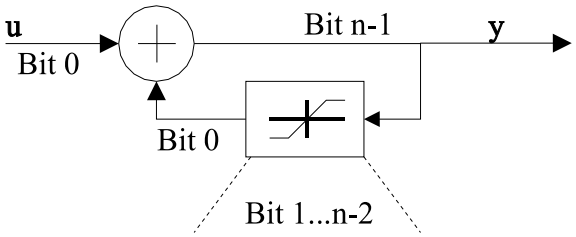

Abbildung 2.9: Anwendungsbeispiel Begrenzer

Kapitel 3

Bitserielle Komponenten

Dieses Kapitel stellt Blöcke vor, die aus den im Kapitel 2 gezeigten arithmetischen Grundkomponenten zusammengesetzt sind.

3.1 Randbedingungen

Die höhere Komplexität dieser Komponenten erlaubt aus sich heraus prinzipiell verschiedene Signalpfade innerhalb einer Komponente. Diese treffen an verschiedenen Stellen wieder aufeinander siehe Bild 3.1.

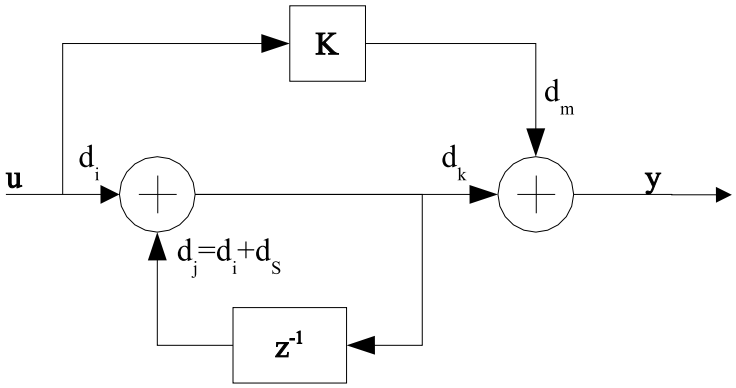

Abbildung 3.1: Mögliche Signalpfade in einer Komponente

Dies kann in einem Vorwärtszweig geschehen wie im Bild über den Addierer und den Verstärker K. Die Verzögerung in einem Signalpfad sei d, treffen zwei Operanden mit den Verzögerungen $\mathrm{d_k}$ und $\mathrm{d_m}$ aufeinander, ist im Operator

$$d_k \bmod n = d_m \bmod n$$

zu gewährleisten. n ist die Verarbeitungswortbreite. Dabei sind Schleifen, im Bild z.B. über z^{-1}, als besondere Pfade zu betrachten, denn mit $d_j = d_i + d_S$ gilt für

$$\begin{aligned} d_i \bmod n &= d_j \bmod n \\ &= (d_i + d_S) \bmod n \end{aligned}$$

und unter Berücksichtigung von

$$0 \leq x \bmod n = x - j \cdot n < n \quad \text{mit } j = 0,1,2,\ldots$$

gilt wiederum

$$\begin{aligned} d_i - j_1 n &= d_i + d_S - j_2 \cdot n \\ d_S &= j_2 \cdot n - j_1 \cdot n \\ &= (j_2 - j_1) n \end{aligned} \tag{3.1}$$

Da sowohl j_1 als auch j_2 natürliche Zahlen sind, gilt das für deren Differenz auch. Somit dürfen in Schleifen nur Verzögerungen auftreten, die ein ganzes Vielfaches der Verarbeitungswortbreite sind. Damit ist die Schleifenbedingung strenger.

3.2 Beschreibungsmittel

Zuerst werden einige Grundlagen digitaler lineare Systeme vorgestellt und dann vor diesem Hintergrund die Implementierung verschiedener Regler oder Filter erläutert. Digitale lineare Systeme lassen sich u.a. mittels Differenzengleichungen, Zustandsraumbeschreibung und z-Übertragungsfunktion beschreiben. Je nach Art und Weise der Besetzung der Matrizen und Vektoren im Zustandsraum spricht man von verschiedenen Normalformen. Diese sind zueinander äquivalent, können also ineinander überführt werden. Siehe dazu [Hos94] und [Föl93]. Im folgenden werden nur Systeme mit einem Eingang u und einem Ausgang y betrachtet.

3.2.1 Differenzengleichung

Die Differenzengleichung beschreibt die Vorgänge mit Hilfe von Werten, die von verschiedenen Abtastzeitpunkten $k-i$ resultieren. Die allgemeine Form der Differenzengleichung lautet:

$$y(k) = f[u(k),\ldots u(k-i),\ldots u(k-m), y(k-1),\ldots y(k-i),\ldots y(k-m)] \tag{3.2}$$

wobei m die Systemordnung darstellt. In dieser Gleichung ist $k-i$ die Verschiebung eines Wertes

um $i \cdot \Delta T$ (ΔT Tastperiode) in die Vergangenheit und somit $\mathrm{u(k)}$ der aktuellste Eingangswert. Diese Verschiebung wird mit Hilfe einer Speicherzelle vorgenommen, welche damit den Zustand des Systems speichert. In linearer Form sieht Gleichung 3.2 folgendermaßen aus:

$$\begin{aligned} y(k) &= b_m \cdot u(k) + b_{m-1} \cdot u(k-1) + \cdots + b_0 \cdot u(k-m) \\ &\quad -a_{m-1} \cdot y(k-1) - a_{m-2} \cdot y(k-2) - \cdots - a_1 \cdot y(k-m+1) - a_0 \cdot y(k-m) \end{aligned} \tag{3.3}$$

3.2.2 z-Übertragungsfunktion

Die z-Übertragungsfunktion $G(z) = y(z)/u(z)$ ist eine Abbildung der Differenzengleichung in den Bildbereich z. Dies wird erreicht, indem man der Verschiebung $k-i$ im Zeitbereich eine Potenz z^i im Bildbereich zuordnet. Damit ändert sich Gleichung 3.3 wie folgt:

$$\begin{aligned} y &= b_m u + b_{m-1} u \cdot z^{-1} + \cdots + b_0 u \cdot z^{-m} + \\ &\quad -a_{m-1} y \cdot z^{-1} - a_{m-2} y \cdot z^{-2} - \cdots - a_0 y \cdot z^{-m} \quad | \cdot z^m \\ G(z) = \frac{y}{u} &= \frac{b_m z^m + b_{m-1} z^{m-1} + \cdots + b_1 z + b_0}{z^m + a_{m-1} z^{m-1} + \cdots + a_1 z + a_0} \end{aligned} \tag{3.4}$$

Diese Form ist sowohl gut zur Repräsentation der Beobachter- als auch der Regelungsnormalform geeignet. Bilder 3.3 und 3.4 zeigen die Strukturen der beiden Realisierungsformen.

3.2.3 Zustandsraummodell

Durch explizites Notieren der Gleichungen zur Berechnung der Zustände und deren Eingangs- und Ausgangskopplungen, lässt sich das Zustandsraummodell (ZRM) gemäß Gleichung 3.5 und 3.6 gewinnen. Tabelle 3.1 zeigt noch einmal die Matrizen und Vektoren.

$$\mathbf{x}(k+1) = \mathbf{A} \cdot \mathbf{x}(k) + \mathbf{b} \cdot u(k) \tag{3.5}$$

$$y(k) = \mathbf{c} \cdot \mathbf{x}(k) + d \cdot u(k) \tag{3.6}$$

Im allgemeinen sind diese z.B. aus der Systemanalyse gewonnenen Matrizen und Vektoren des ZRM sehr unregelmäßig besetzt und lassen nur schwer Rückschlüsse auf das System zu. Für dessen Untersuchung und den Reglerentwurf wurden verschiedenen Normalformen entwickelt, die sich dazu besser eignen. Verfahren zur Transformation in die verschiedenen Normalformen sind z.B. in [Föl90, Föl93] zu finden.

Eine Fehlerbehandlung kann durch eine geeignete Skalierung des Zustandsvektors $\mathbf{x}$, des Eingangssignals u und des Ausgangssignals y vermieden werden. Mit den Skalierungsfaktoren $\mathrm{S_u}$,

Signal/Koeffizient	Definition
Zustandsvektor	$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$
Systemmatrix	$\mathbf{A} = \begin{bmatrix} \overline{a}_{11} & \overline{a}_{12} & \cdots & \overline{a}_{1m} \\ \overline{a}_{21} & \overline{a}_{22} & \cdots & \overline{a}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \overline{a}_{n1} & \overline{a}_{n2} & \cdots & \overline{a}_{m} \end{bmatrix}$
Eingangskopplungen	$\mathbf{b} = \begin{bmatrix} \overline{b}_1 \\ \overline{b}_2 \\ \vdots \\ \overline{b}_m \end{bmatrix}$
Ausgangskopplungen	$\mathbf{c} = \begin{bmatrix} \overline{c}_1 & \overline{c}_2 & \cdots & \overline{c}_m \end{bmatrix}$
Durchgriff	d

Tabelle 3.1: Definition der Signale und Koeffizienten im Zustandsraummodell

$\mathbf{S}_\mathrm{x}$, S_y und dem folgendem Ansatz

$$u = S_u \hat{u} \tag{3.7}$$

$$\mathbf{x} = \mathbf{S}_\mathrm{x} \hat{\mathbf{x}} \tag{3.8}$$

$$y = S_y \hat{y} \tag{3.9}$$

ergibt sich für das Zustandsraummodell

$$\begin{aligned} \mathbf{S}_\mathrm{x} \cdot \hat{\mathbf{x}}(k+1) &= \mathbf{A} \cdot \mathbf{S}_\mathrm{x} \cdot \hat{\mathbf{x}}(k) + \mathbf{b} \cdot S_u \cdot \hat{u}(k) \quad | \ \mathbf{S}_\mathrm{x}^{-1} \cdot \\ S_y \cdot \hat{y}(k) &= \mathbf{c} \cdot \mathbf{S}_\mathrm{x} \cdot \hat{\mathbf{x}}(k) + d \cdot S_u \cdot \hat{u}(k) \quad | : S_y \end{aligned}$$

$$\begin{aligned} \hat{\mathbf{x}}(k+1) &= \mathbf{S}_\mathrm{x}^{-1} \cdot \mathbf{A} \cdot \mathbf{S}_\mathrm{x} \cdot \hat{\mathbf{x}}(k) + \mathbf{S}_\mathrm{x}^{-1} \cdot \mathbf{b} \cdot S_u \cdot \hat{u}(k) \\ \hat{y}(k) &= \frac{1}{S_y} \cdot \mathbf{c} \cdot \mathbf{S}_\mathrm{x} \cdot \hat{\mathbf{x}}(k) + \frac{d}{S_y} \cdot S_u \cdot \hat{u}(k) \end{aligned} \tag{3.10}$$

Damit sehen die neuen Matrizen und Vektoren wie folgt aus

$$\begin{aligned} \hat{\mathbf{A}} &= \mathbf{S}_\mathrm{x}^{-1} \cdot \mathbf{A} \cdot \mathbf{S}_\mathrm{x} \\ \hat{\mathbf{b}} &= \mathbf{S}_\mathrm{x}^{-1} \cdot \mathbf{b} \cdot S_u \\ \hat{\mathbf{c}} &= \frac{1}{S_y} \cdot \mathbf{c} \cdot \mathbf{S}_\mathrm{x} \\ \hat{d} &= \frac{d}{S_y} \cdot S_u \end{aligned}$$

Zur Berechnung der Skalierungsfaktoren wird allgemein der Quotient aus einem aus der Simulation gewonnen Maximalwert (z.B. u_{max}) und einer Vorgabe (z.B. $\hat{u}_{max}$) gebildet. Damit gilt für S_u aus Gleichung 3.7

$$S_u = \frac{u_{max}}{\hat{u}_{max}}$$

Prinzipiell kann nach einer Erweiterung von Gleichung 3.8 mit $\mathbf{I}$, der Vorgabe von $\hat{\mathbf{x}}_{max}$ und einem aus der Simulation gewonnen $\mathbf{x}_{max}$ $\mathbf{S}_x$ folgendermaßen bestimmt werden

$$\begin{aligned} \mathbf{x}_{max} \cdot \mathbf{I} &= \mathbf{S}_x \cdot \hat{\mathbf{x}}_{max} \cdot \mathbf{I} \quad | \cdot (\hat{\mathbf{x}}_{max} \cdot \mathbf{I})^{-1} \\ \mathbf{S}_x &= \mathbf{x}_{max} \cdot \mathbf{I} \cdot (\hat{\mathbf{x}}_{max} \cdot \mathbf{I})^{-1} \end{aligned}$$

Jedoch birgt die Vektormultiplikation $\hat{\mathbf{A}} \cdot \hat{\mathbf{x}}$ die Möglichkeit von Partialsummenüberläufen in sich. Zum Teil ist es möglich, diese durch die Wahl einer geeigneten Form für $\hat{\mathbf{A}}$ zu vermeiden. Jedoch können auch bei der Berechnung von $\hat{y}$ Partialsummenüberläufe auftreten. Darum soll am Beispiel der Berechnung von $\mathbf{S}_y$ eine allgemeine Vorgehensweise zur Vermeidung von Partialsummenüberläufen gezeigt werden. Wenn davon ausgegangen wird, dass das Ausgangssignal

$$\hat{y} = \hat{y}_x + \hat{y}_u$$

die Summe von verstärktem Eingangssignal $\hat{y}_u = \hat{d} \cdot u$ und verstärktem Zustandsvektor $\hat{y}_x$ ist, gilt hier für $\hat{y}_x$, mit $\mathbf{s}_{x_{i,1\ldots m}}$ als der i-ten Zeile von $\mathbf{S}_x$

$$\hat{y}_x = \sum_{i=1}^{m} \hat{y}_{x_i} = \sum_{i=1}^{m} \frac{1}{S_y} \cdot c_i \cdot \mathbf{s}_{x_{i,1\ldots m}} \cdot \hat{\mathbf{x}}$$

S_y ist nun so zu bestimmen, dass gilt:

$$\hat{y}_{min} \leq \hat{y}, \hat{y}_x, \hat{y}_{x_i}, \hat{y}_u \leq \hat{y}_{max}.$$

Hier sind $\hat{y}_{min}$ und $\hat{y}_{max}$ wieder Vorgaben. Die Skalierung verschiebt jedoch nicht die Pole des Systems. Damit bleibt ein instabiles System wie ein I-Anteil instabil und bedarf einer Fehlerbehandlung. Bei den anderen Anteilen, kann so aber auf eine Fehlerbehandlung verzichtet werden und damit auch auf den damit verbundenen Zeitverlust.

3.2.3.1 Modale oder Jordansche Normalform

Durch die Partialbruchzerlegung der z-Übertragungsfunktion derart, dass

$$y(z) = \left[\sum_{i=1}^{m} \frac{K_i z}{z - z_i} + K_0 \right] \cdot u(z) \tag{3.11}$$

gilt, lassen sich die Matrix und Vektoren für die modale Normalform gewinnen.

$$\mathbf{A} = \begin{bmatrix} z_1 & 0 & \cdots & 0 \\ 0 & z_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & z_m \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_m \end{bmatrix} \quad d = K_0$$

Hier können z_i und K_I paarweise konjugiert komplex sein. Durch die diagonale Form von **A** sind die Zustandsvariablen x_i voneinander entkoppelt. Das heißt, die Signale durchlaufen parallele Signalwege und beeinflussen sich nicht gegenseitig. Wenn ein Regler auf Basis der modalen Normalform realisiert wird, sind bestimmten Verhaltensweisen ausgewählte Parameter zugeordnet (siehe Bild 3.2).

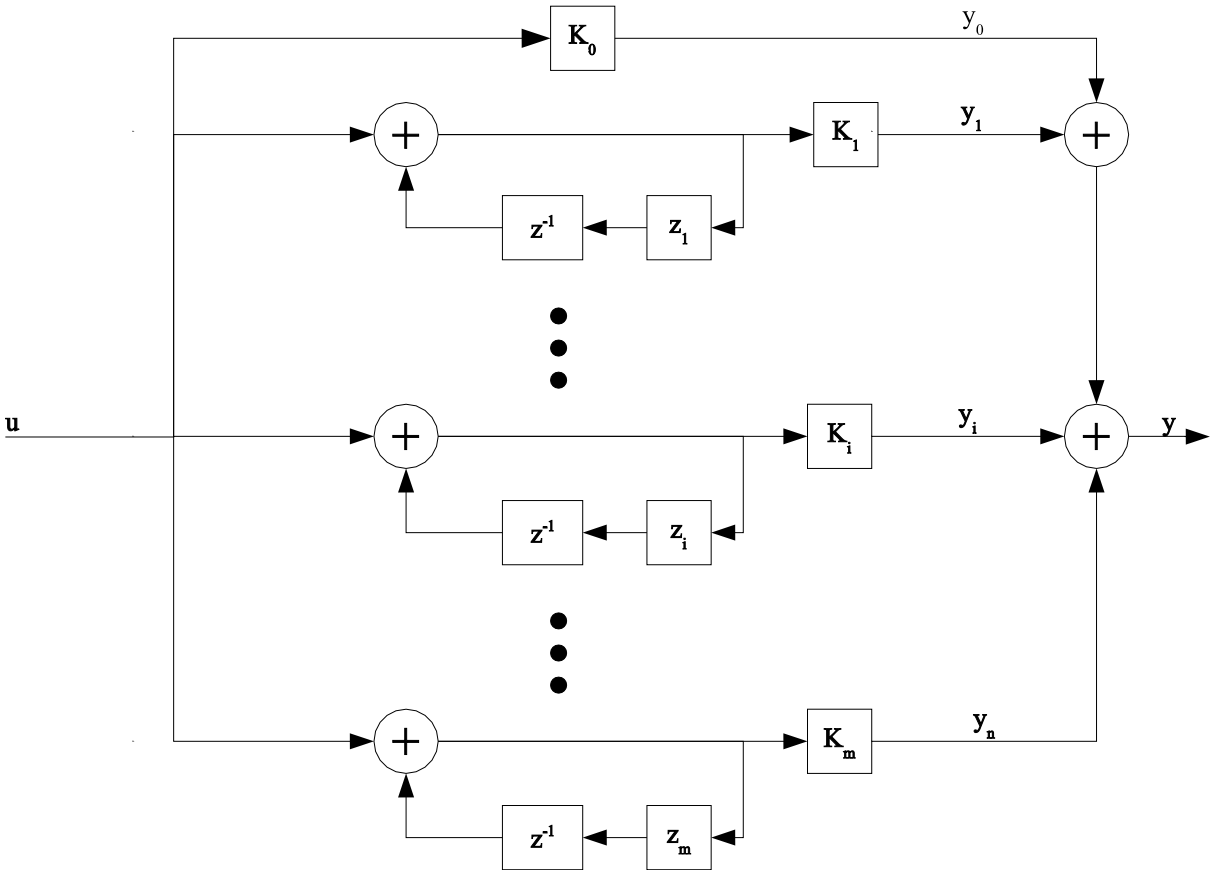

Abbildung 3.2: Blockschaltbild zur Realisierung in modaler Normalform

Für $z_i = 1$ stellt der entsprechende Anteil einen Integrator da (siehe Abschnitt 3.3.3). In diesem Fall ist eine Fehlerbehandlung vorzusehen.

3.2.3.2 Reihenschaltung

Alternativ zur Parallelschaltung bei der modalen Normalform gibt es noch die Reihenschaltung. Dazu muss Gleichung 3.4 getrennt für Zähler und Nenner in Produktterme zerlegt werden.

$$\begin{aligned}\frac{y}{u} &= \frac{(z-z_{z_m})(z-z_{z_{m-1}})\cdots(z-z_{z_0})}{(z-z_{n_m})(z-z_{n_{m-1}})\cdots(z-z_{n_0})} \\ &= G_m G_{m-1}\cdots G_0\end{aligned}$$

Jedes G_i stellt ein PT_1- oder maximal PT_2-Glied dar. Bei der Anordnung ist idealerweise das Element mit der größten Zeitkonstante an den Anfang zu stellen, um die Dynamik gleich einzuschränken. Dadurch kann der erforderliche Wertebereich klein gehalten werden.

3.2.3.3 Beobachternormalform

Eine weitere bekannte Normalform ist die Beobachternormalform. Im folgenden ist der Aufbau der Matrix und Vektoren zu sehen. Die dort auftretenden Werte a_i und b_i haben ihren Ursprung in Gleichung 3.3.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & & \vdots & \\ \vdots & & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & -a_{m-1} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_0 - a_0 b_m \\ b_1 - a_1 b_m \\ \vdots \\ b_{m-1} - a_{m-1} b_m \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad d = b_m$$

Bild 3.3 zeigt das Blockschaltbild zur Realisierung eines Systems in Beobachternormalform. Diese Normalform ist besonders gut zur Implementierung mit bitseriellen Algorithmen geeignet, da nach jedem Addierer eine Speicherzelle vorhanden ist. Anstelle der Speicherzelle kann ein Begrenzer verwendet werden, um einen Überlauf zu verhindern.

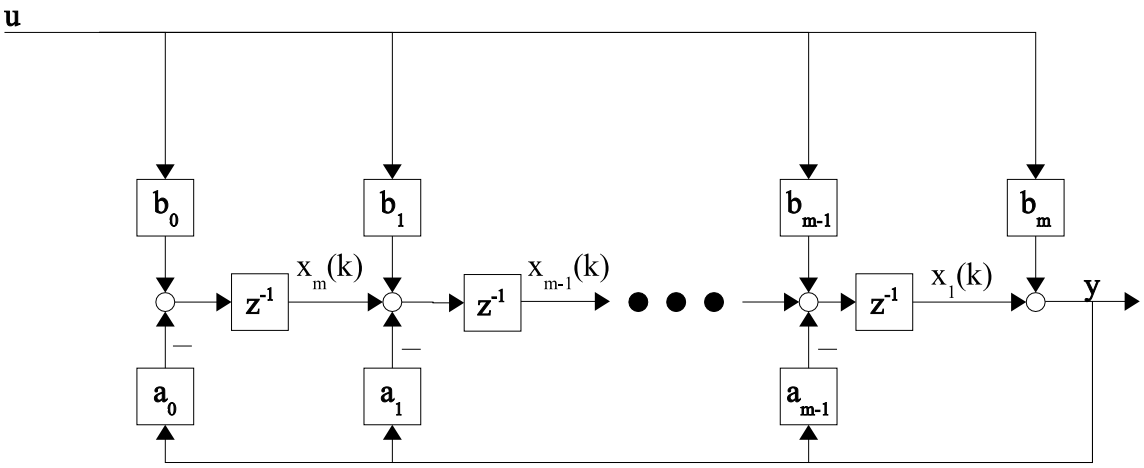

Abbildung 3.3: Blockschaltbild zur Realisierung in Beobachternormalform

3.2.3.4 Regelungsnormalform

Aufbau der Matrix und Vektoren bei der Regelungsnormalform

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & 0 \\ 0 & 0 & 0 & 0 & 1 \\ -a_0 & -a_1 & \cdots & -a_{m-2} & -a_{m-1} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} b_0 - a_0 \cdot b_m & b_1 - a_1 \cdot b_m & \ldots & b_{m-1} - a_{m-1} \cdot b_m \end{bmatrix}^{\mathrm{T}}$$

$$d = b_m$$

Bild 3.4 zeigt den Blockschaltplan zur Realisierung in Regelungsnormalform. Die Addiererketten am Eingang und am Ausgang erlauben keine Fehlerbehandlung ohne Zeitverlust. Darum ist die Verwendung der Regelungsnormalform an eine Skalierung gebunden, die die Fehlerfreiheit garantiert. Allerdings ist der Einsatz dieser Struktur problematisch, da sie sich nicht einfach in Atome zerlegen lässt.

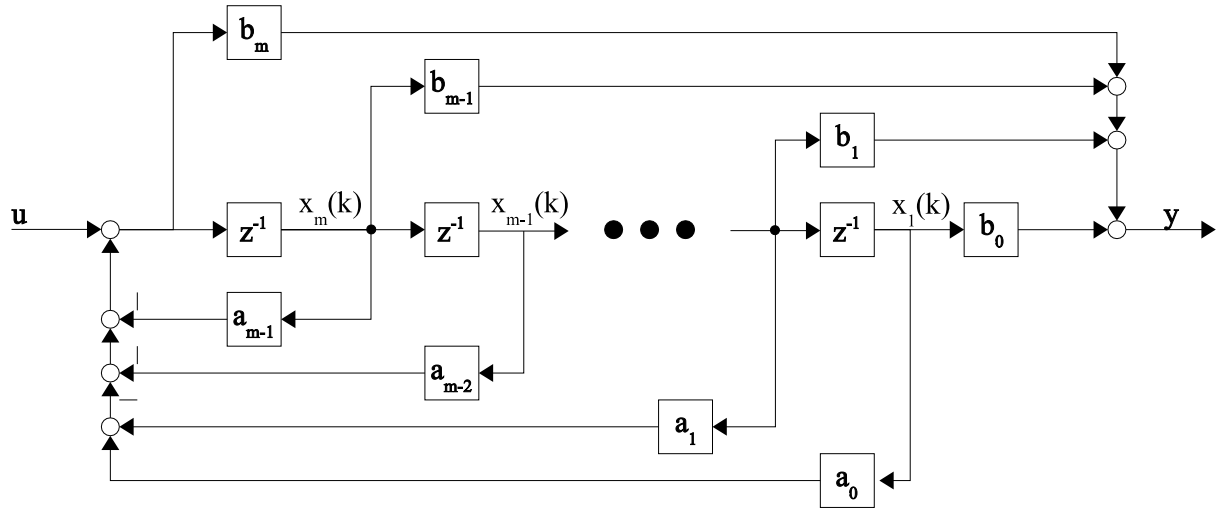

Abbildung 3.4: Blockschaltbild zur Realisierung in Regelungsnormalform

3.3 Elementare Blöcke

3.3.1 Speicherzelle

Die Speicherzelle dient zur Realisierung der Funktion $G(z) = \mathrm{z}^{-1}$.

$$\begin{aligned} y(z) &= u(z)z^{-1} \\ y(k) &= u(k-1) \end{aligned}$$

Die Verschiebung um eine Tastperiode wird durch ein Schieberegister mit der Länge der Verarbeitungswortbreite realisiert.

3.3.2 Differenzierer

In den meisten Fällen ist die Nachbildung eines Differenzierers mit dem Differenzenquotient ausreichend.

$$\begin{aligned} y(t) &= T_V \cdot \dot{u}(t) = T_V \frac{du(t)}{dt} \\ y(k) &= T_V \frac{u(k)-u(k-1)}{\Delta T} \\ y(z) &= K_D\left[u(z)-u(z)\cdot z^{-1}\right] \quad \text{mit } K_D = \frac{T_V}{\Delta T} \end{aligned} \tag{3.12}$$

Hier ist T_V die Vorstellzeit, damit ergibt sich folgende Übertragungsfunktion

$$G(z) = K_D \frac{z-1}{z}.$$

Die Umsetzung von Gleichung 3.12 bedeutet, dass vom aktuellen Wert der der vorherigen Tastperiode zu subtrahieren ist. Bild 3.5 zeigt den realisierten Differenzierer.

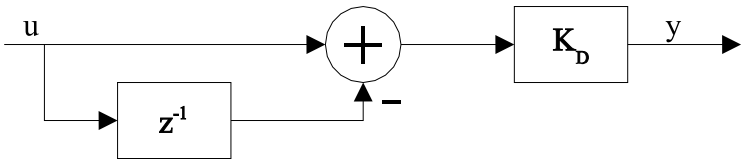

Abbildung 3.5: Differenzierer

3.3.3 Integrator

Zur digitalen Nachbildung der Integration gibt es verschiedene Verfahren, die z.B. auch in [Lut98] beschrieben sind. Bei hohen Abtastfrequenzen ist die Rechteckintegration vorwärts (Euler rechts) hinreichend genau und dient als Ausgangspunkt. T_N ist die Nachstellzeit.

$$\begin{aligned}
\dot{y}(t) &= \frac{1}{T_N} u(t) \\
\frac{y(k)-y(k-1)}{\Delta T} &= \frac{1}{T_N} u(k) \\
y(k) &= \frac{\Delta T}{T_N} u(k) + y(k-1) \\
y(z) &= K_I\, u(z) + y(z) \cdot z^{-1} \quad \text{mit } K_I = \frac{\Delta T}{T_N} && (3.13) \\
G(z) &= K_I \frac{z}{z-1} && (3.14)
\end{aligned}$$

Gleichung 3.13 entspricht einer um eine Tastperiode verzögerten Rückführung des Integratorsignals auf das Eingangssignal. Siehe Bild 3.6.

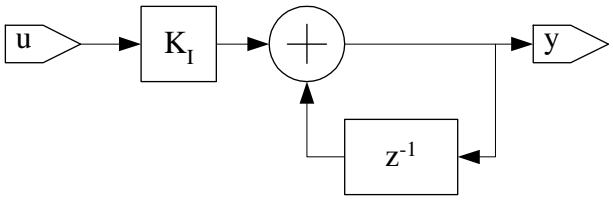

Abbildung 3.6: Integrator

Bei einem Eingangssignal mit konstantem Vorzeichen müsste y über alle Grenzen wachsen, was durch die endliche Wortbreite nicht möglich ist. Bei einem Über- oder Unterlauf würde der Wert schlagartig von einer Grenze zur anderen springen (siehe Bild 3.7a). So ein Verhalten ist nicht annehmbar. Besser ist es, wenn der Wert gegen eine Grenze läuft und dort verbleibt (siehe Bild 3.7b).

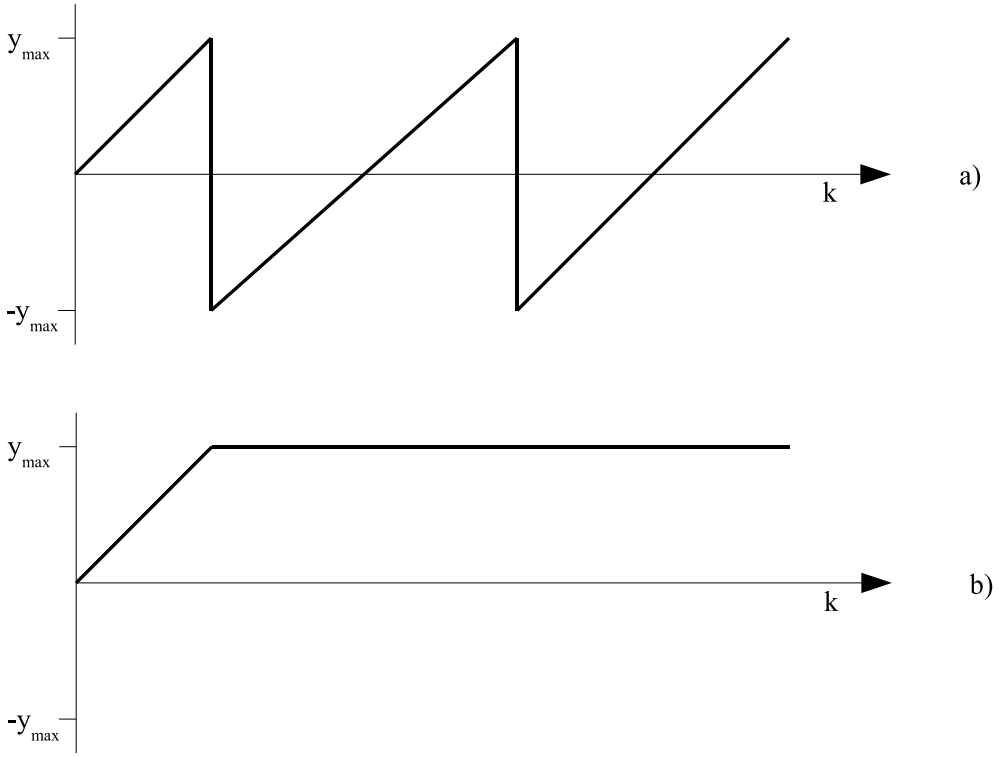

Abbildung 3.7: Verhalten des Integrators a) ohne und b) mit Fehlerbehandlung

Dieses Verhalten muss durch eine Fehlerbehandlung erzwungen werden. Bei der Behandlung des Addierers (siehe Abschnitt 2.5) wurden schon zwei Verfahren vorgestellt. Ein Über- oder Unterlauf wird grundsätzlich verhindert, wenn der Wertebereich der Eingangssignale am Addierer auf die Hälfte eingeschränkt wird. Für u ist dies durch eine geeignete Skalierung von K_I kein Problem, jedoch für y schon. Darum muss in den Rückkopplungszweig ein Element eingefügt werden, das dafür sorgt, dass dieses Signal nicht aus dem Wertebereich von $-2^{n-2}\ldots 0\ldots 2^{n-2}-1$ herausläuft. Dazu wurde ein Begrenzer entwickelt.

Eine andere Möglichkeit besteht in der Verwendung einer anderen Struktur.

Wenn aus Gleichung 3.13 die Ergebnisse für einige Zeitpunkte berechnet werden, erhält man für k=0 und k=1 die in Gleichung 3.15 und 3.16 dargestellten Resultate. Dies wurde in Gleichung 3.17 verallgemeinert.

$$y(0) = K_I u(0) \tag{3.15}$$

$$y(1) = K_I u(1) + K_I u(0) \tag{3.16}$$

$$\begin{aligned} &\vdots \\ y(m) &= K_I\, u(m) + K_I \sum_{i=0}^{m-1} u(i) \end{aligned} \tag{3.17}$$

Damit kann der Integrator in zwei Teile zerlegt werden

$$y(k) = K_I\, u(k) + y_I(k) \tag{3.18}$$

Nach Überführung von Gleichung 3.18 in den Bildbereich und Vergleich mit Gleichung 3.14 nach deren Erweiterung mit u erhält man

$$\begin{aligned} K_I \frac{z}{z-1} u &= K_I\, u + y_I \qquad\qquad |\cdot(z-1) \\ K_I\, u \cdot z &= K_I\, u \cdot z + y_I z - K_I\, u - y_I \quad | - K_I\, u \cdot z \,|\cdot z^{-1} \\ 0 &= y_I - K_I\, u \cdot z^{-1} - y_I z^{-1} \\ y_I &= (K_I\, u + y_I) z^{-1} \end{aligned} \tag{3.19}$$

Daraus ergibt sich der in Bild 3.8 gezeigte Signalfluss. Durch Einsatz eines zweiten Addierers kann ein weiterer Pfad geschaffen werden, der den direkten Durchgriff des Integrators realisiert. Das entspricht einem P-Anteil. Wenn der Integrator im Rahmen eines PID-Reglers zum Einsatz kommt, dann ist dort lediglich der Verstärkungsfaktor des P-Anteils anzupassen.

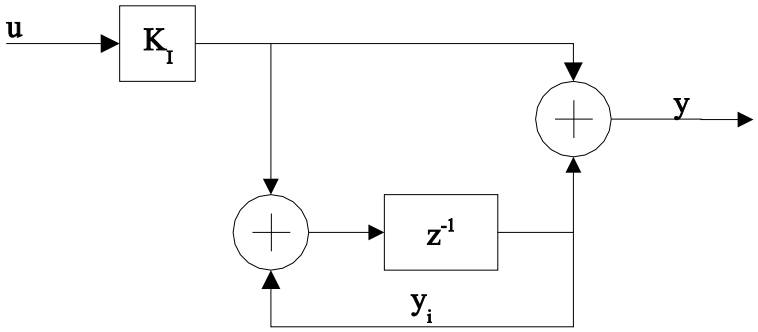

Abbildung 3.8: Integrator mit separatem P-Teil

3.3.4 PT-1

Ausgangspunkt ist die z-Übertragungsfunktion z.B. aus [Föl93].

$$\begin{aligned} G(z) = \frac{y}{u} &= \frac{z}{z-a} \\ y \cdot z - y \cdot a &= z \cdot u \\ y - y \cdot a \cdot z^{-1} &= u \end{aligned} \tag{3.20}$$

$$y = u + y \cdot a \cdot z^{-1}$$

Damit ergibt sich der in Bild 3.9 dargestellte Signalfluss.

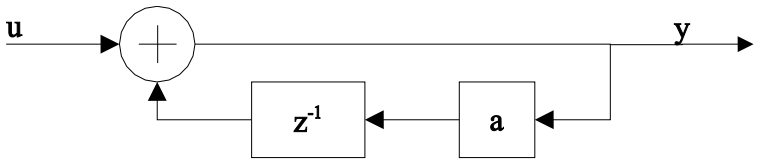

Abbildung 3.9: PT-1 Element

Ersetzt man

$$a = e^{-\hat{a}T} \tag{3.21}$$

gilt

$$a = e^{-\hat{a}T} \leq 1 \text{ mit } \hat{a}, T \geq 0 \tag{3.22}$$

Für die z-Rücktransformation ergibt sich dann:

$$\mathcal{Z}^{-1}\{G(z)\} = e^{-\hat{a}t}. \tag{3.23}$$

Die Annahme in Gleichung 3.22 ist zulässig, da â eine Zeitkonstante und T die Tastperiode darstellen. Beide sind in realen Systemen immer positiv. Diese Bedingung wird im realisierten PT-1 Element überprüft und bei ihrer Verletzung eine Warnung ausgegeben. Darum wurde auf eine Fehlerbehandlung verzichtet.

Prinzipiell bietet aber der Ansatz im Abschnitt 3.3.3 die Möglichkeit der Implementierung einer Fehlerbehandlung.

$$\begin{aligned} y(0) &= u(0) \\ y(1) &= u(1) + a \cdot u(0) \\ &\vdots \\ y(m) &= u(m) + a \sum_{i=0}^{m-1} u(i) \end{aligned} \tag{3.24}$$

Gleichung 3.24 kann wieder als Überlagerung eines P- und I-Anteils betrachtet werden

$$y(k) = u(k) + a \cdot y_I(k)$$

und wird nach Übertragung in den Bildbereich mit Gleichung 3.20 nach deren Erweiterung mit

u gleichgesetzt.

$$\begin{aligned} u + a \cdot y_I &= \frac{z}{z-a} u \\ (u + a \cdot y_I)(z-a) &= z \cdot u \qquad | -z \cdot u \ | : a \\ y_I z &= u + a \cdot y \qquad | \cdot z^{-1} \\ y_I &= (u + a \cdot y_I)\, z^{-1} \end{aligned}$$

Dazu ist der Signalfluss im Bild 3.10 zu sehen. Diese Möglichkeit erkauft man sich allerdings mit den doppelt zu implementierenden Addierern und Verstärkern.

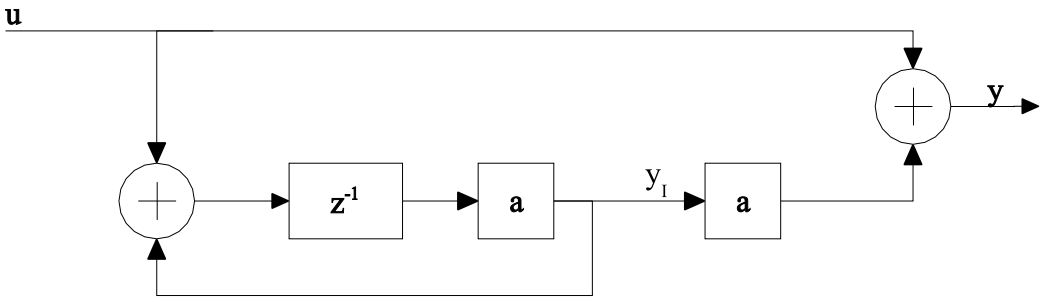

Abbildung 3.10: PT-1 Element mit Fehlerbehandlung

3.3.5 PT-2

Ausgangspunkt ist die Übertragungsfunktion in Gleichung 3.11, jedoch mit zwei Polen, die auch ein konjugiert komplexes Polpaar darstellen können.

$$\begin{aligned} G(z) = \frac{y}{u} &= \frac{z k_0}{z - T_0} + \frac{z k_1}{z - T_1} \\ &= \frac{z k_0 (z - T_1) + z k_1 (z - T_0)}{(z - T_0)(z - T_1)} \\ &= \frac{(k_0 + k_1) z^2 - (k_0 T_1 + k_1 T_0) z}{z^2 + (-T_1 - T_0) z + T_1 T_0} = \frac{b_2 z^2 + b_1 z}{z^2 + a_1 z + a_0} \end{aligned}$$

$$\begin{aligned} y(z^2 + a_1 z + a_0) &= u(b_2 z^2 + b_1 z) \\ y z^2 + y a_1 z + y a_0 &= u b_2 z^2 + u b_1 z \qquad | \cdot z^{-2} \\ y + y a_1 z^{-1} + y a_0 z^{-2} &= u b_2 + u b_1 z^{-1} \\ y &= u b_2 + u b_1 z^{-1} - y a_1 z^{-1} - y a_0 z^{-2} \end{aligned} \tag{3.25}$$

Bild 3.11 zeigt die daraus abgeleitete Implementierung.

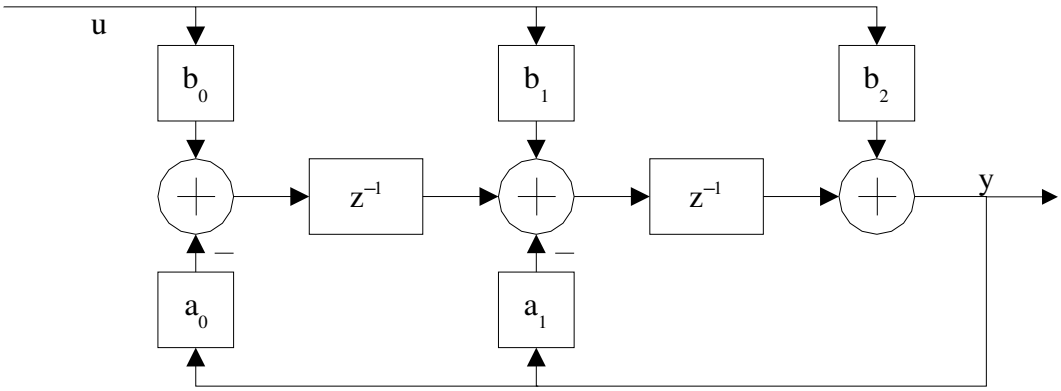

Abbildung 3.11: PT-2 Element

Unter Umständen kann $b_2 = 0$ sein, dann entfällt der entsprechende Addierer. Anforderungen an die Realisierung sind:

- in der Funktion zur Berechnung der Verzögerung der Komponente spielen b_2 und b_1 eine Rolle
- wenn $b_i = 1$ oder $a_i = 1$ ist, kann der jeweilige Verstärker entfallen. Dies ist prinzipiell auch bei $b_i = -1$ oder $a_i = -1$ möglich, jedoch muss dann auch das Vorzeichen behandelt werden.
- wenn $b_i = 0$ oder $a_i = 0$ ist, kann der zugehörigen Addierer durch ein Schieberegister ersetzt werden
- wenn $b_i < 0$ oder $a_i < 0$ ist, dann erfolgt der Austausch des Addierers durch einen Subtrahierer oder umgekehrt, d.h., das Vorzeichen wird in das nachfolgende Element verschoben.

Beim ersten Tap [1] (b_0/a_0) gibt es bei den Ersetzungen eine Besonderheit. Im Unterschied zu den anderen Taps werden hier an einer Stelle zwei Signale zusammengeführt, die direkt von den Verstärkern kommen. Sonst ist immer ein Addierer/Subtrahierer dazwischen. Die wesentliche Konsequenz daraus ist, dass der Austausch Addierer/Subtrahierer nicht ohne Berücksichtigung der Vorzeichen beider Koeffizienten gemacht werden kann, da es kein Element $x = -a - b$ gibt. Der vorhandene Subtrahierer realisiert $x = a - b$. Es gibt 64 Kombinationsmöglichkeiten der Koeffizienten. Diese resultieren aus: $b_0 = 0$, $b_0 < 0$, $|b_0| = 1$, $a_0 = 0$, $a_0 < 0$, $|a_0| = 1$, jedoch schließen sich viele aus, wie z.B. $b_0 = 0$ und $b_0 < 0$. Die wesentliche Konsequenz daraus ist, dass u.U. die Vorzeichenverlagerung nicht erfolgen kann.

Bei der Implementierung wurden die Speicherzellen durch die Fehlerbehandlung bereits in die Addierer/Subtrahierer verlagert. Dies deutet der gestrichelte Kasten im Bild 3.12 an. Da deren

[1] Als Tap (engl. Anzapfung) wird eine Stufe im Filter bezeichnet. Hier sind immer a_i und b_i zu einer Stufe zusammenzufassen.

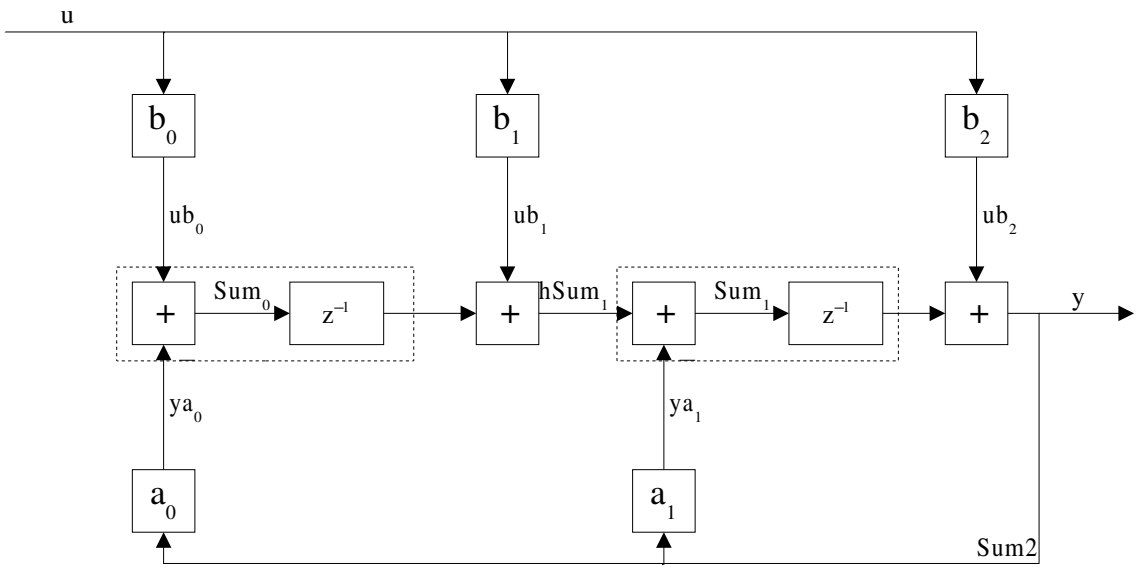

Abbildung 3.12: Realisierung des PT-2 Elements

Ergebnis dabei bereits auf den halben Wertebereich eingeschränkt wird, kann durch eine geeignete Skalierung der b_i ein Fehler am nachfolgenden Addierer/Subtrahierer vermieden werden.

3.4 Zusammengesetzte Blöcke

3.4.1 FIR- und IIR-Filter

Die digitale Signalverarbeitung ist nicht an die Fähigkeiten elektrischer, mechanischer oder fluidischer Bauelemente gebunden und unterliegt damit auch nicht deren Einschränkungen. Damit entstehen Realisierungsprobleme bei digitalen Systemen erst bei deutlich höherer Ordnung als bei analogen Systemen. Weiterhin erlaubt die digitale Signalverarbeitung ein Systemverhalten, welches analog gar nicht machbar wäre. Prinzipiell lassen sich digitale lineare Systeme in zwei Gruppen einteilen. Das sind einerseits Filter mit endlicher Impulsantwort (Finite-Impulse-Response, FIR-Filter) und andererseits unendlicher Impulsantwort (Infinite-Impulse-Response, IIR-Filter).

3.4.1.1 IIR-Filter

Die vollständige Umsetzung von Gleichung 3.4 führt zum IIR-Filter. Die bereits beim PT-2-Element gewonnen Erfahrungen wurden genutzt, um die Struktur in Atome zu zerschlagen, die eine Realisierung in Beobachternormalform zulassen. Es lassen sich drei Atome bilden:

- Beginn der Kette (FirstTap), hier liegen die Koeffizienten a_0 und b_0

- Mittenelement (MiddleTap) mit den Koeffizienten $a_1 \ldots a_{m-1}$ und $b_1 \ldots b_{m-1}$
- Ende der Kette (LastTap) mit dem Koeffizienten b_m

Die zu den a_i gehörenden Addierer nehmen jeweils eine Fehlerbehandlung vor. Eine Begrenzung des Wertebereichs durch eine Skalierung von a_i ist nicht möglich, denn das würde das Ausklammern eines Faktors aus dem Nenner von Gleichung 3.4 bedeuten. Dies verändert jedoch auch a_m zu $a_m \neq 1$. Dieser Koeffizient müsste dann im Rückkopplungszweig eingefügt werden und würde die Skalierung wieder aufheben.

Jedoch ist eine Skalierung über b_i möglich. Denn ein hier ausgeklammerter Faktor könnte problemlos am Ausgang des Filters wieder korrigiert werden, so dass durch eine Begrenzung von Sum_i auf $-2^{n-2} \leq Sum_i < 2^{n-2}$durch eine Fehlerbehandlung und die Skalierung von b_i ein Überlauf von $hSum_i$ vermieden werden kann.

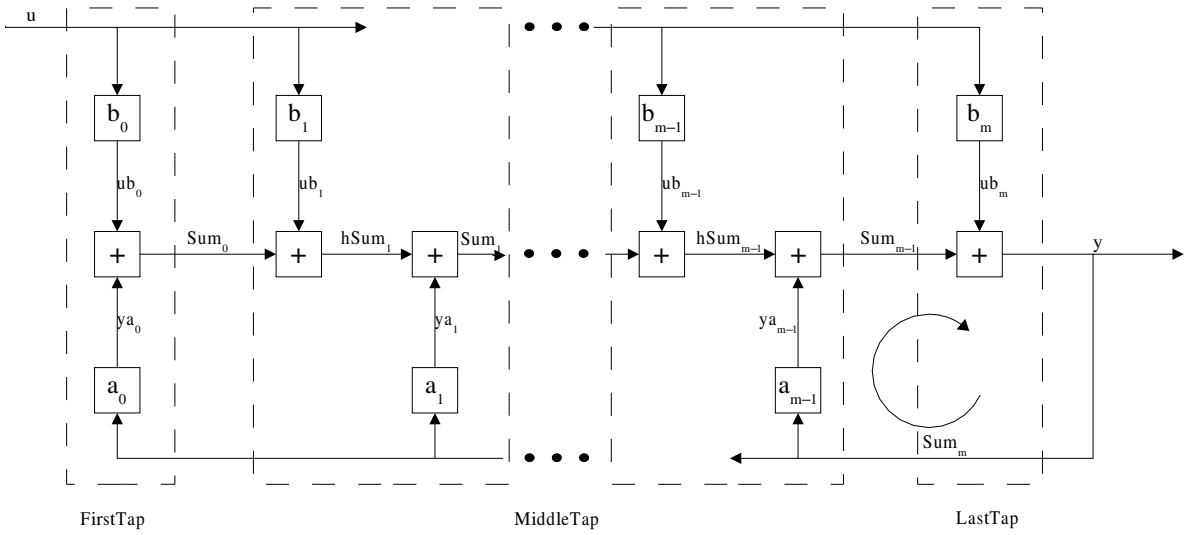

Abbildung 3.13: IIR-Filterrealisierung in Beobachternormalform

Der über a_{m-1} angedeutete Umlauf im Bild 3.13 darf laut Gleichung 3.1 nur n Takte dauern. Die beiden Addierer nutzen jedoch schon n-1 Takte durch die Fehlerbehandlung bei Sum_{m-1}. Der Verstärker verbraucht noch einen weiteren Takt, eine Skalierung würde dann das erlaubte Limit sprengen. Darum ist die Beobachternormalform hier zur Realisierung von IIR-Filtern nur geeignet, wenn für a_i gilt: $|a_i| \leq 1$. Denn bei $|a_i| < 1$ liegt ein stabiles System vor und auf eine Fehlerbehandlung kann verzichtet werden, während bei $|a_i| = 1$ kein Verstärker erforderlich ist.

3.4.1.2 FIR-Filter

FIR-Filter enthalten keine Rückführung, damit entfällt in Gleichung 3.4 der Nenner und es ergibt sich:

$$y = b_m u z^m + b_{m-1} u z^{m-1} + \cdots + b_0 u.$$

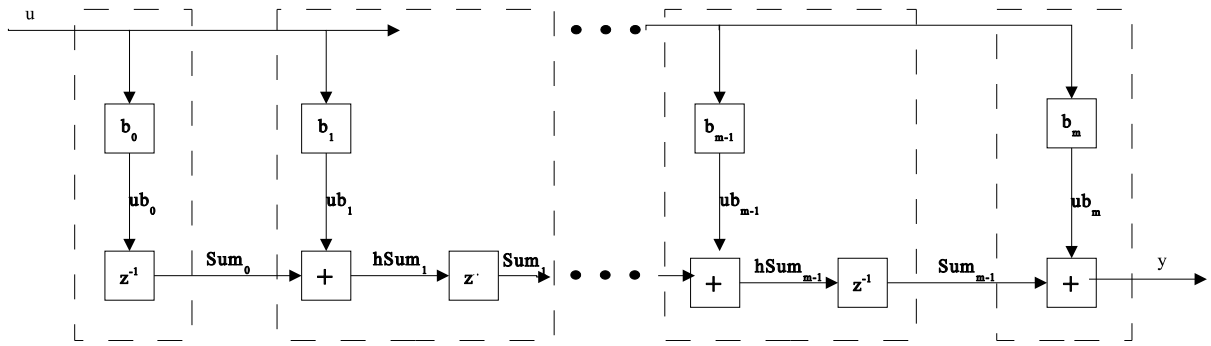

Abbildung 3.14: FIR-Filterrealisierung in Beobachternormalform

Zur Implementierung können dieselben Atome wie für den IIR-Filterverwendet werden, es gilt lediglich für alle $a_i = 0$.

3.4.2 PID-Regler

Der PID-Regler wird gern und häufig aufgrund seines einfachen Aufbaus und der Vertrautheit des Ingenieurs genutzt. Im Fall der modalen Realisierung (siehe Gleichung 3.26) stehen die einzelnen Komponenten für ein nachvollziehbares Verhalten. Weiterhin gibt es klare Zusammenhänge zwischen den Parametern und dem Verhalten. Dies ist bei den anderen aufwändigeren Reglern/Filtern nicht immer der Fall.

$$y = K_P u + K_D (u - u z^{-1}) + K_I u + u_I z^{-1} \tag{3.26}$$

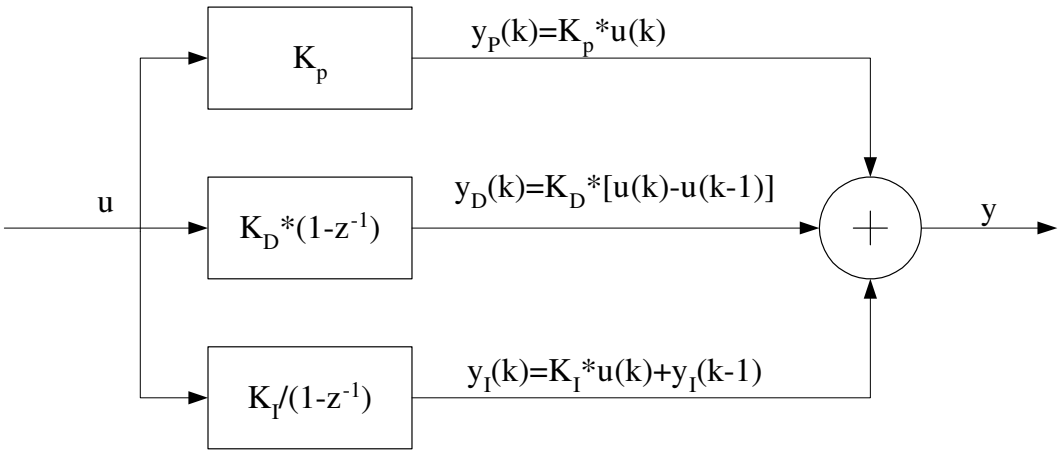

Abbildung 3.15: Allgemeine Form eines PID-Reglers

3.4.2.1 Modale Normalform

Bei der modalen Normalform finden sich die drei Komponenten P-, I- und D-Anteil wieder. Für den I-Anteil gilt das bereits im Abschnitt 3.3.3 bezüglich der erforderlichen Fehlerbehandlung Gesagte. Bild 3.16 zeigt den realisierten PID-Regler, inklusive der Fehlerbehandlung.

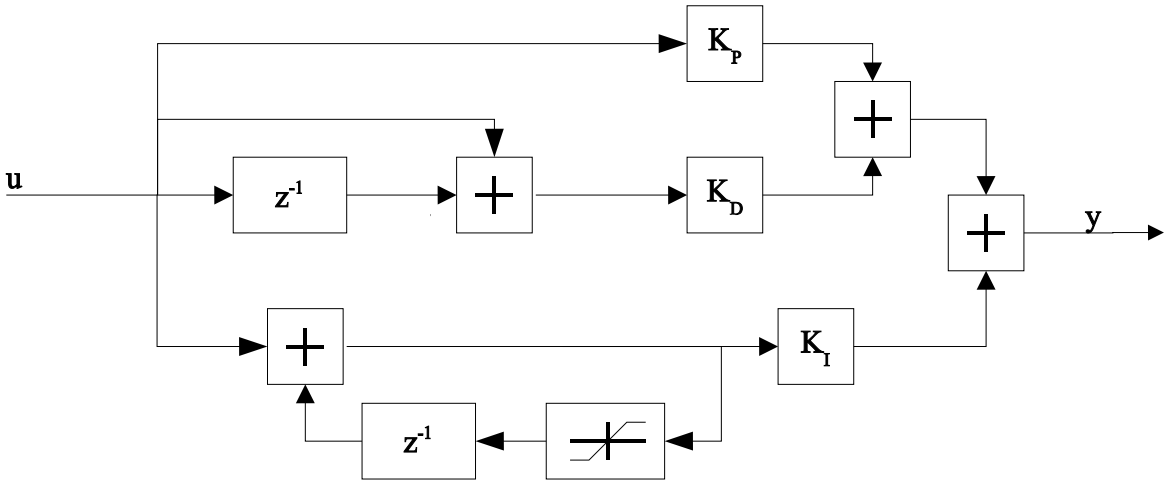

Abbildung 3.16: Realisierung des PID-Reglers in modaler Normalform

3.4.2.2 Beobachternormalform

Gleichung 3.26 lässt sich auch in die allgemeine Form der z-Übertragungsfunktion umformen, womit sich wieder die bekannten Koeffizienten a_i und b_i ergeben. Dabei ist zu beachten, dass der Integrator y_i zurückführt und nicht y, wie es in der z-Übertragungfunktion geschieht. Darum ist der Zwischenschritt laut Gleichung 3.27 notwendig.

$$\begin{aligned} y &= K_P u + K_D (u - u z^{-1}) + K_I u + y_I z^{-1} \\ y_I &= y - y_P - y_D \\ &= y - K_P u - K_D (u - u z^{-1}) \end{aligned} \tag{3.27}$$

$$\begin{aligned} y &= K_P u + K_D (u - u z^{-1}) + K_I u + (y - K_P u - K_D (u - u z^{-1})) z^{-1} \\ &= K_P u + K_D u - K_D u z^{-1} + K_I u + y z^{-1} - K_P u z^{-1} - K_D u z^{-1} + K_D e z^{-2} \\ &= K_D u z^{-2} + (-K_P - 2K_D) u z^{-1} + (K_P + K_D + K_I) u + y z^{-1} \end{aligned} \tag{3.28}$$

Hier sind:

$$\begin{aligned} b_2 &= K_P + K_D + K_I \\ b_1 &= -2K_D - K_P \\ b_0 &= K_D \end{aligned}$$

$$a_1 = 1$$

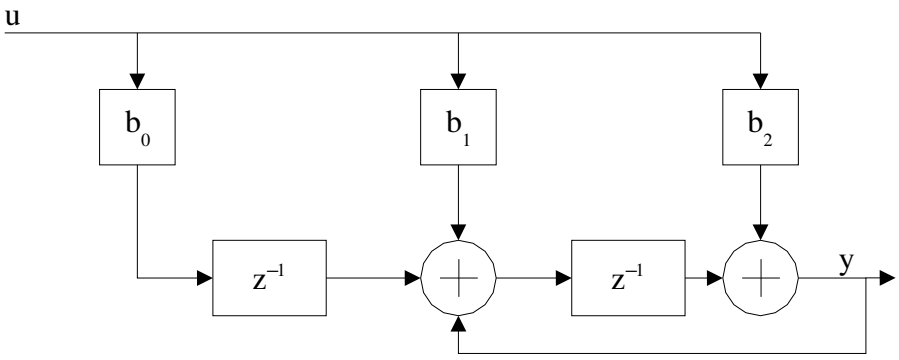

Abbildung 3.17: PID-Regler in Beobachtungsnormalform

Zur Implementierung werden wieder die Atome genutzt, die bei der Gewinnung der Beobachternormalform entstanden sind.

Kapitel 4

Synchronisation

Unter dem Begriff Synchronisation soll hier sowohl die Ausrichtung von Operandenbits auf gleiche Wertigkeit bei Operatoren mit mehreren Eingängen als auch die Kennzeichnung von Operandenbits verstanden werden.

Wie schon bei der Vorstellung der einzelnen Operatoren (siehe Kapitel 2) erwähnt, benötigt die Verarbeitung eines Bits im Durchschnitt einen Takt. Durchlaufen die Operanden nun verschiedene Signalketten, kommen sie bei einer Zusammenführung (Addition, Multiplikation) mit unterschiedlichen Verzögerungen an. Wobei in diesem Zusammenhang nur ungewollte Verzögerungen gewertet werden, denn gewollte Verzögerungen von vielfachen der Tastperiode, resultierend aus dem Algorithmus, verschlechtern das System nicht.

Diese Operatoren (siehe Gleichungen 2.4 bis 2.7) fordern die Ausrichtung der Operandenbits auf gleiche Wertigkeit. Bei der bitparallelen Übertragung ist die Wertigkeit durch die Signalleitung zur Übertragung des jeweiligen Operandenbits gegeben. Im bitseriellen Fall erfolgt die Übertragung aller Signale auf einer einzigen Leitung. Daher ist hier eine andere Möglichkeit zur Festlegung der Wertigkeit von Operandenbits zu finden. Es reicht nicht aus, nur die Wortgrenzen zu markieren, denn in einigen Operatoren sind auch an anderen Stellen Entscheidungen (z.B. bezüglich einer Fehlerbehandlung) zu treffen. Es muss also die Möglichkeit bestehen, einerseits jedes beliebige Bit zu kennzeichnen und andererseits dem Operator mitzuteilen, mit welcher Verzögerung ein Operand eintrifft.

Es bestehen mehrere Möglichkeiten, die Synchronisation vorzunehmen.

4.1 Statische Synchronisation

Statische Synchronisation heißt, dass vor der Logiksynthese alle Angaben zur Synchronisation bekannt sein müssen. Sie sind entweder per Hand oder maschinell in einem Vorverarbeitungsprozess zu ermitteln. Mit Hilfe der Verzögerungswerte erfolgt die Auswahl von Signalen zur

Markierung der einzelnen Operandenbits aus einem Bus. Da ein jedes Bit gekennzeichnet werden können muss, hat der Bus eine Breite, die der Verarbeitungswortbreite entspricht. Jedes Signal ist nur während eines Taktes aktiv. Bild 4.1 stellt den Synchronisationsbus im Vergleich zur bitparallelen Übertagung dar.

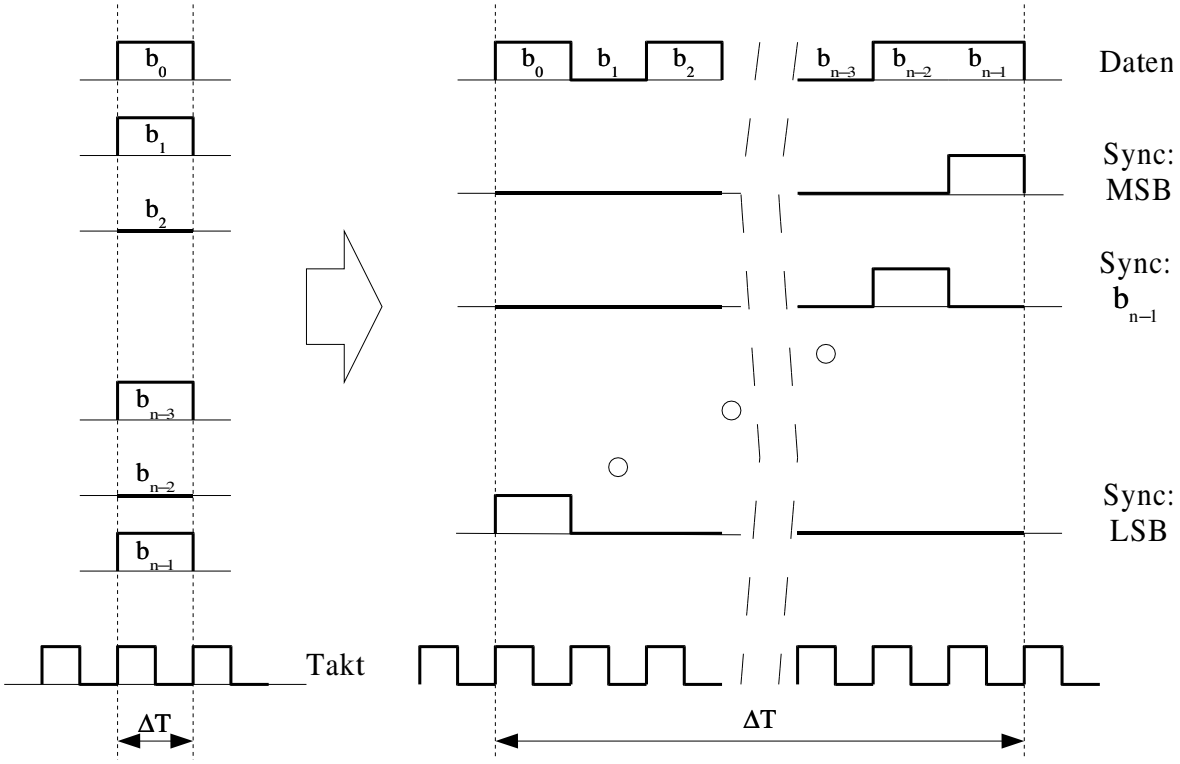

Abbildung 4.1: Festlegung der Wertigkeit von Operandenbits bei bitparalleler und bitserieller Übertragung

Die Ausrichtung der Operandenbits auf gleiche Wertigkeit erfolgt mit Hilfe zusätzlicher Verzögerungsstufen im weniger verzögerten Datenstrom (siehe Bild 4.2). Dazu werden den Operatoren die Verzögerungen der Operanden mitgeteilt. Aus der Differenz der Verzögerungen wird dann die Länge der Verzögerungsstufen berechnet.

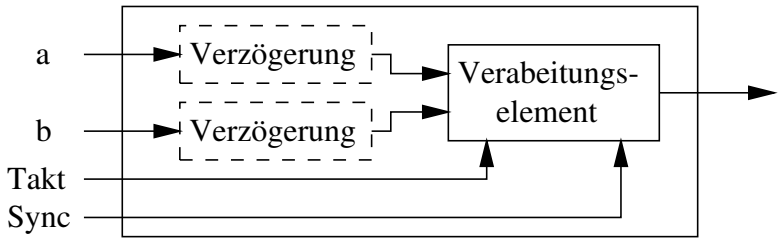

Abbildung 4.2: Prinzipieller Aufbau von Elementen mit zwei Eingängen

Vorteile:

- Minimaler Logikaufwand

Nachteile:

- Aufwendige Analyse des Systems ist erforderlich
- keinerlei Fehlertoleranz

4.2 Dynamische Synchronisation

Während der Logiksynthese stehen keinerlei Informationen über Verzögerungen der Signale zur Verfügung, bzw. sie müssen nicht generiert werden. Dies geschieht während einer zusätzlichen Synchronisationsphase, die der eigentlichen Signalverarbeitung vorgelagert ist. Es besteht auf alle Fälle das Problem, dass alle eventuell benötigten Schieberegister mit synthetisiert werden müssen. Wieviel davon genutzt werden und an welchen Stellen, wird während der Synchronisationsphase entschieden. Dabei kann es passieren, dass nicht genügend Register an einer Stelle zur Verfügung stehen, so dass dann eine Synchronisation nicht möglich ist. Durch eine geeignete Programmierung können solche Fehler erkannt werden, auch ungenutzte Register können so aufgespürt werden. Durch manuellen Eingriff ließe sich das System dann optimieren. Dies wäre jedoch nach jeder Änderung am Algorithmus erforderlich.

Aufgrund der Schleifenbedingung (Gleichung 3.1 auf Seite 37) darf ein Schleifendurchlauf nur n Takte dauern. Je nach Anzahl der durch Operationen verbrauchten Takte wird das Schieberegister der Memoryzelle verkürzt. Dies müsste auch dynamisch erreicht werden.

Vorteile:

- fehlertolerant
- eine automatische Analyse des Systems hinsichtlich der Verzögerungen erforderlich, alle Eingriffe werden vom Menschen vorgenommen

Nachteile:

- höherer Logikaufwand
- Trial and Error, kein automatisches Lösungsverfahren

4.3 Hybrides Verfahren

Hier kämen in den Datenstrom einkodierte Sync-Signale infrage. Ein solches Verfahren wäre auf alle Fälle teilweise dynamisch, da nicht alle Informationen bei der Logiksynthese bekannt

sind. Durch einen zusätzlichen Pegelwechsel innerhalb eines Datenbits wird ein besonderes Bit gekennzeichnet, wahrscheinlich das LSB. Daraus werden Informationen über die anderen Bits abgeleitet. Es ist immer das gleiche Bit zu kennzeichnen.

Nachteile:

- die Kennzeichnung der anderen Bits erfolgt in jedem Operator aufs neue und erfordert jedesmal Logik
- die Daten können maximal mit der halben Taktfrequenz übertragen werden.

4.4 Auswahl

Da die statische Synchronisation eine automatische Implementierung ermöglicht und den geringsten Logikaufwand erfordert, wurde sie hier umgesetzt.

Kapitel 5

Automaten

In Steuer- und Regeleinrichtungen steuern Automaten die Abläufe und schalten Signale. Unter Berücksichtigung der bitseriellen Übertragung der Operanden ist ein Verfahren zur Behandlung der Ein- und Ausgangssignale von Automaten zu finden.

Die Schaltbedingungen sind entweder Ergebnisse von Vergleichen analoger u.U. digitalisierter Signale oder Eingaben (Taster-/Schalterbetätigung) durch den Bediener. Letztere liegen bereits in binärer Form vor, während das Vergleichsergebnis erst gebildet werden muss. Der Vergleich kann nicht direkt als Bedingung an den Automaten geschrieben werden. Denn das verwendete Spezifikationstool erwartet hier, bedingt durch VHDL, logische Operationen. Darum ist der Vergleich an anderer Stelle durchzuführen und das Ergebnis als binäres Signal an den Automaten zu führen.

Ausgangssignale können wieder digital oder binär sein. Während die binären Signale vom Automaten selber generiert werden können, sind die digitalen nur durchgeschaltete und damit gemultiplexte Eingangssignale. Durch die ständig ablaufende Signalverarbeitung sind die Signale auch ständig zu multiplexen. Darum muss dies in jedem Zustand eines Automaten als Aktion angegeben werden, wie Signal y in Bild 5.1. Wenn viele Signale zu multiplexen sind, dann müssen die Aktionen an *alle* Zustände geschrieben werden. Daher ist es sinnvoller, je unabhängigen Multiplexerausgang einen Automaten zu verwenden. Weiterhin ist sicherzustellen, dass nur an den Wortgrenzen die Umschaltung vorgenommen werden kann. Darum müssen die Operanden zuerst synchronisiert und weiterhin ein Signal generiert werden, welches das Wortende dem Automaten mitteilt (Signal sync im Bild 5.1). Da ein solcher Automat nicht losgelöst von seiner Umgebung arbeitet, sind oft noch Signale vorhanden, die das Weiterschalten in Abhängigkeit anderer Ereignisse steuern (siehe Signale freigabe1 und freigabe2 im Bild).

Das Synchronisieren der Operanden und das Generieren des sync-Signals übernimmt eine Synchronisierer-Komponente. Dort werden alle auf einen Ausgang zu multiplexenden Signale hereingeführt. Weiterhin wird die Verzögerung eines jeden Signals übergeben. Mit diesen Informationen werden entsprechende Schieberegister eingebaut und das sync-Signal für den Automaten

aus dem sync-Bus ausgewählt.

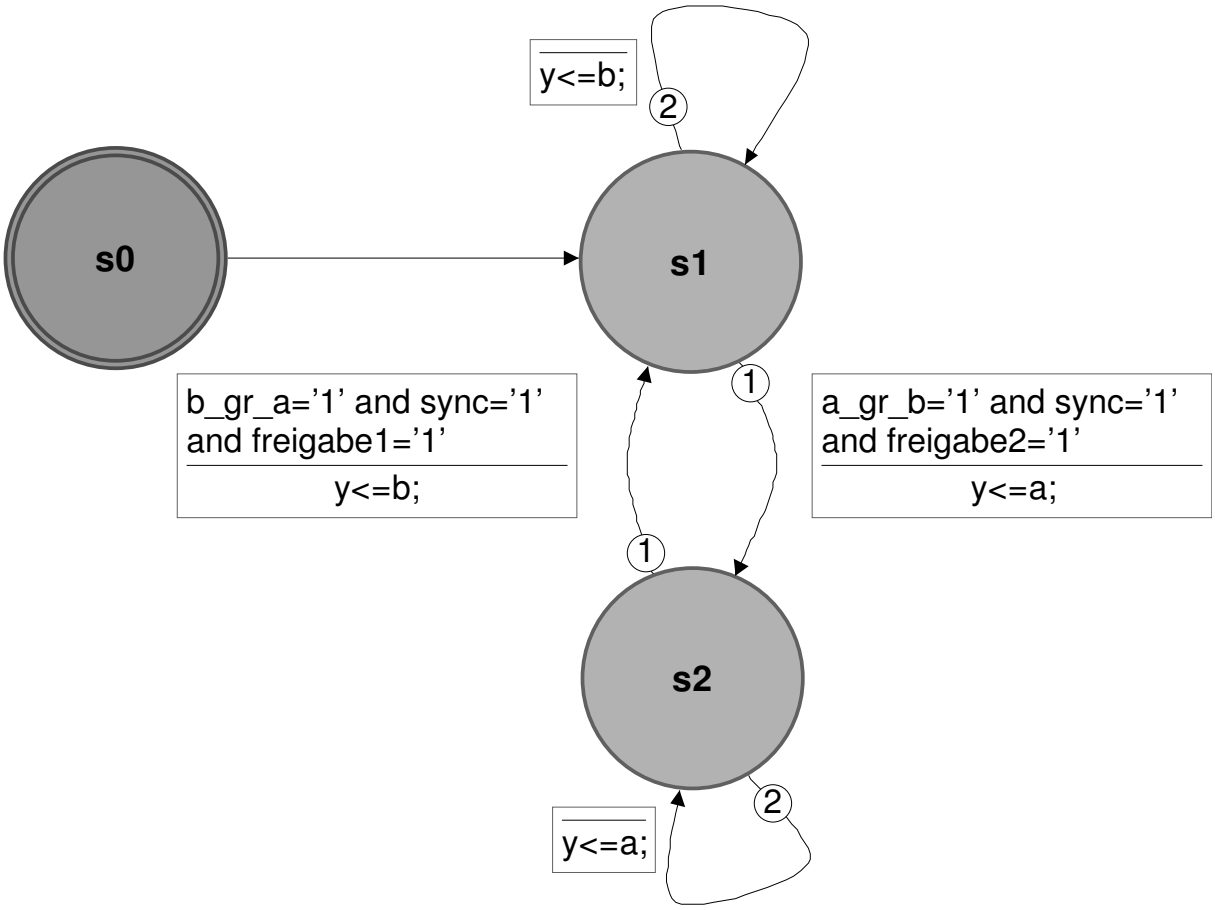

Abbildung 5.1: Automat zum Multiplexen

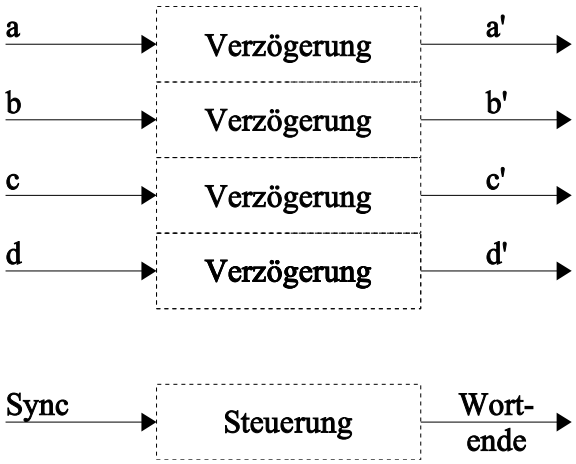

Abbildung 5.2: Aufbau der Synchronisierer-Komponente

Kapitel 6

Automatische Bestimmung der Verzögerungszeiten

Wie bereits im Abschnitt 4.1 auf Seite 56 beschrieben, benötigen die Operatoren zur korrekten Verarbeitung der Operanden Kenntnis über deren Wortgrenzen. Dazu werden den verarbeitenden Komponenten die Verzögerungen der Eingangssignale mittels eines generischen Parameters (Delay-Generic) übergeben. Deren manuelle Berechnung ist schwierig und fehleranfällig, insbesondere in zurückgekoppelten Systemen. In vielen Fällen wird ein Signal in verschiedenen Zweigen einer Hierarchie verwendet. Eine Deklaration als Konstante in einer Architektur ist nicht geeignet, da deren Wert nicht nach oben (in Richtung des Toplevels) durchgereicht wird. So wäre es nicht möglich, die Verzögerung eines Signals aus Block e im Bild 6.1 zu Block d durchzureichen. VHDL ermöglicht dies genauso wenig wie andere Programmiersprachen.

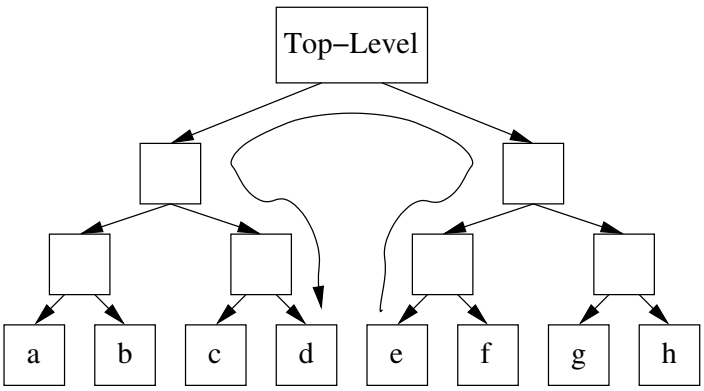

Abbildung 6.1: Beispiel für ein hierarchisches System

Dieses Problem kann durch eine Funktion gelöst werden, die in einem extra Package untergebracht ist. Jede Komponente ruft dann die Funktionen aller ihr unterlagerten Komponenten auf und bestimmt so die Verzögerungen der Signale. Jede Funktion ermittelt anhand von übergebenen Parametern (Eingangsverzögerung, Skalierung, usw.) die Verzögerung des Ausgangssignals

und gibt diesen Wert an die aufrufende Funktion zurück. Damit baut sich parallel zur Hierarchie der Signalverarbeitung eine Hierarchie zur Berechnung der Verzögerungen auf. Jedoch erfolgt deren Implementierung immer noch manuell, auch ergeben sich Probleme bei Rückkopplungen. Denn hier hängt die Verzögerung eines Eingangssignals vom Ausgang derselben Komponente ab. Daher ist nach einem Weg zur automatischen Berechnung und Verteilung der Verzögerungen zu suchen.

6.1 Ausgangsbedingungen

Bei den verwendeten Algorithmen (siehe Kapitel 3) werden Bits an der gleichen Position im Datenstrom als gleichwertig betrachtet. Damit sind nur Verzögerungen d der Datenströme von ganzen Vielfachen der Verarbeitungswortbreite n möglich (siehe Gleichung 3.1 auf Seite 37).

$$d = i \cdot n \text{ mit } i = 0,1,2,... \tag{6.1}$$

Bei einer Schleife treten Funktionen wie

$$y = f(u,y) \tag{6.2}$$

auf. Jedoch benötigt die Berechnung des Ergebnisses der Funktion f(u, y) mindestens einen Takt, d.h. $\mathsf{d} > 0$. Damit gilt für i in Gleichung 6.1 $\mathsf{i} > 0$, und 6.2 ist wie in Gleichung 6.3 zu konkretisieren.

$$y(k) = f[u(k), y(k-i)] \quad \text{mit } i > 0 \tag{6.3}$$

6.2 Modell zur Beschreibung der Verzögerungen

Die Verzögerungen resultieren zwar ausschließlich von den Operatoren, doch sind diese miteinander verknüpft. Damit muss ein Modell für die Verzögerungen sowohl die Verbindungen als auch die durchlaufenen Operatoren wiederspiegeln. Diese beiden Informationen werden in getrennten Gleichungen abgelegt. Bild 6.2 zeigt die Definition der Bezeichner. u_0 und y_0 sind die externen Ein- und Ausgänge eines Blockes, während u_i und y_i mit $\mathsf{i} > 0$ die Ein- und Ausgänge von internen Blöcken sind.

Für die Topologie bezüglich Block 1 in diesem Bild ergibt sich folgende Beziehung:

$$\mathbf{u}_1 = \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot u_0 \tag{6.4}$$

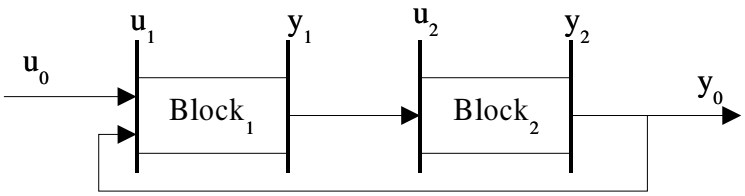

Abbildung 6.2: Definition der Bezeichner bezüglich der Verzögerungen

Diese Gleichung lässt sich verallgemeinern. Dabei ist jedoch zu berücksichtigen, dass immer jeweils nur ein Komponentenausgang y_i mit $\mathbf{P}$ oder ein Eingangssignal u_{0_i} mit $\mathbf{P_0}$ auf u_i abgebildet wird. Dadurch ist höchstens ein Element pro Zeile von $\mathbf{P}$ und $\mathbf{P}_0$ besetzt. Da aber jeder Ausgang auf wenigstens einen Eingang geführt ist, muss jede Spalte mit mindestens einem Element besetzt sein.

$$\mathbf{u}_1 = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} p_{0_1} \\ p_{0_2} \end{bmatrix} \cdot u_0$$

Allgemein ergibt sich folgende Beziehung:

$$\underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_l \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{l1} & p_{l2} & \cdots & p_{ln} \end{bmatrix}}_{\mathbf{P}} \cdot \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}} + \underbrace{\begin{bmatrix} p_{0_{11}} & p_{0_{12}} & \cdots & p_{0_{1k}} \\ p_{0_{21}} & p_{0_{22}} & \cdots & p_{0_{2k}} \\ \vdots & \vdots & \ddots & \vdots \\ p_{0_{l1}} & p_{0_{l2}} & \cdots & p_{0_{lk}} \end{bmatrix}}_{\mathbf{P_0}} \cdot \underbrace{\begin{bmatrix} u_{0_1} \\ u_{0_2} \\ \vdots \\ u_{0_k} \end{bmatrix}}_{\mathbf{u_0}} \tag{6.5}$$

$$\text{mit} \quad p_{0_{ij}}, p_{ij} = \begin{cases} 0 \\ 1 \end{cases} \tag{6.6}$$

Da hier nur Kopplungen betrachtet werden und diese entweder vorhanden sind oder nicht, ist die Einschränkung 6.6 zulässig. Dies trifft auch für die weiteren hier definierten Matrizen zu. Innerhalb eines Blockes hängt die Verzögerung am Ausgang von der des Eingangsignals u_i und der im Block selber generierten variablen v_i und konstanten v_{k_i} Verzögerungen ab. Variable Verzögerungen resultieren aus eventuell einzufügenden Schieberegistern zur Synchronisation, während die konstanten Verzögerungen durch die numerische Verarbeitung der Operanden entstehen. Dies sieht für Block 1 und 2 in Bild 6.2 folgendermaßen aus:

$$\begin{aligned} y_1 &= \mathbf{c}_{\mathbf{v}_1} \cdot \mathbf{v}_1 + \mathbf{c}_{\mathbf{u}_1} \cdot \mathbf{u}_1 + v_{k_1} \\ y_2 &= c_{v_2} \cdot v_2 + c_{u_2} \cdot u_2 + v_{k_2} \end{aligned} \tag{6.7}$$

Hier bildet $\mathbf{c_v}$ die variablen Verzögerungen und $\mathbf{c_u}$ die Eingangsverzögerungen auf das Ausgangssignal ab. Jede dieser Gleichungen beschreibt die Verzögerungen, die in dem jeweiligen Block

entstehen. Damit gibt es ausschließlich Abhängigkeiten der Form $y_i = f(v_i, u_i, v_{k_l})$, was zu einer schwachen Besetzung der Matrizen führt. Allgemein kann Gleichung 6.7 so formuliert werden:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}} = \underbrace{\begin{bmatrix} \mathbf{c_{v_1}} & 0 & \cdots & 0 \\ 0 & \mathbf{c_{v_2}} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{c_{v_m}} \end{bmatrix}}_{\mathbf{C_v}} \cdot \underbrace{\begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v_m} \end{bmatrix}}_{\mathbf{v}} + \underbrace{\begin{bmatrix} \mathbf{c_{u_1}} & 0 & \cdots & 0 \\ 0 & \mathbf{c_{u_2}} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{c_{u_l}} \end{bmatrix}}_{\mathbf{C_v}} \cdot \underbrace{\begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_l \end{bmatrix}}_{\mathbf{u}} + \underbrace{\begin{bmatrix} v_{k_1} \\ v_{k_2} \\ \vdots \\ v_{k_n} \end{bmatrix}}_{\mathbf{v_k}} \tag{6.8}$$

Einschränkungen zur Besetzung der Zeilen und Spalten lassen sich hier nicht machen, da die verschiedensten Abhängigkeiten der Verzögerungen möglich sind. Da sowohl v_i und u_i als auch c_{v_i} und c_{u_i} Vektoren sein können, müssen $\mathbf{C_v}$ und $\mathbf{C_u}$ nicht quadratisch sein. Der Ausgang eines Blocks ist immer mit einem blockinternen Signal verbunden. Somit entspricht auch die Ausgangsverzögerung dem eines internen Signals y_i. Für das Beispiel heißt das:

$$y_0 = y_2$$

und allgemein unter Berücksichtigung eines Durchgriffs:

$$\mathbf{y}_0 = \mathbf{C}_0 \cdot \mathbf{y} + \mathbf{D} \cdot \mathbf{u}_0$$

Damit ergibt sich folgendes Gleichungssystem:

$$\mathbf{u} = \mathbf{P} \cdot \mathbf{y} + \mathbf{P}_0 \cdot \mathbf{u}_0 \tag{6.9}$$

$$\mathbf{y} = \mathbf{C_v} \cdot \mathbf{v} + \mathbf{C_u} \cdot \mathbf{u} + \mathbf{v_k} \tag{6.10}$$

$$\mathbf{y}_0 = \mathbf{C}_0 \cdot \mathbf{y} + \mathbf{D} \cdot \mathbf{u}_0 \tag{6.11}$$

6.3 Zusammenfassen von Hierarchieebenen

Nur einfache Algorithmen lassen sich auf einer Hierarchieebene implementieren. Üblicherweise sind mehrere Ebenen erforderlich. Damit stellt sich die Frage, wie sich diese zusammenfassen lassen.

Ausgangspunkt sind die Gleichungen 6.9 bis 6.11. Diese können z.B. Block 1 in Bild 6.3 beschreiben. Um jedoch zu einer Beschreibung wie in Gleichung 6.8 zu gelangen, ist für jeden Block eine Funktion der Form $y = f(v, u, v_k)$ erforderlich. Dafür ist Gleichung 6.9 in 6.10 einzusetzen.

$$\begin{aligned}
\mathbf{y} &= \mathbf{C_v} \cdot \mathbf{v} + \mathbf{C_u} \cdot (\mathbf{P} \cdot \mathbf{y} + \mathbf{P}_0 \cdot \mathbf{u}_0) + \mathbf{v_k} \\
&= (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \cdot (\mathbf{C_v} \cdot \mathbf{v} + \mathbf{C_u} \cdot \mathbf{P}_0 \cdot \mathbf{u}_0 + \mathbf{v_k}) \\
&= (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \cdot \mathbf{C_v} \cdot \mathbf{v} + (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \cdot \mathbf{C_u} \cdot \mathbf{P}_0 \cdot \mathbf{u}_0 + (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \cdot \mathbf{v_k} \\
&= \widehat{\mathbf{C}}_\mathbf{v} \cdot \mathbf{v} + \widehat{\mathbf{C}}_\mathbf{u} \cdot \mathbf{u}_0 + \widehat{\mathbf{v}}_\mathbf{k} \qquad (6.12)
\end{aligned}$$

Mit Gleichung 6.12 sehen dann die konstanten Matrizen und Vektoren wie folgt aus:

$$\widehat{\mathbf{C}}_\mathbf{v} = (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \mathbf{C_v} \qquad (6.13)$$

$$\widehat{\mathbf{C}}_\mathbf{u} = (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \mathbf{C_u} \cdot \mathbf{P}_0 \qquad (6.14)$$

$$\widehat{\mathbf{v}}_\mathbf{k} = (\mathbf{I} - \mathbf{C_u} \cdot \mathbf{P})^{-1} \mathbf{v_k} \qquad (6.15)$$

$$\widehat{\mathbf{C}}_0 = \mathbf{C}_0 \qquad (6.16)$$

$$\widehat{\mathbf{D}} = \mathbf{D} \qquad (6.17)$$

während die Variablen unverändert bleiben. Auf diese Art und Weise erhält man für jeden i-ten Block einer Ebene:

$$\mathbf{y_i} = \widehat{\mathbf{C}}_{\mathbf{v_i}} \cdot \mathbf{v_i} + \widehat{\mathbf{C}}_{\mathbf{u_i}} \cdot \mathbf{u}_{0_\mathbf{i}} + \widehat{\mathbf{v}}_{\mathbf{k_i}} \qquad (6.18)$$

$$\mathbf{y}_{0_\mathbf{i}} = \mathbf{C}_{0_\mathbf{i}} \cdot \mathbf{y_i} + \mathbf{D_i} \cdot \mathbf{u}_{0_\mathbf{i}} \qquad (6.19)$$

In dieser Gleichung wurden die internen Eingangssignale u_{i_k} eliminiert, da sie immer durch ein y_{i_m} ersetzt werden können. Weiterhin stellt $\mathbf{u}_{0_\mathbf{i}}$ den Vektor der Eingangssignale und $\mathbf{y}_{0_\mathbf{i}}$ den Vektor der Ausgangssignale am entsprechenden Block dar. Im Bild 6.3[1] sind noch einmal alle Signale im Zusammenhang von mehreren Hierarchieebenen zu sehen.

Da die internen Ausgangssignale $\mathbf{y}$ noch vorhanden sind, können sie auch weiterhin berücksichtigt werden. Um sie auch eine Ebene höher verwenden zu können, wird grundsätzlich die rechte Seite von Gleichung 6.19 genutzt. So verbleiben nur die tatsächlichen Ausgänge von verarbeitenden Komponenten und nicht die von verbindenden Komponenten. Damit wird vermieden, dass ein Signal, welches über mehrere Blöcke nur durchgereicht wird, mehrmals in $\mathbf{y}$ enthalten ist. Aus den Gleichungen $\mathbf{y_i}$ für alle n-Blöcke einer Ebene lässt sich wieder eine Gleichung für die generierten Verzögerungen (siehe Gleichung 6.20) gemäß 6.10 unter Einbeziehung von

[1] Im folgenden stellt $\widetilde{\mathbf{x}}$ eine Größe der betrachteten Ebene, $\mathbf{x}$ der unterlagerten und $\widehat{\mathbf{x}}$ eine zusammengefasste Größe dar.

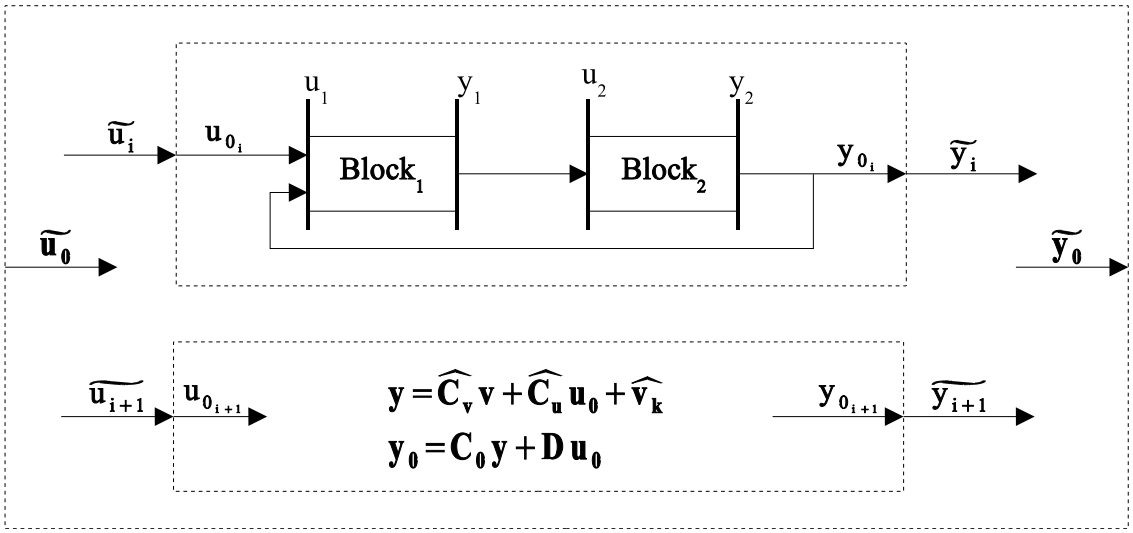

Abbildung 6.3: Verzögerungen in mehreren Hierarchieebenen

Gleichung 6.18 formen.

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_n \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{C}}_{\mathbf{v}_1} & 0 & \cdots & 0 \\ 0 & \widehat{\mathbf{C}}_{\mathbf{v}_2} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \widehat{\mathbf{C}}_{\mathbf{v}_n} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_n \end{bmatrix} + \begin{bmatrix} \widehat{\mathbf{C}}_{\mathbf{u}_1} & 0 & \cdots & 0 \\ 0 & \widehat{\mathbf{C}}_{\mathbf{u}_2} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \widehat{\mathbf{C}}_{\mathbf{u}_n} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_n \end{bmatrix} + \begin{bmatrix} \widehat{\mathbf{v}}_{\mathbf{k}_1} \\ \widehat{\mathbf{v}}_{\mathbf{k}_2} \\ \vdots \\ \widehat{\mathbf{v}}_{\mathbf{k}_n} \end{bmatrix} \tag{6.20}$$

Hier enthält **y** alle Ausgangssignale verarbeitender Komponenten, auch die der aktuellen Ebene und **u** die Eingangssignale der aktuellen Ebene.

Für das im Abschnitt 4.1 vorgestellte Synchronisationsverfahren benötigen die Komponenten die Verzögerungen ihrer Eingangssignale. Da bei der Zusammenfassung einer Ebene **u** eliminiert wird, stehen auf der obersten Ebene nur die dort vorhandenen Eingangssignale zur Verfügung. Um auch die der unteren Ebenen berechnen zu können, ist eine Abbildung der Ausgangssignale auf die Eingangssignale zu schaffen. Prinzipiell beinhaltet Gleichung 6.9 diese Möglichkeit. Darum sind nur **P** und $\mathbf{P}_0$ über alle Ebenen aufzubauen. Nach Bild 6.3 lassen sich die Blockeingangssignale $\begin{bmatrix} u_{0_1} & \cdots & u_{0_n} \end{bmatrix}^T = \mathbf{u}_0 = \begin{bmatrix} \tilde{u}_1 & \cdots & \tilde{u}_n \end{bmatrix}^T = \widetilde{\mathbf{u}}$ und Blockausgangssignale $\begin{bmatrix} y_{0_1} & \cdots & y_{0_n} \end{bmatrix}^T = \mathbf{y}_0 = \begin{bmatrix} \tilde{y}_1 & \cdots & \tilde{y}_n \end{bmatrix}^T = \widetilde{\mathbf{y}}$ zusammenfassen und stellen so den Vektor der Eingangssignale $\widetilde{\mathbf{u}}$ und der Ausgangssignale $\widetilde{\mathbf{y}}$ dar. Damit ergibt sich aus Gleichung 6.9:

$$\mathbf{u} = \mathbf{P} \cdot \mathbf{y} + \mathbf{P}_0 \cdot \widetilde{\mathbf{u}} \tag{6.21}$$

$$\begin{aligned} &= \mathbf{P}\cdot\mathbf{y}+\mathbf{P}_0\cdot(\widetilde{\mathbf{P}}\cdot\widetilde{\mathbf{y}}+\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0) \\ &= \mathbf{P}\cdot\mathbf{y}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\widetilde{\mathbf{y}}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0 \end{aligned} \tag{6.22}$$

und mit Gleichung 6.11:

$$\begin{aligned} \widetilde{\mathbf{y}} &= \mathbf{C}_0\cdot\mathbf{y}+\mathbf{D}\cdot\mathbf{u}_0 \\ &= \mathbf{C}_0\cdot\mathbf{y}+\mathbf{D}\cdot\tilde{\mathbf{u}} \\ &= \mathbf{C}_0\cdot\mathbf{y}+\mathbf{D}\cdot\left(\widetilde{\mathbf{P}}\cdot\widetilde{\mathbf{y}}+\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0\right) \\ \tilde{\mathbf{y}}-\mathbf{D}\cdot\widetilde{\mathbf{P}}\cdot\widetilde{\mathbf{y}} &= \mathbf{C}_0\cdot\mathbf{y}+\mathbf{D}\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0 \\ \tilde{\mathbf{y}} &= \left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{C}_0\cdot\mathbf{y}+\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{D}\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0 \end{aligned} \tag{6.23}$$

nach Einsetzen von Gleichung 6.23 in 6.22 ergibt sich:

$$\begin{aligned} \mathbf{u} &= \mathbf{P}\cdot\mathbf{y}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left[\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{C}_0\cdot\mathbf{y}+\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{D}\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0\right]+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0 \\ &= \mathbf{P}\cdot\mathbf{y}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{C}_0\cdot\mathbf{y}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{D}\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}_0\cdot\widetilde{\mathbf{u}}_0 \\ &= \left[\mathbf{P}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{C}_0\right]\cdot\mathbf{y}+ \\ &\quad\left[\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{D}\cdot\widetilde{\mathbf{P}}_0+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}_0\right]\cdot\widetilde{\mathbf{u}}_0 \end{aligned} \tag{6.24}$$

Hier treffen mit $\mathbf{u}$ und $\mathbf{y}$ sowie $\tilde{\mathbf{u}}_0$ Signale aus verschiedenen Ebenen aufeinander. Wenn man berücksichtigt, dass der Aufbau der Matrizen und Vektoren von unten nach oben erfolgt, enthalten $\mathbf{u}$ und $\mathbf{y}$ die Signale der untersten Ebene. Das sind ausschließlich die Signale, die verarbeitende Komponenten verbinden, während $\tilde{\mathbf{u}}_0$ die Eingangssignale enthält. Nach Durchlaufen aller Hierarchieebenen hat man eine Abbildung der Ausgangssignale auf die Eingangsignale ohne die Signale in den Hierarchieblöcken. Die Matrizen $\widehat{\mathbf{P}}$ und $\widehat{\mathbf{P}}_0$ über zwei Ebenen sind folgendermaßen zu bilden:

$$\widehat{\mathbf{P}} = \mathbf{P}+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{C}_0 \tag{6.25}$$

$$\widehat{\mathbf{P}}_0 = \mathbf{P}_0\cdot\widetilde{\mathbf{P}}\cdot\left(\mathbf{I}-\mathbf{D}\cdot\tilde{\mathbf{P}}\right)^{-1}\mathbf{D}\cdot\widetilde{\mathbf{P}}_0+\mathbf{P}_0\cdot\widetilde{\mathbf{P}}_0 \tag{6.26}$$

6.4 Anwendung des Modells auf die vorhandenen Komponenten

Für bestimmte Basiskomponenten kann *keine* Verzögerungsfunktion mit dem oben geschilderten Verfahren gebildet werden, da diese aus logischen Elementen und Flip-Flops gebildet werden wie zum Beispiel der Addierer (siehe Abschnitt 2.3 auf Seite 18). Für solche Komponenten ist eine Funktion nach Gleichung 6.10 und 6.11 vom Entwickler zu bilden. Bei vielen Komponen-

ten wird $C_0 = 1$ gelten. Zum systematischen Bilden von Gleichung 6.19 ist die Funktion jedoch trotzdem erforderlich. Die numerische Verarbeitung der Operanden nimmt in den meisten Fällen mindestens einen Takt in Anspruch. Dies bedeutet, dass $v_k = 1$ bei der Berechnung der Ausgangsverzögerungen zur Eingangsverzögerung u_0 zu addieren ist.

6.4.1 Addierer, Subtrahierer

Aus Sicht der Verzögerung verhalten sich der Addierer und Subtrahierer gleich. Daher wird im folgenden nur noch vom Addierer gesprochen. Dieser verfügt über zwei Eingänge. Da der Algorithmus (siehe Abschnitt 2.3 auf Seite 18) die Ausrichtung der Operandenbits auf gleiche Wertigkeit erfordert, wird die Verzögerung nur von dem Signal mit der größeren Verzögerung bestimmt. Zur Ausrichtung der Operanden wird in den weniger verzögerten Datenstrom ein Schieberegister eingebaut. Die Länge des Registers r_1 bzw r_2 entspricht der Differenz der Verzögerungen.

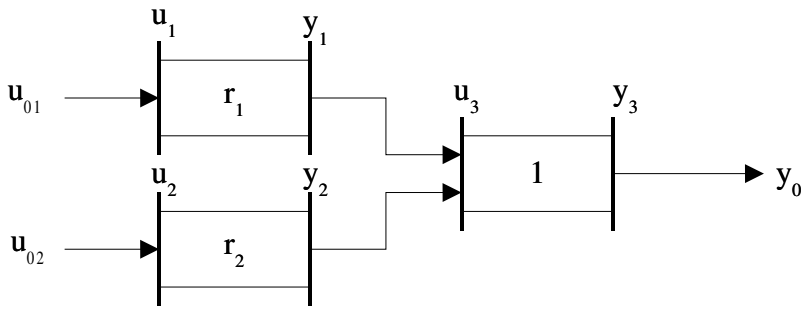

Abbildung 6.4: Definition der Bezeichner am Addierer

Die Topologie lässt sich folgendermaßen beschreiben:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_{31} \\ u_{32} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_{01} \\ u_{02} \end{bmatrix} \tag{6.27}$$

Die Verzögerung entsteht durch die optionalen Schieberegister r_1 und r_2 und die eigentliche Addition. Durch die erforderliche Ausrichtung der Operandenbits muss die Verzögerung bei y_1 und y_2 gleich sein, d.h.

$$\begin{aligned} r_1 + u_1 &= r_2 + u_2 \\ 0 &= r_1 - r_2 + u_1 - u_2 \\ 0 &= \begin{bmatrix} 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \end{aligned} \tag{6.28}$$

Diese Gleichheitsbedingung geht mit in die Optimierung ein. Andererseits führt die Addition dieser Verzögerungen $(y_1 + y_2)$ zu einem falschen, doppelt so großen Ergebnis. Um bei einem einfachen Berechnungsverfahren zu bleiben, kann diese Verdopplung ausgenutzt werden. Dazu werden die Operandenverzögerungen und die doppelte Verzögerung (siehe ② in Gleichung 6.29) durch die Addition zusammengefasst und durch zwei dividiert (Gleichung 6.30).

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ ② \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_{31} \\ u_{32} \end{bmatrix} \tag{6.29}$$

$$y_0 = \begin{bmatrix} 0 & 0 & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{6.30}$$

oder kurz

$$\begin{aligned} y_0 &= \frac{1}{2}(r_1 + u_1 + r_2 + u_2 + 2) \\ &= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + 1 \end{aligned}$$

Unter Einbeziehung von Gleichung 6.28 ergibt sich damit

$$\begin{aligned} \begin{bmatrix} y_0 \\ 0 \end{bmatrix} &= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ y_0 &= \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ 0 \end{bmatrix} \end{aligned} \tag{6.31}$$

also

$$\mathbf{C}_0 = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

6.4.2 Speicherzelle

Die Speicherzelle dient zur Implementation von $\mathbf{z}^{-1}$ (siehe Abschnitt 3.3.1 auf Seite 44), also der Verschiebung eines Signals um eine Tastperiode (n Takte) nach hinten. Ein solches Signal wird im Laufe der weiteren Signalverarbeitung immer zu einem weniger verzögerten Signal addiert. Um das Einfügen weiterer Schieberegister im Addierer zu verhindern, müssen in einem solchen Fall beide Signale mit der gleichen rechnerischen Verzögerung dort ankommen.

Das Beispiel im Bild 6.6 zeigt einen Differenzierer. Zu dessen Realisierung ist vom aktuellen

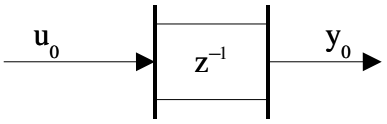

Abbildung 6.5: Definition der Verzögerungen an der Speicherzelle

Tastwert der vorherige zu subtrahieren (siehe Abschnitt 3.3.2 auf Seite 44). Die Verschiebung wird durch Einfügen eines Schieberegisters der Länge n erreicht.

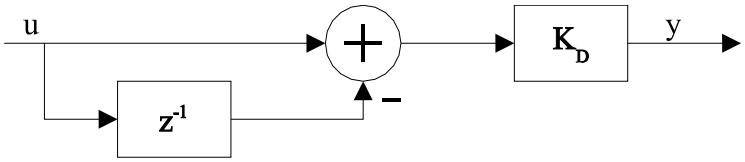

Abbildung 6.6: Speicherzelle im Differenzierer

Damit würde die Verzögerung y_0 am Ausgang der Memoryzelle $n + u_0$ betragen. Wenn dieser Wert dem nachfolgenden Addierer übergeben wird, gleicht dieser die Verzögerung wieder aus, da er beide Operanden auf gleiche Wertigkeit ausrichten will. Darum muss der Wert bei beiden Eingängen gleich sein. Weiterhin handelt es sich um eine gewollte Verzögerung, die nicht korrigiert werden soll. Vor der Forderung der Gleichheit der Verzögerungen am nachgeschalteten Addierer ergibt sich folgende Beziehung:

$$\begin{aligned} d &= u_0 - y_0 \\ y_0 &= -1 \cdot d + 1 \cdot u_0 \end{aligned} \tag{6.32}$$

Da grundsätzlich $u_0 \geq y_0$ gilt, ist immer $d \geq 0$ gegeben. Damit entspricht d der tatsächlichen Differenz der Verzögerungen beider Signale. Das Ziel ist jedoch eine Verschiebung des Datenstromes um eine Tastperiode, also um n Takte. Dementsprechend ist die Differenz dazu einzufügen. Daher gilt für die Implementierung

$$d_i = n - d \tag{6.33}$$

D.h., wenn keine zusätzliche Verzögerungen durch weitere Komponenten auftreten, wird eine Verzögerung, die der Länge der Verarbeitungswortbreite entspricht, eingebaut. Für den Fall weiterer Verzögerungen durch andere Komponenten können diese durch die Memoryzelle ausgeglichen werden. Weiterhin ist $C_0 = 1$.

Es stellt sich die Frage, ob mit der Memoryzelle nur eine rechnerische Beeinflussung der Verzögerung vorgenommen werden soll. Dann kann der ohnehin im Addierer vorhandene Verzögerungsausgleich zur Implementierung genutzt werden. Dazu müsste die Speicherzelle aus dem VHDL-Quellcode entfernt werden. Diesem zusätzlichen Aufwand steht kein Vorteil entgegen, da sich der Logikaufwand nicht reduziert.

6.4.3 Verstärker

Gemäß Abschnitt 2.6.1 auf Seite 27 gibt es zwei verschiedene Verstärkertypen, die sich im Nenner unterscheiden einer mit immer ganzzahligem Verstärkungsfaktor und der andere mit rationalem Verstärkungsfaktor. Der Unterschied besteht im Weglassen von d-niederwertigen Bits beim rationalen Verstärkungsfaktor. Dadurch entsteht eine zusätzliche Verzögerung gegenüber dem Verstärker mit einem ganzzahligen Verstärkungsfaktor. Für alle Verstärker gilt Gleichung 6.34.

$$\begin{aligned} y_0 &= 1+d+u_0 \\ &= 1\cdot u_0+(1+d) \end{aligned} \tag{6.34}$$

$$C_0 = 1$$

6.4.4 Schieberegister

Das Schieberegister verzögert den Datenstrom um k Takte, die im Unterschied zur Speicherzelle auch in die Verzögerung eingehen.

$$y_0 = u_0 + k$$

$$C_0 = 1$$

6.4.5 Multiplikation

Die Multiplikation verwendet ähnlich der Addition zwei Operanden, die aufeinander zu synchronisieren sind. Daher gelten hier dieselben Gleichungen wie beim Addierer. Die mögliche Skalierung des multidiv-Operators führt zu einem zusätzlichen Parameter d in $\mathbf{v_k}$.

$$\begin{aligned} \begin{bmatrix} y_0 \\ 0 \end{bmatrix} &= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1+d \\ 0 \end{bmatrix} \\ y_0 &= \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ 0 \end{bmatrix} \end{aligned} \tag{6.35}$$

6.4.6 Modulo 2^k

Der Operator (siehe Abschnitt 2.6.3 auf Seite 29) schneidet nur die oberen k Bit ab. Damit ergibt sich nur eine Verzögerung durch das Schieben.

$$y_0 = u_0 + 1$$

$$C_0 = 1$$

6.4.7 Div 2^k

Der Operator (siehe Abschnitt 2.6.2 auf Seite 28) schneidet die unteren k Bit ab. Damit ergibt sich folgende Verzögerung:

$$y_0 = u_0 + k + 1$$

$$C_0 = 1$$

6.4.8 Kennlinienapproximation

Die Kennlinienapproximation (siehe Abschnitt 2.9 auf Seite 31) ist ein nichtlineares Element. Damit ergibt sich immer eine Verzögerung größer als die Verarbeitungswortbreite n (d resultiert vom Parameter Teiler)

$$y_0 = u_0 + n + d + 3$$

$$C_0 = 1$$

6.4.9 Differenzierer

Für den Differenzierer nach Abschnitt 3.3.2 auf Seite 44 ergibt sich folgende Beziehung:

$$y_0 = u_0 + 1$$

$$C_0 = 1$$

6.5 Lösen des Gleichungssystems

Das Ziel zur Gewinnung des Gleichungssystem war es, eine Möglichkeit zu finden, die Anzahl der zur Synchronisation erforderlichen Verzögerungselemente r_i, d_{z_i} und der sich daraus ergebenden Verzögerungen y_i berechnen zu können, so dass deren Anzahl minimal ist. Das heißt:

$$\Sigma r_i + \Sigma d_{z_i} + \Sigma y_i = \text{Min!} \tag{6.36}$$

Bei den sich im Abschnitt 6.2 ergebenden Gleichungen handelt es sich ausschließlich um lineare Beziehungen. Dies ergibt ein lineares Optimierungsproblem, siehe [Bro91, S. 695].

Dessen Ausgangspunkt ist eine Gleichung der Form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \text{ mit } \mathbf{x} \geq \mathbf{0} \tag{6.37}$$

$$Q(\mathbf{x}) = \mathbf{p^T} \cdot \mathbf{x} = \text{Min!} \tag{6.38}$$

Hier entspricht Gleichung 6.36 bereits Gleichung 6.38, wobei $Q = \Sigma r_i + \Sigma d_{z_i} + \Sigma y_i$ ist. Weiterhin gilt $\mathbf{p^T} = \mathbf{1}$, denn alle Variablen gehen mit demselben Gewicht ein. Die Bedingung $\mathbf{x} \geq \mathbf{0}$ ist immer erfüllt, da x_i die Signalverzögerungen und die zu implementierenden Verzögerungsstufen darstellen und immer positiv sind.

Jedoch muss Gleichung 6.38 aus Gleichung 6.9 und 6.10 gebildet werden. Da bei den unteren Ebenen bisher **u** eliminiert wurde, geschieht es auch hier.

$$\mathbf{y} = \mathbf{C_v} \cdot \mathbf{v} + \mathbf{C_u} \cdot (\mathbf{P} \cdot \mathbf{y} + \mathbf{P}_0 \cdot \mathbf{u}_0) + \mathbf{v_k} \tag{6.39}$$

Um auf Gleichung 6.37 zu kommen, müssen in **x** alle Variablen und in **b** alle Konstanten zusammengefasst werden. Damit ergibt sich:

$$\begin{aligned}
\mathbf{y} &= \mathbf{C_v} \cdot \mathbf{v} + \mathbf{C_u} \cdot \mathbf{P} \cdot \mathbf{y} + \mathbf{C_u} \cdot \mathbf{P}_0 \cdot \mathbf{u}_0 + \mathbf{v_k} \\
0 &= \mathbf{C_v} \cdot \mathbf{v} + (\mathbf{C_u} \cdot \mathbf{P} - \mathbf{I}) \cdot \mathbf{y} + \mathbf{C_u} \cdot \mathbf{P}_0 \cdot \mathbf{u}_0 + \mathbf{v_k} \\
-\mathbf{v_k} - \mathbf{C_u} \cdot \mathbf{P}_0 \cdot \mathbf{u}_0 &= \mathbf{C_v} \cdot \mathbf{v} + (\mathbf{C_u} \cdot \mathbf{P} - \mathbf{I}) \cdot \mathbf{y} \\
\begin{bmatrix} -\mathbf{I} & -\mathbf{C_u} \cdot \mathbf{P}_0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v_k} \\ \mathbf{u}_0 \end{bmatrix} &= \begin{bmatrix} \mathbf{C_v} & \mathbf{C_u} \cdot \mathbf{P} - \mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v} \\ \mathbf{y} \end{bmatrix}
\end{aligned} \tag{6.40}$$

und somit sind:

$$\mathbf{A} = \begin{bmatrix} \mathbf{C_v} & \mathbf{C_u} \cdot \mathbf{P} - \mathbf{I} \end{bmatrix} \tag{6.41}$$

$$\mathbf{b} = \begin{bmatrix} -\mathbf{I} & -\mathbf{C_u} \cdot \mathbf{P}_0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v_k} \\ \mathbf{u}_0 \end{bmatrix} \tag{6.42}$$

$$\mathbf{x} = \begin{bmatrix} \mathbf{v} \\ \mathbf{y} \end{bmatrix} \tag{6.43}$$

Der Vektor **x** enthält damit alle internen Signale y_i und die unabhängigen Größen v_i, die zur Optimierung zur Verfügung stehen. Verfahren zur Lösung des Optimierungsproblems wurden in verschiedenen Programmen implementiert. Auf [Fou01] ist eine Zusammenstellung solcher Programme zu finden. Im Rahmen dieser Arbeit wird die Optimization Toolbox von Matlab genutzt, siehe [Mat01]. Diese stellt Funktionen zur Lösung linearer Optimierungsprobleme zur Verfügung. Weitere Beispiele zur Anwendung im Rahmen der linearen Optimierung findet man

in [Moh98] und [Bir95].

6.6 Beispiele zur Anwendung des Modells

Dieser Abschnitt schildert zur Veranschaulichung die Anwendung des Modells aus Abschnitt 6.2 an drei Beispielen.

6.6.1 Integrator

Den Anfang macht der im Bild 6.7 dargestellte Integrator als rückgekoppeltes System ohne algebraische Schleife.

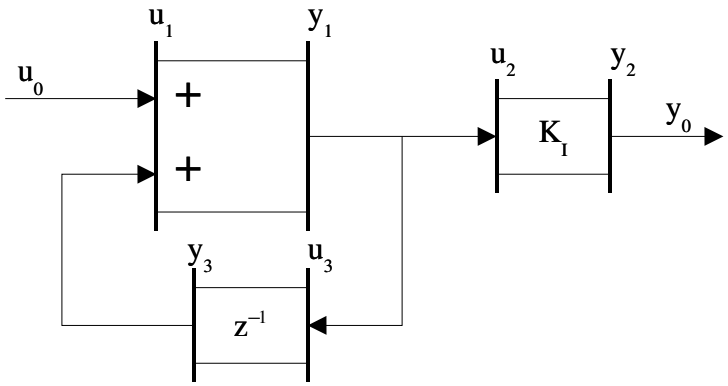

Abbildung 6.7: Verzögerungen am Integrator

Die Beziehungen für die drei Blöcke Addierer, Verstärker und Speicherzelle wurden aus Abschnitt 6.4 übernommen:

$$\begin{aligned}\begin{bmatrix} y_1 \\ 0 \end{bmatrix} &= \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} + 1 \\ y_2 &= 1 \cdot u_2 + (1 + d_{K_I}) \\ y_3 &= -1 \cdot d_z + 1 \cdot u_3 \end{aligned}$$

Die Freiheitsgrade zur Optimierung stellen $\mathbf{v} = \begin{bmatrix} r_1 & r_2 & d_z \end{bmatrix}^T$ dar. Damit kann ein Gleichungssystem gemäß Gleichung 6.10 aufgestellt werden. Die Null in der zweiten Zeile von $\mathbf{y}$ resultiert aus dem Gleichungssystem für den Addierer (siehe Abschnitt 6.4.1) und muss erhalten bleiben, um einen einheitlichen Aufbau von $\mathbf{y}$ zu gewährleisten.

$$\begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \\ d_z \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{12} \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 + d_{K_I} \\ 0 \end{bmatrix} \tag{6.44}$$

Die Blöcke sind folgendermaßen verknüpft:

$$\begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot u_0$$

$$u_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix}$$

$$u_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix}$$

Diese sind laut Gleichung 6.5 folgendermaßen zusammenzufassen:

$$\begin{bmatrix} u_{11} \\ u_{12} \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot u_0. \tag{6.45}$$

Für den Ausgang gilt:

$$y_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix}$$

Damit ergeben sich folgende Konstanten:

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{p}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{C_v} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad \mathbf{C_u} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{C_u} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{c}_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

und Variablen:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} r_1 \\ r_2 \\ d_z \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} u_{11} \\ u_{12} \\ u_2 \\ u_3 \end{bmatrix}$$

Im Sinne der Optimierung stellt aber nur **v** einen Freiheitsgrad dar. Denn **u** und **y** ergeben sich zwangsläufig, wenn für **v** konkrete Werte gewählt werden. Um das Optimierungsproblem zu lösen, sind in den Gleichungen 6.41 und 6.42 die Konstanten einzusetzen und in Gleichung 6.43 die Variablen, wobei letztere **u** nicht enthält, da es eliminiert werden konnte. Die Bildung von **x** ist nur erforderlich, um das Ergebnis der Optimierung interpretieren zu können. Weiterhin muss an dieser Stelle eine Festlegung der Verzögerung des Eingangssignals u_0 gemacht werden. Diese beträgt für ein Gesamtsystem meistens Null. Die Konstante d_{K_I} stellt eine zusätzliche Skalierung im Verstärker (siehe Abschnitt 6.4.3) dar und wurde hier mit Null ersetzt. Dem im Anhang B.2.1 auf Seite 109 befindlichen Matlab-Skript müssen nur die konstanten Matrizen übergeben werden. Daraus errechnet es selbständig **A** und **b** und ruft die Funktion "linprog" zur Lösung des Optimierungsproblems auf. Als Resultat der Optimierung erhält man:

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0 & -1 & 0 & 0 & 0.5 \\ 1 & -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -1 \\ 0 \\ -1 \\ 0 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} r_1 \\ r_2 \\ d_z \\ y_1 \\ 0 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 2 \\ 0 \end{bmatrix} \tag{6.46}$$

und nach Einsetzen von $\mathbf{y}$ in Gleichung 6.45

$$\mathbf{u} = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$$

Das heißt, am Ausgang des Addierers y_1 ergibt sich eine Verzögerung von einem Takt. Diese erhöht sich um einen weiteren Takt an $y_0 = y_2$ durch den Verstärker. Im Addierer werden keine weiteren Schieberegister eingebaut, da $r_1 = r_2 = 0$ ist. Die Speicherzelle enthält entsprechend Gleichung 6.33 bei einer Verarbeitungswortbreite von $n = 16$ ein 15stufiges Schieberegister. Daraus resultiert eine tatsächliche Verschiebung an y_3 von 16 Takten. Das entspricht wie gefordert einer Verschiebung um eine Tastperiode.

Zur Vervollständigung des Beispiels sollen noch die Matrizen $\widehat{\mathbf{C}}_\mathbf{v}$, $\widehat{\mathbf{C}}_\mathbf{u}$ und $\widehat{\mathbf{v}}_\mathbf{k}$ (siehe Gleichung 6.13 bis 6.15) berechnet werden, falls der Integrator in ein größeres System eingebaut werden soll. Dazu dient wieder das Matlab-Skript (siehe B.2.1 auf Seite 109).

$$\widehat{\mathbf{C}}_\mathbf{v} = \begin{bmatrix} 1 & 1 & -1 \\ 0 & -2 & 2 \\ 1 & 1 & -1 \\ 1 & 1 & -2 \end{bmatrix} \quad \widehat{\mathbf{C}}_\mathbf{u} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \widehat{\mathbf{v}}_\mathbf{k} = \begin{bmatrix} 2 \\ -2 \\ 3 \\ 2 \end{bmatrix}$$

Dieses Ergebnis lässt sich validieren, indem es mit dem Resultat der Optimierung (Gleichung 6.46) in Gleichung 6.18 eingesetzt wird und die dann in Gleichung 6.19.

$$y_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \cdot \left(\begin{bmatrix} 1 & 1 & -1 \\ 0 & -2 & 2 \\ 1 & 1 & -1 \\ 1 & 1 & -2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \cdot 0 + \begin{bmatrix} 2 \\ -2 \\ 3 \\ 2 \end{bmatrix} \right) = 2 \tag{6.47}$$

Dies stimmt mit dem Ergebnis der Optimierung überein.

6.6.2 PID-Regler

Hier wird der im vorigen Abschnitt berechnete Integrator mit all seinen Werten verwendet, denn der Verstärkungsfaktor geht mit in die Verzögerung ein. Bild 6.8 zeigt den Regler aus Sicht der Verzögerungen.

Die Topologie wurde gemäß Gleichung 6.11 aufgebaut.

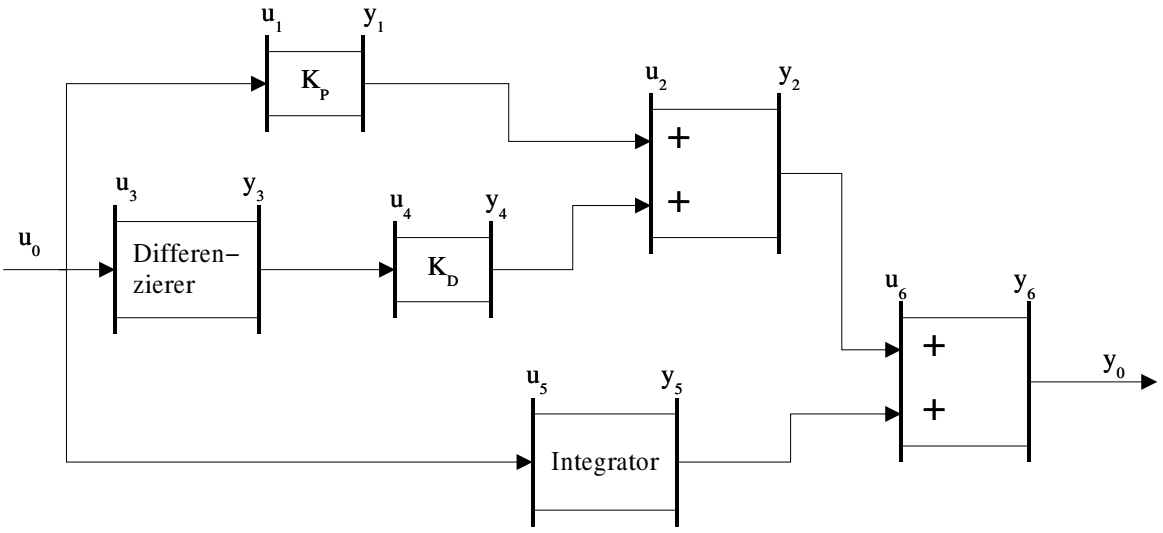

Abbildung 6.8: Verzögerungen am PID-Regler

$$\begin{bmatrix} u_1 \\ u_{21} \\ u_{22} \\ u_3 \\ u_4 \\ u_5 \\ u_{61} \\ u_{62} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{C}0_{gaindiv} \\ \mathbf{C}0_{adder} \\ \mathbf{C}0_{differenzierer} \\ \mathbf{C}0_{gaindiv} \\ \mathbf{C}0_{integrator} \\ \mathbf{C}0_{adder} \end{bmatrix} \cdot \mathbf{I} \cdot \begin{bmatrix} y_1 \\ \mathbf{y}2 \\ y3 \\ y_4 \\ \mathbf{y}5 \\ \mathbf{y}6 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \cdot u_0$$

$$\begin{bmatrix} u_1 \\ u_{21} \\ u_{22} \\ u_3 \\ u_4 \\ u_5 \\ u_{61} \\ u_{62} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ 0 \\ y_3 \\ y_4 \\ y_{51} \\ 0 \\ y_{52} \\ y_{53} \\ y_6 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \cdot u_0$$

Beschreibung der generierten Verzögerungen

$$\begin{bmatrix} y_1 \\ \mathbf{y}_2 \\ y3 \\ y_4 \\ \mathbf{y}_5 \\ \mathbf{y}_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ \mathbf{C_{v_{adder}}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathbf{C_{v_{integrator}}} & 0 \\ 0 & 0 & \mathbf{C_{v_{adder}}} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_2 \\ \mathbf{v}_5 \\ \mathbf{v}_6 \end{bmatrix} + \begin{bmatrix} c_{u_{gaindiv}} \\ \mathbf{C_{u_{adder}}} \\ c_{u_{differenzierer}} \\ c_{u_{gaindiv}} \\ \mathbf{C_{u_{integrator}}} \\ \mathbf{C_{u_{adder}}} \end{bmatrix} \cdot \mathbf{I} \cdot \begin{bmatrix} u_1 \\ \mathbf{u}_2 \\ u_3 \\ u_4 \\ u_5 \\ \mathbf{u}_6 \end{bmatrix} + \begin{bmatrix} v_{k_{gaindiv}} \\ \mathbf{v_{k_{adder}}} \\ v_{k_{differenzierer}} \\ v_{k_{gaindiv}} \\ \mathbf{v_{k_{integrator}}} \\ \mathbf{v_{k_{adder}}} \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ 0 \\ y_3 \\ y_4 \\ y_{51} \\ 0 \\ y_{52} \\ y_{53} \\ y_6 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_{21} \\ r_{22} \\ r_{51} \\ r_{52} \\ d_{Z5} \\ r_{61} \\ r_{62} \end{bmatrix} + \begin{bmatrix} 1 & 0 & & & \cdots & & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & & & & \\ & 1 & -1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ \vdots & & & & & 1 & \vdots \\ & & & & & 0 & \\ & & & & & 1 & \\ & & & & & 1 & 0 \\ & & & & & & \frac{1}{2} \quad \frac{1}{2} \\ 0 & & \cdots & & & 0 & 1 \quad -1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_{21} \\ u_{22} \\ u_3 \\ u_4 \\ u_5 \\ u_{61} \\ u_{62} \end{bmatrix} + \begin{bmatrix} 1+d_{K_P} \\ 1 \\ 0 \\ 1 \\ 1+d_{K_D} \\ 2 \\ -2 \\ 3 \\ 2 \\ 1 \\ 0 \end{bmatrix}$$

Dieses Problem wurde wieder mit Matlab gelöst, jedoch ist das in einem ersten Versuch ermittelte Ergebnis

$$\mathbf{v} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T \text{ und } \mathbf{y} = \begin{bmatrix} 1 & 3 & 0 & 1 & 2 & 1 & 0 & 2 & 0 & 3.5 & 1 \end{bmatrix}^T$$

zwar minimal aber falsch. Dies fällt sofort am gebrochenen Verzögerungswert für $y_{61} = 3.5$ auf. Weiterhin spiegelt y_{62} die Gleichheitsbedingung am Addierer (Block 6) wieder und erfordert $y_{62} = 0$. Bei weiterer Betrachtung sieht man auch, dass $v_6 = r_{62} = 0$ nicht stimmen kann, denn die Verzögerungen der Eingangssignale am Addierer (Block 6) sind mit $y_{21} = 3$ und $y_{52} = 2$ unterschiedlich. Das Problem resultiert daraus, dass die richtige Lösung $y_{61} = 4$, $y_{62} = 0$ und $v_6 = 1$ nicht das absolute Minimum darstellt. Dem könnte durch eine andere Wahl des Gewichtsvektors $\mathbf{p^T}$ in Gleichung 6.38 begegnet werden. Das birgt jedoch die Gefahr in sich, schlecht konditionierte Matrizen zu erzeugen, da die Gewichte zu erhöhen wären und sie gleichzeitig mit in die Optimierung eingehen. Weiterhin bietet die linprog-Funktion von Matlab noch die Möglichkeit, ein Lösungsintervall anzugeben. Dieser Ansatz wurde verfolgt, um eine schlechte Konditionierung zu vermeiden. Die Einschränkung des Lösungsbereichs der Gleicheitsbedingungen auf $\mathbf{y_{max}} = \begin{bmatrix} s & s & 0 & s & s & s & 0 & s & s & s & 0 \end{bmatrix}^T$ führte zur Berechnung des richtigen Ergebnisses. Hier ist s so zu wählen, dass es immer größer als die maximal zu erwartende Verzögerung ist. Damit wird die Gleichheitsbedingung von Gleichung 6.28 realisiert.

$$\mathbf{v} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}^T \text{ und } \mathbf{y} = \begin{bmatrix} 1 & 3 & 0 & 1 & 2 & 1 & 0 & 2 & 0 & 4 & 0 \end{bmatrix}^T$$

Zur Lösung dieses Problems wurde nicht mehr ein einzelnes Skript verwendet, sondern ein modularer Ansatz aus einem Skript und mehreren Funktionen gewählt. Die Funktionen realisieren das Auslesen von Konstanten, das Zusammensetzen der Matrizen und die Optimierung. Die Quelltexte sind im Abschnitt B.2.2 auf Seite 110 zu finden.

6.6.3 System mit algebraischer Schleife

Wie bereits im Abschnitt 6.1 beschrieben, ist die Implementierung algebraischer Schleifen nicht möglich. Darum soll die Überprüfung auf Freiheit von algebraischen Schleifen hier gezeigt werden. Eine algebraische Schleife ist dadurch gekennzeichnet, dass es im gesamten Pfad des zurückgekoppelten Signalflusses keine Speicherzelle gibt. Ausgangspunkt ist das in Bild 6.9 dargestellte System.

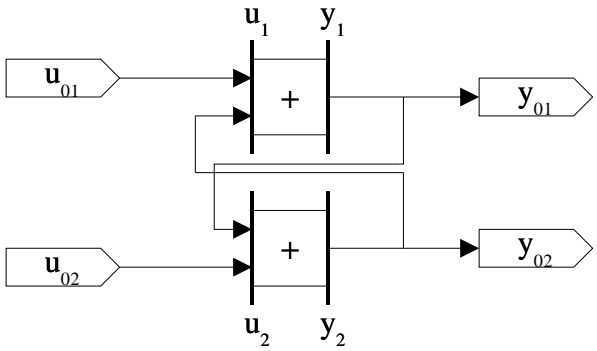

Abbildung 6.9: Beispiel zur Analyse auf algebraische Schleifen

$$\begin{bmatrix} y_1 \\ 0 \\ y_2 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} \\ r_{12} \\ r_{21} \\ r_{22} \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ 0 \\ y_2 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{01} \\ u_{02} \end{bmatrix}$$

Damit ergeben sich folgende Matrizen:

$$\mathbf{C_v} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & -1 \end{bmatrix} \quad \mathbf{C_u} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & -1 \end{bmatrix} \quad \mathbf{v_k} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{P}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Mit diesen Matrizen wurde wieder versucht mittels, Matlab eine Lösung zu finden. Da jedoch Bedingung 6.28 nicht erfüllt werden kann, liefert die Optimierung kein Ergebnis. Leider gibt die Fehlermeldung auch keinen Hinweis auf den Ort der algebraischen Schleife.

6.7 Allgemeiner Ablauf zur Umsetzung des Verfahrens

Anhand der beim Beispiel: “PID-Regler” gewonnen Erfahrungen zeichnet sich ein Weg ab, wie das beschriebene Verfahren automatisiert werden kann. Dazu wird für jede Komponente eine eigene Matlab-Funktion generiert. Diese enthält auch die Aufrufe der unterlagerten Komponenten, so dass sich hier die Hierarchie wiederspiegelt. Auf oberster Ebene befindet sich ein Skript, das die Optimierung vornimmt. Ausgangspunkt ist der VHDL-Quelltext und eine Bibliothek, die Matrizen und Vektoren für bekannte Komponenten enthält. Diese Komponenten sind einerseits die im Abschnitt 6.4 beschriebenen Komponenten und zusätzlich bereits analysierte und zusammengefasste Blöcke, wie z.B. ein PID-Regler.

In der ersten Phase wird der Quelltext, beginnend mit der obersten Ebene, geparst. Dabei sind Informationen über die verwendeten Komponenten und ihre Verbindungen zu gewinnen. Aus den Verbindungen, d.h. den vorhandenen Signalen, ergeben sich die Matrizen $\mathbf{C_0}$, $\mathbf{D}$, $\mathbf{P}$ und $\mathbf{P_0}$. Während $\mathbf{C_u}$, $\mathbf{C_v}$ und $\mathbf{v_k}$ aus den verarbeitenden Komponenten resultieren. Bei jeder anzulegenden Instanz wird in der Bibliothek nachgesehen, ob bereits Informationen über die zugehörige Komponente enthalten sind. Anderenfalls erfolgt die Ausgabe eines Fehlers und der Abbruch des Parsens. Nach dem Parsen einer Komponente wird ihr Code um zusätzliche Funktionen zum Auslesen der Verzögerungen aus einer Tabelle ergänzt und ein Package zur Generierung der Matlab-Funktion angelegt.

Während der Elaboration findet die zweite Phase der Optimierung statt. Dazu wird vor dem Anlegen der ersten Instanz eine Funktion aus dem generierten Package aufgerufen. Diese ruft wiederum die entsprechenden Funktionen in den Packages der unterlagerten Komponenten auf usw., so dass parallel zu der Hierarchie der Signalverarbeitung auch noch eine Hierarchie zur Verarbeitung der Verzögerungen existiert. Jede dieser Funktionen schreibt eine Matlab-Funktion in eine Datei. Die Testbench, als Instanz auf oberster Ebene, veranlasst nun den Start von Matlab zur Lösung des Optimierungsproblems. Aus dem Ergebnis werden die Verzögerungen ausgelesen und in einem Feld abgespeichert. Jede Instanz liest dann dort die für sie erforderlichen Werte aus.

Kapitel 7

Implementierung

7.1 Einbindung in Entwicklungswerkzeuge

Die Implementierung ist nicht völlig losgelöst von der Einbindung des Verfahrens in vorhandene Entwicklungswerkzeuge. Hier dient Mentor Graphics' HDL-Designer als Blockdiagramm-Editor. Dieses Werkzeug ermöglicht die Erweiterung durch in Perl geschriebene Plug-ins. Die eigentliche Optimierung läuft während der Elaboration ab. Um die dazu erforderlichen Vorbereitungen vornehmen zu können, wurde ein Plug-in 'Optimization Preparer' geschrieben. Dies analysiert den VHDL-Quellcode und nimmt daran Veränderungen und Ergänzungen vor. Dazu wird dem Plug-in der Dateiname der Komponente übergeben.

Im Blockdiagrammeditor können die verschiedenen vorgefertigten Komponenten einfach eingefügt und miteinander verbunden werden (siehe Bild 7.1). Den zu jeder Komponente gehörenden generischen Parametern sind Werte zuzuweisen. Da für die Verzögerungswerte und die Indizes keine sinnfälligen Werte bekannt sind, werden sie einfach auf Null gesetzt. Während der Codemanipulation erfolgt die Zuweisung des richtigen Wertes durch einen Verweis auf einen Platz in der Verzögerungswerttabelle. Nach dem Ausfüllen aller leeren Plätze bei den Generics kann der VHDL-Code generiert werden. Im Anschluss daran wird der 'Optimization Preparer' gestartet. Dieser analysiert den Quellcode, fügt die Verweise auf die Plätze in der Verzögerungswerttabelle ein und hängt ein Package an, das den Code zur Generierung einer Matlab-Funktion enthält. Dieses Package ist nur zur Optimierung notwendig, enthält aber VHDL-Ausdrücke, welche nicht synthetisierbar sind. Um es zur Synthese wieder zu entfernen, werden spezielle Kommentare, die als Marken dienen, am Anfang und Ende eingefügt.

Der nachfolgende Start der Simulation veranlasst während der Elaboration das Schreiben der Matlab-Funktionen über einen Funktionsaufruf. Die Verzögerungswerte stehen in einer Konstantentabelle. Deren Initialisierung erfolgt über eine als 'foreign' deklarierte Prozedur und damit über das Foreign-Language-Interface von VHDL. Innerhalb dieser Prozedur wird zuerst Matlab gestartet, das Ende der Verarbeitung abgewartet und dann das Ergebnis eingelesen.

Bild 7.1 zeigt die Bearbeitung des PID-Reglers im Blockdiagrammeditor. Während Bild 7.2 den Design-Browser mit dem 'Optimization Preparer' darstellt.

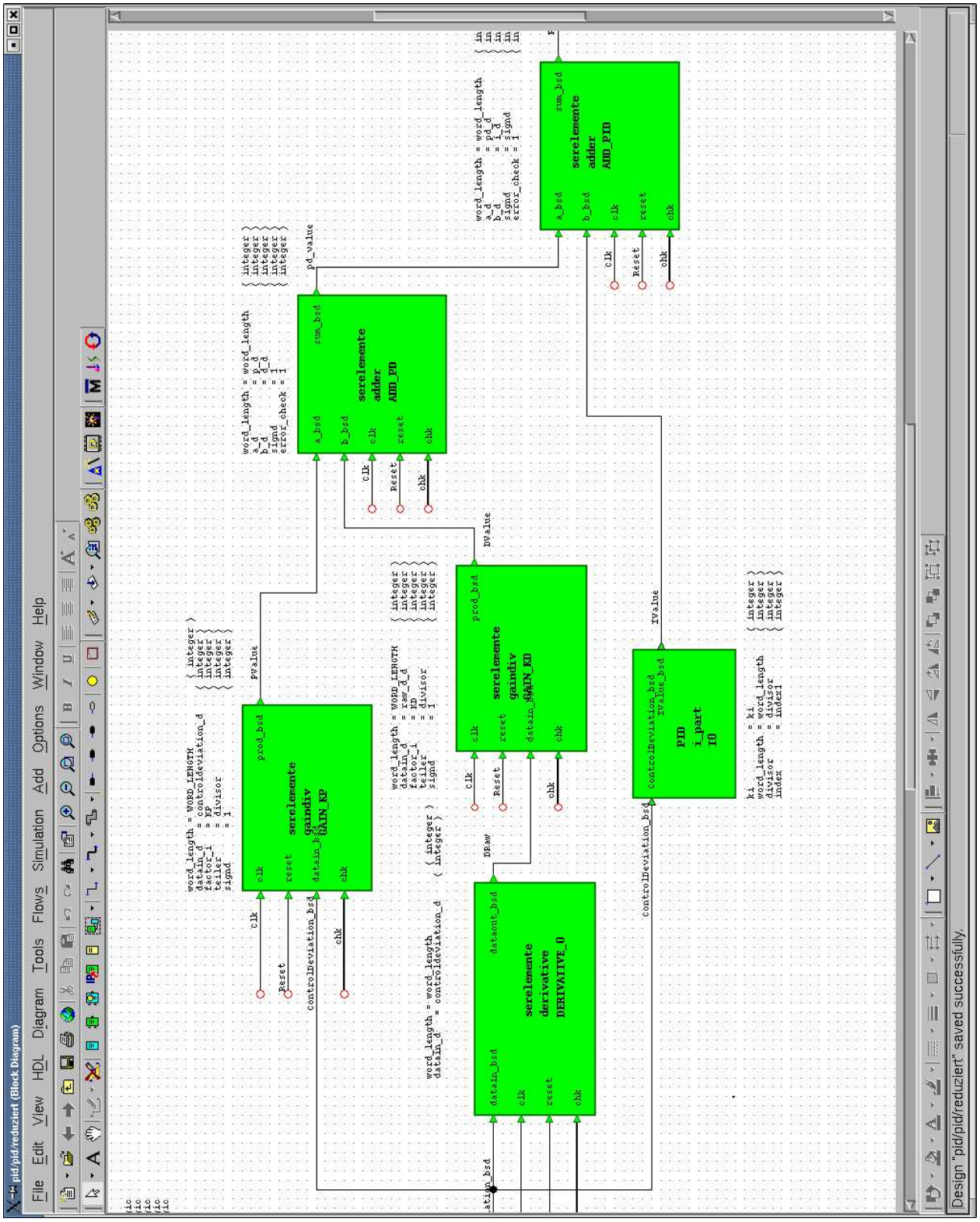

Abbildung 7.1: PID-Regler im Blockdiagrammeditor

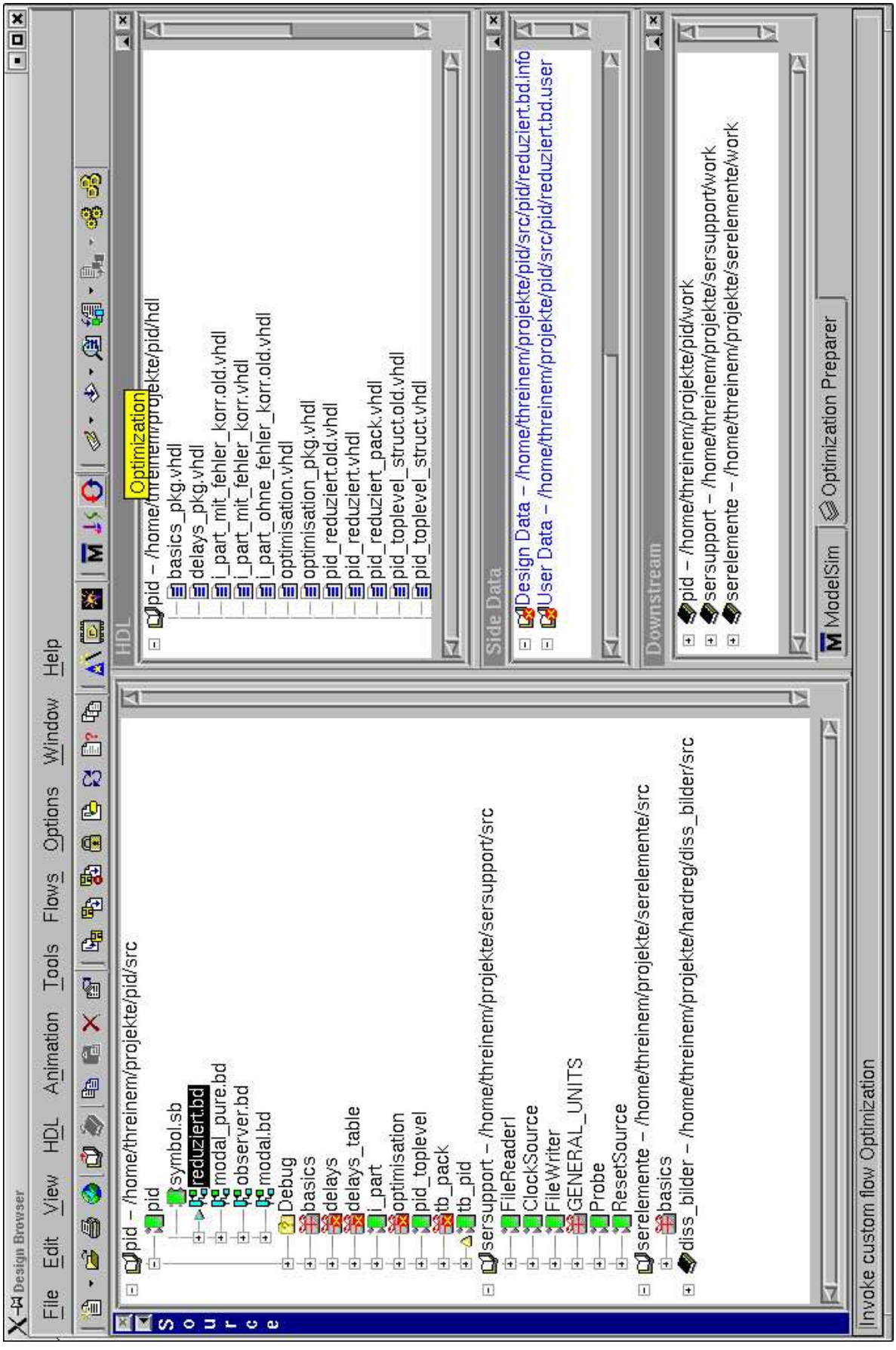

Abbildung 7.2: HDL-Designer

7.1.1 Anforderungen

Zur Anwendung der Gleichung 6.9 bis 6.11 auf Seite 65 sind die Matrizen $\mathbf{C_u}$, $\mathbf{C_v}$, $\mathbf{C}_0$, $\mathbf{D}$, $\mathbf{P}$, $\mathbf{P}_0$ und $\mathbf{v_k}$ über alle Hierarchieebenen aufzustellen. Die Matrizen spiegeln die Verknüpfungen der Signale untereinander wider. Darum müssen zuerst alle auftretenden bitseriellen Signale erfasst werden. Daraus werden dann die Matrizen $\mathbf{C}_0$, $\mathbf{D}$, $\mathbf{P}$, und $\mathbf{P}_0$ generiert, während $\mathbf{C_u}$, $\mathbf{C_v}$ und $\mathbf{v_k}$ aus den verarbeitenden Komponenten resultieren und innerhalb von Hierarchie-Komponenten nur weitergereicht werden. Innerhalb dieser sind alle Matrizen gemäß Gleichung 6.13 bis 6.15 auf Seite 66 sowie Gleichung 6.25 und 6.26 auf Seite 68 zusammenzufassen. Bei einigen Operatoren, z.B. Verstärkern (siehe Abschnitt 6.4.3), hängt die Verzögerung auch von übergebenen Parametern ab. Darum wird grundsätzlich eine Berechnungsvorschrift je Hierarchie-Komponente erstellt und in einer Datei abgespeichert.

Um die Verzögerungen aller Signale eines Designs berechnen zu können, muss ein Abbild geschaffen werden, welches alle anzulegenden Instanzen beinhaltet. Ausgangspunkt ist die Spezifikation mittels eines Blockdiagrammeditors von Mentor Graphics (Renoir bzw. HDL Designer). Der mit diesen Werkzeugen generierte VHDL-Code enthält keine Anweisungen, die eine bedingte Instanzierung ermöglichen würden. Wenn trotzdem Komponenten mit bedingter Instanzierung verwendet werden sollen, sind dafür vom Entwickler selber Funktionen zur Berechnung der Verzögerung zu implementieren.

7.1.2 Konvention

Um das Auffinden der bitseriellen Datensignale zu vereinfachen, lautet deren Bezeichnung immer `name+'_bsd'`. Passend dazu gibt es eine Konstante oder einen generischen Parameter (Delay-Generic), dem die Verzögerung des bitseriellen Signals übergeben wird. Das Format ist `name+'_d'`. Der Takt heißt immer `'clk'`, der Sync-Bus immer `'chk'`, das Reset immer `'reset'`. Diese Konvention ist bei der Erstellung von Komponenten zu berücksichtigen.

Um eine eindeutige Identifizierung des Toplevels zu ermöglichen, deklariert dies immer die folgenden Konstanten.

```
constant u0 : integer := 0;
constant index : integer := 0
```

Auf der obersten Ebene, innerhalb einer VHDL-Hierarchie, kann es keine generischen Parameter geben. Darum sind hier die Startwerte für die Verzögerungen und den Index (siehe Abschnitt: 7.2) über Konstanten zu deklarieren.

Der Zugriff auf die Datenbank mit den Index-Einträgen erfolgt über das “Library-Mapping” von Modelsim. Darum ist es wichtig, dass für den VHDL-Compiler und den “Optimization Preparer”

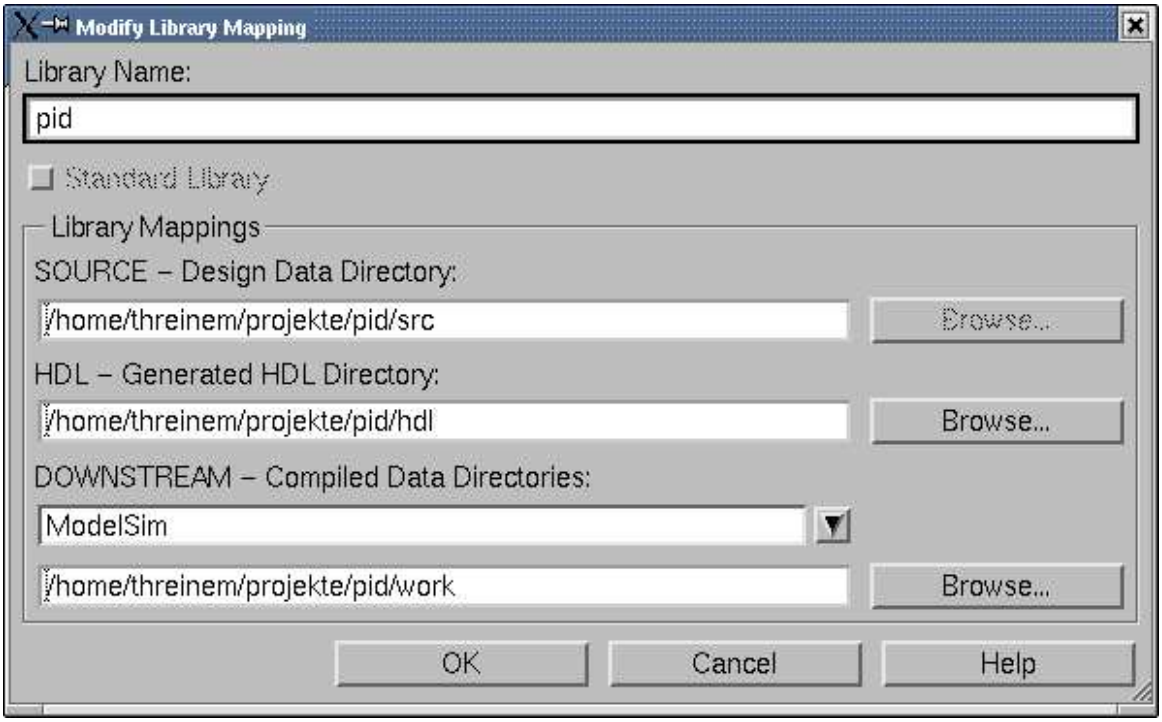

Abbildung 7.3: Library Mapping

dieselben Bibliotheksverzeichnisse (DOWNSTREAM - Compiled Data Directories) eingetragen werden (siehe Bild 7.3).

Als Projektverzeichnis für das aktuelle Projekt wird immer das Verzeichnis über dem Quelltext angenommen. Unter dem Projektverzeichnis befindet sich im Verzeichnis `'work'` die Standard-work-Bibliothek zum Projekt.

7.2 Komponententypen

Je nach Art der vorhanden Ein- und Ausgangssignale wird zwischen verschiedenen Komponententypen unterschieden. An oberster Stelle befindet sich die Testbench. Sie stellt die Simulationsumgebung für das ganze Design dar, versorgt dieses mit Eingangssignalen (Takt, Reset, Daten) und nimmt die Ausgangssignale entgegen. An oberster Stelle des Designs steht eine Komponente (Toplevel), die die Anschlüsse des Schaltkreises eineindeutig abbildet und daher über keine generischen Parameter[1] verfügt. In ihr ist die Konstante `'index'` zu deklarieren. Sie dient als Ausgangspunkt für die Berechnung der Einträge in der Delay-Tabelle (siehe Abschnitt 7.6.1.1). Darunter befinden sich die verarbeitenden und Hierarchie-Komponenten, die in ihrer Gesamtheit das Design darstellen. Da Hierarchie-Komponenten die Verzögerungswerte an die verarbeitenden Komponenten weitergeben, verfügen sie über einen Index-Parameter

[1] Generische Parameter (Generic) dienen zur Steuerung der Hardwareausprägung. Sie werden nur während der Elaboration ausgewertet und nicht zur Laufzeit. Sie sind mit der #define-Direktive in C vergleichbar.

(Index-Generic) zum Auslesen der Werte aus der Delay-Tabelle. Deren Werte werden den Verzögerungsparametern (Delay-Generic) der verarbeitenden Komponenten zugewiesen. Sie nehmen damit die Synchronisation vor. Tabelle 7.1 stellt die Komponententypen gegenüber.

	keine Generics	**Delay-Generics**	**Index-Generics**
keine Ein- und Ausgänge	Testbench	nicht sinnvoll	nicht sinnvoll
Ein- und Ausgänge vorhanden	Toplevel	verarbeitende Komponente	Hierarchie-Komponente

Tabelle 7.1: Komponentenarten in Abhängigkeit zu ihren Ein- und Ausgängen

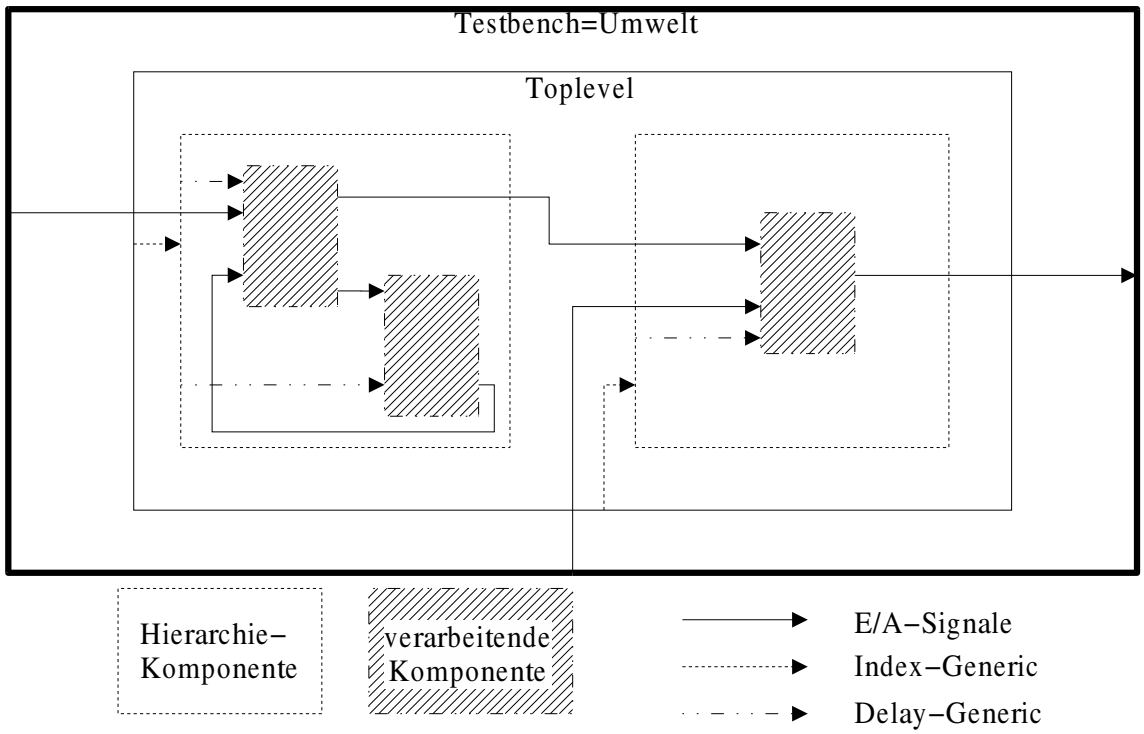

Abbildung 7.4: Die Komponententypen und ihre Kommunikationsbeziehungen

Im Bild 7.4 sind noch einmal alle Komponenten und ihre Kommunikationsbeziehungen zu sehen. Vor dem Hintergrund der Signalverarbeitung beginnen oder enden Signalketten immer in der Umwelt, die hier die Testbench darstellt. Da es in der Umwelt keine generischen Parameter gibt, beginnen Generic-Ketten im Toplevel und pflanzen sich in den verarbeitenden Komponenten fort, wobei beim Übergang von Hierarchie-Komponente zu verarbeitender Komponente ein Wechsel von Index-Generic zu Delay-Generic stattfindet.

7.3 Parsen des Quelltextes

Um die zur Generierung der Matrizen benötigten Informationen zu erhalten, ist der VHDL-Quellcode zu parsen. Dazu kann ein vorhandener Parser genutzt werden oder selbst einer ge-

schrieben werden. Es wurden einige vorhandene Parser getestet. Die Auswahl fiel auf einen in Perl geschriebenen freien Parser von Greg London [Lon02]. Dieser unterstützt einen großen Umfang von VHDL und erlaubte durch die Verfügbarkeit des Perl-Codes die Ergänzung um noch benötigte Sprachkonstrukte. Da der vorhandene Blockdiagramm-Editor durch Perl-Plugins erweitert werden kann, bot sich die Verwendung dieses Parsers an.

7.3.1 Objektmodell

Im folgenden werden die Klassen aufgelistet, die zum Speichern der Informationen dienen. Das % verweist auf einen Hash, das @ auf eine Liste und das $ auf einen Skalar. Wenn diesen Zeichen ein \ vorangestellt ist, wird ein Zeiger auf einen solchen Datentypen verwendet.

- Architecture
  Die Architecture-Klasse wird auf oberster Ebene verwendet und nimmt Verweise auf alle anderen Objekte auf. Dies ist auch im Einklang mit VHDL, denn hier stellt auch eine Architektur die Ausprägung einer Entität dar. Eine Klasse 'Entity' ist nicht erforderlich, da es innerhalb eines HDL-Designer-Blockdiagramms nur eine Architektur gibt.

  ```
  assignment      =>  \%assignment
  components      =>  \%components
  constants       =>  \%constants
  configurations  =>  \%configurations
  instances       =>  \%instances
  library         =>  \%library
  package         =>  \%package
  signals         =>  \%signals
  ```

- Assignment
  ```
  name      =>  $name       # Ziel
  waveform  =>  $waveform   # Quelle
  ```

- Components
  ```
  name      =>  $name
  generics  =>  \%generics
  ports     =>  \%ports
  ```

- Constants
  ```
  name      =>  $name
  pos       =>  $pos
  value     =>  $value
  type      =>  $type
  ```

- Configuration

name	=>	$name	
library	=>	$library	Bibliotheksname
entity	=>	$library_entity[1]	Entitätenname
architecture	=>	$architecture	

- Instance

name	=>	$name	
component	=>	$component	
generics	=>	\@generics	
ports	=>	\@ports	
hgenerics	=>	\%generics	
hports	=>	\%ports	

- Library

entities	=>	\@libs	Eine Liste mit allen deklarierten Bibliotheken.

- Signals

name	=>	$name	
direction	=>	$direction	

7.4 Generierung der Vektoren **u**, $\mathbf{u}_0$, **y** und $\mathbf{y}_0$

Zur Generierung der Vektoren **y**, $\mathbf{y}_0$, **u** und $\mathbf{u}_0$ wird zuerst eine Liste aller bitseriellen Signale der Architektur und der Entität angelegt. Da die Instanzen keine Information zur Signalrichtung liefern, muss diese aus der Komponentendeklaration ausgelesen werden. Somit ist es wichtig, dass beim Parsen auch die Konfigurationsdaten abgespeichert werden. Denn diese enthalten die Zuordnung zwischen Komponente und Instanz. Die Listeneinträge sind Hashes. Jeder Eintrag enthält:

- tatsächlicher Signalname
- formaler Signalname (findet sich in der Komponentendeklaration wieder)
- Instanzname
- Komponente, von der die Instanz abgeleitet wurde
- Signalrichtung (wurde bei Signalen aus der Komponentendeklaration ausgelesen)
- Name der Konstanten, die die Verzögerung enthält

Die Liste wird in zwei Durchläufen, zuerst nach dem formalen Signalnamen und dann nach dem Instanznamen, sortiert. Damit ist eine einheitliche Reihenfolge gewährleistet. Signale der Instanzen gehen in Abhängigkeit ihrer Richtung in u und y ein und Signale der Entität in u_0 und y_0.

Zuweisungen der Art `'signal1 <= signal2;'` können als Umbenennung des Signals betrachtet werden. Darum wird auch jedes weitere Auftreten von `signal1` durch `signal2` ersetzt und somit `signal1` eliminiert.

7.5 Generierung der Matrizen $\mathbf{C_u}$, $\mathbf{C_v}$, $\mathbf{C}_0$, $\mathbf{D}$, $\mathbf{P}$, $\mathbf{P}_0$ und $\mathbf{v_k}$

Die Matrizen $\mathbf{C_u}$, $\mathbf{C_v}$ und $\mathbf{v_k}$ spiegeln die Verbindungen innerhalb verarbeitender Komponenten wider. Darum sind sie vom Entwickler der Basiskomponenten aufzustellen und in einem m-File abzulegen (Beispiel siehe Abschnitt B.2.3). Da diese Komponenten selbstverständlich Ein- und Ausgänge haben, finden sich dort auch die Matrizen $\mathbf{P}$ und $\mathbf{P}_0$ wieder, werden aber mit Gleichung 6.13 bis 6.15 eliminiert. Die Matrizen $\mathbf{P}$, $\mathbf{P}_0$, $\mathbf{C}_0$ und $\mathbf{D}$ bilden die Vektoren $\mathbf{y}$, $\mathbf{y}_0$, $\mathbf{v}$, $\mathbf{u}$ und $\mathbf{u}_0$ innerhalb einer Hierarchie-Komponente aufeinander ab. Dies spiegelt die Verwendung des gleichen Signals im Quelltext wider. Zur Generierung der Matrizen ist daher nur nach dem Auftreten eines Signals in den entsprechenden Vektoren zu suchen.

7.6 Verteilung der Verzögerungswerte an die Instanzen

Wie im Abschnitt 7.2 beschrieben, ist den verarbeitenden Komponenten die Verzögerung ihrer Eingangssignale zu übergeben. Diese Komponenten können wiederum aus verarbeitenden Komponenten instanziert werden, z.B. der Addierer innerhalb des Verstärkers. Dann berechnet die instanzierende Komponente die Verzögerungen anhand ihrer Eingangsverzögerung und übergibt sie. Wenn die Instanzierung jedoch aus einer Hierarchie-Komponente erfolgt, muss diese die bei der Optimierung ermittelten Werte übergeben. Es besteht damit die Aufgabe, die bei der Optimierung erhaltenen Ergebnisse an die entsprechenden Komponenten zu verteilen.

7.6.1 Ablage der Informationen für Simulation und Synthese

Es bestehen die folgenden Möglichkeiten:

- Verwendung einer VHDL-Tabelle: Die Zuordnung zu den Instanzen erfolgt durch die Übergabe eines Indizes, der einen Verweis auf Einträge in dieser Tabelle darstellt.
- Auslesen der Werte aus einer Textdatei mit Hilfe von VHDL-Funktionen: Dazu muss der Dateiname übergeben werden. Jedoch bereitet die Stringübergabe bei der Synthese Probleme. Diese Funktionen wären nur sehr aufwändig zu debuggen, da sie während der Elaboration abgearbeitet werden. Von einer Komponente können mehrere Instanzen angelegt werden. Dort muss eine Zuordnung möglich sein.

Da es gegen die erste Idee keine grundsätzlichen Einwände gibt, wurde sie umgesetzt. Eine Tabelle (Verzögerungswerttabelle) in einem speziellen Package wird für das ganze Design benutzt. Dort werden alle Verzögerungen abgelegt. Der Zugriff auf die Daten zu einer bestimmten Instanz erfolgt über einen Index, der jeder Instanz einer Hierarchie-Komponente übergeben wird und eine laufende Nummer für jede angelegte Instanz einer verarbeitenden Komponente.

7.6.1.1 Aufbau der Verzögerungswerttabelle und Füllen mit Werten

Die Tabelle ist eine Konstante vom Typ:

```
type table is array (0 to 99, 0 to 9) of integer;
constant delay_table : table100 := wrap_read_delay;
```

Die erste Dimension entspricht dem Index, d.h. sie ist der Instanz zugeordnet. Die zweite Dimension entspricht den Verzögerungen, die in der Architektur benötigt werden. Da dieses Array alle Verzögerungen aufnimmt, wird die zweite Dimension von der Architektur mit den meisten Verzögerungen bestimmt. Die erforderliche Größe kann erst während der Elaboration ermittelt werden. Um die Verwendung einer dynamischen Variable zu vermeiden, wird eine feste Größe verwendet. Die Tabelle wird mittels einer externen C-Funktion über das Foreign-Language-Interface (FLI) von VHDL während der Elaboration mit Werten gefüllt. Dazu dient die Prozedur `'read_delays'` im Package `'delays_table'` (siehe Anhang B.4.1 auf Seite 125). Die Prozedur wurde in C realisiert (siehe Anhang B.4.2 auf Seite 127), da die während der Elaboration in VHDL zur Verfügung stehende Funktionalität dazu nicht ausreicht. Es ist weiterhin nicht möglich, einer VHDL-Funktion den Zeiger auf ein Array zu übergeben. Darum musste der Umweg von `'wrap_read_delay'` auf `'read_delay'` über eine Prozedur genommen werden.

Im Anhang B.4.3 auf Seite 130 wurde das Optimierungsergebnis für einen PID-Regler abgebildet. Das Ergebnis besteht aus drei Spalten. Die erste Spalte ist der Index, die zweite Spalte

der Verweis auf die verarbeitenden Komponente und die dritte Spalte enthält die Verzögerung. Damit kann über die ersten beiden Spalten die Zelle in der Verzögerungswerttabelle adressiert werden.

7.6.1.2 Zuordnung der Instanzen zu den Zeilen der Verzögerungswerttabelle

Das Ziel ist die eindeutige Zuordnung zwischen den Instanzen verarbeitender Komponenten und den Einträgen in der Verzögerungswerttabelle. Dabei ist eine relative Adressierung zu berücksichtigen, da Teile des Designs u.U. auch mehrfach innerhalb des Designs verwendet werden können.

Ausgangspunkt ist die Liste des Vektors **u**, denn sie enthält alle bitseriellen Signale, die Eingangssignale einer Komponente sind. Daraus wird ein Hash mit Verzögerungswertkonstanten abgeleitet. Dabei entspricht der Schlüssel dem Konstantennamen und der Wert der Position des Signals in der Liste. Da **u** bereits aus der Liste der sortierten Signale resultiert (siehe Abschnitt 7.4), ist auch die Reihenfolge der Verzögerungswertkonstanten immer einheitlich. Die Anzahl der Hashelemente wird in einer Datenbank abgespeichert, zugeordnet zu ihrer Entität und Architektur. In VHDL kann eine Entität oder Komponente mehrere verschiedene Architekturen haben.

Wenn innerhalb einer Architektur die Instanz einer Hierarchie-Komponente auftritt, ist ihr ein Index zuzuweisen. In alphanumerischer Reihenfolge erfolgt die Wertzuweisung zum Parameter $\mathsf{index} = \mathsf{index} + \mathsf{i}$, wobei i nicht einfach mit jeder Instanz inkrementiert werden kann. Denn dann kommt es zur doppelten Vergabe eines Wertes innerhalb des Designs. Der Indexoffset, um den i erhöht wird, richtet sich danach, wieviele verarbeitende Instanzen in der vorherigen Instanz angelegt wurden und kann aus der Datenbank ausgelesen werden. Innerhalb einer Architektur wird der Indexzähler mit Null initialisiert. Wenn eine Instanz selber verarbeitende Instanzen instanziert, dann benutzt sie $\mathsf{i} = 0$ für die dafür benötigten Verzögerungen, d.h. der Indexzähler ist um Eins zu inkrementieren.

Bild 7.5 stellt hierzu ein Beispiel dar. Instanzen mit großem Buchstaben leiten sich von Hierarchie-Komponenten ab, solche mit kleinem Buchstaben von verarbeitenden Komponenten. Der Wert in der rechten unteren Ecke eines Quadrats stellt den Eintrag in der Verzögerungswerttabelle dar (siehe Tabelle 7.2), den eine Instanz für sich selber benötigt. Instanz A wird $\mathsf{index} = 0$ übergeben. Da sie selber einen Wert für Instanz a aus der Tabelle mit den Verzögerungen auslesen muss, benötigt sie diesen Index für sich selber. An B wird daher $\mathsf{index} + 1$ übergeben. Dieses legt selber zwei Hierarchie-Instanzen (C und D) an, welche die Indizes 1 und 2 für sich belegen. B legt aber keine verarbeitenden Instanzen an. Um eine Dopplung von Einträgen in der Verzögerungswerttabelle zu vermeiden, kann an E nicht $\mathsf{index} + 2 = 2$ übergeben werden. Sondern hier ist die Anzahl der von B angelegten Hierarchie-Instanzen zu berücksichtigen. Daher erfolgt die Übergabe von $\mathsf{index} + 3$.

Das Entwurfswerkzeug (HDL-Designer) verlangt, dass alle verwendeten Konstanten deklariert sind. Daher ist vom Entwickler beim Entwurf als Wert für den Index immer eine Null einzutragen. Dieser Wert wird bei der Code-Manipulation ersetzt. Das Gleiche gilt auch für den zweiten Wert der Tabelle (`'delay_pos'`).

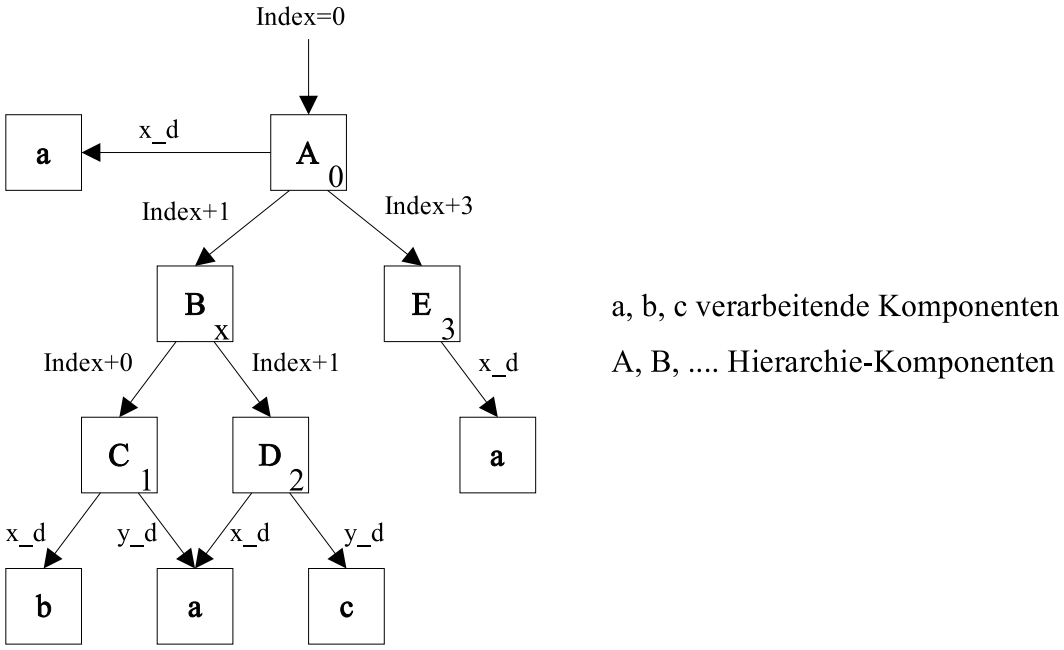

Abbildung 7.5: Beispiel für ein Nummerierungschema

Instanz	Verzögerung	
A	x_d	
C	x_d	y_d
D	x_d	y_d
E	x_d	

Tabelle 7.2: Verzögerungswerttabelle

7.6.2 Sonderfälle

Bei Zählern, Parallel-Seriell-Wandlern und seriellen Quellen bezieht sich das Delay auf das Ausgangssignal, da es kein bitserielles Eingangssignal gibt. Das Schieberegister benötigt keine Angaben zu Verzögerungen, da es alle Bits gleich behandelt. Dafür wird die Memoryzelle zum Einhalten der Schleifenbedingung (siehe 3.1 auf Seite 37) genutzt. Deshalb muss ihre Länge angepasst werden. Die Länge von Memoryzellen ist Teil des Optimierungsergebnisses und muss nur in die Verzögerungswerttabelle mit eingetragen werden.

7.7 Codemanipulation und -generierung

Mit den beim Parsen gewonnenen Informationen und den generierten Matrizen kann dann automatisch der VHDL-Quellcode manipuliert und ergänzt werden. Codemanipulation heißt, dass die Datei noch einmal geöffnet und nach dem Auftreten bestimmter Schlüsselwörter durchsucht wird. Die betreffenden Teile werden dann angepasst und durch Hinzufügen eines Kommentars gekennzeichnet. Das vereinfacht die Fehlersuche. Weiterhin wird noch die Datei als manipuliert markiert, um eine Mehrfachverarbeitung zu vermeiden. Zum Vergleich bleibt die originale Version des Quelltextes unter anderem Namen erhalten.

7.7.1 Behandlung der verschiedenen Komponententypen

7.7.1.1 Testbench

Der Quelltext der Testbench bleibt unverändert. Er muss allerdings einen Funktionsaufruf zur Generierung des m-Files enthalten. Dieser ist bei allen Designs gleich und kann daher vom Entwickler selber eingefügt werden. Lediglich der Verweis auf die Toplevel-Komponente ist anzupassen. Der Quelltext befindet sich im Abschnitt B.2.3. Die Testbench ruft die Funktion zur Generierung des m-Files des Toplevels auf. Daher muss das Package, das diese beinhaltet, mit deklariert werden.

7.7.1.2 Toplevel und Hierarchie-Komponenten

Diese beiden Komponententypen werden gleich behandelt, da sie sich lediglich im Vorhandensein bestimmter Eingangsparameter unterscheiden. In beiden Typen kommen jedoch bitserielle Signale vor und müssen daher behandelt werden.

Bei der Spezifikation im Blockdiagrammeditor ist den generischen Parametern für die Verzögerung eine Konstante zuzuweisen. Diese Konstante ist dann zu deklarieren und mit einem beliebigen Wert zu initialisieren. Aus der Liste mit den sortierten Signalen wird über die Namenskonvention (siehe Abschnitt 7.1.2) ein Hash von Delay-Konstanten abgeleitet. Dabei enthalten die Schlüssel die Konstantennamen und die Werte der Hasheinträge die laufende Nummer innerhalb einer Architektur. Beim Auftreten einer Konstanten im Quelltext wird dann überprüft, ob sie im Delay-Hash vorkommt. In diesem Falle erfolgt eine Initialisierung derart:

```
constant x_d : integer := delay_table (index, Wert_aus_Hash);
```

Desweiteren sind noch die Index-Parameter der Hierarchie-Komponenten anzupassen. Dazu wird bei jeder auftretenden Konstantendeklaration überprüft, ob diese Konstante den Index

für eine Instanz darstellt. Wenn dies der Fall ist, wird ihr der Wert des Indexzählers zugewiesen, siehe Abschnitt 7.6.1.2. Anschließend ist der Indexoffset dieser Komponente aus der Indexdatenbank auszulesen und zum Indexzähler zu addieren.

7.7.2 Erzeugung des m-Files

Die Vorschrift zur Bildung der Matrizen wird in einer Matlab-Funktion für jede Hierarchie-Komponente abgelegt. Erst während der Elaboration des Systems stehen alle Informationen zur Berechnung der Matrizen zur Verfügung. Das sind z.B. Werte von Konstanten, die über generische Parameter an die Komponenten übergeben werden. Um deren Auswertung beim VHDL-Simulator zu belassen, kann das m-File, das die Funktion aufnimmt, nicht direkt beim Parsen generiert werden. Dies geschieht aus einer zusätzlich erstellten VHDL-Funktion heraus. Diese Funktion befindet sich in einem Package, das nach dem Parsen an den Quelltext einer jeden Hierarchie-Komponente angehängt wird. Dieser Funktion werden keine Werte übergeben. Die Wertübergabe erfolgt vollkommen über die Matlab-Schiene. Lediglich die Konstantendeklarationen, außer denen für die Verzögerungen, werden kopiert.

Innerhalb der VHDL-Funktion existiert eine dynamische Variable `'rtext'`, in die während der Elaboration die m-Funktion zeilenweise eingetragen wird. Am Ende der Funktion wird dann der Inhalt dieser Variablen in eine Datei geschrieben.

Generierung des Packages zur Erzeugung des m-Files:

- Namenskonvention für Package `Entityname'_'Architecturename'_pack'`
- Namenskonvention für Funktion `Entityname'_'Architecturename'_delay'`
- Übernahme aller Konstantendeklarationen aus der Entität in den Deklarationsteil des Funktionsbodys, außer denen, die auf die Verzögerungswerttabelle zugreifen
- für jede diese VHDL-Konstanten wird eine Matlabvariable derart definiert:
  `"name="&integer'image(name)&";"`
- Definition der Indizes für den Zugriff auf die `delay_table` derart, dass folgende VHDL-Deklaration
  `constant constname : integer := delay_table (index,k);`.
  ersetzt wird durch
  `add_line (rtext, "constname = [index,k];");`

7.8 Aufbau der Matlab-Funktionen und Skripte

7.8.1 Basisfunktionen

Die Basisfunktionen stellen allgemeine Funktionalitäten zur Verfügung. Dazu zählen das Aufstellen der Matrizen für eine verarbeitende Komponente, das Zusammenführen all dieser Matrizen innerhalb einer Hierarchie-Komponente und die Reduktion auf ein System laut Gleichung 6.9 bis 6.11 auf Seite 65.

7.8.1.1 Aufstellen der konstanten Matrizen für verarbeitende Komponenten

Die Funktion `'constants'` (siehe Anhang B.2.3.1 auf Seite 114) liefert als Rückgabewert die Matrizen und Vektoren für verarbeitende Komponenten, wie sie im Abschnitt 6.4 auf Seite 68 aufgestellt wurden. Zu den Übergabewerten gehören:

- Name von Entität und Architektur
- Teiler: Dieser geht mit in die Verzögerung ein.
- Delay-Position: Sie ermöglicht die Zuweisung eines Wertes in der Verzögerungswerttabelle. Alle Delay-Positionen zusammen stellen den Vektor **y** dar.
- c0: Es legt fest, ob die Komponente einen bitseriellen Ausgang hat und beeinflusst damit die Ausprägung von **y**.
- offset: Dieser Wert wird benötigt, um bei der Memoryzelle die Korrektur entsprechend Gleichung 6.33 vornehmen zu können.

Die Funktion `'constants'` wird nicht direkt aus einer Hierarchie-Komponente aufgerufen, sondern über eine Funktion, die zu einer verarbeitenden Komponente gehört.

7.8.1.2 Verarbeitende Komponenten

Jede verarbeitende Komponente wird durch eine Funktion repräsentiert. Sie hat den Namen: `Entitätsname'_'Architekturname` und hat als Parameter die generischen Parameter der zugehörigen Komponente. Diese Funktion müsste nun alle Matrizen dieser Komponente enthalten. Um jedoch einen gemeinsamen Platz zum Abspeichern aller Matrizen aller verarbeitender Komponenten zu erhalten, bildet die Funktion ihren eigenen Aufruf auf die Funktion `'constants'` ab. Im Anhang B.2.3.6 auf Seite 122 wurde als Beispiel die Funktion `'adder_adder'` angefügt.

7.8.1.3 Zusammensetzen und Flachklopfen der Matrizen

Das bedeutet das Zusammensetzen von $\mathbf{C_u}$, $\mathbf{C_v}$, $\mathbf{v}$ und $\mathbf{v}_k$ gemäß Gleichung 6.8, sowie $\mathbf{P}$ und $\mathbf{P}_0$ gemäß

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u_n} \end{bmatrix} = \begin{bmatrix} \mathbf{P}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{P}_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{P_n} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y_n} \end{bmatrix} + \begin{bmatrix} \mathbf{P}_{0_1} & 0 & \cdots & 0 \\ 0 & \mathbf{P}_{0_2} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{P}_{0_n} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u}_{0_1} \\ \mathbf{u}_{0_2} \\ \vdots \\ \mathbf{u}_{0_n} \end{bmatrix} \tag{7.1}$$

wobei Gleichung 7.1 aus Gleichung 6.21 auf Seite 67 resultiert und die Abbildung aller unterlagerten Ausgänge auf alle unterlagerten Eingänge zusammenfasst. Weiterhin wird $\mathbf{C}_0$ gemäß Gleichung 6.19 auf Seite 66 (entspricht c0 im Quelltext) und $\tilde{\mathbf{C}}_0$ aufgestellt. Dazu dient die Funktion `'compose_matrix'` (siehe Anhang B.2.3.2 auf Seite 118).

Die Funktion `'flat_matrix'` (siehe Anhang B.2.3.3 auf Seite 119) implementiert die Gleichungen 6.13 bis 6.15, 6.25 und 6.26 im Abschnitt 6.3 auf Seite 65.

7.8.2 Hierarchie-Komponenten

Die Funktion zur Berechnung der Matrizen einer Komponente gliedert sich in fünf Teile. Das sind:

1. Definition der Variablen, die die Positionen in der Verzögerungswerttabelle speichern, einschließlich Wertzuweisung
2. Berechnung der Matrizen der eingebetteten Komponenten. Dazu wird zu jeder Instanz ein Funktionsaufruf folgenden Types generiert:
   - Funktionsname: `Entitätsname'_'Architekturname`
   - Die übergebenen Parameter entsprechen den generischen Parametern der zugehörigen Komponente.

   Die aufgerufene Funktion kann entweder eine Hierarchie-Komponente oder eine verarbeitende Komponente repräsentieren.
3. Aufstellen der Matrizen $\tilde{\mathbf{P}}$ und $\tilde{\mathbf{P}}_0$

4. Zusammensetzen der Matrizen (siehe Abschnitt 7.8.1.3 auf der vorherigen Seite), durch Aufruf der Funktion `'compose_matrix'`.

5. Flachklopfen der Matrizen (siehe Abschnitt 7.8.1.3 auf der vorherigen Seite) mit der Funktion `'flat_matrix'`.

Weiterhin existieren noch einige Hilfsvektoren, wie z.B. `'upos'` und `'vpos'`. Diese enthalten die Positionen der Verzögerungswerttabelle, wo die zugehörigen Werte u_i und v_i abzuspeichern sind. Beispielcode für eine Hierarchie-Komponente ist im Anhang B.2.2.1 auf Seite 110 zu finden.

7.8.3 Toplevel

Das Toplevel wird durch ein Skript repräsentiert, das genauso aufgebaut ist, wie die Funktionen der Hierarchie-Komponenten. Es definiert zusätzlich noch die Variable `'index'`. Ein Beispiel ist im Anhang B.2.2.3 auf Seite 113 zu finden.

7.8.4 Testbench

Die Testbench ist für alle Designs gleich, nur der Aufruf des Toplevel-Skripts ist anzupassen. Sie ist folgendermaßen aufgebaut:

1. Erweiterung des Suchpfades um das Verzeichnis mit den allgemeinen Funktionen

2. Aufruf des Toplevels

3. Start der Optimierung

4. Aufbau der Ergebnismatrix

5. Runden auf ganze Zahlen, denn durch der numerischen Verarbeitung ist das Ergebnis nicht immer exakt eine ganze Zahl.

6. Ausgabe der Ergebnismatrix

Ein Beispiel ist im Anhang B.2.3.5 auf Seite 121 zu finden.

7.9 Synthese

Viele bei der Optimierung genutzte VHDL-Ausdrücke lassen sich nicht synthetisieren. Das sind z.B. Dateioperationen und das FLI. Um manuelle Eingriffe bei der Synthese zu vermeiden, werden die bei der Vorbereitung der Optimierung angehängten Pakete wieder entfernt. Das FLI

wird benötigt, um die Ergebnisse der Optimierung während der Elaboration einzulesen. Dies vermeidet eine zusätzlichen Start des Simulators. Die vorliegenden Optimierungsergebnissen werden daher in Vorbereitung der Synthese dierkt in den VHDL-Quellcode übertragen und so die Nutzung des FLI vermieden.

Dies alles geschieht vor dem Hintergrund, dass die Optimierung bereits während der Simulation erfolgte und die entfernten oder ersetzten Codeteile nicht mehr erforderlich sind.

Kapitel 8

Schlussbemerkungen

8.1 Abgrenzung der Arbeit

Mit der geschaffenen Bibliothek bitserieller Elemente und Komponenten können einfach regelungs- und steuerungstechnische Systeme auf Gatterebene realisiert werden. Insbesondere die Synchronisation auf Basis der automatischen Bestimmung der Verzögerungszeiten ist eine große Erleichterung für den Entwickler. Dadurch sind die Ergebnisse universell verwendbar und heben sich so von speziellen Lösungen im Bereich der bitseriellen Arithmetik ab (siehe Abschnitt: 1.5.5 auf Seite 12). Auf der Basis der entwickelten Blockbibliothek und unter Verwendung der Spezifikations- und Synthese-Tools sind komplexe Systeme der Regelungs-, Steuerungstechnik und Signalverarbeitung sehr einfach direkt in Hardware implementierbar. Durch die parallele Implementierung der Signalflüsse lassen sich einerseits sehr hohe Datendurchsätze und sehr schnelle Systeme realisieren. Andererseits werden durch die bitserielle Implementierung nur sehr wenige Ressourcen benötigt, so dass auch größere Systeme äußerst ökonomisch darstellbar sind. Die Verwendung von Standard-Tools und der standardisierten Hardwarebeschreibungssprache VHDL erlaubt ein sehr komfortables Arbeiten von der Spezifikation über die Simulation bis zur Synthese.

Die im Abschnitt 1.4 erwähnten Verfahren vereinfachen alle die Spezifikation von Algorithmen, benötigen aber eine Hardwarebasis zur Implementierung. Dies sind gegenwärtig konventionelle bitparallele Verfahren. Hier besteht die Möglichkeit, bitserielle Algorithmen als Grundlage der Realisierung der Operatoren zu benutzen, um damit deren Implementierung zu vereinfachen.

8.2 Rekonfigurierbares Computing

Die vorliegende Arbeit zeigt, dass auf der Basis bitserieller Algorithmen komplexe Regelungs-, Steuerungs- und Signalverarbeitungssysteme direkt in Hardware implementiert werden können.

Allerdings bezog sich das bislang nur auf eine bestimmte Konfiguration, die auf einem FPGA oder PLD implementiert wurde. Rekonfigurierbares Computing (reconfigurable computing) ist zwischen PLDs und Mikroprozessoren angeordnet. Es handelt sich im Wesentlichen um PLDs, die jedoch im laufenden Betrieb umprogrammiert werden können. Allerdings erfolgt die Umprogrammierung nicht so häufig wie bei Prozessoren (siehe Tabelle 8.1). Damit besteht die Möglichkeit, PLDs an bestimmte Betriebsregime besser anzupassen, ohne dass ständig die gesamte Logik arbeitet und so Gatter und Energie verbraucht werden. Wie ein Mikroprozessor nimmt eine rekonfigurierbare Hardware eine Konfiguration auf, interpretiert diese und führt sie aus. Jedoch werden mehr Daten als bei einem Prozessorbefehl verarbeitet. Hierbei wird der "Single Instruction Multiple Data"-Ansatz intensiviert. Eine Konfiguration ist somit wie ein selbsterstellter Makrobefehl zu verstehen.

	PLD, konfigurierbar	**Rekonfigurierbare Hardware**	**Mikroprozessor**
Bindungszeitpunkt	Laden der Konfiguration	Laden der Konfiguration	Zyklus
Bindungsdauer	Ende der Anwendung	Benutzerdefiniert	Zyklus
Programmierzeit	ms bis s	Zyklus	Zyklus
Anzahl der Befehle	1	> 1	> 1

Tabelle 8.1: Charakteristische Zeiten für programmierbare Einheiten [Deh99]

Verschiedene Technologieansätze kommen für die Realisierung des rekonfigurierbaren Computing in Frage. Das sind partiell rekonfigurierbare FPGAs, wie sie schon von Atmel und Xilinx angeboten werden. Diese laden beim Anlegen der Versorgungsspannung eine Defaultkonfiguration und ersetzen sie nach Bedarf im laufenden Betrieb ganz oder teilweise. Eine andere Möglichkeit wären Multikontext-PLDs.

Die partielle Rekonfigurierbarkeit ist für einige Applikationen noch keine Lösung, falls die entsprechenden Rekonfigurationszeiten zu lang sind. Genau an diesem Punkt setzen die Multicontext-PLDs an: Ein PLD dieser Klasse besitzt zwar nur eine Arbeitsebene, aber mehrere Speicherebenen. Die Rekonfigurierung besteht in diesem Fall aus dem Umschalten der Speicherebene. Einem ausführenden Layer, der dem klassischen PLD entspricht, stehen mehrere Memory-Layer gegenüber, die für die einzelnen Segmente entsprechende Programme bereithalten. Ein weiterer Bausteinabschnitt (Extra-PLD) ist für das Laden der einzelnen Bereiche, also für die Programmauswahl zuständig. Dieser Bereich muss nicht zwangsläufig gesondert im Baustein vorhanden sein (siehe [Sie02b]). Hinsichtlich der bitseriellen Verarbeitung bietet rekonfigurierbares Computing den Vorteil, dass je nach Betriebsregime eine andere Konfiguration und damit andere Algorithmen geladen werden können.

Anhang A

Formelnotation

A.1 Formelnotaion

Skalare in normaler Schriftstärke $y = x^2$

$$y = x^2$$

Vektoren kleine Buchstaben in Fettdruck: $\mathbf{u}_0 = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}$

Matrizen große Buchstaben in Fettdruck: $\mathbf{P} = \begin{bmatrix} p_{11} & \cdots & p_{13} \\ \vdots & \ddots & \vdots \\ p_{31} & \cdots & p_{33} \end{bmatrix}$ mit p_{jk} als Skalar oder

$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{11} & \cdots & \mathbf{p}_{13} \\ \vdots & \ddots & \vdots \\ \mathbf{p}_{31} & \cdots & \mathbf{P}_{33} \end{bmatrix}$, wenn die Elemente Matrizen oder Vektoren sind.

Anhang B

Quelltexte und Skripte

B.1 Beispielpackage für Kennlinienapproximation

```
--------------------------------------------------------------------------
-- Title      : Sampling points for characteristic curve
-- Project    : Hardreg
--------------------------------------------------------------------------
-- File       : lut_sp.vhdl
-- Author     : Thomas Reinemann  <thomas.reinemann@mb.uni-magdeburg.de>
-- Company    : Otto-von-Guericke-Universität Magdeburg
-- Created    : 2000-08-02
-- Last update: 2003-02-24
-- Platform   :
--------------------------------------------------------------------------
-- Description:
--------------------------------------------------------------------------
-- Copyright (c) 2002 Otto-von-Guericke-Universität Magdeburg
--------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2000-08-02  1.0      threinem Created
-- 2002-02-19  1.1      threinem        adopted to 2**n sampling points
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library std;
use std.textio.all;
```

```
library SerElemente;
use serelemente.basics.all;

package KLMemTable is

-- 64 Stützstellen 0..1024 als x-Werte

constant SP_LENGTH_0 : integer := 65;   -- actually used length
constant SP0_0: integer := 65535/2;
constant SP0_1: integer := 65535/2;
constant SP0_2: integer := 32768/2;
constant SP0_3: integer := 21845/2;
constant SP0_4: integer := 16384/2;
constant SP0_5: integer := 13107/2;
constant SP0_6: integer := 10923/2;
constant SP0_7: integer := 9362/2;
constant SP0_8: integer := 8192/2;
constant SP0_9: integer := 7282/2;
constant SP0_10: integer := 6554/2;
constant SP0_11: integer := 5957/2;
constant SP0_12: integer := 5461/2;
constant SP0_13: integer := 5041/2;
constant SP0_14: integer := 4681/2;
constant SP0_15: integer := 4369/2;
constant SP0_16: integer := 4096/2;
constant SP0_17: integer := 3855/2;
constant SP0_18: integer := 3641/2;
constant SP0_19: integer := 3449/2;
constant sp0_20: integer := 3277/2;
constant sp0_21: integer := 3120/2;
constant sp0_22: integer := 2979/2;
constant sp0_23: integer := 2849/2;
constant sp0_24: integer := 2731/2;
constant sp0_25: integer := 2621/2;
constant sp0_26: integer := 2521/2;
constant sp0_27: integer := 2427/2;
constant sp0_28: integer := 2341/2;
constant sp0_29: integer := 2260/2;
constant SP0_30: integer := 2185/2;
```

```
constant SP0_31: integer := 2114/2;
constant SP0_32: integer := 2048/2;
constant SP0_33: integer := 1986/2;
constant SP0_34: integer := 1928/2;
constant SP0_35: integer := 1872/2;
constant SP0_36: integer := 1820/2;
constant SP0_37: integer := 1771/2;
constant SP0_38: integer := 1725/2;
constant SP0_39: integer := 1680/2;
constant SP0_40: integer := 1638/2;
constant SP0_41: integer := 1598/2;
constant SP0_42: integer := 1560/2;
constant SP0_43: integer := 1524/2;
constant SP0_44: integer := 1489/2;
constant SP0_45: integer := 1456/2;
constant SP0_46: integer := 1425/2;
constant SP0_47: integer := 1394/2;
constant SP0_48: integer := 1365/2;
constant SP0_49: integer := 1337/2;
constant SP0_50: integer := 1311/2;
constant SP0_51: integer := 1285/2;
constant SP0_52: integer := 1260/2;
constant SP0_53: integer := 1237/2;
constant SP0_54: integer := 1214/2;
constant SP0_55: integer := 1191/2;
constant SP0_56: integer := 1170/2;
constant SP0_57: integer := 1149/2;
constant SP0_58: integer := 1130/2;
constant SP0_59: integer := 1111/2;
constant SP0_60: integer := 1092/2;
constant SP0_61: integer := 1074/2;
constant SP0_62: integer := 1057/2;
constant SP0_63: integer := 1040/2;
constant SP0_64: integer := 1023/2;

constant SP_LENGTH_1 : integer := 17;   -- actually used length
constant SP1_0: integer := 450;
constant SP1_1: integer := 450;
constant SP1_2: integer := 450;
```

```
constant SP1_3: integer := 448;
constant SP1_4: integer := 313;
constant SP1_5: integer := 245;
constant SP1_6: integer := 200;
constant SP1_7: integer := 170;
constant SP1_8: integer := 148;
constant SP1_9: integer := 131;
constant SP1_10: integer :=118;
constant SP1_11: integer :=107;
constant SP1_12: integer := 98;
constant SP1_13: integer := 90;
constant SP1_14: integer := 84;
constant SP1_15: integer := 78;
constant SP1_16: integer := 73;

type memory is Array (natural range <>, natural range <>) of integer;

constant rom : memory:=
    ((SP0_0,SP0_1,SP0_2,SP0_3,SP0_4,SP0_5,SP0_6,SP0_7,SP0_8,
      SP0_9,SP0_10,SP0_11,SP0_12,SP0_13,SP0_14,SP0_15,SP0_16,
      SP0_17,SP0_18,SP0_19,SP0_20,SP0_21,SP0_22,SP0_23,SP0_24,
      SP0_25,SP0_26,SP0_27,SP0_28,SP0_29,SP0_30,SP0_31,SP0_32,
      SP0_33,SP0_34,SP0_35,SP0_36,SP0_37,SP0_38,SP0_39,SP0_40,
      SP0_41,SP0_42,SP0_43,SP0_44,SP0_45,SP0_46,SP0_47,SP0_48,
      SP0_49,SP0_50,SP0_51,SP0_52,SP0_53,SP0_54,SP0_55,SP0_56,
      SP0_57,SP0_58,SP0_59,SP0_60,SP0_61,SP0_62,SP0_63,SP0_64),
     (SP1_0,SP1_1,SP1_2,SP1_3,SP1_4,SP1_5,SP1_6,SP1_7,SP1_8,
      SP1_9,SP1_10,SP1_11,SP1_12,SP1_13,SP1_14,SP1_15,SP1_16,
      SP1_17,SP1_18,SP1_19,SP1_20,SP1_21,SP1_22,SP1_23,SP1_24,
      SP1_25,SP1_26,SP1_27,SP1_28,SP1_29,SP1_30,SP1_31,SP1_32,
      SP1_33,SP1_34,SP1_35,SP1_36,SP1_37,SP1_38,SP1_39,SP1_40,
      SP1_41,SP1_42,SP1_43,SP1_44,SP1_45,SP1_46,SP1_47,SP1_48,
      SP1_49,SP1_50,SP1_51,SP1_52,SP1_53,SP1_54,SP1_55,SP1_56,
      SP1_57,SP1_58,SP1_59,SP1_60,SP1_61,SP1_62,SP1_63,SP1_64))
constant rom_length : integer_array := (SP_LENGTH_0,SP_LENGTH_1);

end KLMemTable;
```

B.2 Berechnung der Verzögerungen

B.2.1 Integrator

Code zum Beispiel im Abschnitt 6.6.1 auf Seite 75

```
% Optimierung Integrator
clear;
Cv=[1/2 1/2  0
    1   -1   0
    0   0    0
    0   0    -1];
Cu=[1/2 1/2 0 0
    1   -1  0 0
    0   0   1 0
    0   0   0 1];
P=[ 0 0 0 0
    0 0 0 1
    1 0 0 0
    1 0 0 0];
P0=[1;0;0;0];
u0=0;
dzI=0;
vk=[1;0;1+dzI;0];
c0=[0 0 1 0];

% b aufstellen
[vky,vkx]=size(vk);
b= [-eye(vky) -Cu*P0]*[vk;u0];
% A aufstellen

[l,m]=size(P);
[x,n]=size(Cv);
CuP=Cu*P;
CuPeye=eye(size(CuP));
Aeq=[Cv CuP-CuPeye];

[y,lang]=size(Aeq);
beq=b;
```

```
f=ones(1,lang);
lb=zeros(lang,1);
[x,fval,exitflag,output,lambda]=linprog(f,[],[],Aeq,beq,lb);
v=x(1:n)
y=x(n+1:m+n)

u=P*y+P0*u0

[CCu, CCv, vvk] = flat_matrix (Cu, Cv, P, P0, vk);

disp ('Validierung von y');
c0*CCv*v+c0*CCu*u0+c0*vvk
```

B.2.2 PID-Regler

Skripte zum Anwendungsbeispiel PID-Regler im Abschnitt 6.6.2 auf Seite 78

B.2.2.1 Funktion für den PID-Regler

```
function [c]= pid_reduziert (divisor,index,kd,ki,kp,word_length);
%===============Konstanten
index1 = index + 1;
signd= 1;
p_d = [index,0];
d_d = [index,1];
pd_d = [index,2];
i_d = [index,3];
controldeviation_d = [index,4];
raw_d_d = [index,5];
%======== Instance Code =======
%==========add_pd : adder
component(1) =adder_adder (p_d,d_d, 1, 1,word_length);
component(1).c0 =component(1).c0*[0];
%==========add_pid : adder
component(2) =adder_adder (pd_d,i_d, 1,signd,word_length);
component(2).c0 =component(2).c0*[1];
%==========derivative_0 : derivative
component(3) =derivative_derivative (controldeviation_d,word_length);
```

```
component(3).c0 =component(3).c0*[0];
%==========gain_kd : gaindiv
component(4) =gaindiv_gaindiv (raw_d_d,kd, 1,divisor,word_length);
component(4).c0 =component(4).c0*[0];
%==========gain_kp : gaindiv
component(5) =gaindiv_gaindiv
                     (controldeviation_d,kp, 1,divisor,word_length);
component(5).c0 =component(5).c0*[0];
%==========i0 : i_part
component(6) =i_part_mit_fehler_korr (divisor,index1,ki,word_length);
component(6).c0 =component(6).c0*[0];
 P=[ 0 0 0 0 1 0
     0 0 0 1 0 0
     1 0 0 0 0 0
     0 0 0 0 0 1
     0 0 0 0 0 0
     0 0 1 0 0 0
     0 0 0 0 0 0
     0 0 0 0 0 0];
 P0=[ 0
      0
      0
      0
      1
      0
      1
      1];
[c0,C0,Cu,Cv, Pl, P0l, p, upos, v, vk, vpos]=compose_matrix(component);
c.name='pid_reduziert';
c.param=0;
c.c0=c0;
c.P=P;
c.Pl=Pl;
c.P0=P0;
c.P0l=P0l;
c.upos=upos;
c.vpos=vpos;
c.Cv = Cv;
c.Cu = Cu;
```

```
c.C0=C0;
c.vk = vk;
c.v=v;
c.p.v=p.v;
c.p.y=p.y;
c=flat_matrix (c);
```

B.2.2.2 Funktion zur Berechnung des I-Anteils

```
function [c]= i_part_mit_fehler_korr (divisor,index,ki,word_length);
%==============Konstanten
raw_i_d = [index,0];
i_d = [index,1];
controldeviation_d = [index,2];
i_feed_back_d = [index,3];
%======== Instance Code =======
%==========gain_ki1 : gaindiv
component(1) =gaindiv_gaindiv (raw_i_d,ki, 1,divisor,word_length);
component(1).c0 =component(1).c0*[1];
%==========i1 : memorycell
component(2) =memorycell_memorycell (i_d,word_length);
component(2).c0 =component(2).c0*[0];
%==========integrate1 : adder
component(3) =adder_adder
                    (controldeviation_d,i_feed_back_d, 1, 1,word_length);
component(3).c0 =component(3).c0*[0];
%==========limit_2_half1 : lt2hf
component(4) =lt2hf_lt2hf (raw_i_d, 1,word_length);
component(4).c0 =component(4).c0*[0];
 P=[ 0 0 1 0
     0 0 0 1
     0 0 0 0
     0 1 0 0
     0 0 1 0];
 P0=[ 0
      0
      1
      0
      0];
```

```
[c0,C0,Cu,Cv, Pl, P0l, p, upos, v, vk, vpos]=compose_matrix(component);
c.name='i_part_mit_fehler_korr';
c.param=0;
c.c0=c0;
c.P=P;
c.Pl=Pl;
c.P0=P0;
c.P0l=P0l;
c.upos=upos;
c.vpos=vpos;
c.Cv = Cv;
c.Cu = Cu;
c.C0=C0;
c.vk = vk;
c.v=v;
c.p.v=p.v;
c.p.y=p.y;
c=flat_matrix (c);
```

B.2.2.3 Skript für das Toplevel

```
%===============Konstanten
word_length= 16;
tick_cycle= 2;
delay= 0;
kp= 40;
ki= 10;
kd= 30;
divisor= 5;
ki_divisor= 4;
s2p_teiler= 0;
index= 0;
index1 = index + 1;
u0= 0;
pid_d = [index,0];
%========= Instance Code =======
%==========ci_s2p : s2p
component(1) =s2p_s2p ( 0,pid_d,word_length,s2p_teiler,word_length);
component(1).c0 =component(1).c0*[0];
```

```
%==========i2 : pid
component(2) =pid_reduziert (divisor,index1,kd,ki,kp,word_length);
component(2).c0 =component(2).c0*[0];
 P=[ 0 1
     0 0];
 P0=[ 0
      1];
[c0,C0,Cu,Cv, Pl, P0l, p, upos, v, vk, vpos]=compose_matrix(component);
c.name='pid_toplevel_struct';
c.param=0;
c.c0=c0;
c.P=P;
c.Pl=Pl;
c.P0=P0;
c.P0l=P0l;
c.upos=upos;
c.vpos=vpos;
c.Cv = Cv;
c.Cu = Cu;
c.C0=C0;
c.vk = vk;
c.v=v;
c.p.v=p.v;
c.p.y=p.y;
c=flat_matrix (c);
```

B.2.3 Allgemeine Funktionen

B.2.3.1 Funktion zur Bestimmung der Matrizen und Vektoren verarbeitender Komponenten

```
function [c]=constants (name, teiler, c0, delaypos, offset)
% returns some constants

msg=nargchk(4,4,nargin);

if nargin < 5
    disp ('Error in constants');
    disp (msg);
```

```
    return;
end

if nargin > 5
    disp ('Error in constants');
    disp (msg);
    return;
end

c.name=name;
c.param=teiler;
c.c0=[];
c.P=[];
c.Pl=[];
c.P0=[];
c.P0l=[];
c.upos=delaypos;
c.vpos=c.upos;
switch lower(name)
case 'adder_adder'
    c.Cv=[1/2 1/2
          1   -1];
    c.Cu=c.Cv;
    c.C0=[1 0];
    c.vk=[1;0];
    c.v=[1;1];
    c.p.v=[1;1];
    c.p.y=[1;0];
    c.P=[0 0
         0 0];
    c.P0=[1 0
          0 1];
    c.upos=[delaypos [0;0]];
case 'derivative_derivative'
    c.Cv=0;
    c.Cu=1;
    c.C0=1;
    c.vk=1;
    c.v=0;
```

```
    c.p.v=0;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos 0];
case 'gaindiv_gaindiv'
    c.Cv=0;
    c.Cu=1;
    c.C0=1;
    c.vk=1+teiler;
    c.v=0;
    c.p.v=0;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos 0];
case 'lt2hf_lt2hf'
    c.Cv=0;
    c.Cu=1;
    c.C0=1;
    c.vk=15;
    c.v=0;
    c.p.v=0;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos 0];
case 'memorycell_memorycell'
    c.Cv=-1;
    c.Cu=1;
    c.C0=1;
    c.vk=0;
    c.v=1;
    c.p.v=1;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos offset];
case 'multidivpar_multidivpar'
```

```
    c.Cv=0;
    c.Cu=1;
    c.C0=1;
    c.vk=1+teiler;
    c.v=0;
    c.p.v=0;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos 0];
case 's2p_s2p'
    c.Cv=0;
    c.Cu=1;
    c.C0=0;
    c.vk=1;
    c.v=0;
    c.p.v=0;
    c.p.y=1;
    c.P=0;
    c.P0=1;
    c.upos=[delaypos 0];
case 'sub_sub'
    c.Cv=[1/2 1/2
          1   -1];
    c.Cu=Cv;
    c.C0=[1 0];
    c.vk=[1;0];
    c.v=[1;1];
    c.p.v=[1;1];
    c.p.y=[1;0];
    c.P=[0;0];
    c.P0=[1;1];
    c.upos=[delaypos [0;0]];
otherwise disp ('Function constants Unknown component');
          disp ('name = ');
          disp (name);
end

c.c0=c.C0*c0;
```

B.2.3.2 Funktion zum Zusammensetzen der Matrizen

```
function [c0, C0, Cu, Cv, P,  P0,  p, upos, v, vk, vpos]=\
 compose_matrix(component)
% composes matrices from its components

msg=nargchk(1,1,nargin);

if nargin < 1
    disp ('Error in compose_matrix');
    disp (msg);
    return;
end

if nargin > 1
    disp ('Error in compose_matrix');
    disp (msg);
    return;
end

[y, noc] = size (component);
c0=[];
P=[];
P0=[];
C0=[];
Cu=[];
Cv=[];
p.v=[];
p.y=[];
vk=[];
v=[];
upos=[];
vpos=[];
for i=1:noc
    c0=[c0 component(i).c0];
    C0=blkdiag(C0,component(i).C0);
    Cv=blkdiag(Cv,component(i).Cv);
    P=blkdiag(P,component(i).P);
    P0=blkdiag(P0,component(i).P0);
```

```
    if component(i).v == 0
        [y,x]=size(Cv);
        Cv=Cv(:,1:x-1);
    else
        v=[v;component(i).v];
    end
    Cu=blkdiag(Cu,component(i).Cu);
    pp=component(i).p;
    if pp.v > 0
      p.v=[p.v;pp.v];
      vpos=[vpos;component(i).vpos];
    end
    p.y=[p.y;pp.y];
    vk=[vk;component(i).vk];
    upos=[upos;component(i).upos];
end
```

B.2.3.3 Funktion zum Flachklopfen von zwei auf eine Hierarchieebene

```
function [c]=flat_matrix(Cin)
% computes flatted martrices

msg=nargchk(1,1,nargin);

if nargin < 1
    disp ('Error in flat_matrix');
    disp (msg);
    return;
end

if nargin > 1
    disp ('Error in flat_matrix');
    disp (msg);
    return;
end

CuP=Cin.Cu*Cin.P*Cin.C0;
basis=((eye(size(CuP))-CuP)^-1);
CCv=basis*Cin.Cv;
```

```
CCu=basis*Cin.Cu*Cin.P0;
vvk=basis*Cin.vk;
PP=Cin.Pl+Cin.P0l*Cin.P*Cin.C0;
PP0=Cin.P0l*Cin.P0;

c.name=Cin.name;
c.param=0;
c.c0=Cin.c0;
c.P=PP;
c.Pl=Cin.Pl;
c.P0=PP0;
c.P0l=Cin.P0l;
c.upos=Cin.upos;
c.vpos=Cin.vpos;
c.Cv = CCv;
c.Cu = CCu;
c.C0=Cin.c0;
c.vk = vvk;
c.v=Cin.v;
c.p.v=Cin.p.v;
c.p.y=Cin.p.y;
```

B.2.3.4 Funktion zum Optimieren

```
function [u,v,y]=optimieren(Cu, Cv, P, P0, p, u0, vk)
% Optimierung der Verzögerungsstufen

msg=nargchk(7,7,nargin);

if nargin < 7
    disp ('Error in optimieren');
    disp (msg);
    return;
end

if nargin > 7
    disp ('Error in optimieren');
    disp (msg);
    return;
```

```
end

% b aufstellen
[vky,vkx]=size(vk);
beq= [eye(vky) Cu]*[vk;u0];
% A aufstellen

[l,m]=size(P);
[x,n]=size(Cv);
[yy,yx]=size(p.y);

Aeq=[-Cv eye(yy)];

[y,lang]=size(Aeq);
f=ones(lang,1);
lb=zeros(lang,1);
ub=[p.v;p.y]*1000;
[x,fval,exitflag,output,lambda]=linprog(f,[],[],Aeq,beq,lb,ub);
v=x(1:n);
y=x(n+1:m+n);
u=P*y+P0*u0;
```

B.2.3.5 Testbench

```
addpath c:\projekte\linopt
clear;
pid_toplevel_struct;
cflat = c;
[uflat,vflat,yflat]= optimieren_flat (cflat.Cu, cflat.Cv, cflat.P,
                                cflat.P0, cflat.p, u0, cflat.vk);
v=[vflat cflat.vpos];
u=[uflat cflat.upos];
maxv=max(v);
delays=zeros(maxv(2)+1, maxv(3)+1);
[rowsv, columnsv]=size(v);
for i=1:rowsv
    delays(v(i,2)+1,v(i,3)+1)=v(i,1);
end
[rowsu, columnsu]=size(u);
```

```
for i=1:rowsu
    if u(i,columnsu) ~= 0
        u(i,1)=u(i,4) - delays (u(i,2)+1,u(i,3)+1);
    end
end
disp ('Eingangssignalverzoegerungen:');
round([u(:,2) u(:,3) u(:,1)])
```

B.2.3.6 Beispiel einer Funktion für eine verarbeitende Komponente

```
function [c]=adder_adder(delay_pos_a, delay_pos_b,
                         error_check, signd, word_length)

name='adder_adder';
c=constants(name, 0, 1, [delay_pos_a; delay_pos_b], 0);
```

B.3 Quelltextgenerierung

VHDL-Package zu Generierung der m-Funktion

```
use work.optimisation.all;
package pid_reduziert_pack is
  function pid_reduziert_delay

return zeile_rec_ret;
end pid_reduziert_pack;

library sersupport;
library ieee;
library std;
library pid;
library serelemente;
use work.optimisation.all;
use std.textio.all;
use Serelemente.basics.all;
use pid.delays.all;
use sersupport.general_units.all;
```

```
use serelemente.basics.all;
use std.textio.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use pid.basics.all;
use ieee.std_logic_unsigned.all;
use work.i_part_mit_fehler_korr_pack.all;
package body pid_reduziert_pack is
  function pid_reduziert_delay

  return zeile_rec_ret is
  file outfile : text open write_mode is "pid_reduziert.m";
  variable rtext : zeile_rec_ret := zeile_rec_ret'(null, null);
  variable hd : zeile_ptr := zeile_ptr'(null);
  variable letzer : zeile_ptr := zeile_ptr'(null);
-- General constants
--constant index1 : integer := index + 1;
constant signd : integer := 1;

begin
-- initiate sub instances code generation
  rtext := i_part_mit_fehler_korr_delay;
-- Generating m function
  add_line (rtext, "function [c]= pid_reduziert
             (divisor,index,kd,ki,kp,word_length);");
  add_line (rtext, "%===============Konstanten");
  add_line (rtext, "index1 = index + 1;");
  add_line (rtext, "signd= "&integer'image(signd)&";");
  add_line (rtext, "p_d = [index,0];");
  add_line (rtext, "d_d = [index,1];");
  add_line (rtext, "pd_d = [index,2];");
  add_line (rtext, "i_d = [index,3];");
  add_line (rtext, "controldeviation_d = [index,4];");
  add_line (rtext, "raw_d_d = [index,5];");
  add_line (rtext, "%========= Instance Code =======");
  add_line (rtext, "%==========add_pd : adder");
  add_line (rtext, "component(1) =adder_adder (p_d,d_d, 1, 1,word_length);");
  add_line (rtext, "component(1).c0 =component(1).c0*[0];");
  add_line (rtext, "%==========add_pid : adder");
```

```
add_line (rtext, "component(2) =adder_adder
                                (pd_d,i_d, 1,signd,word_length);");
add_line (rtext, "component(2).c0 =component(2).c0*[1];");
add_line (rtext, "%==========derivative_0 : derivative");
add_line (rtext, "component(3) =derivative_derivative
                                 (controldeviation_d,word_length);");
add_line (rtext, "component(3).c0 =component(3).c0*[0];");
add_line (rtext, "%==========gain_kd : gaindiv");
add_line (rtext, "component(4) =gaindiv_gaindiv
                                (raw_d_d,kd, 1,divisor,word_length);");
add_line (rtext, "component(4).c0 =component(4).c0*[0];");
add_line (rtext, "%==========gain_kp : gaindiv");
add_line (rtext, "component(5) =gaindiv_gaindiv
                        (controldeviation_d,kp, 1,divisor,word_length);");
add_line (rtext, "component(5).c0 =component(5).c0*[0];");
add_line (rtext, "%==========i0 : i_part");
add_line (rtext, "component(6) =i_part_mit_fehler_korr
                                (divisor,index1,ki,word_length);");
add_line (rtext, "component(6).c0 =component(6).c0*[0];");
add_line (rtext, " P=[ 0 0 0 0 1 0");
add_line (rtext, "     0 0 0 1 0 0");
add_line (rtext, "     1 0 0 0 0 0");
add_line (rtext, "     0 0 0 0 0 1");
add_line (rtext, "     0 0 0 0 0 0");
add_line (rtext, "     0 0 1 0 0 0");
add_line (rtext, "     0 0 0 0 0 0");
add_line (rtext, "     0 0 0 0 0 0];");
add_line (rtext, " P0=[ 0");
add_line (rtext, "      0");
add_line (rtext, "      0");
add_line (rtext, "      0");
add_line (rtext, "      1");
add_line (rtext, "      0");
add_line (rtext, "      1");
add_line (rtext, "      1];");
add_line (rtext, "[c0,C0,Cu,Cv, Pl, P0l, p, upos, v, vk, vpos]=
                               compose_matrix(component);");
add_line (rtext, "c.name='pid_reduziert';");
add_line (rtext, "c.param=0;");
```

```
  add_line (rtext, "c.c0=c0;");
  add_line (rtext, "c.P=P;");
  add_line (rtext, "c.Pl=Pl;");
  add_line (rtext, "c.P0=P0;");
  add_line (rtext, "c.P0l=P0l;");
  add_line (rtext, "c.upos=upos;");
  add_line (rtext, "c.vpos=vpos;");
  add_line (rtext, "c.Cv = Cv;");
  add_line (rtext, "c.Cu = Cu;");
  add_line (rtext, "c.C0=C0;");
  add_line (rtext, "c.vk = vk;");
  add_line (rtext, "c.v=v;");
  add_line (rtext, "c.p.v=p.v;");
  add_line (rtext, "c.p.y=p.y;");
  add_line (rtext, "c=flat_matrix (c);");
  hd := rtext.hd;
  while hd /= null loop
    letzer := hd;
    writeline (outfile, hd.zeile);
    hd := hd.nxt;
    deallocate (letzer);
  end loop;
  rtext := zeile_rec_ret'(null, null);
  return rtext;
end;
end pid_reduziert_pack;
```

B.4 Optimierung

B.4.1 VHDL-Package zur Aufnahme der Verzögerungswerte

```
-------------------------------------------------------------------------
-- Title      : contains all delays of a design
-- Project    : Serielle Elemente
-------------------------------------------------------------------------
-- File       : delay_table.vhdl
-- Author     : Thomas Reinemann  <thomas.reinemann@mb.uni-magdeburg.de>
```

```
-- Company    : Otto-von-Guericke-Universität Magdeburg
-- Created    : 2002-03-04
-- Last update: 2003-02-24
-- Platform   :
-------------------------------------------------------------------------
-- Description:
-------------------------------------------------------------------------
-- Copyright (c) 2002 Otto-von-Guericke-Universität Magdeburg
-------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2002-03-04  1.0      threinem Created
-------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

package delays_table is
  type table is array (natural range <>, natural range <>) of integer;
  type table_100 is array (0 to 99, 0 to 9) of integer;

procedure read_delays ( delay_table : out table_100 );
attribute foreign of read_delays : procedure is
                                          "read_delays ./delays.dll";
function wrap_read_delay  return table_100;

end delays_table;

use work.delays_table.all;
package body delays_table is

-- purpose: reads the delays from the Matlab result via FLI
    procedure read_delays ( delay_table : out table_100 ) is
    begin
      assert false report
        "ERROR: foreign subprogram not called" severity note;
    end read_delays;

-- purpose: wraps the read_delay procedure call
```

```
  function wrap_read_delay
    return table_100 is

    variable delay_table : table_100;

  begin  -- wrap_read_delay

    for i in 0 to 99 loop
      for j in 0 to 9 loop
        delay_table (i,j):=0;
      end loop;  -- j
    end loop;  -- i
    read_delays (delay_table);
    return delay_table;
  end wrap_read_delay;

end delays_table;
```

B.4.2 C-Funktion zum Füllen der Verzögerungswerttabelle

```
#include <mti.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "delays.h"

#define MAXZEILENLAENGE 250
enum ZUSTAENDE {INIT, SUCESSFULL, VERZOEGERUNGEN_HD, VERZOEGERUNGEN, ENDE};

extern "C"
{
  void read_delays (mtiVariableIdT varid);
}

void read_delays (mtiVariableIdT varid) {

  mtiVariableIdT * elem_list1;
  mtiVariableIdT * elem_list2;
  mtiVariableIdT * delay_table;
```

```
char *filename = ".\\opti_res.txt";
char zeile[MAXZEILENLAENGE];
char *s;
int rc, zeilepos, spaltepos, verzoegerung;
long length, width;
long* lhelp;
FILE *optires;
enum ZUSTAENDE zustand=INIT;
  delay *delays = NULL;

mti_PrintFormatted ("=======read_delays cpp======\n");
elem_list1 = mti_GetVarSubelements(varid, NULL);
elem_list2 = mti_GetVarSubelements(elem_list1[0], NULL);

lhelp = (long*)  mti_GetVarValueIndirect(elem_list2[0], NULL);
length = *lhelp;
lhelp = (long*)   mti_GetVarValueIndirect(elem_list2[1], NULL);
width = *lhelp;
mti_PrintFormatted ("length = %d and width = %d\n",length, width);

optires = fopen (filename, "r");
if (optires == NULL) {
    mti_PrintFormatted
      ("Unable to open optimization result file %s!\n", filename);
    mti_Break ();
} else {
  while (!feof(optires)) {
    s = fgets(zeile, MAXZEILENLAENGE, optires);
    if ((s != NULL) && (strlen(s) > 1)) {
      switch (zustand) {
      case INIT:
        rc = (int)strstr (zeile, "Optimization terminated successfully");
        if (rc) {
          zustand = SUCESSFULL;
          mti_PrintFormatted ( "Zustand SUCESSFULL erreicht\n");
        }
      break;
      case SUCESSFULL:
        rc = (int)strstr (zeile, "Eingangssignalverzoegerungen");
```

```
          if (rc) {
            zustand = VERZOEGERUNGEN_HD;
            mti_PrintFormatted ( "Zustand VERZOEGERUNGEN_HD erreicht\n");
          }
        break;
        case VERZOEGERUNGEN_HD:
          rc = (int)strstr (zeile, "ans");
          if (rc) {
            zustand = VERZOEGERUNGEN;
            mti_PrintFormatted ( "Zustand VERZOEGERUNGEN erreicht\n");
          }
        break;
        case VERZOEGERUNGEN:
          rc = (int)strstr (zeile, ">>");
          if (rc) {
            zustand = ENDE;
            mti_PrintFormatted ( "Zustand ENDE erreicht\n");
          }
          else {
            sscanf (zeile,"%d %d %d",&zeilepos, &spaltepos, &verzoegerung);
            if (!delays)
              delays = new delay (zeilepos, spaltepos, verzoegerung);
            else
               new delay (zeilepos, spaltepos, verzoegerung);
          }
        break;
        case ENDE:
          mti_PrintFormatted ( "Zustand ENDE erreicht\n");
        break;
        default :
          mti_PrintFormatted
    ("Falscher Zustand File %s Zeile %d\n",__FILE__, __LINE__);
        }
      }
    }
    fclose (optires);
}
//  delays->write_delays ();
if (zustand == ENDE) {
```

```
    delays->set_values (elem_list1);
  } else {
    mti_PrintFormatted
      ("Fehler beim Parsen der Resultate im Zustand %d\n", zustand);
    mti_Break ();
  }
  delete delays;

  mti_VsimFree( elem_list2 );
  mti_VsimFree( elem_list1 );
}
```

B.4.3 Ein typisches Optimierungsergebnis

```
Warning: Unrecognized MATLAB option "".

                          < M A T L A B >
                Copyright 1984-2002 The MathWorks, Inc.
                    Version 6.5.0.180913a Release 13
                              Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> >> >> >> >> Optimization terminated successfully.
>> >> >> >> >> >> >> >> >> Eingangssignalverzoegerungen:
>>
ans =

     0     0     9
     1     0     6
     1     1     7
     1     2     8
     1     3     7
     1     4     0
     1     5     1
     1     4     0
```

```
    2     0     1
    2     1     0
    2     2     0
    2     3     0
    2     0     1

>>
```

Index

Literaturverzeichnis

[And03a] Andraka, Ray: *FPGA DSP Performance Comparision.* www.andraka.com/dsp.htm, last visited 21.01.2003, 01 2003.

[And03b] Andraka, Ray: *Multiplikation in FPGAs.* www.andraka.com/multipli.htm, last visited 21.01.2003, 01 2003.

[Aue95] Auer, Adolf: *FPGA: feldprogrammierbare Gate arrays.* Huethig, Heidelberg, 1995.

[Bir95] Biran, A.; Breiner, M.: *Matlab for Engineers.* Addison-Wesley Publishers Ltd., Wokingham, England, 1995.

[Bla97] Bland, I.; Megson, G.: *Efficient operator pipelining in a bit serial genetic algorithm engine.* Electronic Letters, 33(12):1026–1028, 1997.

[Blu02] Blume, H.: *Midel-based Exploration of the Design Space for Heterogeneous Systems on Chip. Proceedings of the Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, 2002. VDI Verlag.

[Bro91] Bronstein, I.N.; Semendjajew, K.A.: *Taschenbuch der Mathematik.* Teubner Verlagsgesellschaft, Stuttgart-Leipzig, 1991.

[Deh99] Dehon, A.; Wawrzynek, J.: *Reconfigurable Computing: What, Why and Implications for Design Automation. Design Automation Conference*, San Francisco, 1999. IEEE Computer Society Press.

[Erc02] Ercegovac, Milos D.: *Digit-serial arithmetic.* www.cs.ucla.edu/~milos/ch9.pdf, December 2002.

[Fou01] Fourer, Robert: *Linear Programming Frequently Asked Questions.* www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html, 2001.

[Föl90] Föllinger, Otto: *Regelungstechnik.* Hüthig, Heidelberg, 1990.

[Föl93] Föllinger, Otto: *Lineare Abtastsysteme.* Oldenbourg, München, 1993.

[Gra99] Gray, David: *Introduction to the formal design of real-time systems.* Springer, London, 1999.

[Har98] Hardt, W.; Kleinjohann, B.: *FLYSIG: Dataflow Oriented Delay-Insensitive Processorfor Rapid Prototyping of Signal Processing.* Sixth IEEE International Workshop on Rapid System Prototyping, Juni 1998.

[Hla92] Hlander, A.; Svensson, B.: *Floating point calculations in bit-serial SIMD computers. The Fourth Swedish Workshop on Computer System Architecture*, Linköping, Sweden, 1992.

[Hos94] Hostetter, Santina; Stubberud;: *Digital Control System Design.* Saunders College Publishing, Orlando, Florida, 1994.

[Kop98] Kopetz, Hermann: *Real-time systems: design principles for distributed applications.* Kluwer, Boston, 1998.

[Kor93] Koren, Israel: *Computer Arithmetic Algorithms.* Prentice Hall, Englewood Cliffs, 1993.

[Lee97] Lee, H.H.; Sobelman, G.E.: *FPGA-Based FIR Filters Using Digit-Serial Arithmetic. Proceedings of IEEE international ASIC Conference*, Napa Valley California, USA, 9 1997. IEEE Computer Society Press.

[Leo00] Leong, J.M.P.: *A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley California, USA, 2000. IEEE Computer Society Press.

[Lon02] London, Greg: *Hardware-VHDL-Parser.* search.cpan.org/author/GSLONDON/Hardware-Vhdl-Parser-0.12/, 11 2002.

[Lut98] Lutz, H.; Wendt, W.: *Taschenbuch der Regelungstechnik.* Verlag Harri Deutsch, Frankfurt am Main, 1998.

[Mat01] Mathworks: *Linear Programming Frequently Asked Questions.* www.matlab.com, 2001.

[Moh98] Mohr, Richard: *Numerische Methoden in der Technik.* Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 1998.

[Nil97] Nilsson, P.; Torkelson, M.: *A Custom Digital Intermediate Frequency Filter for the American Mobile Telephone System.* IEEE journal of solid-state circuits, 32(6), June 1997.

[Pag96] Page, I.: *Constructing hardware-software systems from a single description.* Journal of VLSI Signal Processing, 12(1):87–107, 1996.

[Rei98] Reisch, Michael: *Elektronische Bauelemente.* Springer, Berlin-Heidelberg, 1998.

[Sie01a] Siemers, Christian: *Die Makimoto-Welle.* Design&Elektronik, 2001(3):48–55, 2001.

[Sie01b] Siemers, Christian: *Rechenfabrik.* c't Magazin für Computertechnik, 2001(15):170–179, 2001.

[Sie02a] Siemers, Christian: *Kostenfaktor.* c't Magazin für Computertechnik, 2002(10):43, 2002.

[Sie02b] Siemers, Christian: *Reconfigurable Computing.* Design&Elektronik, 2002(2), 2002.

[Sti01] Stiller, Andreas: *Prozessorgeflüster.* c't Magazin für Computertechnik, 2001(5), 2001.

[Val98] Valls, J.; Peiro, M.M.: *Design and FPGA implementation of Digit-Serial FIR filters. Proceedings of IEEE International Conference on Electronics, Circuits and Systems*, Napa Valley California, USA, 1998. IEEE Computer Society Press.