DIPLOMARBEIT

Rekonfigurierbare Befehlssatzerweiterung zur Leistungssteigerung von Universalprozessoren

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dr. Dietmar Dietrich
Dr. Florian Schupfer
Dipl.-Ing. Michael Rathmair

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Günther Mader
Matr.Nr. 0826653
Reinprechtsdorferstraße 34/17, 1050 Wien

08.04.2014 _____

Kurzfassung

Rechenleistung, Energieeffizienz und Entwicklungsaufwand sind wesentliche Parameter bei der Entwicklung von Embedded Systems. Um diese Anforderungen zu erfüllen reicht es meist nicht aus einen leistungsstarken Universalprozessor mit mehreren Kernen zu verwenden. Der Trend geht daher in Richtung Verwendung von spezialisierter Hardware zur Unterstützung der Universalrechenkerne. So verfügen moderne Systeme oft neben mehreren Hauptprozessoren über eine immer größer werdende Anzahl von zusätzlichen Berechnungseinheiten, welche die Abarbeitung komplexer Funktionen beschleunigen [4]. Ziel ist die Optimierung des Verhältnisses von Rechenleistung und Energieeffizienz für das jeweilige Anwendungsgebiet. Um den Entwicklungsaufwand in Grenzen zu halten ist dabei eine effiziente Wiederverwendung des Designs erforderlich. Deshalb beschäftigt sich diese Diplomarbeit mit rekonfigurierbarer Hardware, welche sowohl Wiederverwendung des Designs bei der Hardwareentwicklung als auch von Hardwareressourcen während des Betriebs ermöglicht. Es wird ein Ansatz erarbeitet, der es erlaubt den Befehlssatz eines Universalprozessors zu erweitern. Dieser sieht eine Art Modulsystem vor um auch Änderungen und Erweiterungen des Applikationsgebietes für zukünftige Anwendungen abzudecken. Es besteht aus dem eigentlichen Prozessor mit einer generischen Schnittstelle und beliebigen rekonfigurierbaren Erweiterungsfunktionseinheiten, die daran angeschlossen werden können. Berechnungen und Konfigurierung der Erweiterungsblöcke werden über die Schnittstelle mit Erweiterungsbefehlen ausgeführt. Das ermöglicht hardware- und softwareseitig eine effiziente Wiederverwendung von Ressourcen, Design und Entwicklungsumgebung. Durch eine geeignete Demoimplemtierung wird die Funktionsfähigkeit des vorgestellten Ansatzes gezeigt und die durch die rekonfigurierbaren Erweiterungsinstruktion erzielbaren Steigerungen von Rechenleistung und Energieeffizienz dargestellt.

Abstract

Processing power, energy efficiency and development effort are essential parameters in the development of embedded systems. To meet these requirements, the use of a powerful general purpose processor with multiple cores is mostly not enough. Therefore the trend goes towards the use of specialized hardware to support the general purpose computing cores. Modern systems often are equipped with several processors and a permanently increasing number of additional computation units, which accelerate the execution of complex functions [4]. The aim is to optimize the ratio of computing performance and energy efficiency for the desired application. In order to keep the development effort within reasonable limits, efficient reuse of the design is required. To achieve this goal this thesis deals with reconfigurable hardware, which enables reuse of hardware design in the development stage as well as reuse of hardware resources during operation. An approach that allows to extend the instruction set of a general-purpose processor is developed. This provides a kind of module system to cover modifications and extensions of the application area also for future applications. It consists of the main processor with a generic interface and reconfigurable functional units as extensions which can be connected thereto. Calculations and configuration of the extension blocks are executed with extension commands via the interface. This enables hardware- and software-efficient reuse of resources, design and development environment. By demonstrating the functionality with a suitable implementation, the proposed approach is shown. Moreover the increase in computing power and energy efficiency achieved by the reconfigurable expansion instructions are illustrated.

Inhaltsverzeichnis

Abkürzungen

ADC	Analog/Digital Converter
ADDR	Address
ADSP	Application Domain Specific Processor
ALU	Arithmetical and Logical Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
CISC	Complex Instruction Set Computer
CLK	CLocK (Taktsignal)
COM	COMmand (Spezialbefehl)
CORDIC	COordinate Rotation DIgital Computer
CPLD	Complex Programmable Logic Device
DEC	DECode
DLX	DeLuXe
DPGA	Dynamically Programmable Gate Array
E/A	Ein-/Ausgabe
EN	ENable
EX	EXecute
F	Fetch
FFT	Fast Fourier Transform
FIR	Finite Impuls Response
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
GR	GRant
I	Interrupt
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
IoT	Internet of Things
IP	Intellectual Property
IRQ	Interrupt ReQuest
LED	Light Emitting Diode
LUT	Look Up Table
MAC	Multiply and ACcumulate
MARS	Microprocessor without interlocked pipeline stages Assembler and Runtime Simulator
MEM	MEMory-access
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory Management Unit
MUX	MUltipleXer
NAN	Not A Number
OP	OPerand
PE	Processing Element

PFU	Programmable Functional Unit
PRISC	PRogrammable Instruction Set Computer
PTR	PoinTeR
R31	Register 31 (Rücksprungregister)
RDY	ReaDY
ReRISC	Reconfigurable Reduced Instruction Set Computer
RES	RESult
REQ	REQuest
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SA	Shift Amount
SFR	Special Function Register
SOC	System On Chip
SP	Stack Pointer
SYNC	SYNCronization (Pipeline-Synchronität)
TFLOPS	Tera FLoating-point Operations Per Second
VHDL	Very high speed integrated circuit Hardware Description Language
WR	WRite
WB	Write-Back

Mathematische Symbole

Symbol	Definition	Einheit
α	Winkel einer CORDIC Teilrotation	rad
θ	Soll-Rotationswinkel CORDIC-Algorithmus	rad
$\widetilde{\theta}$	Ist-Rotationswinkel CORDIC-Algorithmus	rad
σ	Vorzeichen einer CORDIC Teilrotation	1
f_{abs}	absoluter Fehler	1
f_{rel}	relativer Fehler	1
i	CORDIC Iterationsindex (ganze Zahl)	1
in_1	erster Eingangsoperand CORDIC-Erweiterung	1
in_2	zweiter Eingangsoperand CORDIC-Erweiterung	1
k	Ergebnis bei Ganzzahldivision	1
K	Dehnungsfaktor beim CORDIC-Algorithmus	1
m	Index CORDIC-Koordinatensystem {-1,0,1}	1
max	obere Integrationsgrenze	1
n	Anzahl CORDIC-Iterationen (ganze Zahl)	1
out_{ist}	Ergebnis CORDIC-Hardware	1
out_{soll}	exaktes Ergebnis CORDIC-Funktion	1
pos	mittlere Bitposition	1
R	Operator CORDIC-Teilrotation	1
$\overrightarrow{v}$	allgemeiner Vektor	1
x	Funktionsargument	1
x'	modifiziertes Funktionsargument	1
z^{-1}	Verzögerung um einen Taktzyklus	1
x_i, y_i, z_i	CORDIC Iterationsvariablen	1

1 Einleitung

Praktisch jedes digitale Gerät verfügt über programmierbare Elemente, meist in Form von Prozessoren. Zukunftsvisionen wie z. B. das IoT (**I**nternet **o**f **T**hings) sehen sogar vor, verschiedenste Objekte des täglichen Lebens mit Intelligenz - also Prozessoren - auszustatten [5]. Diese Objekte sind miteinander vernetzt und verfügen oftmals über Sensoren, was eine Ausführung in Echtzeit und das Übertragen von Daten erfordert. So sollen durch die Verfügbarkeit und Auswertung sehr vieler Daten Dinge des täglichen Lebens vereinfacht und ein effizienterer Umgang mit Ressourcen wie etwa Energie ermöglicht werden. Weil dazu eine große Anzahl unterschiedlichster intelligenter Objekte erforderlich ist, müssen die darin eingebetteten Systeme sehr kostengünstig und für die jeweilige Applikation geeignet sein. Zudem sind viele dieser Systeme batteriebetrieben und schwer erreichbar, wodurch auch die Energieeffizienz ein wichtiger Faktor ist. Somit hängt die Verwirklichung des IoTs vor allem von der Verfügbarkeit energie- und kosteneffizienter Prozessoren mit ausreichender Rechenleistung ab.

Auch wenn das IoT noch nicht Realität geworden ist, stellen Embedded Systems bereits heute einen Großteil des Hardwaremarktes dar und dieser Trend setzt sich fort [9]. Es wird davon ausgegangen, dass diese Systeme immer schneller und funktioneller werden, wobei eine Steigerung um den Faktor zwei alle zwei Jahre eine gängige Annahme ist [4]. Dass es jedoch schwierig werden wird zukünftige Erwartungen an Rechenleistung und Energieverbrauch zu erfüllen zeigt Abbildung 1.1 für den Bereich mobiler Consumerelektronik, welcher stark im Wachsen ist [2]. Die Bewältigung dieser zukünftigen Anforderung, auch bezüglich Entwicklungsaufwand und -kosten bilden die Motivation dieser Diplomarbeit und werden daher im folgenden Abschnitt genauer behandelt.

1.1 Motivation und Zielsetzung

Die Entwicklung neuer Produkte ist oft von starkem Kostendruck und kurzer Time-to-Market geprägt. Deshalb muss der Entwicklungsaufwand trotz ständiger Erhöhung der Funktionalität so gering wie möglich ausfallen. Das setzt eine Steigerung der Produktivität voraus, die auf Wiederverwertung und Minimierung des dabei entstehenden Overheads basiert. So ist es laut Prognosen im Consumerbereich bis zum Jahr 2026 erforderlich 98% eines Designs wieder zu verwenden [4].

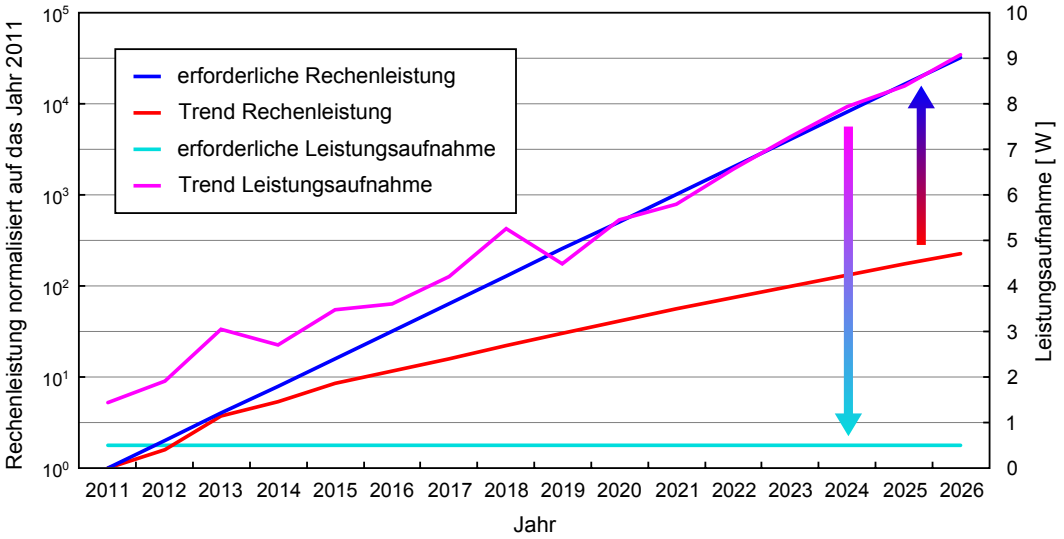

Abbildung 1.1: Entwicklung wichtiger Systemparameter im mobilen Consumerbereich [4]

Deshalb wäre ein Universalprozessor das ideale Bauteil für neue Entwicklungen. Denn durch seinen universellen Aufbau ist der Prozessor in der Lage annähernd beliebige Aufgaben zu erfüllen und somit praktisch alle Anwendungsgebiete und Funktionen abzudecken. Auch seitens der Software ergeben sich durch Einsatz eines Universalprozessors Vorteile, weil immer der gleiche Befehlssatz zur Anwendung kommt. Dadurch können immer die selben Entwicklungsumgebungen verwendet und ältere Programme kompatibel gehalten werden.

Allerdings besteht auch ein ständiger Bedarf zur Erhöhung der Rechenleistung, die notwendig ist um die geforderte Funktionalität innerhalb eines vorgegebenen Zeitrahmens bereit zu stellen bzw. echtzeitfähige Systeme zu ermöglichen. Um das zu erreichen gibt es bei Universalprozessoren mehrere Möglichkeiten:

- neue Fertigungstechnologie / neuartige Transistoren
- Erhöhung der Taktfrequenz
- Verbesserung der Architektur / Steigerung der Komplexität eines Prozessorkerns
- größere Anzahl an Recheneinheiten

Während in den ersten Jahrzehnten der Mikroprozessorentwicklung die Leistungssteigerung vor allem auf den ersten drei Punkten basierte, muss heute ein anderer Weg gewählt werden. Zwar entwickelt sich die Fertigungstechnologie ständig weiter, allerdings kann die historisch verwirklichte Steigerung der Taktfrequenz um den Faktor 1,4 pro Technologiegeneration nicht mehr fortgesetzt werden [4]. Auch die Komplexität von Prozessorkernen ist mittlerweile an einem Punkt angelangt, an dem sich nur mehr wenig ändert. So verbleibt unter Annahme der selben zugrunde liegenden Technologie nur noch die Erhöhung der Anzahl an Recheneinheiten. Das findet bereits statt und ist in Abbildung 1.2 (TFLOPS = **T**era **F**oating **P**oint **O**perations **P**er **S**econd) für die kommenden Jahre am Beispiel stationärer Consumerelektronik (z. B. Spielkonsolen) dargestellt.

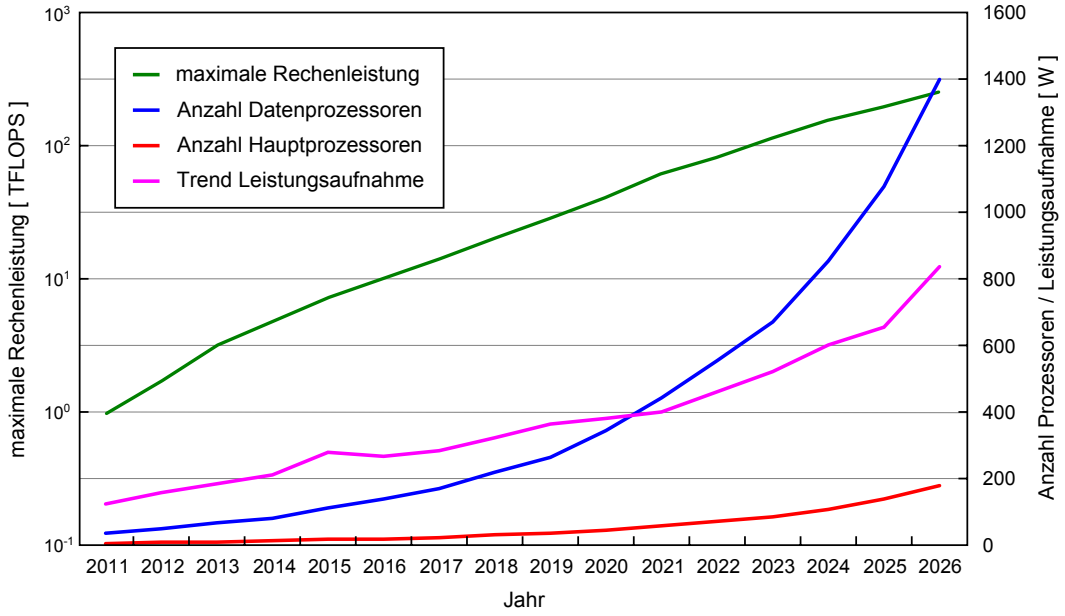

Abbildung 1.2: Entwicklung wichtiger Systemparameter im stationären Consumerbereich [4]

Ein großes Problem, das mit der Duplikation von Rechenkernen einhergeht ist die Erhöhung der Leistungsaufnahme (siehe Abbildung 1.2). Bei Geräten mit Anschluss an das Stromnetz besteht die Schwierigkeit darin, die erforderliche elektrische Leistung in den Chip hinein zu bekommen und die entstehende Abwärme wieder abzuführen. Das kann bei Embedded Systems bereits ein Problem hinsichtlich der Kühlung des Chips darstellen [4]. Bei mobilen und batteriebetriebenen Geräten ergibt sich jedoch ein weitaus gravierenderes Problem, da diese nur einen sehr beschränkten Energievorrat zur Verfügung haben. Weil sich Energiespeicher nur sehr langsam weiterentwickeln, bleiben die Anforderungen an die Leistungsaufnahme für die kommenden Jahre voraussichtlich konstant (siehe Abbildung 1.1). Die dargestellte Grenze von 0,5 W gilt für mobile Consumerelektronik mit Ausnahme größerer Geräte wie Tablets und wird bereits überschritten [4]. So sind Universalprozessoren alleine oft nicht in der Lage die hohen Anforderungen an Rechenleistung und Energieeffizienz von Embedded Systems zu erfüllen.

Wird daher das Hauptaugenmerk des Designs eines solchen Systems auf die Effizienz der Hardware gelegt, besteht eine Optimierungsmöglichkeit darin, die Hardware exakt auf die Aufgabenstellung anzupassen. Neben einer möglichen Leistungssteigerung können damit Energiebedarf und die benötigte Chipfläche gesenkt werden [VS07, Kapitel 2.2.2]. Allerdings bringt eine derartige Anpassung auch eine Einschränkung der Funktionalität mit sich, sodass ein anwendungsspezifisch entworfener digitaler Schaltkreis für andere Anwendungen nicht mehr geeignet ist. Deshalb spielen bei diesen sogenannten ASICs (**A**pplication **S**pecific **I**ntegrated **C**ircuit) Entwicklungszeit und -kosten eine wesentliche Rolle [7].

Es gibt nicht nur die beiden konträren Ansätze von ASIC und Universalprozessor, sondern auch verschiedene Abstufungen, bei denen der Befehlssatz und die Hardware teilweise auf das Anwendungsgebiet angepasst sind. Man unterscheidet dabei zwischen ADSPs (**A**pplication **D**omain **S**pecific **P**rocessor) und ASIPs (**A**pplication **S**pecific **I**nstruction-Set **P**rocessor), wobei z. B. DSPs (**D**igital **S**ignal **P**rocessor) in die Kategorie der ADSPs fallen [GB11]. Diese Typen verfügen jedoch über eine statische Anpassung, sodass um das Beispiel des DSPs fortzuführen, der Befehlssatz speziell für Signalverarbeitungsalgorithmen ausgelegt ist. Wenn sich also das Anwendungsgebiet ändert, ist auch hier der Prozessor nicht mehr optimal geeignet und muss möglicherweise ersetzt werden.

Wegen Rechenleistung, Entwicklungsaufwand und Energieeffizienz wäre deshalb eine Kombination der Vorteile von ASICs, ADSPs und Universalprozessoren wünschenswert, um eine einzige Hardwarelösung für alle Problemstellungen zu haben. Dabei garantiert der Universalprozessor, dass alle aktuellen und zukünftigen Anwendungsgebiete abgedeckt werden und bestehende Software kompatibel bleibt. Der ADSP stellt immer die passenden Maschinenbefehle zur Verfügung, um eine effiziente Programmierung unter Benutzung von vorhandenen Entwicklungsumgebungen zu ermöglichen. Der ASIC schließlich gewährleistet eine optimale Hardware, welche im Stande ist alle Anforderungen in Bezug auf Rechenleistung und Energieeffizienz zu erfüllen. Da diese Hardware somit für alle Problemstellungen geeignet ist, erfüllt sich auch die Forderung nach einer effizienten Wiederverwertung des Designs.

Als Folge der in diesem Abschnitt durchgeführten Überlegungen bezüglich Prozessoreigenschaften wird die wissenschaftliche Problemstellung für den weiteren Verlauf der Diplomarbeit abgeleitet:

- Optimierung eines Universalprozessors hinsichtlich Rechenleistung und Energieeffizienz durch Befehlssatzerweiterung
- Beibehaltung von Standardbefehlssatz zur Maximierung der Wiederverwendung von Software und Entwicklungswerkzeugen
- zukünftige Erweiterbarkeit / Variation des Applikationsgebietes ohne Modifikation des Prozessorkerns zur Maximierung der Wiederverwendung des Hardwaredesigns und Minimierung des dabei entstehenden Overheads

1.2 Aufgabenstellung

Die Aufgabe dieser Diplomarbeit ist es, eine Lösung für die in Kapitel 1.1 präsentierte Problemstellung zu erarbeiten und eine eigene Prozessorarchitektur auf Basis dieser Anforderungen zu entwickeln. Das setzt einen flexiblen Aufbau des Prozessors voraus, wobei eine zweigeteilte Flexibilisierung vorgesehen ist.

Den ersten Teil der Flexibilisierung bilden spezielle Hardwareerweiterungen, die den Befehlssatz des Prozessors erweitern und optimal auf das aktuelle Aufgabenfeld angepasst werden können. Damit eine bestmögliche Ausnutzung der Erweiterungen erreicht werden kann, sollen diese rekonfigurierbar sein, zudem soll es möglich sein die Hardware auch während der Programmausführung über Maschinenbefehle zu rekonfigurieren. So kann der erweiterte Befehlssatz des Prozessors auf die Anforderungen einer laufenden Anwendung flexibel angepasst werden, um eine effiziente Abarbeitung der benötigten Operationen zu ermöglichen. Für die Erweiterungen bzw. die durch

sie zur Verfügung gestellten Befehle bedeutet das, dass sie je nach Konfiguration unterschiedliche Operationen ausführen können. Durch diese Form der Anpassung kann daher nicht nur die Rechenleistung gesteigert, sondern auch der Energieverbrauch durch die eine Verringerung der auszuführenden Befehle reduziert werden.

Der zweite Teil der Prozessorflexibilisierung sieht vor, die Prozessorerweiterungen austauschbar zu gestalten, um den Chip bereits vor der Fertigung konfigurieren zu können. Diese Möglichkeit zielt damit sowohl auf zukünftige Anwendungen, als auch auf eine optimale Ausstattung des Prozessors für existierende Anwendungen ab. Somit kann jeder Prozessor individuell auf seine spätere Aufgabe angepasst werden, wobei sogar Bibliotheken für Erweiterungen vorstellbar wären.

Auch für den Fall, dass eine gewünschte Erweiterung noch nicht existieren sollte, reduziert sich der Entwicklungs- und Testaufwand erheblich, da der eigentliche Prozessor nicht verändert wird und sich der Entwicklungsaufwand nur auf die neu zu entwerfenden Erweiterung bezieht. Durch die Trennung von Erweiterungsbefehlen und allgemeinen Prozessorkern können die Entwicklungskosten gesenkt und die Time-to-Market reduziert werden. Ein weiterer Vorteil bei der Verwendung eines fixen Prozessorkerns ist, dass der Standardbefehlssatz des Prozessors gleich bleibt und somit Kompatibilität zu früheren Programmen erhalten bleibt. Zudem können auch bereits vorhandene Compiler weiter verwendet werden und müssen nur für die Verwendung des erweiterten Befehlssatzes angepasst werden.

Das vorgestellte Konzept ist in Abbildung 1.3 dargestellt und soll im Zuge der Diplomarbeit zu einem konkreten Entwurf eines Prozessordesigns weiterentwickelt werden. Daraus soll eine Beispielimplementierung mittels VHDL (**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage) erstellt werden, wobei das Ziel der Implementierung ein funktionierender Prototyp in einem FPGA (**F**ield **P**rogrammable **G**ate **A**rray) ist. Als Ausgangspunkt dafür dient ein RISC (**R**educed **I**nstruction **S**et **C**omputer) Prozessor. Dieser besitzt im Vergleich zum CISC (**C**omplex **I**nstruction **S**et **C**omputer) Prozessor eine reduzierte Komplexität und ist daher besser geeignet um Modifikationen vorzunehmen. Der implementierte Prototyp soll dabei neben dem eigentlichen Prozessor über zwei zur Laufzeit rekonfigurierbare Hardwareerweiterungen verfügen. Die Rekonfiguration der Erweiterungseinheiten soll jedoch nicht auf Basis einer dynamischen Konfiguration des FPGAs erfolgen, sondern durch eine Rekonfiguration deren Struktur ermöglicht werden, um nicht auf FPGAs als Zielhardware beschränkt zu sein. Für das Hinzufügen oder gegebenenfalls Entfernen von Erweiterungskomponenten kann aber durchaus die Konfiguration des FPGAs dienen, da diese Art der Konfigurierung ohnehin vor einer eventuellen Fertigung vorgesehen ist.

Um die Funktionsfähigkeit des Prototypen zu zeigen, sollen Demonstrationsprogramme erstellt werden Es wird auf bereits existierende Assembler zurückgegriffen, da der Standardbefehlssatz des implementierten Prozessors kompatibel zu gängigen RISC-Architekturen sein soll. Mit Hilfe dieser Beispielprogramme soll abschließend die Rekonfiguration der Hardware gezeigt und die erzielbaren Leistungssteigerungen untersucht werden.

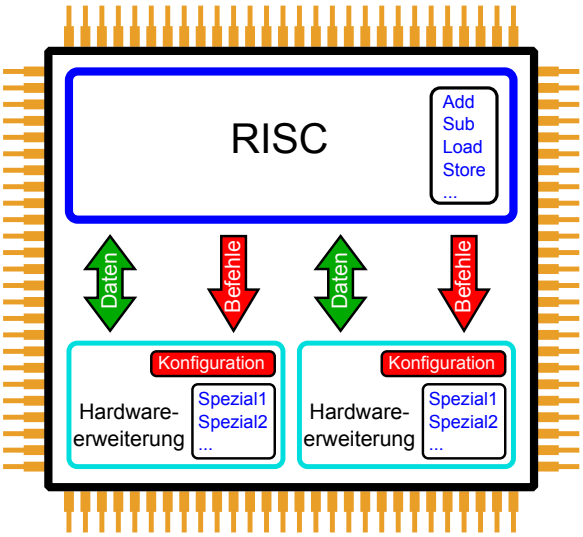

Abbildung 1.3: RISC mit rekonfigurierbaren Befehlssatzerweiterungen

1.3 Gliederung der Arbeit

Im ersten Teil dieser Arbeit werden die Möglichkeiten von rekonfigurierbaren Befehlssatzerweiterungen untersucht. Bestehende Arbeiten und Lösungen auf diesem Gebiet werden dazu in Kapitel 2 diskutiert und mit dem eigenen Ansatz verglichen. Nach diesem Überblick über den Stand der Technik wird in Kapitel 3 der Ansatz zur Verwirklichung des Konzepts aus Kapitel 1.2 im Detail vorgestellt. Darauf aufbauend wird ein beispielhaftes Hardwaredesign entwickelt, welches in Kapitel 4 vorgestellt wird.

Auf die zur Demonstration des Prototypen entwickelten Beispielprogramme wird in Kapitel 5 eingegangen. Dabei widmet sich das Kapitel im ersten Teil dem Testen der Hardware und der Erstellung dieser Tests. Für die Tests wurden im Zuge der Diplomarbeit Simulationsmodelle in MATLAB geschrieben, die das Verhalten der Ein- und Ausgänge unter Berücksichtigung des Timings darstellen. Dadurch ist es möglich, Stimulidaten für die Testbenches automatisch zu generieren. Im zweiten Teilabschnitt werden die implementierten Erweiterungsbefehle mit den Softwarependants des Standardbefehlssatzes ohne Hardwareerweiterungen gegenüber gestellt. Dabei wird sowohl auf die Leistungssteigerung, als auch den erhöhten Hardwareaufwand durch die Erweiterungen eingegangen. Außerdem werden insbesondere die bei der Implementierung aufgetretenen Probleme angeführt und diskutiert.

Den Abschluss der Arbeit bildet Kapitel 6, in dem die Arbeit und deren wichtigste Resultate zusammengefasst werden. Weiters gibt dieses Kapitel einen kurzen Ausblick auf mögliche Weiterentwicklungen. In Kapitel A werden zusätzliche Informationen für die Programmierung des implementierten Prozessors bereitgestellt. Außerdem befindet sich hier eine Auflistung der Erweiterungsbefehle für die Spezialhardwareerweiterungen.

2 Stand der Technik

Dieses Kapital widmet sich dem Stand der Technik und stellt vorhandene Konzepte kurz vor. Zuerst wird das Prinzip der Befehlssatzerweiterung vorgestellt und daran anschließend exemplarisch vorangegangene Arbeiten auf diesem Gebiet chronologisch präsentiert. Es wird auf die Vor- und Nachteile der Arbeiten eingegangen und insbesondere die Unterschiede zu dieser Diplomarbeit behandelt und diskutiert.

2.1 Befehlssatzerweiterungen

Mit dem Standardbefehlssatz eines Universalprozessors kann zwar annähernd beliebige Funktionalität bereitgestellt werden, allerdings ist das in manchen Fällen sehr ineffizient und die Kosten in Form von Taktzyklen und benötigter Energie für diese Operationen sind signifikant. Treten solch ineffiziente Operationen häufig auf, kann dies bezüglich der benötigten Ausführungszeit ein Problem darstellen. Deshalb werden Prozessoren schon seit mehreren Jahrzehnten mit zusätzlichen Befehlen ausgestattet, die ansonsten sehr ineffiziente Operationen als direkte Maschinenbefehle bereitstellen. In den 1980er Jahren, als die Transistorzahl pro Chip noch auf wenige hunderttausend limitiert war [Bon98], wurden beispielsweise Coprozessoren entwickelt, die den Befehlssatz des Hauptprozessors mit Operationen basierend auf dem IEEE (**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers) Gleitkommazahlenformat (siehe Kapitel A.1) erweitern. Ein ausgewähltes Beispiel dafür ist der MC68881 Coprozessor von Motorola, welcher als Erweiterung der M68000 Familie konzipiert wurde [HC83]. Diese Form der Erweiterung zeichnet sich dadurch aus, dass Coprozessor und Hauptprozessor meist parallel arbeiten und zum Teil über eine eigene Speicheranbindung verfügen [GB11]. Während im Einführungsbeispiel der Coprozessor einen eigenen Chip darstellt, sind in aktuellen Implementierungen die Coprozessoren auf dem gleichen Chip wie die Hauptprozessoren verbaut. Beispiele dafür lassen sich sogar bei den in aktuellen FPGAs integrierten Prozessoren finden, wie etwa die Serie 7 von Xilinx FPGAs zeigt [13, Kapitel 3.2].

Eine weitere Form der Befehlssatzerweiterung stellt die Erweiterung über zusätzlich integrierte Funktionseinheiten dar. Diese Erweiterungsform entspricht dem Ansatz der Diplomarbeit und wird deshalb als einzige im Folgenden untersucht. Dabei handelt es sich um eine enge Kopplung mit dem Prozessor, wobei die Funktionseinheit parallel zur Ausführungseinheit im Datenpfad liegt [GB11]. Die wesentlichen Vorteile dieser Art der Anbindung ist der zu vernachlässigende Kommunikations-Overhead zwischen erweiterter Funktionseinheit und restlichem Prozessor, sowie die hohe erreichbare Kommunikationsbandbreite. Abbildung 2.1 zeigt dazu ein einfaches Beispiel mit einer Funktionseinheit als Erweiterung, welche wie die prozessorinterne Ausführungseinheit über zwei Eingänge und einen Ausgang verfügt.

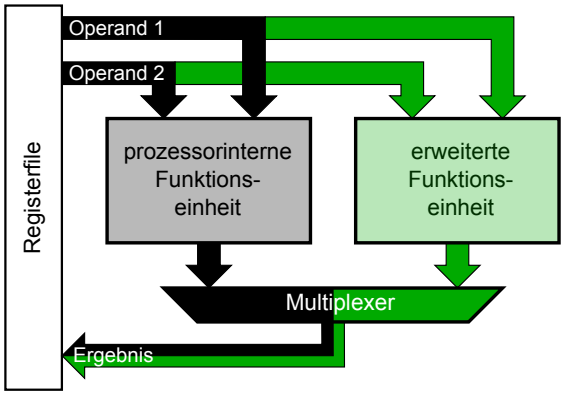

Abbildung 2.1: Datenpfad erweitert mit zusätzlicher Funktionseinheit

Dabei ist ersichtlich, dass in diesem Beispiel alle Operationen auf Prozessorregistern ausgeführt werden. Um Beispiele für diesen Aufbau zu geben werden im Folgenden ausgewählte existierende Architekturen behandelt, die über solche Funktionseinheiten erweitert wurden. Dabei liegt das Hauptaugenmerk der Betrachtungen auf der Rekonfiguration und Flexibilität dieser Erweiterungen, was auch der Zielsetzung dieser Arbeit entspricht.

2.2 Die PRISC-1 Architektur

Ein frühes Beispiel für die Befehlssatzerweiterung über eine zusätzliche rekonfigurierbare Funktionseinheit stellt die PRISC-1 Architektur dar [RS94], wobei PRISC für **PR**ogrammable **I**nstruction **S**et **C**omputer steht. Dieser Ansatz beinhaltet einige Kernpunkte der Diplomarbeit, allerdings gibt es auch manche Nachteile, die im Folgenden beleuchtet werden.

In der PRISC-1 Architektur wird ein universeller RISC Prozessor mit einer rekonfigurierbaren Funktionseinheit erweitert, welche als PFU (**P**rogrammable **F**unctional **U**nit) bezeichnet wird. Den Ausgangspunkt bildet ein MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) Prozessor, wie er auch in der Diplomarbeit verwendet wird. Der MIPS Prozessor ist ein 32 Bit RISC Prozessor, der 1984 an der Stanford Universität entwickelt wurde [RPJ+84] und in den Kapiteln 3.2 und 4.2 genauer behandelt wird. Die PFU ist ähnlich aufgebaut wie die rekonfigurierbare fine-grain Logik eines FPGAs, jedoch mit dem Ziel eine geringere Laufzeit zu besitzen als die normale, fest verdrahtete Ausführungseinheit des Prozessors. Durch die geringere Signallaufzeit wird gewährleistet, dass die erzielbare Taktfrequenz des gesamten Prozessorsystems in erster Näherung nicht negativ von der Erweiterung beeinflusst wird. Deshalb wurde ein einfacher Aufbau gewählt, welcher aus nur vier Logikebenen besteht, wobei jeweils Interconnection-Matrizen und

LUTs (**L**ook **U**p **T**able) abwechselnd angeordnet sind. Sowohl die Interconnection-Matrizen, als auch die LUTs sind zur Laufzeit rekonfigurierbar, was mehrere hundert Taktzyklen in Anspruch nimmt. Die Ermittlung und Generierung der passenden Konfigurationen für diese Erweiterungseinheit erfolgt bei der Programmerstellung. Dabei werden nur solche Funktionen generiert, die zum einen mit den physikalischen Ressourcen implementiert werden können und zum anderen unter Beachtung der Rekonfigurationszeit einen Geschwindigkeitsvorteil verschaffen. Die Konfigurationen der extrahierten Funktionen werden dann in die Datei des ausführbaren Programms eingebunden. Durch Einführung eines neuen Befehlsformats werden die Zusatzfunktionen schließlich aufgerufen, wobei die Funktion selbst über ein 11 Bit langes Feld im Befehl codiert wird. Somit stehen 2047 unterschiedliche Funktionen zur Verfügung. Sobald eine solche Zusatzfunktion ausgeführt werden soll, wird zuerst der 11 Bit lange Funktionscode überprüft und festgestellt, ob die richtige Konfiguration geladen ist. Ist das der Fall, dann beträgt die Ausführungszeit genau einen Taktzyklus. Andernfalls wird eine Ausnahmebehandlung durchgeführt und es vergehen einige hundert Taktzyklen, bis die richtige Konfiguration geladen ist und der Befehl ausgeführt werden kann. Während dieser Zeit ist der gesamte Prozessor blockiert.

Durch den besprochenen Aufbau der PRISC-1 Architektur ergeben sich jedoch einige Nachteile, welche im Folgenden aufgezählt und erörtert werden.

- **Beschränkung auf eine PFU**

  Im Gegensatz zu dem in dieser Diplomarbeit gewählten Ansatz wird nur eine PFU unterstützt, welche eine spezielle Funktion bereitstellt. Wird eine andere Funktion benötigt, muss die PFU jedes Mal umkonfiguriert werden.

- **enormer Konfigurations-Overhead**

  Da der Prozessor bei jeder Rekonfigurierung aufgrund der Logikgranularität mehrere hundert Taktzyklen angehalten werden muss, reduziert sich der erzielbare Leistungsgewinn erheblich. So kann es vor allem in Kombination mit dem vorherigen Punkt vorkommen, dass viele potentiell beschleunigende Funktionen nicht mehr rentabel sind und daher wegfallen.

- **Einschränkung der Verarbeitungszeit auf einen Taktzyklus**

  Die Hardware von PRISC-1 ist zwar nicht auf bestimmte Anwendungen beschränkt, allerdings können durch die geringe zur Verfügung stehende Verarbeitungszeit nur wenig komplexe logische Funktionen konfiguriert werden. Aus diesem Grund lässt der in Kapitel 3 gewählte Ansatz die Beschränkung auf nur einen Taktzyklus Verarbeitungszeit fallen, was folgende Auswirkungen hat:

  - Es ist eine zusätzliche Kommunikation mit den anderen Prozessoreinheiten erforderlich. Der Hardwareaufwand dafür ist allerdings relativ gering und die Ausführungszeit wird nicht negativ beeinflusst.
  - Die minimale Ausführungszeit der Erweiterungsbefehle wird von einem Taktzyklus auf drei erhöht, da beim gewählten Ansatz der in Abbildung 2.1 gezeigte Multiplexer in eine spätere Stufe der Prozessor-Pipeline verlegt wird.
  - Theoretisch kann jede beliebige Operation mit passender Anzahl an Ein- und Ausgängen mit einer PFU realisiert werden.
  - Durch die Verschiebung des Multiplexers wird die Anzahl an Logikebenen in der Ausführungseinheit um die des Multiplexers verringert. Wenn die Ausführungsstufe den kritischen Pfad für die Signallaufzeit der Prozessor-Pipeline darstellt, lässt sich dadurch eine etwas höhere Taktfrequenz erzielen.

- **vorgeschriebene Verwendung des Funktionscodes**

  Durch dedizierte Verwendung des Funktionscodes zur Funktionsselektion geht die Möglichkeit zur freien Verwendung dieser Bits bei der Kommunikation mit der PFU verloren. Deshalb werden die Funktionscodes beim Ansatz dieser Arbeit im Unterschied zur PRISC-1 Architektur direkt an den Erweiterungsfunktionsblock übergeben, wodurch sich folgende Konsequenzen ergeben:

  – Die Funktionscode-Bits können entweder zur Selektion einer bestimmten Konfiguration, als konstanter Operand oder als Konfiguration selbst interpretiert werden.

  – Bei Vorhandensein einer falschen PFU-Konfiguration kann keine automatische Ausnahmebehandlung durchgeführt werden. Deshalb ist es Aufgabe des Compilers bzw. der Erweiterungseinheit, jeweils die richtige Konfiguration zu laden.

2.3 Die OneChip Architektur

Eine weitere Arbeit auf dem Gebiet der rekonfigurierbaren Befehlssatzerweiterungen stellt die 'OneChip' Architektur der Universität von Toronto dar [WC96]. Hier wird ebenfalls ein MIPS-ähnlicher Prozessor mit zusätzlichen Funktionseinheiten erweitert. Im Vergleich zur PRISC-1 Architektur wird allerdings die Beschränkung auf nur einen Taktzyklus Verarbeitungszeit in den PFUs aufgehoben.

Das Resultat dieser Arbeit ist ein Prozessor, dessen interne Ausführungseinheit mit ein oder mehreren zusätzlichen konfigurierbaren Funktionseinheiten erweitert werden kann. Dabei ist der Prozessorkern vollständig mit dem Standardbefehlssatz kompatibel und als fest verdrahtete Implementierung vorgesehen, wodurch die Kompatibilität zu älteren Programmen und Compilern erhalten bleibt. Die FPGA-ähnliche rekonfigurierbare Logik ist im Design um den Prozessorkern angeordnet und verfügt über zwei zusätzliche Speicher, wovon einer als Statusspeicher für die Funktionseinheiten dient und der zweite Speicher vorcompilierte Images für Konfigurationen der PFUs enthält. Durch diesen Aufbau und der Komprimierung der Konfigurationen ist es laut [WC96] theoretisch möglich die für die rekonfigurierbare Logik erforderliche Chipfläche stark zu reduzieren. Da diese Logik den Großteil der benötigten Chipfläche des vorgeschlagenen Designs bildet, profitiert auch die Gesamtfläche erheblich von diesen Optimierungen.

Da der Prototyp jedoch auf einem Multi-FPGA-System realisiert wurde, konnte von den oben genannten Optimierungen zur FPGA-Logik und dem fest verdrahteten Prozessorkern nicht Gebrauch gemacht werden. Somit ist das System aufgrund von Hardwarelimitierungen auf eine einzige PFU beschränkt und kann nur mit 1,25 MHz betrieben werden. Neben den Einschränkungen des Prototypen hat das Konzept dieses Systems allerdings auch den Nachteil, dass die Konfigurierung der PFUs nur während des Startvorganges möglich ist und somit eine dynamische Rekonfigurierung wegfällt. Während des Betriebs kann daher lediglich per Multiplexer zwischen den bereits konfigurierten PFUs umgeschaltet werden.

Eine direkte Weiterentwicklung hierzu stellt 'OneChip-98' [JC99] dar. Hier wird eine Möglichkeit der dynamischen Rekonfigurierung zur Laufzeit vorgeschlagen, welche mit vorgeladenen Konfigurationen arbeitet. Dafür wird ein DPGA (**D**ynamically **P**rogrammable **G**ate **A**rray) verwendet, welches im Unterschied zum FPGA mehrere Konfigurationen parallel halten kann [TCE+95]. Dadurch besteht die Rekonfiguration aus einem Kontext-Switch. Dieser benötigt einen Taktzyklus und ist daher um viele Größenordnungen schneller als die Rekonfiguration eines FPGAs. Wird

nun ein Erweiterungsbefehl decodiert, bekommt ihn das DPGA zur Verfügung gestellt. Dieses schaltet dann bei Bedarf auf die entsprechende Konfiguration um und arbeitet den Befehl ab. Aufgrund der begrenzten Anzahl an vorgeladenen Konfigurationen, kann es jedoch vorkommen, dass die gewünschte Konfiguration nicht vorgeladen ist und somit erst geladen werden muss. Das kann per Compilerdirektive oder bei Bedarf erfolgen, wobei Letzteres allerdings eine erhebliche Befehlsverzögerung verursacht. Durch das Vorladen der neuen Konfiguration muss eine andere überschrieben werden (z. B. die längste nicht verwendete). Der Prozessor muss aber während dieses Vorganges im Gegensatz zur PRISC-1 Architektur nicht angehalten werden, da der rekonfigurierbare Teil das Laden der neuen Konfiguration selbst vornimmt.

Die zweite Weiterentwicklung von OneChip-98 stellt eine direkte Speicherschnittstelle dar, die es der PFU ermöglicht direkt auf Speicherinhalten zu operieren. Dadurch müssen diese nicht extra in das Registerfile des Prozessors geladen und anschließend wieder in den Speicher zurückgeschrieben werden. Das hat vor allem für die Verarbeitung von Datenblöcken wie z. B. bei der FFT (**F**ast **F**ourier **T**ransform) Vorteile und die PFU kann unabhängig vom Prozessorkern parallel zu diesem arbeiten.

In dieser Diplomarbeit wurde solch eine Speicheranbindung zwar auch in Betracht gezogen, jedoch weist diese Form der Erweiterung bereits Charakterzüge einer Coprozessoranbindung auf. Das ist zum einen nicht Ziel der Arbeit und zum anderen gibt es auch bereits eine Spezialisierung auf blockbasierte Verarbeitung vor. Neben dem zusätzlichen Hardwareaufwand für die Speicheranbindung kommt es bei diesem Ansatz im Fall von Datenabhängigkeiten zwischen Prozessor und PFU zwangsläufig zu längeren Bearbeitungszeiten, da dazwischen immer Speicherzugriffe erforderlich sind. Außerdem besteht auch ohne eine zusätzliche Speicheranbindung die Möglichkeit der Parallelverarbeitung von Prozessor und PFU, da der restliche Prozessor bereits parallel dazu andere Befehle bearbeiten kann (siehe Kapitel 3). Die einzige Einschränkung im Vergleich zur direkten Speicheranbindung ist daher nur die begrenzte Anzahl von Eingangsoperanden und Ergebnissen pro Erweiterungsbefehl.

Bezüglich der dynamischen Rekonfigurierung der PFUs zur Laufzeit können leider keine weiteren Aussagen getroffen werden, da aufgrund von Hardwarebeschränkungen der DPGA-Ansatz und seine Funktionalität nicht in den OneChip-98 Prototypen übernommen wurden. Somit wird diese Architektur auch wieder nur beim Hochfahren einmalig konfiguriert und der weitere Programmablauf muss sich auf eine PFU-Funktionalität beschränken.

2.4 Die ReRISC Architektur

Die ReRISC (**Re**configurable **RISC**) Architektur [VKTN06] stellt eine aktuelle Arbeit auf dem Gebiet der rekonfigurierbaren Befehlssatzerweiterungen dar. Als Basis wird auch hier wie bei OneChip-98 ein MIPS ähnlicher DLX (**D**e**L**u**X**e) Prozessor mit fünfstufiger Pipeline verwendet. Bezüglich der Befehlssatzerweiterung ist die Architektur ähnlich aufgebaut wie die PRISC-1 Architektur mit zwei signifikanten Unterschieden. Zum einen verwendet sie für die PFU im Gegensatz zu PRISC-1 coarse-grain Logik (siehe Kapitel 3.4.2) und zum anderen erstreckt sich die PFU über zwei Pipeline-Stufen des Prozessors. Ersteres hat den Vorteil, dass die maximale Rekonfigurationszeit wesentlich geringer ausfällt und die rekonfigurierbare Hardware effizienter arbeiten kann. Der zweite Unterschied sorgt dafür, dass auch komplexere Funktionen mit zwei Taktzyklen Verarbeitungszeit realisiert werden können. Zudem besteht die Möglichkeit, das Ergebnis sowohl nach der ersten, als auch nach der zweiten PFU-Stufe zurück in den prozessorinternen Datenpfad

zu routen. Komplexere Funktionen, die mehr als zwei Taktzyklen benötigen sind zwar möglich, jedoch muss dafür der Prozessor angehalten werden. Ein weiterer Unterschied von ReRISC zu den bereits behandelten Architekturen besteht in der Anzahl an Operandeneingängen der PFU, welche hier auf vier verdoppelt ist. Somit gibt es mehr Freiheitsgrade bei den möglichen Spezialbefehlen. Um gewünschte Funktionalität und Rekonfigurierbarkeit bereitstellen zu können, ist die PFU aus folgenden drei Ebenen aufgebaut:

- Prozessierungsebene
- Interconnection-Ebene
- Konfigurationsebene

Die Prozessierungsebene besteht aus einem Array von coarse-grain PEs (**P**rocessing **E**lement), die jeweils eine einfache logische oder arithmetische Operation mit zwei Eingängen und einem Ausgang ausführen können. Am Ausgang eines solchen PEs befindet sich ein Multiplexer, mit dem zwischen direkten und registerten Ausgang selektiert werden kann. Dadurch ist es möglich bei Auswahl des direkten Ausgangs mehrere PEs innerhalb eines Taktzyklus in Serie zu schalten, solange die Laufzeit des kritischen Prozessorpfads nicht überschritten wird. Die Interconnection-Ebene ist für die Verbindung der PEs untereinander und zum restlichen Prozessor verantwortlich. Damit bildet sie neben den Multiplexern in den PEs den Hauptbestandteil der Rekonfigurierbarkeit. Um die dynamische Rekonfiguration der PFU zur Laufzeit kümmert sich die Konfigurationsebene, wobei das Rekonfigurieren ähnlich funktioniert wie es bei der OneChip-98 Architektur vorgeschlagen wurde. In einem lokalen Speicher befindet sich eine begrenzte Anzahl an Konfigurationen, auf die sofort zurückgegriffen werden kann. Wenn ein PFU-Befehl ausgeführt wird, überprüft die Konfigurationsebene, ob die erforderliche Konfiguration vorhanden ist und wendet diese bei Vorhandensein gleich an. Andernfalls wird ein Miss-Flag gesetzt und die erforderliche Konfiguration muss in den lokalen Speicher geladen werden, was aufgrund der coarse-grain Logik nur wenige Taktzyklen benötigt.

Ein sehr großer Vorteil dieser Architektur ist das Vorhandensein einer vollautomatischen Entwicklungsumgebung, welche die rekonfigurierbare Logik vor dem Benutzer verbirgt. Das wird dadurch erreicht, dass die Erweiterungsbefehle selbstständig ermittelt werden und die dafür notwendige Hardware automatisch generiert wird. Somit hat die Erweiterung keinen Nachteil auf die Programmierung, während sie bei der Programmausführung für erhebliche Beschleunigungen sorgt. Die Bereitstellung einer solchen Entwicklungsumgebung ist jedoch nicht das Ziel der Diplomarbeit und daher für einen Vergleich der Architekturen zweitrangig.

Auch wenn die ReRISC-Architektur im Vergleich zu seinen Vorgängern erhebliche Verbesserungen aufweist und diese auch wirklich in den Prototypen integriert wurden, gibt es dennoch Einschränkungen in der Flexibilität der erweiterten Funktionseinheiten. Durch den Aufbau der Anbindung der PFU wird eine maximale Verarbeitungszeit von nur zwei Taktzyklen nahegelegt. Auch wenn theoretisch längere Zeiten erlaubt sind, reduziert sich der dadurch mögliche Leistungsgewinn aufgrund der Tatsache, dass der Prozessor dazu angehalten werden muss. Aus diesem Grund wird eine Parallelverarbeitung von Prozessor und PFU verhindert. Diese Einschränkung gilt nicht für den in Kapitel 3 gewählten Ansatz, welcher außerdem im Gegensatz zur hier behandelten Architektur mehrere physikalische PFUs unterstützt, um explizit von deren Parallelverarbeitung Gebrauch zu machen.

2.5 Vergleich der Architekturen

Um einen Überblick über die in diesem Kapitel besprochenen Architekturen zu erhalten, sind die wichtigsten Merkmale dieser Architekturen in Tabelle 2.1 aufgelistet. Ebenfalls aufgelistet ist der eigene Ansatz, welcher im folgenden Kapitel im Detail besprochen wird und zum Ziel hat, möglichst viele Vorteile der bisherigen Arbeiten zu vereinen. Ein besonders wichtiger Punkt dabei ist die Flexibilität der Erweiterungseinheiten, die von keiner der vorgestellten Architekturen erreicht wird. So ist es in den bisher diskutierten Arbeiten nicht vorgesehen echte Hardware einfach auszutauschen um sie durch neue zu ersetzen und damit die Ausstattung des Prozessors vor der Fertigung anzupassen.

Tabelle 2.1: Vergleich der präsentierten Architekturen und des eigenen Ansatzes

Architektur	**PRISC-1**	**OneChip**	**OneChip-98**	**ReRISC**	**eigener Ansatz**
Basisprozessor	MIPS R2000	MIPS R4400	DLX	DLX	MIPS R2000
kompatibel zu Standardbefehlssatz	ja	ja	ja	ja	ja
Anzahl PFUs	1	1 - ?*	1	1	1 - 4/8*
Taktzyklen PFU	1	1 -	1 -	1 - 2	3 -
Rekonfigurierbarkeit	zur Laufzeit	beim Einschalten	zur Laufzeit*	zur Laufzeit	zur Laufzeit**
Rekonfigurationszeit	viele Zyklen	-	kurz*,***	0/1***	0/1**
Prozessor stoppt bei Rekonfiguration	ja	-	nein	ja	nein**
Logikgranularität	fein	fein	fein	grob	grob**
PFU-Datenanbindung	Registerfile 2 Eingänge	Registerfile 2 Eingänge	Speicher	Registerfile 4 Eingänge	Registerfile 2 Eingänge

*nicht im Prototyp
**Implementierungsabhängig
***nur wenn Konfiguration lokal vorhanden

3 Gewählter Ansatz

In diesem Kapitel wird der eigene Lösungsansatz beschrieben. Dafür wird zuerst die Problembeschreibung erörtert, bevor auf die Details der Lösung eingegangen wird. Außerdem wird auch theoretischer Hintergrund behandelt, um konkrete Entscheidungen besser nachvollzielbar zu machen.

3.1 Zielsetzung

Wie in der Einleitung Kapitel 1.2 bereits kurz umrissen, soll in einem Top-Down Entwurf ein RISC Prozessor mit rekonfigurierbaren Befehlssatzerweiterungen entwickelt werden. Dabei soll der Standardbefehlssatz des Prozessors unverändert bleiben, um mit vorhandenen Programmen und Compilern kompatibel zu sein. Die Erweiterungen sollen eng mit dem Prozessor als zusätzliche Funktionseinheiten gekoppelt sein um eine schnelle und effektive Kommunikation zu ermöglichen. Dazu werden die Erweiterungseinheiten parallel in den Datenpfad des Prozessors integriert.

Um eine Applikationsunabhängigkeit zu schaffen und auch zukünftige Befehlssatzerweiterungen zu ermöglichen, sollen beliebige Erweiterungen unterstützt werden. Dadurch wird es später möglich sein, den Prozessor vor der Fertigung auf die Anforderungen der späteren Applikation gezielt abzustimmen. Dies garantiert, dass applikations- oder applikationsgebietspezifische Erweiterungen, sowie Universalerweiterungen je nach Anforderung beliebig zusammengestellt werden können. Somit kann der Prozessor auf verschiedenste Weise optimiert werden wie z. B. auf Chipfläche, Rechenleistung, universeller Einsatzfähigkeit oder Energieverbrauch. Da diese Optimierungsziele sehr unterschiedlich sind, wird auch der dazu erforderliche Hardwareaufbau der Spezialerweiterungen sehr unterschiedlich sein. Deshalb ist es Teil der Zielsetzung, dass theoretisch jede Form von fine- und coarse-grain Logik, sowie fest verdrahteter Logik unterstützt werden soll.

Weiters wird gefordert, den Prozessor mit mehreren Funktionseinheiten erweitern zu können, um etwa statt einer universell konfigurierbaren Einheit mehrere, dafür aber spezialisiertere zu verwenden. Spezialisiertere Logik ist nicht nur schneller und energieeffizienter, sondern benötigt auch weniger Chipfläche als universell konfigurierbare Logik [VS07, Kapitel 2.3]. Außerdem kann so bei besonders hohen Anforderungen an die Rechenleistung die Parallelität der Einheiten ausgenutzt werden, wenn diese längere Operationen ausführen. Das setzt voraus, dass längere Verarbeitungszeiten in den Erweiterungsblöcken auch unterstützt werden. Dafür muss ein Mechanismus zur Synchronisation mit den anderen Prozessoreinheiten entwickelt werden, um keine Fehler durch

Ressourcenkonflikte zu verursachen. Dieses Problem würde ohne Erweiterungen nicht auftreten, weil die Prozessor-Pipeline mit ihrem statischen Ablauf die Ressourcenzugriffe eindeutig vorgibt.

Die eben genannten Anforderungen sollen nach Entwicklung des Konzepts in einen Prototypen implementiert werden. Ein schematischer Aufbau solch eines Systems ist in Kapitel 1.2 Abbildung 1.3 dargestellt. Dabei ist es das Ziel neben dem Prozessor zwei rekonfigurierbare coarse-grain Erweiterungseinheiten zu entwickeln, welche komplexe Operationen ausführen und über wenige Bits zu konfigurieren sind. Diese Erweiterungen sind ein CORDIC-Algorithmus (**CO**ordinate **R**otation **DI**gital **C**omputer) in Hardware und ein digitales Filter. Außerdem soll eine dritte Erweiterungseinheit implementiert werden, die mit der Pipeline des Prozessors synchronisiert ist. Diese Erweiterung soll Konvertierungsfunktionen zwischen dem IEEE754 Gleitkommaformat [IEE08] und dem Zweierkomplement Fixkommaformat bereitstellen. Durch die Synchronisation mit der Prozessor-Pipeline ergeben sich mögliche Vorteile, von denen für die Anbindung der Erweiterungseinheit Gebrauch gemacht werden soll. So kann zum einen der Kommunikations-Overhead reduziert werden und zum anderen werden Ressourcenkonflikte ausgeschlossen. Die detaillierte Implementierung dieser Einheiten ist in Kapitel 4 zu finden.

Natürlich sind manuelle Implementierungen von Erweiterungseinheiten und deren Programmcodeanpassung für die Praxis unpraktikabel. Der Idealfall wäre eine vollautomatische Entwicklungsumgebung, in der der Code einer Hochsprache zuerst analysiert wird und unter optimaler Ausnutzung der Hardware ein Maschinenprogramm erstellt wird. Dabei sollten auch die Konfigurationen der Erweiterungseinheiten mit den zugehörigen Maschinenbefehlen automatisch erzeugt werden. Sofern die Zielhardware noch nicht existiert, sollte die Entwicklungsumgebung außerdem in der Lage sein entsprechende Erweiterungskomponenten selbst zu generieren bzw. sie aus einer vorhandenen Bibliothek auszuwählen. Nach Erstellung der Hardware oder bei vorgegebener Hardware sollten die Befehlssatzerweiterungen gewinnbringend in den Maschinencode eingearbeitet werden, ohne dass der Benutzer eingreifen muss.

3.2 Verwendeter RISC Prozessor

Um den gewählten Ansatz der Implementierung zu präsentieren ist Kenntnis über den verwendeten Prozessor unerlässlich, welcher daher im Folgenden beschrieben wird.

Die Basis des Designs bildet ein RISC Prozessor, wobei ein RISC folgende Eigenschaften aufweist [Jam95, BMS13]:

- Load/Store Architektur
- Operationen werden auf Registern ausgeführt
- wenige, einfache Befehle konstanter Länge
- kurze und einfach gestaltete Pipeline, meist ein Taktzyklus pro Befehl

Der verwendete MIPS R2000 [KH92, Kapitel 2] ähnliche Prozessor besitzt eine fünfstufige Pipeline, welche in Abbildung 3.1 vereinfacht dargestellt ist. In der ersten Stufe, der Fetch-Stufe wird ein neuer Befehl vom Programmspeicher geladen. Dieser Befehl wird in der Decode-Stufe decodiert und die benötigten Operanden werden aus den Registern geladen. Die Execute-Stufe ist für

die eigentliche Befehlsabarbeitung verantwortlich. Hier werden sämtliche logische und arithmetische Operationen, sowie Speicheradressberechnungen ausgeführt. In der Memory-Access-Stufe wird im Falle von Lade- oder Speicherbefehlen auf den Datenspeicher zugegriffen. Das Ergebnis der Execute-Stufe bzw. der geladene Speicherinhalt des Datenspeichers wird in der letzten Stufe, der Write-Back-Stufe in das Ergebnisregister zurückgeschrieben.

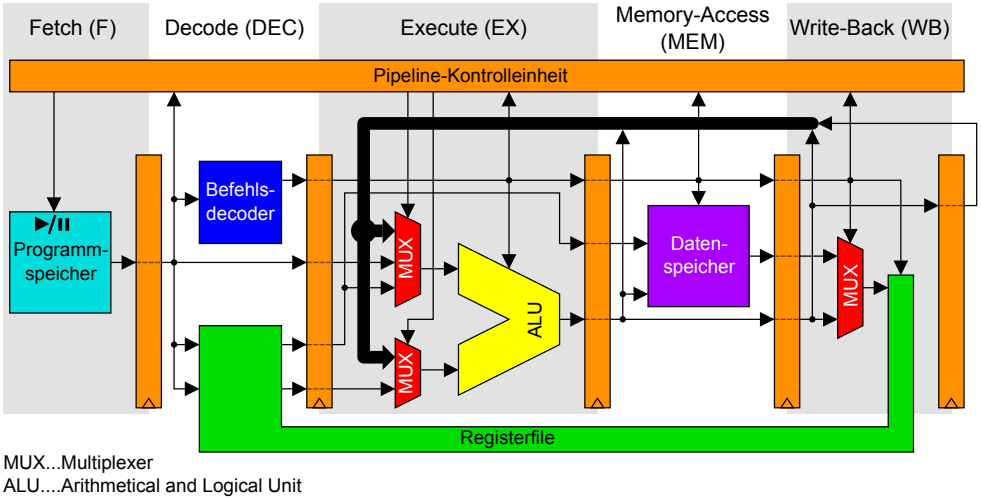

Abbildung 3.1: Aufbau der MIPS Pipeline

Da das Ergebnisregister aufgrund des Pipeline-Aufbaus erst nach drei Taktzyklen den korrekten Wert enthält, kann es bei zu früher Weiterverarbeitung dieses Wertes zu Datenhazards kommen. Deshalb kümmert sich eine Kontrolleinheit um Datenabhängigkeiten und führt bei Bedarf ein noch nicht in das Registerfile zurückgeschriebenes Ergebnis per Multiplexer an den Eingang der Ausführungseinheit zurück. Im Falle von Ladebefehlen ist das nur bedingt möglich, da das geladene Datum erst in der letzten Pipeline-Stufe zur Verfügung steht, sodass die Pipeline angehalten werden muss, wenn zu früh auf dieses Datum zugegriffen wird. Auch darum kümmert sich die Kontrolleinheit.

Die Ausführungseinheit führt 32 Bit Operationen mit maximal zwei Eingangsoperanden und einem Ergebnis durch. Daher liegt es nahe, dass auch die erweiterten Funktionseinheiten mit dieser Anzahl an Operanden arbeiten, wodurch hier kein zusätzlicher Hardwareaufwand erforderlich ist. Um den Prozessor optimal an die Erweiterungseinheiten anzupassen und aufgrund des relativ einfachen Aufbaus wurde beschlossen, diesen selbst zu implementieren. Eine andere Möglichkeit wäre gewesen, auf vorhandene freie Lizenzen zurückzugreifen. Aufgrund der engen Kopplung von Prozessor und erweiterten Funktionseinheiten muss dabei aber tief in die Pipeline und deren Kontrolleinheit eingegriffen werden. Der daraus entstandene Aufwand wäre vergleichbar mit der durchgeführten spezifischen Implementierung. Die verbleibenden unbedingt benötigten Prozessoreinheiten wie Registerfile und Speicher stellen Standardeinheiten dar und sind daher in vereinfachter Form mit geringem Aufwand zu implementieren.

Der Standardbefehlssatz eines MIPS Prozessors wird durch den implementierten Prozessor eingehalten, mit der Ausnahme, dass einzelne Befehle (z. B. Division) nicht unterstützt werden. Diese sind für das Konzept der rekonfigurierbaren Befehlssatzerweiterungen unwesentlich, hätten aber einen erhöhten Entwicklungsaufwand erfordert und eventuell sogar zu Platzproblemen im verwendeten FPGA geführt. Der Befehlssatz verfügt über drei verschiedene Formate [KH92, Seite 3-1], deren Aufbau in Tabelle 3.1 dargestellt ist. In der ersten Zeile der Tabelle sind die Bitpositionen

im Befehlswort angegeben.

Tabelle 3.1: MIPS Befehlsformate

	31-26	**25-21**	**20-16**	**15-11**	**10-6**	**5-0**
R-Format	Opcode (000000)	Rs	Rt	Rd	SA	Function
I-Format	Opcode	Rs	Rt	Immediate		
J-Format	Opcode	Target				

Im R-Format sind alle Operationen codiert, die nur auf und mit Registern arbeiten. Der Operationscode, kurz Opcode ist dabei immer '000000' und die auszuführende Operation wird im Feld 'Function' codiert. Die beiden Eingangsoperanden sind die Registerinhalte der mit Rs und Rt codierten Register und das Ergebnis wird in Register Rt zurückgeschrieben. Durch diese Codierung können genau 32 verschiedene Register angesprochen werden, was auch der tatsächlichen Anzahl im Prozessor entspricht. Das Feld 'SA' (**S**hift **A**mount) wird nur für fixe Schiebebefehle benötigt und gibt an, um wie viele Bits ein Datum geschoben werden soll.

Befehle, die mit Konstanten arbeiten sind im I-Format codiert. Dazu gehören sowohl Lade- und Speicherbefehle, deren Adresse aus der übergebenen Konstanten plus einem Registerinhalt berechnet wird, als auch sämtliche Operationen, die eine Konstante als Operand haben. Die Konstante, in der Tabelle als 'Immediate' bezeichnet, wird dabei als 16 Bit vorzeichenbehaftete Zahl im Zweierkomplement interpretiert. Das mit Rs codierte Register enthält den zweiten Eingangsoperanden und das Ergebnis bzw. das geladene Datum wird in das Register Rt geschrieben.

Als drittes Befehlsformat gilt das J-Format, welches für Sprünge zu einer fixen Programmadresse, dem 'Target', verwendet wird. Da der Adressbereich des Prozessors 32 Bit beträgt und die Sprungadresse im Befehl nur 26 Bit lang ist, werden die oberen 6 Bit des Programmzählers für die Zieladresse verwendet.

Natürlich ergeben nicht alle Bitkombinationen gültige Maschinenbefehle, so sind z. B. auch im Opcode noch Kombinationen frei, welche beim MIPS für die Ansteuerung von Coprozessoren verwendet werden und in dieser Diplomarbeit zum Ansprechen der Erweiterungseinheiten dienen (siehe Kapitel 3.3.5).

3.3 Erweiterungskonzept

Das Erweiterungskonzept bestimmt die Anbindung der erweiterten Funktionseinheiten an den Basisprozessor und bildet den Kernpunkt dieser Arbeit. Mit diesem Konzept zur Erweiterung werden die Möglichkeiten im Design der Erweiterungskomponenten vorgegeben und der erzielbare Rechenleistungsgewinn, sowie die Erweiterbarkeit sind davon abhängig. Um die gewünschten Anforderungen zu erfüllen, wird eine generische Schnittstelle zwischen dem Prozessor und den Erweiterungseinheiten definiert. Dadurch wird die universelle Erweiterbarkeit und der einfache Austausch von Erweiterungskomponenten ermöglicht. Die Schnittstelle verfügt dabei über separate Leitungen für die beiden Kommunikationsrichtungen vom Prozessor zu den Erweiterungseinheiten und umgekehrt. In den folgenden Unterkapiteln werden diese Kommunikationsrichtungen im Detail vorgestellt und das Kommunikationsprotokoll spezifiziert.

3.3.1 Schnittstelle vom Prozessor zu Erweiterungseinheiten

Für die Kommunikation zu den Erweiterungsfunktionseinheiten ist eine einfache, gemeinsam genutzte Ausführung der meisten Steuer- und Datenleitungen ausreichend. Zwei Steuerleitungen müssen allerdings für jede Erweiterung gesondert vorhanden sein, um jeweils den richtigen Block anzusprechen. Der Aufbau der Schnittstelle ist in Abbildung 3.2 dargestellt. Die zwei 32 Bit Ausgangsoperanden der Schnittstelle sind äquivalent zu den Operanden der R-Befehle im Prozessorstandardbefehlssatz und werden direkt vom Registerfile bereitgestellt. Sie stellen die Eingangsoperanden für die Spezialerweiterungen dar. Ein weiterer Ausgangsparameter der Schnittstelle ist die Zielregisteradresse des Spezialbefehls. Diese ist erforderlich, da der Prozessor aufgrund der Allgemeinheit des Erweiterungskonzepts keine Information über die Ausführungsdauer einer Spezialoperation hat und es zudem sein kann, dass eine Erweiterungseinheit während der Abarbeitung eines Befehls weitere Befehle und damit auch Ergebnisregisteradressen aufnehmen kann. Durch die Bereitstellung der Zielregisteradresse als Ausgangsparameter der Schnittstelle muss sich der Prozessor nicht um die Zwischenspeicherung dieser Adressen kümmern und den Erweiterungseinheiten sind in dieser Hinsicht alle Möglichkeiten offen.

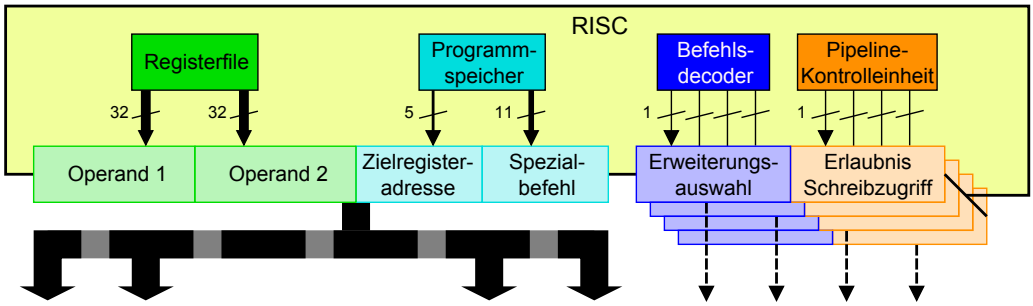

Abbildung 3.2: Schnittstelle vom Prozessor zu Spezialhardwareerweiterungen

Als gemeinsam genutzte Steuer- bzw. Datenleitungen wird ein 11 Bit breites Stück des aktuellen Maschinenbefehls ausgegeben. Dieser Teilbefehl wird im Folgenden als 'Spezialbefehl' bezeichnet. Hier sind zweierlei Besonderheiten gegeben. Zum einen wird dieser Teil des Befehls direkt aus dem Programmspeicher, also von der Fetch-Pipeline-Stufe gelesen. Zum anderen wird nicht vorgegeben, wie diese 11 Bit zu interpretieren sind. Aufgrund der ersten Besonderheit kann ein Erweiterungsblock vorbereitende Arbeiten für den kommenden Befehl während der Decodierungsstufe des Prozessors durchzuführen. Solche vorbereitende Arbeiten können etwa Zugriffe auf lokale Speicher, Laden von Konfigurationen, usw. sein. Somit können unter Umständen Befehlsabarbeitungen in den Erweiterungen beschleunigt werden. Durch die Interpretationsfreiheit des Spezialbefehls kann dieser von den Erweiterungen entweder als Funktionscode, als Operand oder als Konfiguration aufgefasst werden. Dabei ist auch eine Kombination dieser drei Varianten möglich.

Wie in Abbildung 3.2 dargestellt, werden für jedes Erweiterungsmodul zwei Steuerleitungen realisiert. Eine Steuerleitung stellt ein Auswahlsignal zur Verfügung, mit dem der gewünschte Funktionsblock angesprochen wird. Diese Steuerinformation ist jedoch erst einen Taktzyklus nach dem Spezialbefehl verfügbar, da der Befehl im Prozessor erst als Erweiterungsbefehl decodiert werden muss. Im Anschluss daran kann dann die korrekte Erweiterung ausgewählt werden. Das muss von den Spezialfunktionseinheiten berücksichtigt werden, um nicht zu früh falsche Operationen durchzuführen.

Da Erweiterungsbefehle verschieden lange Ausführungszeiten haben können und aufgrund maximaler Rechenleistung parallele Befehlsausführung zugelassen wird, kann es vorkommen, dass mehrere Erweiterungseinheiten gleichzeitig ein Register beschreiben wollen. Weil jedoch das Registerfile nur einen Schreibzugriff pro Taktzyklus erlaubt, stellt dieser Fall einen Ressourcenkonflikt dar und muss geregelt werden. Einen Teil dieser Zugriffsregelung bildet die zweite Steuerleitung, die einer Erweiterungseinheit signalisiert, wenn sie Schreibzugriff auf das Registerfile bekommt. Es wäre auch möglich das Registerfile auf die maximale Anzahl an Schreibzugriffen auszulegen. Allerdings würde dadurch der Hardwareaufwand erheblich steigen und Ressourcenkonflikte könnten trotzdem noch auftreten, wenn etwa mehrere Funktionseinheiten zur gleichen Zeit den gleichen Registerinhalt beschreiben wollten. Diese Konflikte müssten dann ebenfalls gelöst werden. Da dieser übermäßige zusätzliche Aufwand die dadurch gewonnene Leistungssteigerung vermutlich nicht rechtfertigen würde, wurde der Ansatz der Zugriffssteuerung gewählt. Dieser wird in Kapitel 3.3.4 noch im Detail behandelt.

3.3.2 Schnittstelle von Erweiterungseinheiten zum Prozessor

Für die Kommunikation von den Erweiterungseinheiten zum Prozessor sind für die maximal unterstützte Anzahl an Erweiterungen jeweils separate Steuer- und Datenleitungen vorgesehen, um eine schnelle und kollisionsfreie Kommunikation zu gewährleisten. Der Aufbau dieses Teils der Schnittstelle ist in Abbildung 3.3 dargestellt. Über den 32 Bit breiten Ergebniseingang gelangt das von der jeweiligen Erweiterungseinheit produzierte Ergebnis wieder zurück in den Prozessor. Zusätzlich beinhaltet die Eingangsschnittstelle die Adresse des Registers, in der das Ergebnis geschrieben werden soll. Dieser Registeradresseingang ist 5 Bit breit und entspricht somit dem Registeradressbereich des Prozessors.

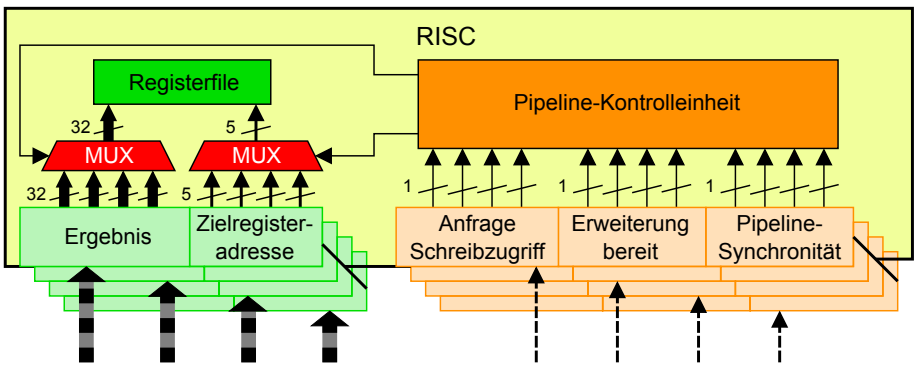

Abbildung 3.3: Schnittstelle von Spezialhardwareerweiterungen zum Prozessor

Die restlichen drei Leitungen der Schnittstelle dienen als Steuerleitungen. Eine davon ist für die Beantragung des Schreibzugriffs auf das Registerfile zuständig, was für die Zugriffskontrolle erforderlich ist. Die zweite Steuerleitung signalisiert dem Prozessor die Bereitschaft der Erweiterungseinheit zur Aufnahme eines Befehls. Dies bedeutet, dass sie in der Lage ist einen neuen Befehl aufzunehmen. Da die generische Schnittstelle zwei Operationsmodi anhand der Befehlsverarbeitungsdauer einer Erweiterungseinheit unterscheidet, muss diese Information über die letzte Steuerleitung bereitgestellt werden. Dabei gibt ein gesetzter Zustand der Leitung an, dass die am entsprechenden Schnittstellenport befindliche Erweiterung mit der Prozessor-Pipeline synchronisiert ist. Als synchronisiert wird dabei eine gleiche Länge der Datenpfade von Erweiterungseinheit und den dazu parallelen Prozessoreinheiten verstanden. Falls diese Synchronisationsbedingung

erfüllt ist, kann auf jede weitere Steuerkommunikation verzichtet und das Ergebnis ohne Zugriffssteuerung in das Zielregister geschrieben werden. Dieser Vorgang benötigt um einen Taktzyklus weniger als die vollständige Schnittstellenkommunikation, wodurch das Ergebnis schneller verfügbar wird. Es ist auch ein Wechsel von Pipeline-Synchronisation auf beliebige Ausführungsdauer und umgekehrt möglich, jedoch muss der Erweiterungsblock dazu mindestens einen Taktzyklus davor Inaktivität signalisieren. Nur so kann gewährleistet werden, dass während des Wechsels kein Befehl mit falschem Kommunikationsschema an die Erweiterungseinheit gesendet wird. Somit kann auch hinsichtlich der Bearbeitungszeit von Erweiterungsbefehlen beliebig rekonfiguriert werden.

3.3.3 Kommunikationsablauf

Der Kommunikationsablauf gestaltet sich aufgrund der engen Kopplung zwischen Prozessor und Erweiterungen sehr einfach und ist in Abbildung 3.4 dargestellt. Als Allererstes muss eine erweiterte Funktionseinheit signalisieren, ob sie mit der Prozessor-Pipeline synchron läuft oder nicht. Frühestens zeitgleich zu dieser Information darf die Einheit Bereitschaft melden. Nun muss die Erweiterung warten, bis sie vom Prozessor über die Schnittstelle angesprochen wird. Während dieser Zeit wird ständig das 11 Bit lange Stück aller bearbeiteten Maschinenbefehle an die Einheiten gesendet, welche diese Eingangsinformationen ignorieren müssen. Erst wenn vom Prozessor das Auswahlsignal für die entsprechende Erweiterung gesetzt wird, bedeutet das, dass der zuvor empfangene Spezialbefehl gültig ist. Gleichzeitig werden auch die zwei Eingangsoperanden, welche auch Konfigurationsinformationen enthalten können, von der generischen Schnittstelle bereitgestellt.

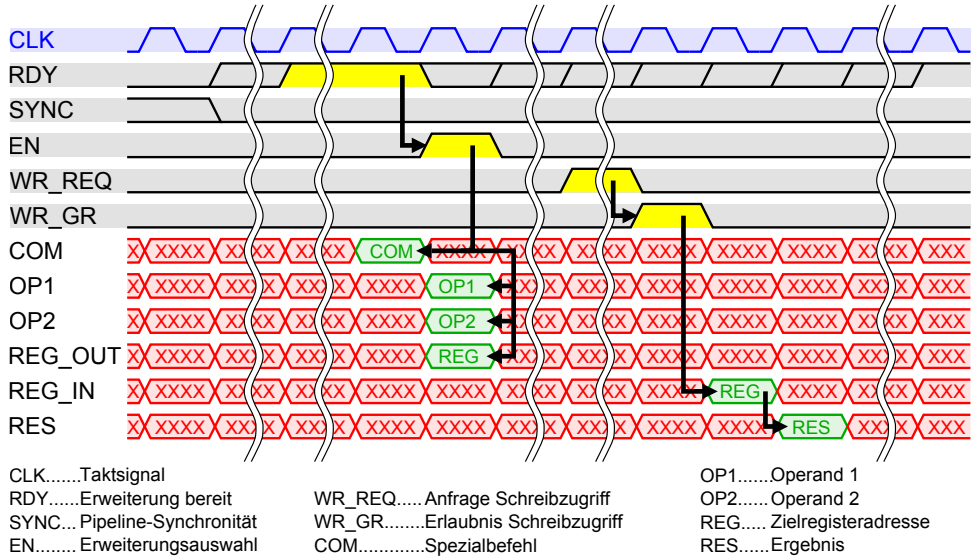

Abbildung 3.4: Kommunikationsablauf zwischen Prozessor und zusätzlicher Funktionseinheit

Ein weiteres mögliches Szenario ergibt sich wenn der Prozessor einen Spezialbefehl geladen hat, jedoch die entsprechende Erweiterung nicht bereit ist. In diesem Fall muss der Prozessor auf die Erweiterung warten, bis diese Bereitschaft signalisiert. Derartige Situationen sollten allerdings vom Compiler bzw. Programmierer vermieden werden, da die Erweiterungsblöcke die Programmabarbeitung beschleunigen sollen, anstatt sie zu behindern.

Für den nächsten Schritt ist entscheidend, ob die Erweiterungseinheit zur Prozessor-Pipeline synchron läuft oder nicht. Besteht Synchronität zum Prozessor, erübrigt sich jede weitere Kommunikation und der Prozessor erwartet nach einer Pipeline-Stufe die Zielregisteradresse, gefolgt vom Operationsergebnis nach der zweiten Stufe. Wenn aus irgendeinem Grund wie z. B. bei einem Rekonfigurationsbefehl kein Ergebnis produziert wird, muss sichergestellt werden, dass kein gültiger Registerinhalt fälschlicherweise überschrieben wird. Das ist Aufgabe der Erweiterung, welche am besten das Register null mit dem Wert null überschreibt. Aufgrund der Tatsache dass dieses Register beim verwendeten Prozessor per Definition den Wert null enthält [KH92, Seite 2-6], ist gewährleistet, dass nichts überschrieben wird. Der Wert null ist jedoch wichtig, da das Register in der Beispielimplementierung im Gegensatz zum Originalprozessor nicht fest auf null verdrahtet ist.

Ist die Erweiterungseinheit nicht mit der Prozessor-Pipeline synchronisiert wie in Abbildung 3.4, muss sie vor Fertigstellung einer Operation Schreibzugriff auf das Registerfile beantragen. Wenn keine Ressourcenkonflikte auftreten, wird dieser auch gewährt. Erfolgt keine Freigabe, so ist es die Aufgabe der Erweiterungseinheit weiter den Zugriff zu beantragen, sofern noch Bedarf besteht. Diese Prozedur wiederholt sich so lange, bis entweder der Schreibzugriff aus irgend einem Grund irrelevant geworden ist oder die Freigabe erteilt wird. Danach muss die Erweiterung im nächsten Taktzyklus die Adresse des Zielregisters und spätestens nach der zweiten steigenden Taktflanke das Operationsergebnis bereitstellen. Das heißt, die minimal mögliche Zeit für eine Operation ist um einen Taktzyklus länger als im synchronen Fall. Das ist auch der Hauptgrund für die Einführung und Unterstützung des synchronen Falls. Somit sind ab einer Verarbeitungszeit von zwei Taktzyklen beliebige Ausführungszeiten möglich. Zu beachten ist allerdings, dass bei Datenabhängigkeiten die Rückführung des Ergebnisses zurück zur Ausführungsstufe immer noch einen weiteren Taktzyklus in Anspruch nimmt. Deshalb ist die frühestmögliche Weiterverarbeitung des Resultats eines Spezialbefehls erst nach drei Taktzyklen, also zwei Befehlen dazwischen möglich. Bestehen noch kürzere Abhängigkeiten, werden von der Pipeline-Kontrolleinheit im synchronen Fall automatisch Leerbefehle eingefügt. Im nicht synchronen Fall ist das automatische Einfügen von Leerbefehlen nicht möglich, weshalb bei Datenabhängigkeiten gewartet werden muss, bis das Ergebnis vorhanden ist. Erfolgt ein verfrühter Lesezugriff auf das Ergebnisregister, wird auch keine Ausnahmebehandlung oder Ähnliches durchgeführt und die Aufgabe solche Fehler zu verhindern wird wieder der Programmerstellung übertragen.

Sofern im nicht synchronen Fall kein Schreibzugriff seitens der Erweiterung erforderlich ist, wird dieser auch nicht beantragt. Somit wird auch nichts in das Registerfile geschrieben. Der Prozessor wartet also nicht explizit bei Spezialbefehlen auf einen Schreibzugriff. Dadurch ergeben sich folgende zusätzliche Möglichkeiten:

- **Operationen mit beliebig vielen Eingangsoperanden**

  Mächtige Spezialoperationen mit mehreren Eingangsoperanden können über eine Folge von Spezialbefehlen codiert werden, mit denen jeweils bis zu zwei Operanden übergeben werden.

- **lokale Registerspeicherung in Erweiterungseinheiten**

  Sofern ein Spezialbefehl ein temporäres Zwischenergebnis oder mehrere Ergebnisse produziert, können diese in der Erweiterungseinheit zwischengespeichert werden, um den Overhead für das Speichern in das prozessorinterne Registerfile einzusparen bzw. die anderen Ergebnisse nicht zu verlieren. Über weitere Spezialbefehle können diese Ergebnisse im Anschluss weiter verarbeitet oder nacheinander ausgelesen werden.

- **Verwendung von Erweiterungen als zusätzliche Prozessorports**

  Durch die Entkopplung der beiden Kommunikationsrichtungen der generischen Schnittstelle wird es den Erweiterungseinheiten ermöglicht, auch als E/A-Einheiten (**E**in-/**A**usgabe) mit integrierter Verarbeitung zu agieren. So kann die Einheit als Reaktion auf bestimmte Änderungen der E/A-Pins direkt in das Registerfile des Prozessors schreiben und zusätzlich einen Interrupt auslösen. Umgekehrt kann der Prozessor zu beliebigen Zeitpunkten Daten an die E/A-Einheit für die Weiterverarbeitung oder direkte Ausgabe senden.

3.3.4 Lösen von Ressourcenkonflikten

Aufgrund der möglichen parallelen Befehlsabarbeitung durch die zusätzlichen Funktionseinheiten und deren unterschiedliche Ausführungszeiten kann es wie bereits erwähnt zu Zugriffskonflikten auf das Registerfile kommen. Das passiert, wenn zwei Operationsergebnisse zum gleichen Zeitpunkt fertig werden. In diesem Fall ist eindeutig geregelt, wie die Vergabe des Schreibzugriffs erfolgt.

Da davon ausgegangen wird, dass in den erweiternden Funktionseinheiten komplexere Befehle als im Prozessor selbst ausgeführt werden, haben immer die Hardwareerweiterungen Vorrang. Das bedeutet somit, dass die prozessorinterne Befehlsausführung immer unterbrochen wird, wenn sowohl die interne Ausführungseinheit als auch eine Erweiterungseinheit Ergebnisse schreiben wollen. Die weitere Selektion der Zugriffsvergabe ist ähnlich einfach und effizient gelöst. Alle Anschlüsse für Erweiterungskomponenten sind mit einer Nummer von null bis zur maximalen Anzahl an Erweiterungen minus eins versehen. Dabei hat die Erweiterung mit der Nummer null die höchste Priorität und die mit der größten Nummer die niedrigste. Somit bekommt bei gleichzeitigem Zugriffsantrag mehrerer Erweiterungen immer die mit der niedrigsten Nummer den Zugriff auf das Registerfile. Bei mehr als zwei gleichzeitigen Zugriffen ist das Prinzip dasselbe, nur dass im nächsten Taktzyklus der Schreibzugriff von den verbleibenden Einheiten erneut beantragt werden muss. Dieses Prinzip wird so lange ausgeführt, bis kein Ressourcenkonflikt mehr besteht und die prozessorinterne Ausführungseinheit wieder den Zugriff bekommt. Im schlimmsten Fall wird daher die Prozessorausführung für mehrere Taktzyklen unterbrochen. Um diesem negativen Effekt entgegenzuwirken gibt es zwei Möglichkeiten:

- Bei Sprung- oder Datentransferbefehlen zum Speicher benötigt der Prozessor keinen Schreibzugriff auf das Registerfile und muss somit nicht angehalten werden. Daher können bei Kenntnis der Ausführungszeiten der Erweiterungseinheiten und geschickter Anordnung der Maschinenbefehle solche Wartezyklen teilweise verhindert werden.
- Die Erweiterungen können so entworfen werden, dass diese synchron zur Prozessor-Pipeline arbeiten. Das ist jedoch Aufgrund der Komplexität der gewünschten Spezialbefehle nicht immer möglich.

Ein großer Vorteil der vorgestellten Zugriffsvergabe ist die Tatsache, dass der sofortige Schreibzugriff der Erweiterungseinheit mit der höchsten Priorität immer garantiert ist. Somit können an diese Erweiterungsschnittstelle mit höchster zugeordneter Zugriffspriorität Module angebunden werden, welche über eine eigene Pipeline verfügen. Es ist dabei ausgeschlossen, dass die Pipeline wegen Zugriffsproblemen angehalten werden muss und die Erweiterungseinheit ist daher immer in der Lage einen neuen Befehl pro Taktzyklus aufzunehmen (siehe CORDIC-Implementierung Kapitel 4.3). Deshalb muss sich die Erweiterung weder um das Anhalten der Pipeline kümmern, noch um das Steuersignal, welches anzeigt dass die Erweiterung bereit ist.

3.3.5 Erweiterungsbefehle

Für die Befehlssatzerweiterung wird der Standardbefehlssatz des MIPS Prozessors um ein weiteres Befehlsformat ergänzt. Dieses Befehlsformat ist eine Kombination aus dem R-Format und den Coprozessoropcodes [KH92, Seite 3-14] des MIPS Prozessors die dadurch ersetzt werden. Weil davon ausgegangen wird, dass die Spezialhardwareerweiterungen die Funktionalität von Coprozessoren übernehmen können, stellt dies in der Regel kein Problem bezüglich der Funktionalität dar. Somit wirkt sich das Weiterverwenden der Coprozessoropcodes positiv auf die Anzahl möglicher Erweiterungen aus. Prinzipiell wäre jedoch auch eine Koexistenz der beiden Erweiterungstypen möglich, dazu müssten nur die entsprechenden Steuerleitungen des konkurrierenden Erweiterungsanschlusses entsprechend gesetzt werden um für den Prozessor transparent zu sein. Da jedoch der implementierte Prototyp aus Kapitel 4.2 eine Unterstützung für Coprozessoren nicht vorsieht, wird auf dieses Thema nicht näher eingegangen.

Tabelle 3.2: Befehlsformate für zusätzliche Befehle

	31-26	**25-21**	**20-16**	**15-11**	**10-6**	**5-0**
Coprozessor-befehl	Opcode (0100XX)	Format	Rs	Rt	Rd	Function
Erweiterungs-befehl	Opcode (010XXX)	Rs	Rt	Rd	Special	

In Tabelle 3.2 ist die Kodierung der Erweiterungsbefehle dargestellt und zum Vergleich einem Coprozessorformat gegenübergestellt. Rs und Rt sind wie beim R-Format die Registeradressen der Eingangsoperanden und Rd ist die Registeradresse des Ergebnisses. Das Feld 'Special' steht für den 11 Bit langen Spezialbefehl, dessen Interpretation von der jeweiligen Erweiterungseinheit abhängt. Wie aus den Opcodes aus Tabelle 3.2 ersichtlich, werden bis zu vier Coprozessoren unterstützt, jedoch könnten im Befehlssatz problemlos acht untergebracht werden. Diese maximale Anzahl von acht Erweiterungseinheiten ist im Befehlsformat vorgesehen, wobei für die Implementierung des Prototypen die Anzahl auf vier reduziert wurde, da nur drei Erweiterungen entwickelt wurden.

3.4 Erweiterungseinheiten

Der bisher besprochene Ansatz gibt aus Gründen der Allgemeinheit und Erweiterbarkeit keinen bestimmten Aufbau der Erweiterungseinheiten vor. Lediglich die Schnittstelle zum Prozessor und das Kommunikationsprotokoll sind fixiert. Dadurch können praktisch beliebige Erweiterungen implementiert werden, welche sowohl rekonfigurierbar oder fest verdrahtet sein können. Für die Anbindung an die Schnittstelle und zum Steuern der Rekonfigurierung sollte jede Erweiterungseinheit allerdings über einen kleinen fest verdrahteten Teil verfügen.

3.4.1 Beispiele und deren Rekonfigurierung

Beispiele für mögliche Erweiterungen lassen sich in den Implementierungen aus Kapitel 2 finden, wo großteils die Konzepte mit geringen Modifikationen in Frage kommen würden. Ein weiteres mögliches Beispiel wäre auch die Implementierung einer rekonfigurierbaren Finite-state-machine, welche in [GDHG10] präsentiert wird. Als einzige Einschränkung gibt es keinen direkten Zugriff der Erweiterungen auf Daten- oder Programmspeicher, um explizit bei Funktionseinheiten und einer einfachen Schnittstelle zu bleiben. Allerdings besteht die Möglichkeit, die Erweiterungseinheiten mit einem lokalen Speicher auszustatten, um etwa Konfigurationsdaten oder implementationsabhängige Daten zu halten. Diese Speicher könnten z. B. über eine zusätzliche Schnittstelle oder dedizierte Pins am Chip geladen werden. Um jedoch ganz allgemein zu bleiben, wird davon ausgegangen, dass keine zusätzlichen Maßnahmen ergriffen werden. Somit müssen die eventuell vorhandenen lokalen Speicher über die generische Schnittstelle geladen werden. Das kann beispielsweise in der Initialisierungsphase des Systems passieren, um etwa alle für die Applikation benötigten Konfigurationen vom Programmspeicher in den lokalen Konfigurationsspeicher einer Erweiterungseinheit zu laden. Dadurch ergibt sich der Vorteil, dass keine außerordentlichen Software- oder Hardwaremaßnahmen getroffen werden müssen, da alles mit dem normalen erweiterten Befehlssatz des Prozessors machbar ist.

Besteht im weiteren Programmablauf der Bedarf eine andere Konfiguration zu laden, können aufgrund der Schnittstellendefinition pro Taktzyklus zwei Registerinhalte an die Erweiterungseinheit übergeben werden. Das entspricht maximal 64 variablen Konfigurationsbits und auch ein Teil des 11 Bit breiten Spezialbefehls kann für das Laden einer neuen Konfiguration benutzt werden. Eine derartige Prozedur ist allerdings nur für eine coarse-grain rekonfigurierbare Logik ratsam, da fine-grain Logik sehr viele Konfigurationsbits [VS07, Kapitel 2.3] braucht, was sich auf die Anzahl der für die Konfiguration benötigten Taktzyklen negativ auswirkt. Im Unterschied dazu kann coarse-grain Logik jedoch so entworfen werden, dass ein Taktzyklus für die Rekonfiguration ausreicht. Für sehr hohe Optimierungsmaßnahmen kann die Konfiguration mit dem aktuellen Befehl für eine Operation mitgeliefert werden. Das ist zum einen möglich, wenn nur ein Operand für die Operation benötigt wird und das zweite Ausgangsregister die neue Konfiguration enthält. Zum anderen kann auch der Spezialbefehl die Konfiguration oder Teile davon enthalten. Die beiden im Prototypen implementierten Erweiterungsfunktionseinheiten CORDIC und digitales Filter sind zum Teil von dieser Art der Rekonfigurierbarkeit und lassen sich daher praktisch ohne Zeitverlust rekonfigurieren.

3.4.2 Granularität der rekonfigurierbaren Logik

Um rekonfigurierbare Logik für Erweiterungseinheiten zu entwerfen, ist es von großem Vorteil das spätere Einsatzgebiet des Prozessors zu kennen. Obwohl sich mit rekonfigurierbarer Logik praktisch alle beliebigen digitalen Schaltungen realisieren lassen, können durch entsprechende Applikationskenntnisse die Strukturen daraufhin optimiert werden. Aus diesem Grund gibt es auch verschiedene Granularitäten bei rekonfigurierbarer Logik mit unterschiedlichen Vor- und Nachteilen. Um die konkrete Wahl für die Beispielimplementierung von Kapitel 4 nachvollziehen zu können, werden die Eigenschaften dieser Granularitäten im Folgenden verglichen.

Prinzipiell wird rekonfigurierbare Logik in fine- und coarse-grain unterschieden [VS07, Kapitel 2.3] und [HD07, Kapitel 2.1]. Wie sich aus dem Namen bereits schließen lässt, bezieht sich diese Einteilung auf die Größe der elementaren, konfigurierbaren Einheiten der Logik. Fine-grain Logik ist aufgrund ihres Aufbaus aus einzeln konfigurierbaren Verbindungen und logischen Verknüpfungen Bit-orientiert. Dadurch ist sie universell einsetzbar und jede beliebige digitale Schaltung, die nicht zu groß für die vorhandenen Hardwareressourcen ist lässt sich damit realisieren. Dieser universelle Aufbau hat jedoch im Vergleich zur coarse-grain Logik einige Nachteile. So ist aufgrund der großen Anzahl an Konfigurationsmöglichkeiten auch ein großer Konfigurationsoverhead und -speicher erforderlich um die Funktionen zu definieren. Da jedes rekonfigurierbare Element einen eigenen Speicher darstellt, ist zudem die Flächenausnutzung schlecht und der Energiebedarf vergleichsweise hoch. Weiters ist das Timing solcher Schaltungen im Allgemeinen eher schlecht, da ein Signal über viele solcher rekonfigurierbaren Schalter und teilweise langer Strecken laufen muss. Deshalb kommt fine-grain Logik größtenteils dann zum Einsatz, wenn das Applikationsgebiet so sehr variabel sein soll, was z. B. bei FPGAs und CPLDs (**C**omplex **P**rogrammable **L**ogic **D**evice) der Fall ist.

Im Unterschied dazu ist rekonfigurierbare coarse-grain Logik nicht vollständig universell einsetzbar. Der Aufbau besteht aus programmierbaren fest verdrahteten PEs, die vorgegebene Operationen auf Wortebene ausführen. Wortebene beschreibt Operationen mit mehreren parallelen Bits je nach Architekturbreite, wie z. B. Additionen, Schiebeoperationen, usw. Auch das Verbindungsnetzwerk der PEs ist nicht bitweise organisiert, sondern über programmierbare Busverbindungen realisiert. Durch diesen Aufbau besitzt coarse-grain Logik genug Flexibilität um für mehrere verschiedene Anwendungen geeignet zu sein, wobei jedoch eine komplette Anwendungsunabhängigkeit verloren geht. Allerdings bringt die Einschränkung der Konfigurierbarkeit auch erhebliche Vorteile mit sich. So ergibt sich neben einem wesentlich verringerten Konfigurationsoverhead und dem damit zusammenhängenden Speicher- und Zeitbedarf auch eine deutlich effizientere Chipflächennutzung. Eine ineffiziente Ausnutzung der Fläche ergibt sich nur, wenn Operationen auf Bit-Ebene auszuführen sind, außerdem ist die Funktionsausführung, bei gleichzeitiger Steigerung der Energieeffizienz erheblich schneller. Somit ist coarse-grain Logik vor allem für Datenpfadoperationen in einem bestimmten Applikationsgebiet ausgelegt.

Tabelle 3.3: Vergleich der Eigenschaften von rekonfigurierbarer fine- und coarse-grain Logik

	fine-grain	**coarse-grain**
Flexibilität	universell	beschränkt
Konfigurationsoverhead	sehr hoch	wenig bis sehr gering
Leistung	mittel bis gering	hoch
Energieeffizienz	gering	hoch
Flächenausnutzung	ineffizient	effizient
Operationsebene	Bit	Wort
Einsatz	FPGA, CPLD	Applikationsgebiet

Um einen Überblick über die vorhin beschriebenen Eigenschaften zu bekommen, sind diese nochmals in Tabelle 3.3 zusammengefasst. Da sich die vorliegende Diplomarbeit mit der Leistungssteigerung von Prozessoren beschäftigt, wird für die Beispielimplementierung aufgrund der hohen erzielbaren Rechenleistung eine coarse-grain Logik verwendet.

3.5 Herangehensweise

Die Herangehensweise zur Erstellung des Prototypen wie er in Kapitel 4 beschrieben wird, ist in Abbildung 3.5 dargestellt. Nach der Spezifikation des Prozessors und der Schnittstelle zu den erweiterten Funktionseinheiten besteht die weitere Aufgabe darin, die Erweiterungseinheiten zu spezifizieren. Da der genaue Aufbau dieser Komponenten nur als Beispiel anzusehen ist, sind diese Komponenten nicht Teil des gewählten Ansatzes und werden daher erst in Kapitel 4 behandelt.

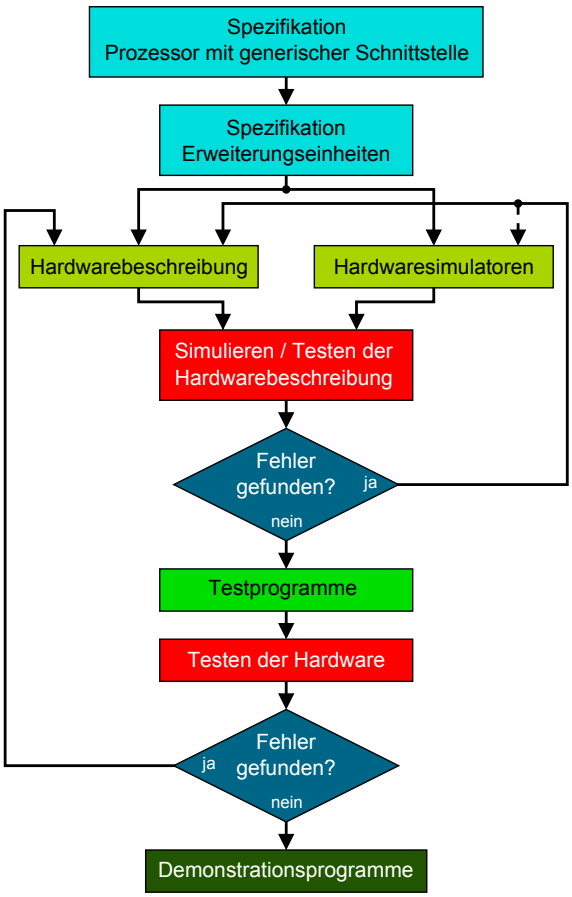

Abbildung 3.5: Top-Down Entwurf zur Erstellung des Prototypen

Der nächste Schritt in Richtung Prototyp ist eine high-level Beschreibung der Hardware. Dafür wird die Hardwarebeschreibungssprache VHDL verwendet. Um die Hardwareerweiterungen später vernünftig testen zu können und Implementierungsdetails festzulegen, wurden parallel zur Beschreibung Simulatoren für diese Hardware in MATLAB erstellt. Diese Simulatoren simulieren das komplette äußere Verhalten der Hardware inklusive Timing und werden in Kapitel 5.1 noch genauer behandelt. Überprüfen lässt sich das Simulationsergebnis dabei teilweise über die in MATLAB integrierten Funktionen.

Nach erfolgter Hardwarebeschreibung beginnt das Simulieren und Testen der beschriebenen Strukturen. Dafür werden Testbenches erstellt, welche als Testmuster die von den Simulatoren generierten Ausgangsdaten verwenden. Durch einen iterativen Prozess aus Simulation, Fehlersuche und Korrektur konnten Software- und Hardwaresimulation in Einklang gebracht werden. Somit können Fehler in der Beschreibung zuverlässig identifiziert und damit vermieden werden. Ist die

Hardwarebeschreibung fertig gestellt, verbleibt noch die Synthese der Hardware. Dazu muss jedoch die Zielhardware bekannt sein. Aus Aufwands- und Kostengründen wird für die Diplomarbeit kein eigens entworfener Siliziumchip, sondern ein Xilinx FPGA verwendet. Dieses ist zwar nicht so leistungsfähig wie ein Chip mit fest verdrahteter Logik, allerdings lässt sich das Konzept hier sogar mit dem Austausch von Erweiterungseinheiten demonstrieren. Für die Synthese müssen lediglich das Pin-Layout und die Anforderungen an das Timing festgelegt werden.

Nach erfolgreicher Synthese können Testprogramme für den Prozessor erstellt werden, um diesen im tatsächlichen Einsatz zu testen. Dazu werden chipinterne Signale aufgezeichnet und ausgewertet. Durch eine ausgiebige Signalanalyse konnten auch letzte Fehler behoben werden, sodass der Prozessor problemlos und stabil läuft. Alle Einzelheiten dazu finden sich wieder in Kapitel 5.1.

Der letzte Schritt der praktischen Arbeit besteht dann in der Erstellung von Beispielprogrammen, die auf anschauliche Weise die Funktionsfähigkeit und Rekonfigurierbarkeit des Prototypen anhand von Echtzeit Audioverarbeitung demonstrieren.

4 Beispielimplementierung

Das Kapitel Beispielimplementierung widmet sich der Beschreibung des implementierten Prototypen. Es werden Implementierungsdetails beschrieben und auf implementierungsspezifische Besonderheiten hingewiesen. Dabei werden zuerst die für die Entwicklung notwendigen Softwarewerkzeuge und die verwendete Hardware vorgestellt, bevor auf den eigentlichen Prozessor und seine Erweiterungen eingegangen wird.

4.1 Verwendete Entwicklungsumgebung und Hardware

Die Auswahl der verwendeten Entwicklungsumgebung und des FPGA-Boards wurde hauptsächlich anhand von frei verfügbaren Ressourcen getroffen. Sowohl Hardware als auch Software stammt großteils von der Firma Xilinx, wodurch Kompatibilitätsprobleme unwahrscheinlich sind. Für die Beschreibung und Synthese der Hardware, bis hin zum Download des Bitstreams wurden die Xilinx-Entwicklungsumgebung der Version 14.x und weiterführende Werkzeuge verwendet. Lediglich für die Simulation musste auf eine andere Software namens QuestaSim Version 10.1 zurückgegriffen werden, da der in der Entwicklungsumgebung integrierte Simulator zu große Stabilitätsprobleme aufwieß. Auch für das Debugging am Chip wurde die Xilinx Software ChipScope verwendet, mit deren Hilfe auch die Abbildungen 5.2 und 5.4 in Kapitel 5.2 über Signalverläufe im Chip erstellt werden konnten.

Als FPGA-Board kam ein ML405 Evaluationboard (siehe Abbildung 4.1) zum Einsatz welches neben einiger Peripherie ein Virtex 4 FPGA (XC4VFX20-FF672) mit integriertem Mikroprozessor beherbergt [10]. Der integrierte Prozessor, sowie die meisten Peripheriekomponenten werden jedoch nicht verwendet. Lediglich die darauf befindlichen Taster, LEDs (**L**ight **E**mitting **D**iode), Audiobuchsen und ein AC97 Audiochip werden für Demonstrationszwecke benötigt.

4.2 Prozessor

Für den Prototypen wird laut Spezifikation ein 32 Bit MIPS R2000 ähnlicher Prozessor verwendet. Da der Aufbau der Prozessor-Pipeline bereits in Kapitel 3.2 behandelt worden ist, fokusiert sich der folgende Abschnitt vor allem auf den Aufbau der einzelnen Funktionsblöcke. Im Anschluss daran wird die Vorgehensweise bei der Programmierung des Prozessors kurz vorgestellt.

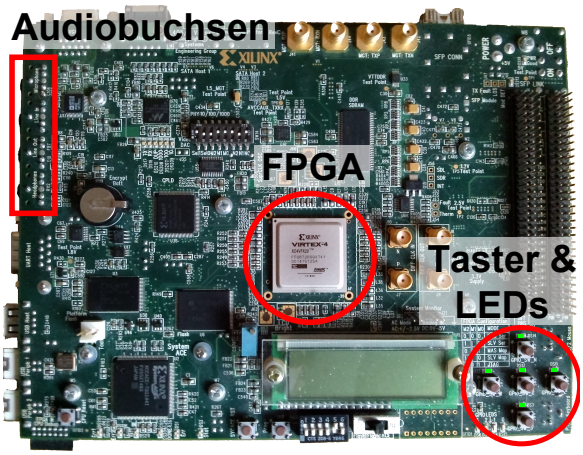

Abbildung 4.1: ML405 Evaluationboard

4.2.1 Aufbau

Der Aufbau des MIPS R2000 Prozessors ist etwas komplizierter, als es für die Demonstration der Diplomarbeit erforderlich ist. Deshalb wurde der Prozessor für die Implementierung in mancher Hinsicht stark vereinfacht und folgende Funktionseinheiten weggelassen [KH92, Kapitel 2]:

- Speicherverwaltungseinheit MMU (**M**emory **M**anagement **U**nit)
- Cache Kontrolleinheit
- System Control Coprozessor
- Coprozessorschnittstelle

Der Wegfall der Cache- und Speicherkontrolleinheit wird dadurch erreicht, dass für den Prototypen keine externen Speicher vorgesehen werden. Somit besteht der gesamte zur Verfügung stehende Speicher nur aus einem jeweils 64 KByte großen Daten- und Programmcache. Diese reichen für die Implementierung der einfachen Demonstrationsprogramme aus und werden im Folgenden lediglich als Daten- und Programmspeicher bezeichnet.

Durch den Wegfall der Coprozessorschnittstelle erübrigt sich aufgrund der fehlenden Anbindung auch der System Control Coprozessor. Dieser Coprozessor ist beim ursprünglichen Prozessor für die Koordination der Aufgaben zuständig, die bei der vorgestellten vereinfachten Implementierung wegfallen. So existieren keine unterschiedlichen Operationsmodi wie Benutzer- oder Kernelmodus. Außerdem gibt es bis auf die Interrupt-Behandlung keine Ausnahmebehandlungen und die Behandlung von Zahlenüberläufen wird durch einen normalen maskierbaren Interrupt ersetzt. Da die anderen Aufgaben dieses Coprozessors mit der Cache- und Speicherverwaltung zusammenhängen, sind auch diese hier überflüssig.

In Abbildung 4.2 ist der Aufbau des vereinfachten Prozessors der Diplomarbeit inklusive dem Erweiterungskonzept gezeigt. Programm- und Datenspeicher sind getrennt, wodurch pro Taktzyklus ein Zugriff auf beide Speicher gleichzeitig stattfinden kann. Um jedoch Konstanten und Konfigurationen vernünftig aus dem Programmspeicher laden zu können, ist der mögliche Adressbereich

des Datenspeichers genau in der Mitte in zwei Blöcke unterteilt. Da der Datenspeicher byteadressierbar ist, während der Programmspeicher in der Implementierung nur über eine Wortadressierung verfügt, sind die möglichen Adressbereiche 2 GB bzw. 8 GB groß. Der untere Bereich adressiert dabei den normalen Datenspeicher und mit dem oberen Block können die unteren 8 GB des Programmspeichers adressiert werden. Da weder Daten, noch Programmspeicher bei der Implementierung auch nur annähernd so groß sind, wird bei Überlauf der Adresse über die letzte Speicherstelle eine simple Modulofunktion verwendet. Das heißt, es werden nur die unteren Adressleitungen verwendet ohne bei Adressüberlauf eine Ausnahmebehandlung durchzuführen. Bei Speicherzugriffen ist zu beachten, dass die Speicher zwar byteweise adressiert werden, jedoch nach Datenwörtern ausgerichtet sind. So können bei Datenwörtern aufgrund der Länge von vier Byte pro Wort Speicherzugriffe nur auf Adressen mit ganzzahligen Vielfachen von vier erfolgen. Das Gleiche gilt für Halbwörter, allerdings mit dem Faktor zwei statt vier. Daher werden bei der Adressierung, sofern es sich nicht um einen Bytezugriff handelt, die unteren ein oder zwei Adressleitungen ignoriert.

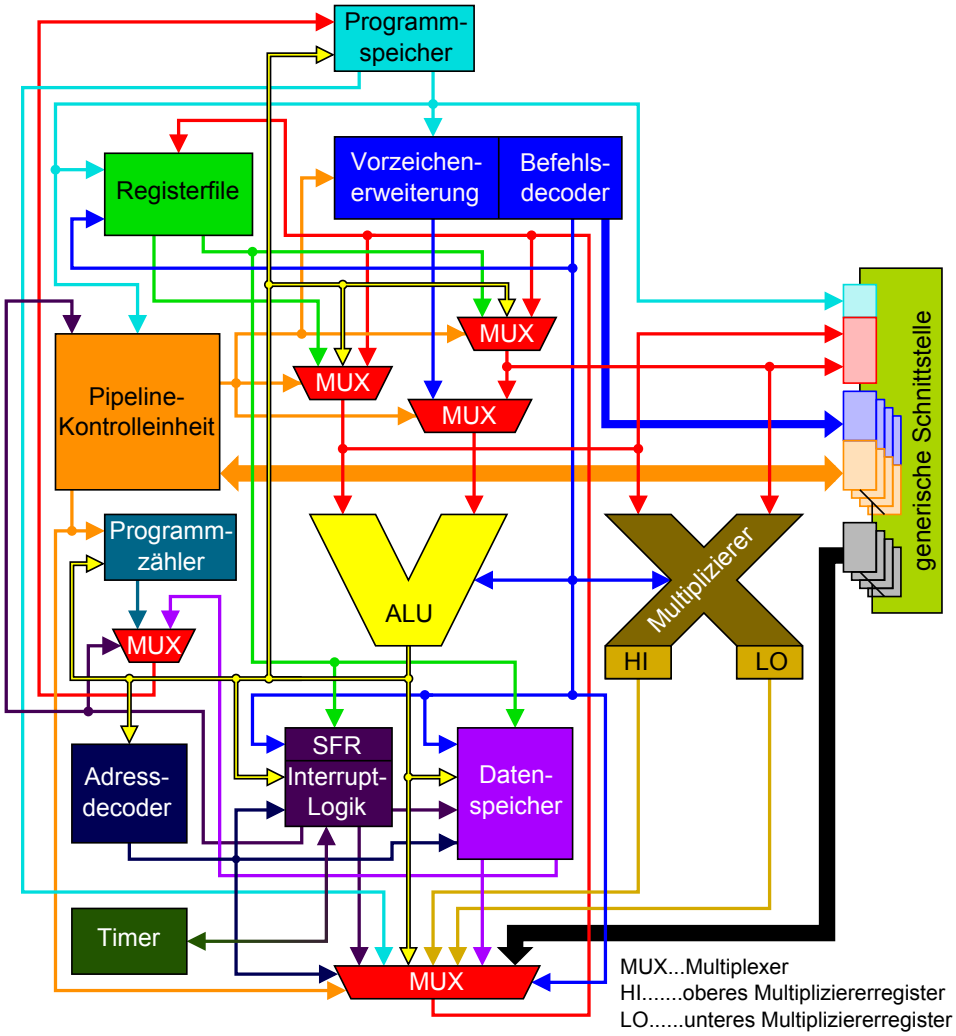

Abbildung 4.2: Aufbau des Prototypenprozessors

Um in der Reihenfolge der Prozessor-Pipeline zu bleiben, wird als nächstes das Registerfile behandelt. Dieses verfügt über 32 Register, wobei im Unterschied zum R2000 auch das Register null beschrieben werden kann. Dadurch kann es bei der Programmierung mit entsprechender Vorsicht auch als Universalregister verwendet werden. Pro Taktzyklus können zwei beliebige Registerinhalte gelesen und einer geschrieben werden. Dabei ist die Reihenfolge zuerst lesen, dann schreiben. Als Besonderheit besitzt das Registerfile die Fähigkeit zur automatischen Stackpointer-Verwaltung, sofern diese Funktion entsprechend konfiguriert wird. Damit können kurze Interrupts und Unterprogrammaufrufe stark beschleunigt werden. Außerdem kann das Register, in dem normalerweise die Stackadresse gespeichert wird, nach der Stackpointer-Initialisierung auch als Universalregister verwendet werden.

Ein Funktionsblock, der ebenfalls in der zweiten Pipeline-Stufe liegt, ist der Befehlsdecoder. Dieser hat die Aufgabe den vom Programmspeicher übernommenen Befehl zu decodieren und entsprechende Steuersignale für die Ausführungseinheit zu erzeugen. Beispiele für Steuersignale sind dabei die Auswahl der Eingangsoperanden und die Operationsselektion. Außerdem ist der Befehlsdecoder für die Erweiterung des Vorzeichens bei I-Befehlen verantwortlich, da die im Befehl enthaltenen konstanten Operanden nur 16 Bit lang sind.

Als nächstes kommt die Ausführungseinheit oder auch ALU (**A**rithmetical and **L**ogical **U**nit) genannt, die sämtliche logische und arithmetische Operationen mit den Eingangsoperanden durchführt. Zu diesen Operationen gehört auch die Berechnung von Speicheradressen für den Zugriff auf den Datenspeicher. Außerdem werden für Speicheroperationen, die kein ganzes Datenwort betreffen, die kürzeren Bitblöcke Byte und Halbwort auf die entsprechenden Speicheradressen ausgerichtet. Dabei kommt im Gegensatz zum R2000, welcher in dieser Hinsicht konfiguriert werden kann, immer die Little-Endian Speicherorganisation zum Einsatz. Das bedeutet, dass das niederwertigste Byte auch auf der niederwertigsten Adresse gespeichert wird.

Die einzige arithmetische Operation, die sich nicht im ALU-Funktionsblock befindet, ist die Multiplikation, da diese auch im FPGA auf einen dedizierten Multiplizierer ausgelagert ist. Zum Multiplizererblock gehören auch dessen Ergebnisregister. Durch die Auslagerung auf den fest verdrahteten Multiplizierer kann die Ausführungszeit von zwölf Taktzyklen beim R2000 auf drei reduziert werden. Somit kann direkt im nächsten Befehl auf das Multiplikationsergebnis zugegriffen werden, wobei der Zugriff dafür erst in der letzten Pipeline-Stufe erfolgt. Von der Pipeline-Kontrolleinheit werden solche Befehle daher wie Ladebefehle behandelt.

Eng gekoppelt mit der Ausführungseinheit und in der Hardwarebeschreibung sogar im selben Modul befindet sich der Programmzähler. Dieser berechnet immer die Adresse des nächsten auszuführenden Befehls. Die enge Kopplung kommt dadurch zustande, dass es bedingte Sprünge gibt, die eine vorherige Auswertung durch die ALU erfordern. Im Unterschied zum R2000 Prozessor wird ein Sprungbefehl erst zwei Taktzyklen, also um einen Takt später als beim R2000, nach dem Laden aus dem Programmspeicher ausgeführt. Der Grund für diese spätere Ausführung liegt in der Befehls-Pipeline des implementierten Prozessors, in der die Ausführungsstufe keine zwei Phasen zum Auswerten und Springen besitzt. Während beim R2000 der nach dem Sprung folgende Befehl in jedem Fall noch ausgeführt wird, fügt der implementierte Prozessor automatisch zwei Leerbefehle ein. Dadurch ergibt sich neben einer einfacheren Interrupt-Logik auch eine einfachere Assemblerprogrammierung auf Kosten der Rechenleistung.

In der Speicherzugriffsstufe der Pipeline befindet sich neben dem weiter oben bereits behandelten Datenspeicher das implementierungsspezifische SFR (**S**pecial **F**unction **R**egister). Dieses Register ist Memory-mapped, das heißt es wird in den unteren Adressbereich des Datenspeichers eingebettet und verhält sich bezüglich Zugriffen auch so. Es beinhaltet 16 Datenwörter, über die spezielle

Prozessorfunktionen gesteuert werden. Dazu gehört unter anderem die Steuerung und der Zugriff auf die E/A-Ports. Weiters können hier die 16 möglichen Interrupts aktiviert oder deaktiviert werden und die Triggerflanke der acht externen Interrupts bestimmt werden. Auch das Rücksetzen von IRQs (**I**nterrupt **R**e**Q**uest), die Steuerung des Timers und der automatischen Stackpointer-Verwaltung werden über diese Funktionseinheit vorgenommen. Die genaue Registerbelegung dafür ist in Kapitel A.2 zu finden. Da sowohl die Kontrolle der externen Ports, als auch die Interrupt-Konfigurationen in diesem Register sind, befindet sich auch hier die Interrupt-Kontrolllogik. Diese Logik generiert bei einem Interrupt-Aufruf die Zeiger auf die Interrupt-Adresse und das Steuersignal für den Programmzähler und die Pipeline-Kontrolleinheit. Die Interrupt-Adressen liegen dabei programmierbar auf den unteren 16 Datenwörtern im Datenspeicher, wofür dieser über einen zweiten Leseport verfügt. Um jeweils den richtigen Speicher anzusprechen gibt es einen einfach aufgebauten Adressdecoder, der anhand der oberen Adressleitungen die Auswahlsignale für die Speicher generiert.

Die letzte Stufe der Pipeline besteht dann nur noch aus einem Multiplexer, der das in das Registerfile zu schreibende Datum auswählt. Dabei müssen allerdings Byte- und Halbwort-Daten, die vom Datenspeicher kommen noch richtig ausgerichtet werden und bei Bedarf noch das Vorzeichen erweitert werden. Auch die Ergebnisse der erweiterten Funktionseinheiten kommen über diesen Multiplexer wieder in den Standarddatenpfad des Prozessors zurück.

Um auf ALU-Ergebnisse zurückgreifen zu können, die noch nicht wieder in das Registerfile geschrieben wurden, ist vor den Operandeneingängen der ALU je ein Multiplexer geschaltet, der das Ergebnis jeder Pipeline-Stufe auswählen kann. Die Steuerung dafür übernimmt die Pipeline-Kontrolleinheit, welche anhand der Befehlsabfolge Datenabhängigkeiten erkennt. Bei Lade- und Pipeline-synchronen Erweiterungsbefehlen fügt diese Einheit bei Datenabhängigkeiten auch automatisch Leerbefehle ein, um die Assemblerprogrammierung zu vereinfachen. Beim MIPS R2000 Prozessor ist das jedoch nicht der Fall. Die wichtigste Aufgabe dieser Einheit ist allerdings die Kontrolle der erweiterten Funktionseinheiten. So kümmert sich die Kontrolleinheit um die Auswahl der Erweiterungseinheit für einen bestimmten Befehl und behandelt Ressourcenkonflikte beim Schreibzugriff auf das Registerfile.

Als zusätzlichen Block verfügt der Prozessor noch über einen konfigurierbaren Timer, um längere Wartezeiten vernünftig implementieren zu können. Die Steuerung des Blocks erfolgt über das SFR und der Block selbst ist in der Lage einen Überlaufs- und Vergleichs-Interrupt auszulösen. Somit könnte z. B. der Prozessor in einen Schlafmodus versetzt werden um Energie zu sparen, sofern die dafür nötigen Implementierungen noch gemacht werden.

4.2.2 Programmierung

Obwohl sich im Aufbau des Prozessors doch erhebliche Unterschiede zum R2000 Basisprozessor ergeben, ist der MIPS I Befehlssatz [KH92, Tabelle 2-1] praktisch unverändert implementiert. Die einzigen Ausnahmen sind die beiden Divisionsbefehle, die Befehle zum Beschreiben der beiden Spezialregister für Multiplizierer und Dividierer und die Spezialbefehle 'SYSCALL' und 'BREAK'. Diese Befehle wurden aus Aufwandsgründen weggelassen und durch Leerbefehle ersetzt. Ein weiterer Grund für das Weglassen der Division besteht in der Tatsache, dass diese über die später besprochene CORDIC-Erweiterung ebenfalls durchgeführt werden kann. Dabei benötigt die Erweiterung nicht nur weniger Taktzyklen, sondern arbeitet auch mit Gleitkommazahlen. Die restlichen Befehle stimmen allerdings exakt mit dem spezifizierten Befehlssatz überein, sodass für die

Erstellung von Programmen für den Prototypen existierende MIPS Assembler verwendet werden können.

Bei der Verwendung von höheren Programmiersprachen ist allerdings darauf zu achten, dass die nicht implementierten Befehle vom Compiler nicht verwendet werden. Außerdem müsste der Compiler bei Sprüngen und Verzweigungen immer von einem 'non-delayed Branch' ausgehen. Ein non-delayed Branch bedeutet dabei, dass Befehle, die hinter dem Sprung stehen nur dann ausgeführt werden, wenn nicht gesprungen wird. Der MIPS R2000 Prozessor verfügt jedoch über einen sogenannten Delay-Slot [KH92, Seite 3-11], das heißt der Sprung wird erst nach dem darauf folgenden Befehl ausgeführt. Zu beachten ist außerdem die nicht vorhandene Ausnahmebehandlung. Aus diesen Gründen ist die Verwendung von höheren Programmiersprachen ohne Modifikationen des Compilers oder entsprechender Nachbearbeitung des Maschinenprogramms nicht möglich und sämtliche Programmierarbeiten müssen in der Assemblersprache durchgeführt werden.

4.2.3 Interrupt-Abarbeitung

Die erweiterten Funktionseinheiten können, wie bereits mehrfach erwähnt, beliebig lange Ausführungszeiten haben, weshalb es von Seite der Hardware kein automatisches Warten auf deren Ergebnisse gibt. Der Grund dafür ist, dass der Prozessor gar nicht wissen kann, ob ein Spezialbefehl überhaupt Ergebnisse liefert. Darum stellen Interrupts ein wichtiges Mittel dar, um auf Ergebnisse von Erweiterungen zu reagieren. Eine andere Möglichkeit wäre natürlich ein Modell der jeweiligen Erweiterung bereitzustellen, damit der Compiler die richtige Anzahl an Befehlen einfügt, um auf das erwartete Ergebnis zu warten. Aufgrund der manuellen Assemblerprogrammierung ist jedoch bei längeren Befehlsabarbeitungen der Erweiterungseinheiten die Interrupt-Methode vorzuziehen. Deshalb wird der Ablauf bei Interrupt-Aufrufen im Folgenden genauer untersucht.

Grundsätzlich werden alle Interrupts von Funktionseinheiten als interne Interrupts mit höherer Priorität als externe behandelt. Dabei hat der Zahlenüberlauf-Interrupt die höchste Priorität, gefolgt von den erweiterten Funktionseinheiten. So wird bei gleichzeitiger Anfrage mehrerer Interrupts immer der mit der höchsten Priorität zuerst ausgeführt. Die Prioritätsreihenfolge für die Erweiterungseinheiten ist die Gleiche wie bei der Registerzugriffsvergabe. Dadurch kann gewährleistet werden, dass das Ergebnis der Funktionseinheit beim Interrupt-Aufruf auch wirklich bereits im Register vorhanden ist.

Ein IRQ wird bei den erweiterten Funktionseinheiten durch die Beantragung des Schreibzugriffs auf das Registerfile getätigt. Sofern folgende zusätzliche Bedingungen erfüllt sind, wird mit der Auslöseprozedur des Interrupts fortgefahren:

- Interrupts sind aktiviert
- aktueller Interrupt ist nicht ausmaskiert
- Anfrage hat die höchste Priorität
- keine Kollision mit früheren Interrupt-Aufrufen

Nach positiver Überprüfung dieser Bedingungen wird im nächsten Taktzyklus der Zeiger auf die Interrupt-Adresse an den Datenspeicher übergeben. Dieser liefert in der darauf folgenden

Taktperiode die Interrupt-Adresse für den Programmspeicher. Daraufhin wird die normale Befehlsabarbeitung bei der Interrupt-Adresse fortgesetzt. Somit dauert es vier Taktzyklen vom IRQ bis zur Abarbeitung des ersten Interrupbefehls, also genau die Zeit, die es dauert, bis das Ergebnis einer Erweiterungseinheit verfügbar wird. Von den beiden sich in der Befehls-Pipeline befindlichen Befehle wird der erste noch normal ausgeführt und der zweite vorerst verworfen, um nach Rücksprung des Interrupts hier fortzufahren. Statt des zweiten Befehls wird die im Rücksprungregister befindliche Adresse auf den Stack abgelegt und der aktuelle Stand des Programmzählers in das Rücksprungregister geschrieben. Bei der Interrupt-Abarbeitung wird garantiert, dass die ersten sieben Befehle in jedem Fall ausgeführt werden, bevor der nächste Interrupt ausgeführt werden kann. Das ist wichtig, wenn z. B. der Stackpointer nicht von der Hardware verwaltet wird und somit noch verändert werden muss, oder wenn der Interrupt auf keinen Fall unterbrochen werden darf. Dazu müssen die Interrupts jedoch aufgrund der Verzögerungen von Pipeline- und Interrupt-Logik sofort deaktiviert werden. Der Ablauf der eben beschriebenen Prozedur ist in Abbildung 4.3 anhand der fünf Pipeline-Stufen dargestellt. Zum Vergleich befindet sich darunter der Signalverlauf wenn eine Erweiterungseinheit Schreibzugriff auf ein Register beantragt, was eine mögliche Interrupt-Quelle darstellt.

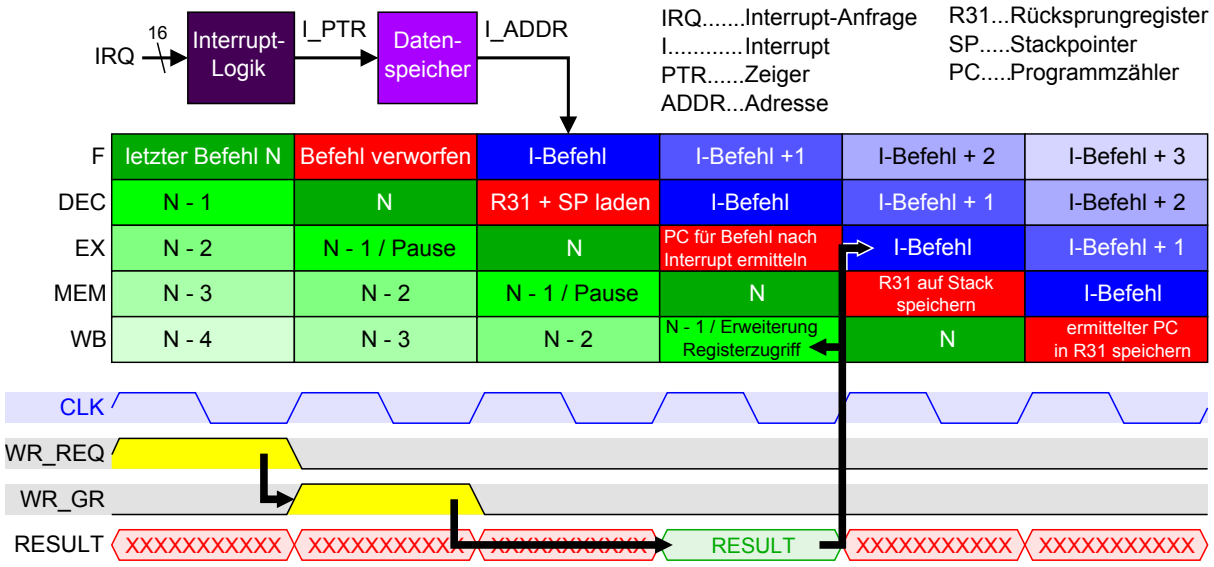

Abbildung 4.3: Ablauf eines Interrupt-Aufrufes

Nach Abarbeitung des Interrupts ist zu beachten, dass das Rücksprungregister noch per Software wiederhergestellt werden muss, wofür die Interrupt-Rücksprungadresse in ein anderes Register verschoben werden muss. Dieses Register kann somit vom übrigen Programm nicht genutzt werden, was einen Nachteil dieser Implementierung darstellt. Um den Verlust des Registers zu kompensieren, wurde die Möglichkeit einer automatischen Stackpointer-Verwaltung geschaffen, wodurch das ursprüngliche Stackpointer-Register als Universalregister frei wird.

4.3 Transzendente Berechnungen (CORDIC)

Die erste rekonfigurierbare Erweiterungseinheit für den Prozessor ist eine CORDIC-Einheit. Bevor jedoch auf Implementierungsdetails eingegangen wird, werden der Basisalgorithmus und dessen Modifikationen erklärt.

4.3.1 Basisalgorithmus

Der CORDIC-Algorithmus wurde ursprünglich für Echtzeitberechnung trigonometrischer Funktionen in Navigationssystemen entwickelt [Vol59]. Es handelt sich dabei um einen iterativen Algorithmus, der sich vor allem durch die Einfachheit der involvierten Operationen auszeichnet.

Die ursprünglichste Form wurde für das Ausführen von zirkularen Vektorrotationen ausgelegt. Dabei wird von einem zweidimensionalen Startvektor ausgegangen, der um einen Winkel θ gedreht wird. In Gleichung 4.1 wird so eine Rotation durchgeführt wobei x und y die kartesischen Koordinaten des Vektors sind.

$$\begin{bmatrix} x \\ y \end{bmatrix}_{neu} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{alt} = \cos(\theta) \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{alt} \tag{4.1}$$

Der letzte Teil der Gleichung stellt eine reine Umformung dar. Nun können mehrere dieser Rotationen um verschiedene Winkel hintereinander ausgeführt werden. So kann beispielsweise die Rotation aus Gleichung 4.1 auch durch mehrere Teilrotationen erreicht werden, wobei die Summe der Teilwinkel gleich θ ist. Gleichung 4.2 zeigt so eine Folge von Teilrotationen. Die verwendeten Teilwinkel bestehen aus einem Vorzeichen σ_i und dem Betrag des Winkels α_i. Der letzte Teil der Gleichung ist lediglich eine Vereinfachung der Darstellung, wobei die verwendeten Konstrukte in Gleichung 4.3 definiert sind.

$$\begin{bmatrix} x \\ y \end{bmatrix}_{neu} = \prod_i \cos(\sigma_i \alpha_i) \begin{bmatrix} 1 & -\tan(\sigma_i \alpha_i) \\ \tan(\sigma_i \alpha_i) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{alt} = K \prod_i R_i \begin{bmatrix} x \\ y \end{bmatrix}_{alt} \tag{4.2}$$

$$\theta = \sum_i \sigma_i \alpha_i, \qquad \sigma_i = \pm 1, \qquad K = \prod_i \cos(\alpha_i), \qquad R_i = \begin{bmatrix} 1 & -\sigma_i \tan(\alpha_i) \\ \sigma_i \tan(\alpha_i) & 1 \end{bmatrix} \tag{4.3}$$

Der Faktor K fasst dabei die Skalierungsfaktoren aller mit R_i beschriebenen Teilrotationen zusammen, wobei zu beachten ist, dass R_i nicht nur eine Drehung, sondern auch eine Streckung eines Vektors bewirkt. Der Kernpunkt des Algorithmus ist nun die konkrete Wahl der Teilwinkel. Denn durch diese können die Teilrotationen so stark vereinfacht werden, dass pro Rotation lediglich zwei Additionen bzw. Subtraktionen und Schiebeoperationen ausreichend sind. Das wird dadurch erreicht, dass der Tangens der Teilwinkel als negative Zweierpotenz gewählt wird, wodurch sich die Gleichung 4.4 ergibt.

$$R_i = \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix}, \qquad \alpha_i = \arctan(2^{-i}), \qquad K = \prod_i \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{4.4}$$

Damit können für $n \to \infty$ Teilrotationen beliebige Gesamtdrehungen erzeugt werden, solange der Rotationswinkel kleiner oder gleich der Summe der Teilwinkelbeträge ist [HD07, Kapitel 25]. Doch auch für eine geringe endliche Anzahl an Teilrotationen können diese Winkel sehr gut

angenähert werden. Nach Ausführen aller dazu erforderlichen Operationen ist ein um den Winkel $\widetilde{\theta} \approx \theta$ gedrehter und um K gestreckter Vektor verfügbar. Der Faktor K ist für eine feste Anzahl an Teilrotationen konstant und kann wie auch die verwendeten Teilwinkel vorberechnet werden. Sofern die Dehnung des Vektors für weitere Berechnungen störend ist und nicht anderweitig eliminiert werden kann, ist eine anschließende Multiplikation mit der Inversen von K erforderlich.

Bei der Auswahl der Teilrotationen unterscheidet der CORDIC-Algorithmus zwei Operationsmodi. Beim ersten Modus, dem Rotationsmodus wird wie vorhin beschrieben ein Eingangsvektor durch eine Folge von Teilrotationen um einen bestimmten Winkel gedreht. Eine Teilrotation wird dazu im weiteren Verlauf als Interationsschritt bezeichnet und ist in den Gleichungen 4.5 bis 4.7 explizit dargestellt.

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{4.5}$$
$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{4.6}$$
$$z_{i+1} = z_i - \sigma_i \arctan(2^{-i}) \tag{4.7}$$

Die Iterationsvariable z_i dient dabei zur Akkumulation der Teilwinkel und wird vor dem ersten Iterationsschritt mit θ initialisiert. Da es bei der Rotation darum geht den Winkel θ anzunähern und z_i den noch verbleibenden Rotationswinkel angibt, ist es das Ziel, z_i gegen null gehen zu lassen. Dazu muss das Vorzeichen σ_i des nächsten Teilwinkels so gewählt werden, dass $|z_{i+1}|$ kleinstmöglich wird. Somit hat σ_i immer das gleiche Vorzeichen wie z_i bzw. ist bei $z_i = 0$ per Definition eins. Mit Hilfe der eben beschriebenen Vorgangsweise können die trigonometrischen Funktionen $\sin(\theta)$ und $\cos(\theta)$ berechnet werden. Dazu muss der Algorithmus lediglich mit dem Einheitsvektor auf der x-Achse, also $x_0 = 1$ und $y_0 = 0$ initialisiert werden. Statt $x_0 = 1$ kann natürlich auch $x_0 = 1/K$ gesetzt werden, um von vornherein die richtige Skalierung zu erhalten. Über die Drehung um den Winkel θ ergeben sich nach dann Gleichung 4.1 die gewünschten Größen. Durch die Wahl von x_0 als euklidische Vektornorm $\|\vec{v}\|_2$ und z_0 dessen Phase ϕ wird sogar eine Koordinatentransformation von Polar- auf kartesische Koordinaten erreicht (siehe Gleichung 4.8).

$$\begin{aligned} x_n &= \|\vec{v}\|_2 \cos(\phi) = x_0 \cos(z_0) \\ y_n &= \|\vec{v}\|_2 \sin(\phi) = x_0 \sin(z_0) \end{aligned} \tag{4.8}$$

Der zweite Operationsmodus ist der Vektormodus. Hier wird versucht, ausgehend von einem beliebigen Startvektor diesen auf die x-Achse zu drehen. Dadurch besitzt der resultierende Vektor keine y-Komponente mehr, sondern nur noch eine x-Komponente, die damit gleichzeitig die Norm des Vektors $\vec{v}$ ist. Erreicht wird das dadurch, dass diesmal die y-Komponente des Vektors, also y_i gegen null iteriert wird. Dadurch ergibt sich aus Gleichung 4.6, dass das Vorzeichen des nächsten Teilwinkels das invertierte Vorzeichen von y_i ist bzw. bei $y_i = 0$ auf minus eins gesetzt wird. Da mit der Variable z_i, welche zuvor mit null initialisiert werden muss, alle Teilwinkel aufsummiert werden, ergibt sich so die Phase ϕ des Ausgangsvektors. Es wird also eine Transformation von kartesische- auf Polarkoordinaten durchgeführt (siehe Gleichung 4.9). Dabei berücksichtigt die Funktion 'arctan2' im Unterschied zu 'arctan' die Vorzeichen beider Koordinaten und liefert daher über den gesamten Kreis den richtigen Winkel, wodurch eine eventuelle Korrektur um π entfällt.

$$\begin{aligned} \|\vec{v}\|_2 = x_n &= \sqrt{x_0^2 + y_0^2} \\ \phi = z_n &= \arctan2(y, x) \end{aligned} \tag{4.9}$$

In Abbildung 4.4 ist die Funktionsweise des CORDIC-Algorithmus anhand des Vektormodus für die ersten vier Iterationsschritte dargestellt.

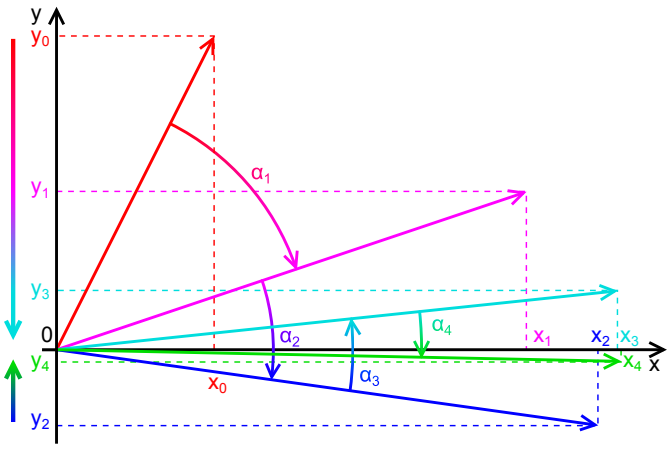

Abbildung 4.4: Vektormodus des CORDIC-Algorithmus

4.3.2 Modifikationen

Wie in [Vol59] bereits angemerkt, kann der Algorithmus nicht nur zur Berechnung trigonometrischer Funktionen herangezogen werden, sondern viel allgemeiner verwendet werden. Das ist möglich durch die Ausweitung des Algorithmus auf verschiedene Koordinatensysteme wie lineares bzw. hyperbolisches [HD07, Kapitel 25]. Für die Änderung auf das lineare Koordinatensystem wird der Kosinus in den Teilrotationen einfach durch eins und der Sinus durch null ersetzt. Ähnlich funktioniert es für das hyperbolische System, indem die trigonometrischen Funktionen durch deren hyperbolische Äquivalente ersetzt werden, was natürlich in beiden Fällen Auswirkungen auf den Dehnungsfaktor K hat. Zu beachten ist außerdem, dass auch das Vorzeichen des Sinus hyperbolicus positiv ist. In Gleichung 4.10 ist das Ergebnis der Veränderungen am Basisalgorithmus anhand eines Iterationschrittes gezeigt.

$$
\begin{aligned}
x_{i+1} &= x_i - m\sigma_i y_i 2^{-i} \\
y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\
z_{i+1} &= \begin{cases} z_i - \sigma_i \arctan(2^{-i}), & m = +1, \quad \text{zirkular} \\ z_i - \sigma_i 2^{-i}, & m = 0, \quad \text{linear} \\ z_i - \sigma_i \operatorname{artanh}(2^{-i}), & m = -1, \quad \text{hyperbolisch} \end{cases}
\end{aligned}
\tag{4.10}
$$

Dabei dient der Parameter m zur Auswahl des Koordinatensystems und kann nur die drei Werte $\{-1, 0, +1\}$ annehmen. Durch die Anwendung dieser Koordinatensysteme können nun unter der Annahme, dass nur zwei Variablen ungleich null initialisiert werden folgende weitere Funktionen realisiert werden:

- $x_n = \cosh(z_0)$ und $y_n = \sinh(z_0)$ (Rotationsmodus, $m = 1$)
- $x_n = \sqrt{x_0^2 - y_0^2}$ und $z_n = \operatorname{artanh}(\frac{y_0}{x_0})$ (Vektormodus, $m = 1$)
- $y_n = x_0 z_0$ (Rotationsmodus, $m = 0$)
- $z_n = \frac{y_0}{x_0}$ (Vektormodus, $m = 0$)

Durch eine andere Initialisierung der Eingangsgrößen können jedoch noch mehr Funktionen berechnet werden [Wal71]. Diese basieren alle auf dem hyperbolischen Koordinatensystem. So kann etwa der natürliche Logarithmus einer Zahl über die Areatangens hyperbolicus Funktion berechnet werden, weil Gleichung 4.11 gilt.

$$\begin{aligned} \tanh a &= \frac{\mathrm{e}^{2a}-1}{\mathrm{e}^{2a}+1} \\ \tanh\frac{\ln b}{2} &= \frac{b-1}{b+1}, & a &= \frac{\ln b}{2} \\ \Rightarrow \ln(b) &= 2\ \operatorname{artanh}\left(\frac{b-1}{b+1}\right), & x_0 &= b+1, \quad y_0 = b-1, \quad z_0 = 0 \end{aligned} \tag{4.11}$$

Auf ähnliche Art und Weise können auch Ausdrücke für die Wurzel- und Exponentialfunktion hergeleitet werden, wobei Gleichung 4.12 die Ergebnisse zeigt.

$$\begin{aligned} \mathrm{e}^{\theta} &= \cosh(\theta)+\sinh(\theta), & x_0 &= y_0 = 1, & z_0 &= \theta \\ \sqrt{w} &= \sqrt{x^2-y^2}, & x_0 &= w+\frac{1}{4}, \quad y_0 = w-\frac{1}{4}, & z_0 &= 0 \end{aligned} \tag{4.12}$$

4.3.3 Konvergenzbereich und dessen Erweiterung

Durch den Aufbau des Algorithmus, welcher im Wesentlichen versucht eine bestimmte Anfangsauslenkung durch eine Folge von Additionen und Subtraktionen auf null zu bringen, ergeben sich Einschränkungen auf die maximal mögliche Anfangsauslenkung. Das macht sich als Konvergenzbereichsbeschränkungen für die auszuführenden Operationen bemerkbar, welche auch im Fall unendlich vieler Iterationsschritte, beginnend von einem festgelegten Startindex, bestehen bleiben. Aufgrund der Tatsache, dass die Additionen und Subtraktionen abnehmende Größe haben, werden die Konvergenzbereiche hauptsächlich von den ersten paar Iterationen bestimmt, wie Tabelle 4.1 zeigt. In dieser Tabelle sind außerdem die Konvergenzbereiche für eine unendliche Anzahl an Iterationen und die für die Implementierung geltenden dargestellt. Als Einheit für das zirkulare Koordinatensystem wird Radiant verwendet, die anderen Konvergenzgrößen sind einheitenlos. Der Startindex gibt die Iterationsnummer i der ersten Iteration in Gleichung 4.10 an und ist entscheidend für den Konvergenzbereich.

Tabelle 4.1: Konvergenzbereichsbeschränkungen

Koordinatensystem	Startindex	maximale Konvergenz	Konvergenz 5 Iterationen	Konvergenz Implementierung
zirkular	0	≈ 1,7432866	≈ 1,6807982	≈ 1,7432864
linear	0	2,0	1,9375	≈ 1,9999998
hyperbolisch	1	≈ 1,1181730	≈ 1,0555393	≈ 1,1181725

Die in der Tabelle angegebenen Werte gelten dabei für den Betrag der Iterationsvariablen z und stellen sowohl Einschränkungen für den Rotationsmodus, als auch für den Vektormodus dar. Denn für den Vektormodus kann das Ergebnis nach Ausführung des CORDIC-Algorithmus den Konvergenzbereich von z nicht überschreiten. Dabei wurde eine Initialisierung von $z = 0$ angenommen. Somit ergeben sich teilweise auch Einschränkungen für die Konvergenz der beiden anderen Variablen x und y, die zu beachten sind.

Um den Konvergenzbereich zu erweitern gibt es nun mehrere Möglichkeiten. Eine Möglichkeit, welche z. B. in [HHB91] Anwendung findet, ist die Einführung weiterer Iterationen zu Beginn des Algorithmus. Das heißt, es werden zusätzliche Iterationen mit dem Index $i \geq 0$ eingeführt. Somit kann der Konvergenzbereich je nach Anforderung auf einen festen größeren Wert erhöht werden.

In der nachfolgend präsentierten Implementierung kommt dieses Verfahren nicht zum Einsatz. Stattdessen beschränken sich die Konvergenzbereichserweiterungen auf die wichtigeren Funktionen. Im Fall des zirkularen Koordinatensystems ergibt sich der Vorteil, dass der Winkel nur in einem Intervall von $[-\pi, \pi)$ eindeutig bestimmt ist. Jeder andere Winkel lässt sich durch eine einfache Modulo-Operation mit 2π auf dieses Intervall transformieren. Deshalb ist eine Ausweitung des Konvergenzbereichs auf $\pm\pi$ völlig ausreichend. Da der Standardalgorithmus bereits Winkel von $\pm\frac{\pi}{2}$ unterstützt, muss für den Vektormodus nur noch der Fall der negativen x-Koordinate behandelt werden. Das geschieht einfach indem der Vektor vor Ausführung des CORDIC-Algorithmus um den halben Kreis, also um π gedreht wird [And98]. Dazu sind lediglich die Vorzeichen der beiden Koordinaten zu invertieren und zum Ergebniswinkel des Algorithmus ist der Winkel π zu addieren. Für den Rotationsmodus gestaltet sich das Ganze etwas schwieriger, denn hier ist der Winkel nicht eindeutig und die Modulo-Operation muss explizit durchgeführt werden. Am einfachsten geschieht dies, wenn der Winkel zuerst in eine andere Zahlenrepräsentation übergeführt wird. Diese Winkelrepräsentation ist ähnlich wie in [Dag59], wobei sich die Bits auf den Halbkreis beziehen. Das heißt, das Vorkommabit steht für π und die Nachkommabits für $\frac{\pi}{2}$, $\frac{\pi}{4}$, usw. Als Unterschied zu [Dag59] wird nun dem ersten Vorkommabit nicht π, sondern $\frac{\pi}{2}$ zugewiesen. Somit kann aus den ersten zwei Vorkommabinärstellen einfach der Quadrant des Winkels abgelesen werden und durch Vernachlässigung der anderen Vorkommastellen erfolgt die Operation modulo 2π. Dadurch ist es möglich die Berechnung der Winkelfunktionen auf dem ersten Quadranten durchzuführen. Nach der Ausführung des Algorithmus können durch entsprechende Vertauschungen von Vorzeichen und CORDIC-Ausgängen Sinus und Kosinus korrekt zurückgeliefert werden.

Für das lineare Koordinatensystem ergibt sich eine sehr simple Vorgehensweise. Wenn für den CORDIC-Algorithmus von normalisierten Zahlen ausgegangen wird, bei denen immer das erste Bit mit der Wertigkeit eins gesetzt ist, ist der Konvergenzbereich aus Tabelle 4.1 für alle beliebigen Eingangskombinationen völlig ausreichend. Dazu muss im Vergleich zu mancher Literatur wie z. B. auch in [HD07, Kapitel 25] eine zusätzliche Iteration mit dem Index $i = 0$ durchgeführt werden, damit der Konvergenzbereich gleich zwei ist. Die Einschränkung auf normalisierte Zahlen stellt dabei keine wirkliche Einschränkung dar, da die Mantisse des IEEE754 Gleitkommazahlenformats bereits normalisiert ist, sofern Spezialfälle ausgenommen werden. Die Beschreibung dieses Formats ist im Anhang Kapitel A.1 zu finden.

Da sich eine allgemeine Konvergenzbereichserweiterung beim hyperbolischen Koordinatensystem schwierig gestaltet, wird nur die davon abgeleitete Exponentialfunktion behandelt. Während die direkten Funktionen sehr enge Einschränkungen haben, gilt das nicht für die Logarithmus- und Wurzelfunktion. Aufgrund der speziellen Variableninitialisierung ergeben sich hier bei Verwendung der Mantisse des Gleitkommazahlenformats praktisch keine Einschränkungen. Probleme gibt es wie auch bei der Division lediglich beim Vorhandensein denormalisierter Zahlen. Das sind Zahlen, die mit dem Zahlenformat nicht mehr in voller Auflösung dargestellt werden können. Die für die Verwendung des Gleitkommaformats benötigte Vor- und Nachbearbeitung wird bei der Implementierung Kapitel 4.3.5 behandelt. Bei der Exponentialfunktion ist jedoch eine Konvergenzbereichserweiterung sehr wichtig, da das Argument der Funktion für praktische Zwecke oft außerhalb des Bereichs liegt. Dabei kommt zu Gute, dass die Form der Exponentialfunktion unabhängig vom Startpunkt des Exponenten ist. Dadurch kann vom Exponenten wie in Gleichung 4.13

ein konstanter Wert addiert oder subtrahiert werden und als multiplikativer konstanter Vorfaktor berücksichtigt werden.

$$e^x = e^{x-c+c} = e^{x-c}e^c = Ce^{x-c}, \qquad C = e^c, \qquad C, c ...\text{Konstante} \tag{4.13}$$

Somit besteht die Aufgabe der Konvergenzbereichserweiterung nur mehr in der konkreten Auswahl dieser Konstanten um eine einfache Automation in Hardware zu ermöglichen. Um auf eine Form mit Zweierpotenzen zu kommen, wird eine Modulooperation mit $\ln(2)$ benötigt (siehe Gleichung 4.14). Die Modulooperation kann dabei durch eine Multiplikation mit $\frac{1}{\ln(2)}$ und anschließendem Trennen von Vor- und Nachkommabits erreicht werden. Dabei ist allerdings zu beachten, dass sowohl die Mantisse von Gleitkommazahlen, als auch $\frac{1}{\ln(2)}$ größer als eins sind, was im Algorithmus zu berücksichtigen ist um im Konvergenzbereich zu bleiben.

$$\frac{x}{\ln(2)} = k + \frac{x'}{\ln(2)}, \qquad k \in \mathbb{Z}, \qquad x' ...\text{Divisionsrest} \tag{4.14}$$

$$\Rightarrow e^x = e^{x'+k\ln(2)} = e^{x'}e^{k\ln(2)} = 2^k e^{x'} \tag{4.15}$$

Der aus Gleichung 4.15 resultierende Vorfaktor kann nun direkt im Gleitkommaformat interpretiert werden, wobei k der Exponent ist. Die verbleibenden Operationen zur korrekten Umwandlung in das Gleitkommazahlenformat sind die selben wie bei allen anderen CORDIC-Funktionen auch.

4.3.4 Berechnungsgenauigkeit

Da neben des iterativen Charakters des CORDIC-Algorithmus auch die endliche Rechengenauigkeit numerischer Operationen für die Genauigkeit der Implementierung maßgeblich ist, widmet sich der folgende Abschnitt diesem Thema.

Es existieren zwar zahlreiche theoretische Überlegungen zu diesem Problem, wie etwa auch in [HD07, Kapitel 25], allerdings reagieren verschiedene CORDIC-Funktionen unterschiedlich auf die Parameter des Algorithmus. Die Algorithmusparameter sind dabei die Anzahl an Iterationen und die Wortbreite für arithmetische Operationen. Außerdem soll ein guter Kompromiss zwischen Genauigkeit, Hardwareaufwand und Rechenzeit gefunden werden. Aus diesem Grund wurden auf einem selbst erstellten Simulator Hardwaresimulationen durchgeführt, in denen die vorhin genannten Parameter variiert werden. Das Ziel war eine Berechnungsgenauigkeit, die annähernd in der gleichen Größenordnung liegt wie die des verwendeten Ein- und Ausgangsdatentyps. Dieser Datentyp ist das IEEE754 Gleitkommazahlenformat mit einfacher Genauigkeit, was einer relativen Auflösung von 24 Bit entspricht. Dafür wären laut [HD07, Kapitel 25] 27 Iterationen mit einer Wortlänge von etwas über 32 Bit erforderlich. Die durchgeführten Simulationen zeigen jedoch, dass mit wesentlich geringerem Aufwand eine ähnliche Genauigkeit erreicht werden kann. So wurden die Parameter für die Implementierung mit globalen Konstanten in der Hardwarebeschreibung auf 23 Iterationen mit 27 Bit Auflösung festgesetzt. Dadurch ist es auch einfach möglich, bei Bedarf diese Parameter vor der Implementierung noch zu verändern. Abbildung 4.5 zeigt dazu Simulationsergebnisse für den mittleren relativen Fehler der Exponentialfunktion. Diese Form des Berechnungsfehlers in Abhängigkeit der CORDIC-Parameter ist repräsentativ für die meisten

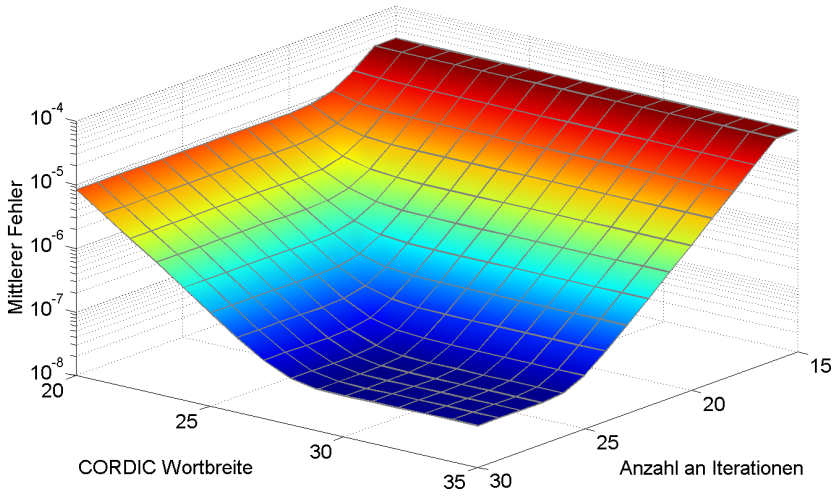

Abbildung 4.5: Simulationsergebnis des mittleren relativen Fehlers der Exponentialfunktion

CORDIC-Funktionen. Allerdings sind sämtliche Wurzelfunktionen wesentlich robuster gegen eine Reduktion der Iterationsschritte.

Die resultierenden mittleren und maximalen Fehler der CORDIC-Funktionen wurden mit jeweils zehn Millionen Simulationsläufen ermittelt und sind in Tabelle 4.2 dargestellt. Die Funktionen sind bereits die der Implementierung und deshalb als Funktion der Eingangsoperanden in_1 und in_2 angegeben. Dabei ist der Fehler teilweise je nach Aussteuerbereich der Funktionen als relativer f_{rel} oder absoluter Fehler f_{abs} angegeben (siehe Gleichung 4.16), was auch in der Tabelle vermerkt ist. Zu beachten ist hier, dass der verwendete Datentyp eine relative Genauigkeit von ca. $1{,}2 \cdot 10^{-7}$ hat und als exakte Lösungen Berechnungen mit doppelter, also 64 Bit Gleitkommagenauigkeit verstanden werden. Der Vollständigkeit halber wird auch der Konvergenzbereich der Funktionen angeführt.

$$
\begin{aligned}
f_{abs} &= |out_{soll} - out_{ist}|, \\
f_{rel} &= \left| \frac{out_{soll} - out_{ist}}{out_{soll}} \right|,
\end{aligned}
\qquad
\begin{aligned}
&out_{soll}\text{...exaktes Ergebnis} \\
&out_{ist}\text{....CORDIC Ergebnis}
\end{aligned}
\tag{4.16}
$$

Während die meisten Funktionen sowohl beim mittleren, als auch beim maximalen Fehler ein gutes Verhalten zeigen, stechen manche Einträge aufgrund extrem schlechter minimaler Genauigkeit hervor. Bei der Exponentialfunktion und der Multiplikation liegt das an der Verwendung von denormalisierten Zahlen (siehe Kapitel A.1). Diese sind so klein, dass auch der kleinste Exponent des Zahlenformats nicht mehr zur Darstellung ausreicht und daher die Mantisse nach links verschoben wird. Dadurch verringert sich die relative Auflösung und ist im schlechtesten Fall sogar nur mehr ein Bit, was beim kleinsten Rundungsfehler bereits eine Abweichung von 100% verursacht. Genau dies ist abgeschwächter Form bei den beiden vorhin genannten Funktionen teilweise passiert und hat in weiterer Folge auch Auswirkungen auf den mittleren Fehler.

Funktionen, die ebenfalls aufgrund ihres maximalen Fehlers ins Auge stechen sind die Logarithmusfunktionen. Dabei sind deren große Abweichungen nicht durch das Zahlenformat bedingt, da denormalisierte Zahlen hier ausgenommen sind. Der Hauptgrund liegt in ihrer Initialisierung (siehe Gleichung 4.11). Ist die Mantisse des Eingangsoperanden sehr nahe bei eins, so sind die

Tabelle 4.2: Abweichung der CORDIC-Ergebnisse von den exakten Werten und Konvergenzbereich

Funktion	**mittlerer Fehler**	**maximaler Fehler**	**Art des Fehlers**	**Konvergenzbereich**
$in_1 \cos(in_2)$	$9{,}3582{\cdot}10^{-8}$	$7{,}1526{\cdot}10^{-7}$	$\frac{1}{in_1} f_{abs}$	gesamter Zahlenbereich
$in_1 \sin(in_2)$	$9{,}3597{\cdot}10^{-8}$	$7{,}1526{\cdot}10^{-7}$	$\frac{1}{in_1} f_{abs}$	gesamter Zahlenbereich
$\arctan2(in_2, in_1)$	$1{,}8720{\cdot}10^{-7}$	$8{,}0268{\cdot}10^{-7}$	f_{abs}	gesamter Zahlenbereich
$\sqrt{in_1^2 + in_2^2}$	$5{,}1306{\cdot}10^{-8}$	$2{,}3842{\cdot}10^{-7}$	f_{rel}	gesamter Zahlenbereich
$in_1 in_2$	$2{,}3190{\cdot}10^{-7}$	$0{,}2500^*$	f_{rel}	gesamter Zahlenbereich
$\frac{in_2}{in_1}$	$1{,}2650{\cdot}10^{-7}$	$7{,}2776{\cdot}10^{-7}$	f_{rel}	normalisierte Zahlen
$in_1 \cosh(in_2)$	$9{,}8530{\cdot}10^{-8}$	$7{,}9980{\cdot}10^{-7}$	$\frac{1}{in_1} f_{abs}$	$z_{0,max} = 1{,}1182$
$in_1 \sinh(in_2)$	$2{,}5693{\cdot}10^{-7}$	$1{,}0664{\cdot}10^{-6}$	$\frac{1}{in_1} f_{abs}$	$z_{0,max} = 1{,}1182$
$\operatorname{artanh}(\frac{in_2}{in_1})$	$2.4217{\cdot}10^{-7}$	$9.5367{\cdot}10^{-7}$	f_{abs}	$\left(\frac{y_0}{x_0}\right)_{max} = 0{,}8069$
$\sqrt{in_1^2 - in_2^2}$	$4{,}8417{\cdot}10^{-8}$	$6{,}4717{\cdot}10^{-7}$	f_{rel}	$\left(\frac{y_0}{x_0}\right)_{max} = 0{,}8069$
$\sqrt{\lvert in_1 \rvert}$	$6{,}4904{\cdot}10^{-8}$	$5{,}0482{\cdot}10^{-7}$	f_{rel}	normalisierte Zahlen**
$\mathrm{ld}(\lvert in_1 \rvert)$	$1{,}1355{\cdot}10^{-7}$	$0{,}0527$	f_{rel}	normalisierte Zahlen**
$\ln(\lvert in_1 \rvert)$	$1{,}1921{\cdot}10^{-7}$	$0{,}0527$	f_{rel}	normalisierte Zahlen**
$\log(\lvert in_1 \rvert)$	$1{,}1725{\cdot}10^{-7}$	$0{,}0527$	f_{rel}	normalisierte Zahlen**
$in_1 \mathrm{e}^{in_2}$	$2{,}5062{\cdot}10^{-7}$	$0{,}0034^*$	f_{rel}	gesamter Zahlenbereich

*Genauigkeitsverlust wegen denormalisierten Zahlen
**teilweise auch denormalisierte Zahlen

Abweichungen von eins nur mehr mit sehr wenigen Bits codiert. Die y-Variable des CORDIC-Algorithmus wird daher nur mit sehr schlechter Genauigkeit initialisiert, während der Wert der x-Variable um viele Größenordnungen höher ist. Dadurch dass die ohnehin schon geringe Auflösung im Laufe des Algorithmus immer weiter nach links verschoben und gerundet wird, ergibt sich ein relativ ungenaues Ergebnis. Der zweite Grund für den ungewöhnlich hohen relativen Fehler ist die geringe Größenordnung des exakten Ergebnisses, denn die Logarithmusfunktion liefert für Argumente sehr nahe bei eins ein Ergebnis, das sehr nahe bei null ist. Das kann auf einfache Weise durch die Taylorreihenentwicklung des natürlichen Logarithmus um den Punkt $x = 1$ nachvollzogen werden, wie Gleichung 4.17 zeigt. Genau durch diesen extrem kleinen exakten Ergebniswert wird der ohnehin schon relativ große absolute Fehler noch einmal verstärkt.

$$\mathrm{taylor}(\ln(x), x)|_{x=1} = x - 1, \qquad x \to 1 \Rightarrow \ln(x) \to 0 \tag{4.17}$$

4.3.5 Implementierung

Die Hardwareimplementierung des CORDIC-Algorithmus beinhaltet alle oben genannten Funktionen und Eigenschaften. Die Hardware führt 23 Iterationen mit einer Breite von 27 Bit aus. Obwohl der Algorithmus selbst mit dem Fixkommazahlenformat arbeitet, sind Eingangs- und Ausgangsgrößen jeweils im Gleitkommaformat codiert. Das hat neben der Eigenschaft der normalisierten Zahlen auch den Vorteil, dass ein wesentlich höherer Dynamikbereich unterstützt wird. Davon profitieren vor allem die Exponential-, Wurzel- und Logarithmusfunktion. Vom Algorithmus selbst wird dabei jeweils nur die Mantisse verarbeitet, alles andere wird in der Initialisierungsphase bzw. in der Nachbearbeitung erledigt.

Die Implementierung besitzt eine eigene 31-stufige Pipeline, welche aus coarse-grain rekonfigurierbarer Logik aufgebaut ist. Die Rekonfiguration findet dabei auf Pipeline-Ebene statt. Außerdem ist die Erweiterung im Stande parallel zur Rekonfiguration einen Befehl auszuführen. Das funktioniert indem die für einen Befehl benötigte Konfiguration mit diesem durch die Pipeline wandert und entsprechende Multiplexer schaltet. Somit wird maximaler Durchsatz ohne zusätzlichen Zeitaufwand für die Rekonfiguration garantiert. Die Konfiguration kann dabei entweder über die unteren 5 Bit des Spezialbefehls übergeben werden oder aus dem lokalen Speicher geladen werden. Für die zweite Variante muss dieser allerdings zuerst per Spezialbefehl mit einem Inhalt eines Prozessorregisters beschrieben werden.

Der prinzipielle Aufbau der CORDIC-Erweiterung ist in Abbildung 4.6 dargestellt. Die Kontrolleinheit kümmert sich dabei um die Kommunikation mit der Schnittstelle und verwaltet den lokalen Konfigurationsspeicher. Sie generiert auch die Konfigurationsdaten für die Pipeline.

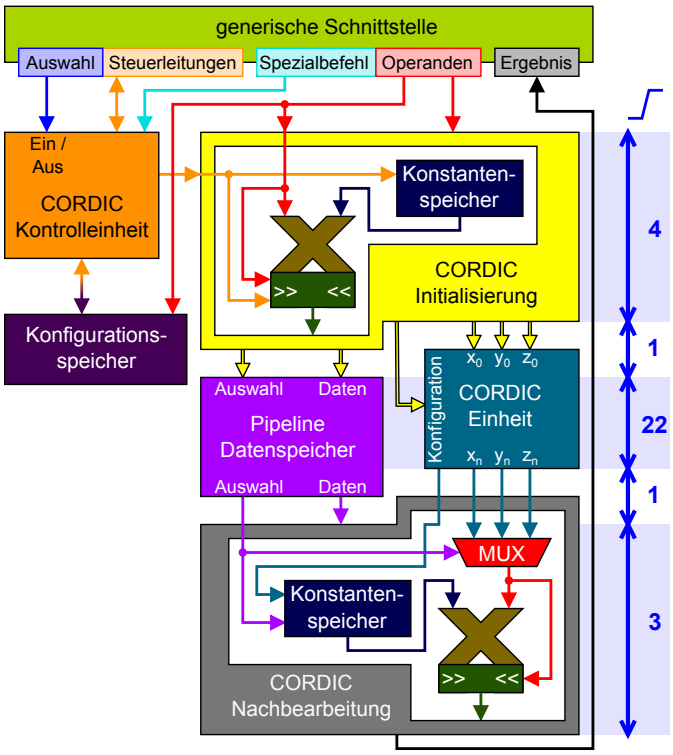

Abbildung 4.6: Aufbau der CORDIC-Erweiterung

Der Initialisierungsblock besetzt die ersten vier Pipeline-Stufen des Hauptdatenpfades bzw. die ersten fünf Pipeline-Stufen des Nebenpfades. In diesem Block findet die Decodierung der Eingangsdaten im IEEE754 Gleitkommazahlenformat mit einfacher Genauigkeit statt. Dabei wird das Zahlenformat fast vollständig unterstützt, bis auf die Unterscheidung von unendlich und NAN (**N**ot **A** **N**umber). Hier findet auch die Aufspaltung in Haupt- und Nebendatenpfad statt, wobei die Mantissen im Haupt und die Exponenten im Nebendatenpfad verarbeitet werden. Die wichtigste Aufgabe des Initialisierungsblocks ist die korrekte Initialisierung der folgenden CORDIC-Einheit, wofür die von der Kontrolleinheit übernommene Konfiguration zum Einsatz kommt. Außerdem generiert sie die vereinfachte Konfiguration des CORDIC-Algorithmus. Die wichtigste Komponente in dieser Einheit ist ein Multiplizierer. Dieser ist für die Konvergenzbereichserweiterungen zuständig, indem die Zahlen so skaliert werden, dass die erforderlichen Moduloopera-

tionen einfach per Bitselektion durchgeführt werden können. Die Multiplikation wird aber auch bei allen anderen Funktionen des gleichen Koordinatensystems verwendet, damit überall die selbe Skalierung zum Einsatz kommt. Das Multiplikationsergebnis muss anschließend wieder über Schiebeoperationen für den CORDIC-Algorithmus ausgerichtet und auf die passende Wortlänge verkürzt werden. Das entspricht einer Rundung zur nächst kleineren Zahl.

Die nächste Einheit des Hauptdatenpfades ist die CORDIC-Einheit. Sie implementiert 23 Iterationen von Gleichung 4.10 wofür auch 23 Pipeline-Stufen benötigt werden. Als einzige Konfigurationsparameter kommen dabei der Parameter m aus der Gleichung und der CORDIC-Modus zum Einsatz, welche mit insgesamt 3 Bit codiert sind und nacheinander auf die Pipeline-Stufen angewendet werden. Die Iterationsfolge startet beim zirkularen und linearen Koordinatensystem beim Index $i = 0$ und beim hyperbolischen bei eins. Zu beachten ist, dass beim hyperbolischen Modus die Iterationsschritte 4 und 13 wiederholt werden müssen, um Konvergenz zu garantieren [Wal71]. Dadurch müssen in manchen Pipeline-Stufen je nach Konfiguration x und y verschieden weit nach rechts geschoben werden. Jede Pipeline-Stufe hat ihre eigene LUT für die zu addierenden z-Werte, wobei es nur drei verschiedene Einträge gibt. Das wird durch die Multiplikationen in der Initialisierungsstufe erreicht, welche für alle Funktionen eines Koordinatensystems den gleichen Skalierungsfaktor verwendet. So sind alle Einträge der LUT für das zirkulare Koordinatensystem mit $\frac{2}{\pi}$ skaliert und die des hyperbolischen mit $\frac{1}{\ln(2)}$. Der Aufbau einer solchen Pipeline-Stufe ist in Abbildung 4.7 veranschaulicht. Die Rundungsmethode bei den Schiebe- und Addieroperationen ist dabei eine symmetrische, um maximale Rechengenauigkeit bei gegebener Wortlänge zu erreichen. Dazu wird der Carry-in Eingang der Addierer bzw. Subtrahierer verwendet.

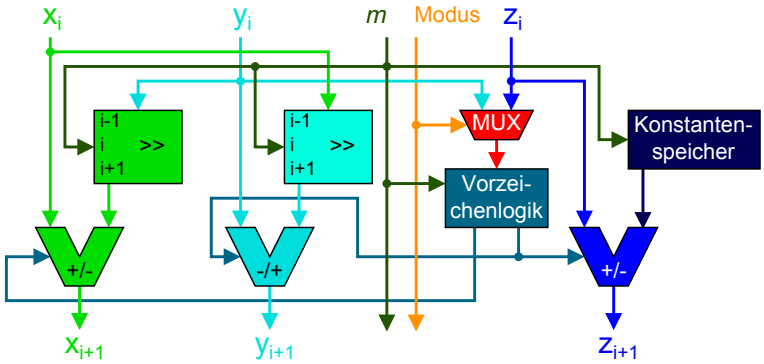

Abbildung 4.7: Aufbau eines CORDIC-Iterationsschrittes

Um die Daten des Nebendatenpfades temporär zu speichern befindet sich in diesem Datenpfad parallel zur CORDIC-Prozessierung ein kleiner Speicher. Dieser speichert die Adresse des Ergebnisregisters, die CORDIC-Ergebnisauswahl, den Exponenten und das Vorzeichen des Ergebnisses.

Der letzte Funktionsblock in der CORDIC-Pipeline ist die Nachbearbeitung mit vier Pipeline-Stufen. Hier werden Haupt- und Nebendatenpfad wieder zusammengeführt und das Endergebnis produziert. Um die Skalierung des CORDIC-Algorithmus und der Initialisierungseinheit zu kompensieren, befindet sich auch in diesem Block ein Multiplizierer zum Rückskalieren des CORDIC-Ergebnisses. Dieser Multiplizierer wird auch dazu verwendet, abgeleitete Logarithmusfunktionen wie Zehner- und Zweierlogarithmus bereitzustellen. Weiters muss das im Festkommaformat codierte Ergebnis noch auf die Gleitkommadarstellung transformiert werden, was eine Schiebeoperation für die Mantisse und eine Addition für den Exponenten erfordert. Dabei wird das IEEE-Format wieder bis auf die Unterscheidung von unendlich und NAN vollständig unterstützt.

Die Mantisse wird allerdings nur aus dem CORDIC-Ergebnis ausgeschnitten, wodurch hier eine Rundung die nächst kleinere darstellbare Zahl stattfindet. Das resultierende Ergebnis wird anschließend mit der Adresse des Ergebnisregisters direkt der Schnittstelle des Prozessors übergeben.

Da die Pipeline der CORDIC-Erweiterungseinheit nicht für Stopps ausgelegt ist und daher immer durcharbeitet, muss die Erweiterung immer Schreibzugriff auf das Registerfile des Prozessors bekommen. Das impliziert, dass sie die Erweiterung mit der höchsten Priorität sein muss und deshalb vorzugsweise an den Erweiterungsanschluss mit der Nummer null anzuschließen ist.

4.4 Konvertierung des Zahlenformats

Die Erweiterung für die Zahlenformatkonvertierung wurde aus zwei Gründen für die Implementierung ausgewählt. Zum einen verfügt der Basisprozessor nur über Festkommaarithmetik, weshalb für die Nutzung der CORDIC-Erweiterung eine Konvertierung erforderlich ist und zum anderen kann die Konvertierung in der gleichen Länge wie die Prozessor-Pipeline implementiert werden. Somit stellt diese Erweiterung ein Beispiel für eine zur Prozessor-Pipeline synchrone Erweiterung dar.

Die Aufgabe der Erweiterung ist es also Konvertierungsfunktionen zwischen Gleitkomma- und Festkommaformat bereitzustellen. Als Besonderheit kann beim Festkommaformat das Komma an jeder beliebigen Stelle in Bezug auf das Gleitkommaformat liegen. Diese Position kann entweder mit dem Spezialbefehl mitangegeben werden oder ähnlich wie bei der CORDIC-Erweiterung konfiguriert werden. Die Konfiguration erfolgt dabei ebenfalls über einen Spezialbefehl und einen Registerinhalt, welcher als zweiter Operand angegeben wird. Zu beachten ist, dass diese Art der Konfiguration erst mit einem Takt Verzögerung wirksam wird, wodurch ein eventuell direkt darauf folgender Befehl noch die alte Konfiguration verwendet. Wird die Positionen des Kommas jedoch mit dem Spezialbefehl übergeben, ergibt sich keine Verzögerung. Für die Einstellung des Dezimalpunkts gilt, dass beim Positionswert null das Komma gleich nach dem Vorzeichenbit steht. Es wird also standardmäßig von der I.Q-Darstellung 1.31 ausgegangen. Dabei gibt I die Anzahl an Vorkommabits und Q die der Nachkommabits an [Dob04, Kapitel 4.1.1]. Ein positiver Wert verschiebt das Komma der Festkommarepräsentation nach rechts und bei einem negativen Wert ist es umgekehrt. Das heißt, ein positiver Wert der Kommaposition bedeutet auch einen größeren Exponenten der umgewandelten Gleitkommazahl.

Der Aufbau der Erweiterung ist wieder durch eine Pipeline gekennzeichnet und ist in Abbildung 4.8 dargestellt. Die Erweiterung macht auch davon Gebrauch, dass der Spezialbefehl schon vor der Erweiterungsauswahl und den Eingangsoperanden verfügbar ist. So kann in einer vorbereitenden Pipeline-Stufe bereits eine Anpassung des Exponenten auf die Gleitkommadarstellung erfolgen, welche über einen konstanten Offset verfügt.

Um die weitere Funktionsbeschreibung zu vereinfachen, wird im Folgenden zuerst von einer Konvertierung des Festkommaformats in das Gleitkommaformat ausgegangen. In der ersten Pipeline-Stufe des Datenpfades erfolgt die Vorzeichendetektion und eine anschließende Betragsbildung über die Bildung des Zweierkomplements. Außerdem wird die ungefähre Position des ersten gesetzten Bits ermittelt. In der nächsten Pipeline-Stufe wird das Datum ungefähr auf die Mantisse des Gleitkommaformats ausgerichtet und der Exponent anhand der Vorberechnung und der Position des ersten gesetzten Bits berechnet. Die letzte Stufe der Pipeline beinhaltet nur mehr kombinatorische Logik, welche direkt mit der Schnittstelle zum Prozessor verbunden ist. Diese

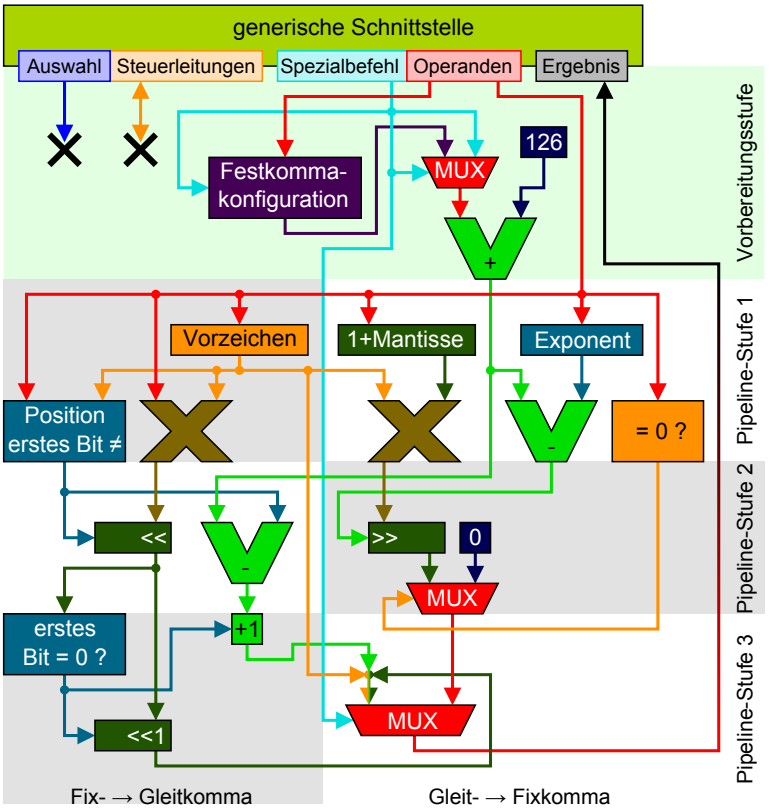

Abbildung 4.8: Aufbau der Erweiterungseinheit zur Zahlenformatkonvertierung

Logik kümmert sich um die exakte Ausrichtung der Mantisse und korrigiert bei Bedarf den Exponenten. Außerdem wird die Gleitkommazahl bestehend aus Vorzeichen, Exponent und Mantisse zusammengesetzt.

Nun wird der umgekehrte Fall einer Konvertierung vom Gleitkomma- in das Festkommaformat betrachtet. In der ersten Pipeline-Stufe wird wieder das Vorzeichen ermittelt und bei einer negativen Eingangszahl das Zweierkomplement der Mantisse gebildet. Außerdem wird eine Überprüfung auf die Zahl null durchgeführt, da diese eine denormalisierte Zahl und somit einen wichtigen Sonderfall darstellt. Währenddessen wird ermittelt, wie weit die Mantisse für die gewünschte Darstellung verschoben werden muss. Die zweite Pipeline-Stufe führt diese Verschiebung dann aus und liefert bereits das Endergebnis, das nur noch um das Vorzeichen in der letzten Stufe ergänzt wird.

Eine Kommunikation mit dem Prozessor ist nicht erforderlich, da die Pipeline der Erweiterung synchron zur Prozessor-Pipeline läuft. Deshalb sind auch Probleme mit dem Schreibzugriff auf das Registerfile ausgeschlossen und die Pipeline muss niemals angehalten werden. Lediglich die Adresse des Zielregisters muss in einem Schieberegister temporär gespeichert und gleichzeitig mit dem Ergebnis der Schnittstelle bereitgestellt werden. Im Gegensatz zur CORDIC-Erweiterung verfügt diese Erweiterung nicht über Sonderbehandlungen für zu große und denormalisierte Zahlen mit Ausnahme der Null. Der Grund dafür ist die Annahme, dass Festkommazahlen für solche Extremwerte praktisch nie zum Einsatz kommen.

4.5 Digitales Filter

Das digitale Filter ist die letzte implementierte Spezialerweiterungseinheit. Sie verfügt wie die zuvor behandelte CORDIC-Erweiterung über rekonfigurierbare coarse-grain Logik. Um einen Überblick über die Funktionsweise zu bekommen, wird zuerst der Aufbau der Hardware erklärt. Anschließend wird nach einem kurzen theoretischen Teil auf die rekonfigurierbare Filterstruktur und deren Steuerung eingegangen.

4.5.1 Aufbau

Der Aufbau dieser Einheit ist in Abbildung 4.9 dargestellt. Da das Filter im Gegensatz zu den beiden vorhergehenden Einheiten nicht über eine Pipeline verfügt muss die Hauptkontrolleinheit neben der Kommunikation mit der Prozessorschnittstelle auch die Ablaufsteuerung übernehmen. Der Vorteil dieses Aufbaus ist, dass das Filter so wesentlich flexibler und daher auch besser zu konfigurieren ist. Außerdem werden dadurch wesentlich weniger Hardwareressourcen für die Implementierung benötigt, mit dem Nachteil, dass immer nur ein Filterbefehl zu einem Zeitpunkt verarbeitet werden kann. Weitere Aufgaben der Kontrolleinheit sind die Speicherung der aktuellen Konfiguration und das Abarbeiten von Konfigurationsbefehlen, was somit parallel zur Filterung erfolgen kann. Diese Abarbeitung erfolgt mit der Prozessortaktfrequenz, sodass jeder Konfigurationsbefehl genau einen Taktzyklus braucht. Dazu verfügt die Kontrolleinheit über drei Befehlsregister, die maximal zwei Befehle aufnehmen können.

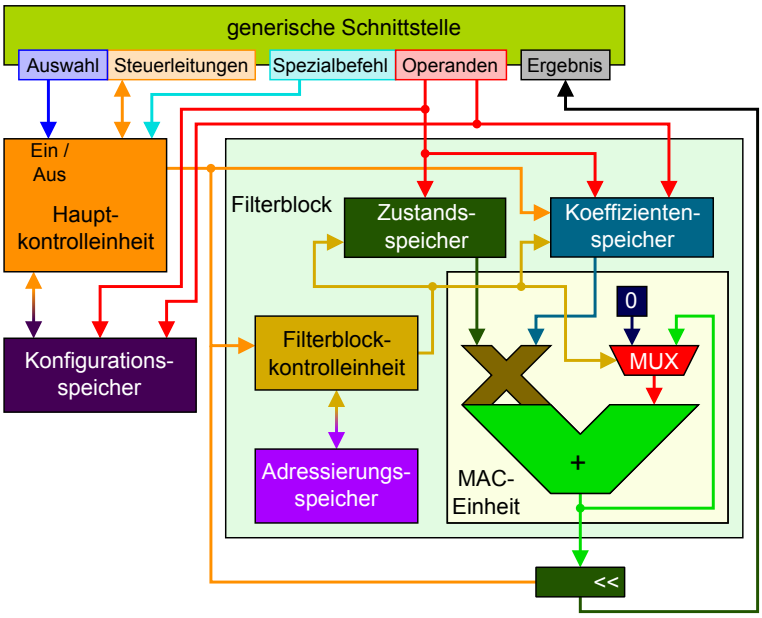

Abbildung 4.9: Aufbau der digitalen Filter-Erweiterungseinheit

Zum eigentlichen Filterblock - das ist der Teil der Erweiterung, der sich nur um die Filterung kümmert - gehört eine weitere Kontrolleinheit für die Detailablaufsteuerung, eine Ausführungseinheit und drei lokale Speicher. Die Ausführungseinheit besteht aus einer einzigen MAC-Einheit (**M**ultiply and **AC**cumulate), welche um den Durchsatz zu erhöhen über eine dreistufige Pipeline verfügt. Diese wird bei der Befehlsabarbeitung so gesteuert, dass der Multiplizierer ständig mit Daten versorgt wird und somit jeden Taktzyklus eine MAC-Operation ausgeführt wird. Aufgrund

der Einfachheit dieses Aufbaus ist es zumindest für die FPGA-Implementierung möglich, die Taktfrequenz der MAC-Einheit zu verdoppeln, wodurch die Filterzeit halbiert wird. Die Wortbreite für die Multiplikation ist dabei im Gegensatz zu der des Prozessors nur 18 Bit , was für viele Anwendungen ausreichend ist, wie kommerziell verfügbare DSPs wie z. B. [1, 8] zeigen. Um jedoch bei dem akkumulierten Multiplikationsergebnis keine Genauigkeit zu verlieren und Zahlenüberläufe zu vermeiden, arbeitet der Akkumulator mit 48 Bit .

Damit die Ausführungseinheit jederzeit mit genug Daten versorgt werden kann, ist sie direkt an zwei Speicher angeschlossen. Der eine Speicher enthält dabei die Filterkoeffizienten, während der andere die Filterzustände enthält. Somit stehen jeden Taktzyklus zwei Eingangsoperanden für die Multiplikation zur Verfügung. Die Verwaltung und Adressierung dieser Speicher übernimmt die Kontrolleinheit des Filterblocks, welche zusätzlich noch über einen weiteren lokalen Speicher verfügt. Dieser Speicher beinhaltet Adress-Offsets, um die Adressierung des Zustandsspeichers effektiv zu ermöglichen. Denn eigentlich wandern die Filterzustände durch die Filterstruktur durch, was ein Schieberegister anstatt eines einfachen Speichers erfordern würde. Da Speicher jedoch wesentlich effizienter zu implementieren sind, findet die Adressierungsvariante Anwendung. Neben der Speicherverwaltung übernimmt die Kontrolleinheit des Filterblocks auch dessen Ablaufsteuerung, wozu die Änderung der Adress-Offsets gehört. Ebenfalls dazu gehört die Aufnahme und Speicherung eines neuen Abtastwertes bzw. neuer Filterkoeffizienten, die Rücklieferung des Filterergebnisses und die Kommunikation mit der Hauptkontrolleinheit. Der Ablauf der Filterung ist dabei zur Laufzeit konfigurierbar, sodass das Filter je nach Konfiguration entweder als FIR- (**F**inite **I**mpuls **R**esponse) oder als IIR-Filter (**I**nfinite **I**mpulse **R**esponse) arbeitet. Diese Konfiguration wird dabei mit dem Spezialbefehl übergeben und kommt jeweils sofort zum Einsatz.

Außerhalb des Filterblocks befindet sich noch ein Shifter, der das Filterergebnis um maximal sieben Binärstellen verschieben kann, um den 48 Bit breiten Akkumulatorausgang optimal auf die 32 Bit Wortbreite des Prozessors auslegen zu können.

4.5.2 FIR- und IIR-Filter

Die Ablaufsteuerung des Filterblocks lässt zwei Filterstrukturen, FIR und IIR zu. Dabei ist die Anzahl der MAC-Operationen für eine Filterung im Filterblock auf fünf beschränkt, um bei IIR-Filtern starke Abweichungen der Filtercharakteristik durch die endliche Wortlängeneffekte zu verhindern [Dob07, Kapitel 6.4]. Abbildung 4.10 zeigt den Aufbau dieser zwei Filterarten unter der genannten Einschränkung, wodurch nur vier Speicherelemente für den Filterzustand gebraucht werden. Dabei gibt z^{-1} eine Verzögerung um einen Taktzyklus an und die dargestellten arithmetischen Operationen sind die MAC-Operationen der Implementierung. Die Multiplikationskoeffizienten sind die Filterkoeffizienten und die verzögerten Abtastwerte des Eingangssignals bzw. der Rückkopplung sind die gespeicherten Filterzustände. Vom jeweiligen Aufbau lassen sich die Namensgebungen der Filter ableiten. Da das FIR-Filter über eine einfache Schiebekette verfügt, klingt das Filterausgangssignal bei einer endlichen Anregung des Eingangs auch in endlicher Zeit wieder ab. Beim IIR-Filter ist das aufgrund der Rückkopplung nicht der Fall, kann jedoch trotzdem aufgrund der endlichen numerischen Genauigkeit passieren. Genau diese Rückkopplung ist auch der Grund, warum es bei IIR-Filtern zu Instabilitäten kommen kann. Außerdem können sich so durch die Kombination aus Rückkopplung und endlicher numerischer Genauigkeit auch bei einem an sich stabilen Filter dauerhafte Schwingungen ergeben [Dob04, Kapitel A.7.3].

Bei der Hardwareimplementierung ist zu beachten, dass für die Rückkopplung wie auch für den Filtereingang nur 18 Bit Werte vorgesehen sind. Für die Rückkopplung wird daher aus dem 48 Bit

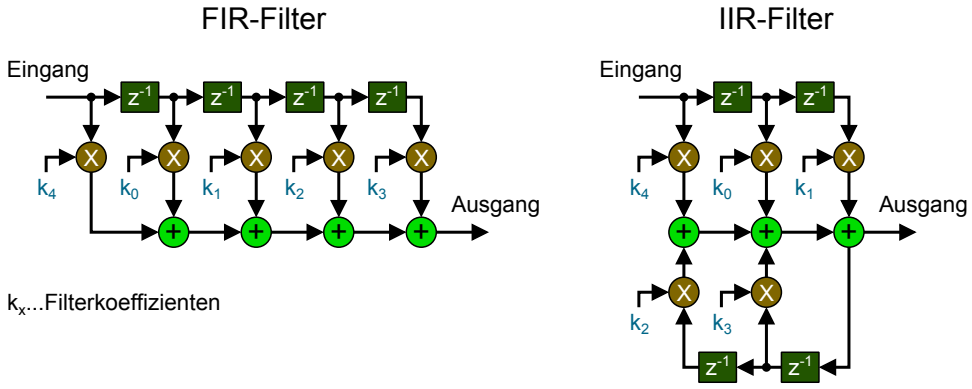

Abbildung 4.10: Filterstrukturen

Akkumulator ein 18 Bit Signal ausgeschnitten, weswegen durch das Filterdesign in Software gewährleistet werden muss, dass hier keine Überläufe stattfinden. Dabei geht die Hardware von einer Zweierkomplement Festkommaimplementierung im I.Q-Format 1.17 aus. Das heißt, beim IIR-Filter darf der Akkumulatorinhalt diesen Wertebereich nach erfolgter Filterung nie überschreiten. Die Konvertierung von den 32 Bit Prozessorwortbreite auf die 18 Bit des Filters passiert durch die Wahl der 18 höherwertigen Bits. Daher hat das Filtereingangssignal im Prozessor ebenfalls nur eine Vorkommastelle (I.Q-Format 1.31) und die unteren 14 Nachkommastellen werden verworfen. Beim Filterausgangssignal sind jedoch alle 32 Bit des Ergebnisregisters signifikant und können durch die Schiebeoperation am Filterausgang vom I.Q-Format 8.24 bis 1.31 ausgerichtet werden.

4.5.3 Programmierung und Konfiguration

Nach der Definition der möglichen Filterstrukturen und der Erklärung des Hardwarefilterblocks widmet sich der folgende Teil der Konfiguration der gesamten Erweiterungseinheit. Um nicht auf so kurze Filter mit nur fünf MAC-Operationen beschränkt zu sein, wie es aus dem Aufbau des Filterblocks hervorgeht, können mehrere Filter zusammengeschaltet werden. Damit sind jedoch nicht zusätzliche Hardwareressourcen gemeint, sondern nur mehrere virtuelle Filterblöcke, welche im weiteren Verlauf als 'Filterstufen' bezeichnet werden. Das wird dadurch ermöglicht, dass der reale Filterblock für 256 verschiedene Filter mit vier Verzögerungselementen ausgelegt ist, wodurch sich auch so viele eigenständige Filterstufen realisieren lassen. Durch Zusammenschalten mehrerer oder aller dieser Stufen können so wesentlich längere Filter mit bis zu einer Länge von 1025 Koeffizienten realisiert werden. Diese Anzahl ergibt sich dabei aus der Gesamtzahl an Verzögerungselementen plus eins für das Filtereingangssignal.

Das Zusammenschalten der Filterstufen kann nun per Spezialbefehl konfiguriert werden. Dabei können allerdings nur aufeinanderfolgende Stufen für die automatische Abarbeitung zusammengeschaltet werden, wozu lediglich die Start- und Endstufe angegeben werden müssen. Sollte aus irgend einem Grund eine aufeinanderfolgende Zusammenschaltung nicht möglich sein, können auch mehrere Filterfolgen kombiniert werden. Allerdings ist die Programmierung dann etwas komplizierter, da zuerst der erste Teil des Filters konfiguriert und anschließend der Filterbefehl ausgeführt werden muss. Für jeden zusätzlichen Filterteil muss diese Vorgehensweise ebenfalls durchgeführt werden, nur dass anstatt eines normalen Filterbefehls ein Fortsetzungsbefehl verwendet wird. Durch diesen Fortsetzungsbefehl führt die Hardware den Speicher der Filterzustände

bei der gewünschten Filterstufe fort und übernimmt den Akkumulatorinhalt des letzten Filterteils. Ein Grund solche Filter aufzubauen könnte z. B. der Wunsch nach einer Kombination aus FIR- und IIR-Filtern sein, was sonst nicht möglich wäre. In Abbildung 4.11 ist so eine mögliche Zusammenschaltung von Filterstufen gezeigt. Dabei fällt auf, dass nur in der ersten Filterstufe fünf MAC-Operationen ausgeführt werden, während die restlichen Stufen jeweils nur vier ausführen. Dadurch müssen pro Stufe auch nur vier Zustände gespeichert werden.

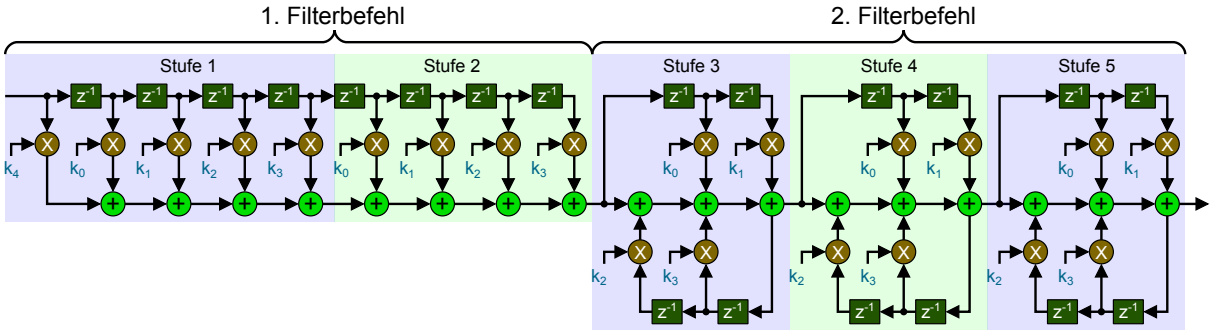

Abbildung 4.11: Beispiel einer Filterkonfiguration

Die Konfiguration erfolgt über Spezialbefehle, die parallel zu einem Filterbefehl ausgeführt werden können und nur einen Taktzyklus benötigen. Als Konfigurationsdaten dienen die Registerinhalte des Prozessors, die über die generische Schnittstelle an die Erweiterungseinheit übermittelt werden. Zu den Konfigurationsmöglichkeiten gehört neben der Filterzusammenschaltung auch das Auswählen einer Filterstufe, sowie das Konfigurieren von Filterkoeffizienten. Pro Taktzyklus können entweder zwei Filterkoeffizienten der aktuell ausgewählten Filterstufe gesetzt werden oder nur ein Koeffizient an einer bestimmter Position. Die Angabe von Stufe und Position in der Filterstufe erfolgt dabei über den zweiten Eingangsoperanden der Erweiterungseinheit. Um die Schnittstelle zum Prozessor optimal ausnutzen zu können, besteht die Möglichkeit einer automatischen Inkrementierung der aktuellen Filterstufe nach Abarbeitung eines Befehls. Das gilt sowohl für Konfigurations-, als auch für Filterbefehle. Somit kann die explizite Selektion der Filterstufe oft vermieden werden. Außerdem kann bei Filterbefehlen über den zweiten Eingangsoperanden bereits die nächste Filterstufe ausgewählt werden. Dadurch können hardwareseitig die für den nächsten Filterbefehl benötigten Operanden vorgeladen werden, was einen sofortigen Start der MAC-Operationen ermöglicht.

Auch um den Zustand eines Filters zu löschen existiert ein Spezialbefehl, welcher wie ein Filterbefehl behandelt wird und genau so lange braucht. Da die Kontrolleinheit bis zu zwei Befehle aufnehmen kann, kann ein Filterbefehl bereits in Warteschleife sein, bevor ein anderer Filterbefehl fertig ist. In diesem Fall signalisiert die Filtereinheit dem Prozessor, dass sie nicht bereit ist weitere Befehle aufzunehmen. Das Gleiche gilt auch dann, wenn ein Filterergebnis auf die Übermittlung ins Zielregister warten muss und bereits ein weiterer Filterbefehl im Befehlsregister der Erweiterung ist. Denn die Filterkontrolleinheit kann nur maximal zwei Ergebnisse speichern, wenn sie keinen Schreibzugriff auf das Registerfile des Prozessors bekommt. Durch diese Signalisierung weiß der Prozessor, wenn die Erweiterung nicht ansprechbar ist und stoppt bei weiteren Befehlen an diese Erweiterung seine Pipeline. So können Fehler aufgrund von Ressourcenproblemen ausgeschlossen werden.

Sollten die Ressourcen einer Filtererweiterung aufgrund eines extrem langen Filters oder zu vieler Filterkanäle nicht ausreichen, besteht natürlich die Möglichkeit den Prozessor mit mehreren solchen Einheiten auszustatten.

5 Evaluierung und Ergebnisse

In diesem Kapitel wird die Beispielimplementierung aus Kapitel 4 untersucht. Da vor einer Leistungsevaluierung die korrekte Funktionsweise der Hardware gegeben sein muss, beschäftigt sich der erste Teil dieses Kapitels mit den Hardwaretests. Erst danach werden die Implementierungsergebnisse und die darauf laufende Software präsentiert. Mit Hilfe dieser Vorarbeiten können anschließend aussagekräftige Vergleiche getroffen werden, wie sie im letzten Teil des Kapitels besprochen werden.

5.1 Systemtest

Das erfolgreiche Testen der implementierten Funktionseinheiten ist wohl der aufwendigste Teil dieser Diplomarbeit. Dabei stellen nicht die Tests der Untereinheiten das große Problem dar, sondern die Gesamttests. Denn kleine Untereinheiten lassen sich mit relativ geringem Aufwand vollständig testen. Doch selbst wenn nur fehlerfreie Komponenten zu einer Funktionseinheit oder einem System kombiniert werden, kann das System trotzdem Fehler haben. Diese Fehler entstehen entweder durch eine falsche Verdrahtung oder sind konzeptioneller Natur. Da jedoch die Anzahl interner Zustände des Systems exponentiell mit dessen Komplexität steigt und das System zudem im mathematischem Sinn nichtlinear ist, sind vollständige Tests praktisch ausgeschlossen. Außerdem ist die Vorgehensweise bei weitem nicht so simpel wie beispielsweise beim Testen der Prozessorausführungseinheit. Hier reicht es aus, ein Eingangssignal anzulegen, nach einer bestimmten Verzögerungszeit das Ausgangssignal zu überprüfen und diesen Vorgang vielmals zu wiederholen. Aufgrund der Einfachheit dieses Vorgangs wird auf solche Unterkomponententests im Folgenden nicht näher eingegangen. Dazu wird auch die Konvertierungserweiterung gezählt, da diese eine einfache rückkopplungsfreie Pipeline besitzt und die Funktionsweise eindeutig definiert ist. Bei den Gesamttests hingegen müssen Eingangsfolgen angelegt werden, um die vorhandenen internen Speicher und Zustände zu berücksichtigen. Das lässt nicht nur die Anzahl der benötigten Testvektoren enorm ansteigen, sondern macht auch eine Auswertung der Ausgangssignale schwierig.

5.1.1 Prozessor

Da der Prozessor über einen relativ simplen Aufbau mit einer kurzen Pipeline verfügt, sind auch Daten- und Kontrollpfad einfach gehalten. Deshalb ist der Test dieses Subsystems noch relativ einfach durchführbar und kann sogar manuell erfolgen. Es wird dabei davon ausgegangen, dass alle Funktions- und Unterfunktionseinheiten fehlerfrei sind. Zuerst wird kontrolliert, dass alle Speicher und Kontrollsignale korrekt initialisiert werden. Mit diesen Vorbedingungen lassen sich wesentliche Vereinfachungen treffen.

Da von korrekter Speicherinitialisierung ausgegangen wird, dürfen im Programmspeicher nur gültige Maschinenbefehle vorhanden sein, wovon es nur eine sehr begrenzte Anzahl gibt. Zudem wurden die Adressierungen von Speicher und Register bereits getestet, wodurch sich entsprechende Befehle auf eine oder wenige Beispieladressen reduzieren lassen. Sämtliche E/A-Operationen bis auf Interrupts sind intern als normale Speicherzugriffe abgebildet und fallen daher unter die Speichertests. Bei den Interrupts kommt die Tatsache zu Gute, dass diese unabhängig vom normalen Programmablauf erfolgen und daher separat getestet werden können. Es ist lediglich darauf zu achten, dass die Rücksprungadresse korrekt ermittelt und gespeichert wird. Aus diesem Grund gibt es keine vollständige Entkopplung vom normalen Programmablauf und die Kombination mit Sprungbefehlen muss getestet werden. Bezüglich der normalen Befehlsabarbeitung vereinfacht sich die Vorgangsweise zusätzlich, da die Pipeline nur fünf Stufen besitzt und es daher keine längeren Abhängigkeiten geben kann.

Damit verbleibt nur noch die Behandlung der Schnittstelle zu den Erweiterungseinheiten. Diese kann komplett mit Unterkomponententests der Kontrolleinheit und des Multiplexers vor dem Registerfile abgedeckt werden. Denn die Auswahl der Erweiterungseinheit, die Vergabe des Registerzugriffs und die Datenselektion sind direkte Funktionen von Eingangsgrößen zu genau einem Zeitpunkt. Das einzige Signal der Kontrolleinheit, das für den Prozessor relevant ist und längere Auswirkungen verursacht, ist das Anhalten der Pipeline. Dieses Signal wird jedoch auch bei den normalen Befehlssatztests überprüft.

Somit verbleiben folgende Punkte zum Testen:

- Befehlssatz des Prozessors
- Befehlsfolgen mit kurzen Daten- oder Kontrollabhängigkeiten
- Programmsprünge
- Interrupts

Unter der Annahme, dass Register und Speicher auf gewünschte Werte initialisiert werden, reichen jeweils Folgen von fünf Maschinenbefehlen oder weniger aus um alle genannten Punkte zu testen. Es sind dabei nicht einmal alle Kombinationen erforderlich, sondern nur solche, die bezüglich des Daten- oder Kontrollpfads überhaupt Auswirkungen aufeinander haben können. Da diese Auswahl sehr überschaubar ist, kann das Testen manuell und ohne aufwendige Testdatenerzeugung und -kontrolle erfolgen. Die einzige sich dabei ergebende Schwierigkeit besteht darin, keine für Tests relevanten Befehlsfolgen zu übersehen.

5.1.2 CORDIC-Erweiterung und Simulator

Beim Test der CORDIC-Erweiterungseinheit stellt deren Aufbau kein Problem dar. Zwar ist die Pipeline mit 31 Stufen wesentlich länger als die des Prozessors, doch gibt es hier überhaupt keine Abhängigkeiten neben dem normalen Durchlauf der Pipeline. Somit sollten Tests der einzelnen Pipeline-Stufen ausreichend sein. Da jedoch die Hardware hier einen vollständigen mathematischen Algorithmus abbildet, müssen nicht nur die Pipeline-Stufen richtig arbeiten, sondern es muss auch verifiziert werden, dass der Algorithmus korrekt auf die Hardware abgebildet wurde. Ein weiteres Problem ergibt sich dadurch, dass die bereitgestellten mathematischen Funktionen nur gute Näherungen der exakten Lösung sind. Daher reicht es nicht, entsprechende Eingangswerte anzulegen, den Pipeline-Durchlauf abzuwarten und die Ergebnisse bitweise mit denen einer Berechnungssoftware zu vergleichen. Denn hier werden sich oft algorithmus- und rundungsbedingte Abweichungen ergeben, die so als Hardwarefehler interpretiert würden.

Unter anderem aus diesem Grund wurde ein Simulator entwickelt, welcher im ersten Schritt das Verhalten des Algorithmus nachstellt. Dazu wird sowohl die Initialisierung der CORDIC-Variablen, als auch jeder einzelne Iterationsschritt, sowie die Nachbearbeitung nachgestellt. Der Simulator ist dabei auch im Stande sich selbst zu überprüfen, indem die Abweichungen von den exakten mathematischen Funktionen ermittelt werden und bei zu großen Differenzen Fehler gemeldet werden. Somit kann sichergestellt werden, dass nicht Simulator und die spätere Hardware die selben Fehler aufweisen. Außerdem können durch die Selbstkontrolle Modifikationen am Algorithmus vorgenommen werden und die Korrektheit dieser Änderungen sofort überprüft werden. Das war vor allem bei der Erweiterung des Standard-CORDIC-Algorithmus auf mehrere Koordinatensysteme und Funktionen wichtig. Denn nur so konnten verschiedene Abläufe bei der Variableninitialisierung und der Nachbearbeitung mit vernünftigem Aufwand ausprobiert und getestet werden. Ziel dieser Variationen war ein möglichst gut für die Hardware geeigneter Ablauf, um einerseits Ressourcen zu sparen und andererseits eine geringe zusätzliche Anzahl an Pipeline-Stufen zu erhalten.

Im nächsten Schritt wurden die einzelnen Berechnungsschritte des Simulators mit einer konfigurierbaren Genauigkeit ausgestattet. So wurden Gleitkommaberechnungen durch Integerberechnungen mit einstellbarer Wortlänge ersetzt. Dadurch ist es möglich auch endliche Wortlängeneffekte der späteren Hardware wie Rundungsfehler und Überläufe zu simulieren. Somit ist ein bitweiser Vergleich von Hardware- und Simulatorergebnis möglich, was für automatische Hardwaretests erforderlich ist. Weiters lässt sich der Einfluss der Wortlänge auf die Genauigkeit der CORDIC-Ergebnisse simulieren, was auch zur Festsetzung dieser in der Implementierung verwendet wurde.

Um nun die korrekte Funktionsweise der Hardwarebeschreibung der CORDIC-Erweiterung zu testen, ist es somit nur mehr erforderlich aus den simulierten Ein- und Ausgangsgrößen entsprechende Anregungen für die Testbench zu generieren. Das erfolgt durch eine einfache Umwandlung dieser Größen in das Binärformat und anschließender Speicherung in einer Datei. Diese Datei wird von der Testbench automatisch eingelesen und die darin enthaltenen Signale auf die zu testende Hardware angewandt. Somit können beliebig lange Tests praktisch ohne Aufwand durchgeführt werden, womit nachträgliche Änderungen der Hardware auch kein Problem darstellen.

5.1.3 Digitales Filter

Das Testen des digitalen Filters gestaltet sich von allen Funktionseinheiten am schwierigsten. Denn es verfügt nicht wie die anderen Teilsysteme über den strukturierten Funktionsablauf einer Pipeline und bietet außerdem die meisten Konfigurationsmöglichkeiten. So stellt diese Erweiterungseinheit eine große rekonfigurierbare Finite-state-machine, ohne fester Verzögerung zwischen Ein- und Ausgangssignalen dar. Zudem können Befehle und Ergebnisse zwischengespeichert werden und Rekonfigurationen auch während einer Befehlsausführung getätigt werden. Aufgrund dieser Eigenschaften und dem Vorhandensein mehrerer lokaler Speicher ist ein manuelles Testen wie beim Prozessor nicht möglich.

Die lokalen Speicher und die MAC-Einheit haben jedoch fixe Zeitverzögerungen zwischen Ein- und Ausgängen und verfügen auch nicht über etwaige Rückkopplungen. Deshalb können diese Untereinheiten genauso wie die anderen getestet werden. Die nächste Funktionsebene ist der Filterblock, welcher zumindest eine einfache State-machine in Form eines Zählers darstellt. Dieser Zähler zählt je nach Konfiguration entweder bis vier oder bis fünf. Daher kann das Verhalten relativ einfach nachmodelliert werden, um Stimuli für die Testbench dieses Funktionsblocks zu erzeugen. Das wurde auch gemacht und ein entsprechendes MATLAB-Skript erstellt.

Der aufwändigste Teil ist jedoch das Testen des gesamten Filters. Denn neben dem eigentlichen Filter, stellt auch das Testen der Schnittstellenkommunikation eine große Herausforderung dar. Dazu ist wieder ein Simulator erforderlich, der neben den normalen Berechnungen und Verwaltung der lokalen Speicher auch das variable Timing des Filters modelliert. Natürlich müssen sämtliche Softwareberechnungen auch in diesem Fall mit der gleichen numerischen Auflösung wie in der Hardware durchgeführt werden. Der Simulator benutzt nicht nur eine vorgegeben Konfiguration, sondern führt auch selbst Rekonfigurationen während des Betriebs aus. Im Gegensatz zur CORDIC-Erweiterung kann der Simulator aber nicht so einfach selbst getestet werden und darum wurde hier ein anderer Weg beschritten. Dieser Weg macht sich die Tatsache zu Nutze, dass Softwaremodellierung und Hardwarebeschreibung zwar das gleiche Ein-/Ausgangsverhalten besitzen sollen, aber intern völlig anders aufgebaut sind. Somit ist es extrem unwahrscheinlich, dass beide Beschreibungen die gleichen Fehler aufweisen. Deshalb wurden nach einer manuell durchgeführten Überprüfung des Simulators ganz normal die Hardwaretests durchgeführt. Bei Unstimmigkeiten zwischen Hardware und Simulator wurden diese überprüft und entsprechende Korrekturen vorgenommen. Dabei handelte es sich meist um Fehlerbehebungen in der Hardwarebeschreibung oder Timingfehler in der Softwaremodellierung. Nachdem alle wesentlichen Unstimmigkeiten beseitigt wurden, ist die Hardware voll funktionsfähig, was auch die erfolgreiche Implementierung der Beispielprogramme in Kapitel 5.2.5 bestätigt.

5.1.4 Aufgetretene Probleme

Wie bei fast allen Entwicklungsaufgaben, gab es auch beim praktischen Teil der Diplomarbeit einige Probleme. Ein paar wesentliche davon werden im Folgenden kurz präsentiert.

Fehler im Filtersimulator:

Ein Problem, das nach wie vor besteht, bezieht sich auf den Simulator des digitalen Filters. Der Simulator weist nämlich in Sonderfällen noch immer ein etwas anderes Timingverhalten auf als die Hardwarebeschreibung. Da jedoch in diesen Fällen die Hardwarebeschreibung das korrekte zeigt,

wurde eine auf Fehlerbehebung des Simulators verzichtet. Der Grund dafür ist, dass diese Sonderfälle extrem selten auftreten und deshalb mit sehr hohem Zeitaufwand bei der Ursachensuche verbunden sind. So treten solche Fehler nur wenige Male in mehreren Millionen Taktzyklen beim Testen auf. Dabei geht es immer um einen Taktzyklus Unterschied beim Filterergebnis zwischen Simulation und Hardware. Verursacht wird dieser Fehler durch die benötigte Pufferung von Filterbefehlen und Ergebnissen bei Ressourcenkonflikten zwischen Prozessor und Filtererweiterung. Offenbar beginnt dann die Filterung unter bestimmten Umständen in Simulation und Hardware zu unterschiedlichen Zeitpunkten. Eine Möglichkeit diesen Fehler zu umgehen wäre natürlich die Softwaremodellierung der Hardware anzupassen, wodurch jedoch die Diversität verloren geht. Das bedeutet, wenn der Simulator exakt gleich arbeitet wie die Hardware, besitzt er auch zwangsweise die gleichen Fehler, die somit nicht gefunden werden können. Daher wird hier ein Simulator mit bekannten und sehr selten auftretenden Timingfehlern als die bessere, wenn auch nicht perfekte Variante empfunden.

Sogar wenn fälschlicherweise angenommen würde, dass das Problem in der Hardware und nicht im Simulator liegt, wäre das kein großes Problem. Denn aufgrund der Schnittstellendefinition zwischen Prozessor und Erweiterungseinheiten wartet der Prozessor nicht explizit auf das Ergebnis einer Erweiterungseinheit. Vielmehr ist es die Aufgabe der Erweiterung Bescheid zu geben, wenn ein Ergebnis fertig ist. Somit verzögert sich im schlimmsten Fall nur das Ergebnis um ein oder mehrere Takte. Da bei Erweiterungseinheiten mit ungewissen Latenzzeiten sowieso softwareseitig mit Interrupts oder Schleifen zur Abfrage eines Flags gearbeitet werden soll, würde das auch für die Programmierung kein Problem darstellen.

Nicht erreichtes Timing:

Ein weiteres Problem, diesmal jedoch auf die Implementierung bezogen, trat durch eine irreführende Chipbezeichnung des verwendeten FPGAs auf. Denn aufgrund dieser Bezeichnung wurde ein höherer Geschwindigkeitsgrad des Chips angenommen [12]. Bei Verwendung des Entwicklungsboards und des echten Geschwindigkeitsgrades konnte das Timing dadurch nicht mehr erreicht werden, wodurch Änderungen im Design notwendig waren. Dazu wurden soweit es die Spezifikation zuließ Pipeline-Stufen sowohl im Prozessor selbst, als auch in den Erweiterungseinheiten verschoben. Außerdem war es notwendig bei der Filtererweiterung eine zusätzliche Verzögerung um einen Taktzyklus der Domäne mit doppelter Taktgeschwindigkeit einzufügen. Das machte natürlich erneutes Testen und eine Änderung des Timings beim Filtersimulator erforderlich, was die Arbeit mehrere Tage verzögerte. Weiters wurde die Optimierungsstufe der verwendeten Softwarewerkzeuge erhöht, um auch hier noch Verbesserungen zu erwirken. Durch oben genannte Änderungen war es möglich, das Timing des Designs auch für die geringere Leistung des FPGAs auszulegen. Eine Alternative wäre die Halbierung des Systemtakts von 100 MHz auf 50 MHz gewesen, was mit einer Halbierung der Rechenleistung des Prozessors einhergegangen wäre und daher als absolute Notfalllösung gedacht war.

Implementierung des Programmspeichers:

Bei der Implementierung des Programmspeichers, im Folgenden einfach als ROM (**R**ead **O**nly **M**emory) bezeichnet, gab und gibt es Probleme seitens der Entwicklungssoftware. Um unabhängig von der Zielhardware zu sein, wurde der ROM ursprünglich über eine allgemeine Verhaltensbeschreibung implementiert. Das Setzen der Speicherinhalte wurde vor der Synthese mit Hilfe einer Schleife durchgeführt, die diese Information aus einer Datei ausgelesen hat. Dabei war es nicht

erforderlich den gesamten Speicherinhalt in der Datei bereitzustellen, sondern lediglich den benötigten Teil des Speichers. Über entsprechende Direktiven in der Datei konnten die Speicherinhalte sogar an beliebige Stellen geschrieben werden, ohne sämtliche vorherige Adressen beschreiben zu müssen. Diese Vorgehensweise funktionierte im Prinzip zwar problemlos, allerdings mussten sämtliche für die Implementierung notwendigen Prozesse von der Kommandozeile aus gestartet werden, da die graphische Benutzeroberfläche hier zu limitiert ist. Es ist nämlich nicht möglich, mehr als 64 Schleifendurchläufe durchzuführen, da ansonsten das Synthesewerkzeug die Synthese der Schaltung abbricht. Das Problem daran ist, dass der Wert 64 fix vorgegeben ist, was für ein normales Programm viel zu wenig ist. Um weiterhin die GUI (**G**raphical **U**ser **I**nterface) verwenden zu können, musste somit statt der Verhaltensbeschreibung ein IP-Core (**I**ntellectual **P**roperty) des ROMs verwendet werden. Dieser ist spezifisch für die Zielhardware und muss um ein neues Programm für den Prozessor zu laden regeneriert werden, was aufgrund von nicht nachvollziehbarer Ursachen oft fehlschlägt. Das macht es erforderlich den Regenerationsvorgang mehrmals zu wiederholen und verlängert somit die Synthesezeit erheblich. Prinzipiell wäre es zwar möglich, die Initialisierung des ROMs auch direkt in die Binärdatei zum Download in das FPGA einzuflechten, jedoch ist das mit erheblichen manuellen Aufwand verbunden. Da es nicht gelungen ist die zuletzt genannte Methode zu verwenden, muss nun für eine Änderung der Prozessorsoftware immer die komplette Synthese mit vorheriger Regeneration des ROMs erfolgen.

Programmerstellung mit Erweiterungsbefehlen:

Ein weiteres Problem bezogen auf die Prozessorsoftware war dessen Erstellung. Es können zwar bestehende Assembler wie z. B. der in [VS06] beschriebene verwendet werden, allerdings sind Interrupts anders implementiert und die Erweiterungsbefehle sind nicht bekannt. Die Unterschiede in der Implementierung der Interrupts wurden bereits in Kapitel 4.2 erörtert. Da jedoch ein vorhandener Assembler besser ist als gar keiner, wurde der zuvor erwähnte MARS-Assembler (**M**IPS **A**ssembler and **R**untime **S**imulator) für die Programmerstellung verwendet. Dabei wurden für jeden Interrupt und das Hauptprogramm eigene Dateien mit den jeweiligen Programmfragmenten erstellt. Für eine spätere Nachbearbeitung wurden an den entsprechenden Stellen im Programm nicht erlaubte Befehle und Coprozessorbefehle als Markierung verwendet. Letztere haben den Vorteil, dass sie auch Registeradressen und Funktionscodes enthalten. Für diese Rohdateien wurden MATLAB-Skripts geschrieben, die daraus eine vollständige Speicherbelegung des ROMs zusammenstellen und die Markerbefehle durch gültige Erweiterungsbefehle ersetzen. Dadurch wird die Programmerstellung mit Benutzung der Erweiterungsbefehle nur unwesentlich erschwert und gestaltet sich ähnlich der normalen Assemblerprogrammierung des MIPS R2000 Prozessors.

Designfehler:

Nach der ersten erfolgreichen Synthese des Prozessors mit fertigem Programm konnte die Hardware in Betrieb genommen werden. Für einfache Programme ohne Interrupts funktionierte der Prozessor fehlerfrei. Allerdings gab es manchmal bei Interrupt-Ereignissen nicht nachvollziehbare Probleme, dass Interrupts deaktiviert oder Ausgänge falsch gesetzt wurden. Mit dem Hardwaredebug-Werkzeug 'ChipScope' wurden diese Probleme genau untersucht und festgestellt, dass manche Register während des Programmablaufs wirklich falsch gesetzt wurden. Nach kleineren Korrekturen im Programmcode war das Problem teilweise noch immer vorhanden. Wie sich durch detailliertes Verfolgen der Signale durch den Chip herausstellte, wurden offenbar beim Testen des Designs in Kombination mit Interrupt-Aufrufen manche ungünstige Befehlskonstellationen übersehen. Diese Konstellationen gehören zwar zu den in Kapitel 5.1.1 aufgeführten Testkategorien, wurden

allerdings nicht bedacht. So traten Fehler genau dann auf, wenn Programmsprünge mit nachfolgenden registerbasierten Operationen durch Interrupts unterbrochen wurden. In diesen Fällen wurden teilweise Register- und Speicherinhalte falsch beschrieben. Nach Korrektur dieses Designfehlers gab es selbst bei stundenlanger Programmlaufzeit und vielen Interrupts keine Fehler mehr, was auch durch Kontrolle der wichtigsten Chipsignale mit Hilfe von ChipScope bestätigt wurde.

Kommunikation mit dem AC97 Audiochip:

Da für die anschauliche Demonstration des Prozessors eine Echtzeit-Audioverarbeitung als Anwendung ausgewählt wurde, war auch eine Ansteuerung des sich auf dem Entwicklungsboard befindlichen AC97 Audiocodecs [3] erforderlich. Dafür stellt Xilinx einen IP-Core zur Verfügung, der die Kommunikation mit diesem Chip übernimmt. Dieser IP-Core wurde auf die Prozessorimplementierung so angepasst, dass er direkt an den E/A-Anschlüssen angeschlossen ist und auch externe Interrupt-Leitungen des Prozessors verwendet. Es handelt sich also nicht um einen Teil des Prozessors, sondern um dessen Peripherie. Leider gab und gibt es nach wie vor Probleme durch Fehler im IP-Core, was auch durch [6] bestätigt wird. Denn beim Starten muss der Codec richtig initialisiert werden, um überhaupt damit arbeiten zu können. Das funktioniert über die Konfiguration von Registern im AC97-Chip, wofür der Prozessor mit diesem über den IP-Core kommunizieren muss. Auf Softwareseite erfolgt die Konfiguration über das Senden eines Konfigurationsbefehls und Übermitteln der Konfiguration. Danach wird per Warteschleife gewartet, bis der Codec bereit für den nächsten Befehl ist. Genau hier kann es sein, dass ein Fehler passiert und der IP-Core eine erneute Bereitschaft des Codecs nicht mehr signalisiert. Daher bleibt das Programm in der Warteschleife stecken und wird nicht weiter ausgeführt. Um dieses Problem zu umgehen könnte zwar ein Time-out auf Softwareseite eingeführt werden, allerdings ist damit nicht gewährleistet, dass die Kommunikation mit dem Codec überhaupt funktioniert. Außerdem verschafft hier ein einfaches Betätigen des Reset-Tasters Abhilfe, was für Demonstrationszwecke ausreichend ist. Ein Fehler auf Seite des Prozessors wurde zwar in Betracht gezogen, konnte aber per ChipScope ausgeschlossen werden und eine Fehlerbehebung im IP-Core war nicht Aufgabe der Diplomarbeit.

Synthesefehler:

Ein weitaus gravierenderes Problem ergab sich bei der Synthese des Prozessors mit unterschiedlichen Hardwarekonfigurationen. Dabei ist das Austauschen von Hardwareerweiterungen an der generischen Prozessorschnittstelle gemeint, was ja auch Teil der Konfigurierbarkeit sein soll. Bei gewissen Konfigurationen funktionierte trotz passendem Timing und korrekt gesetzten Constraints unter anderem die Filtererweiterung nicht mehr. Mit ChipScope konnte festgestellt werden, dass diese immer den Wert null zurückliefert. Es wurde ebenfalls überprüft, ob die geänderte Hardwarekonfiguration trotz deren Nichtbenutzung Änderungen in Speicherelementen und Registern im Prozessor verursacht, jedoch konnte keine Beeinflussung festgestellt werden. Somit hatte bei gleichem Prozessor und dessen Programmierung nur das bloße Vorhandensein von anderen Erweiterungen Einfluss auf die Funktion des digitalen Filters und des Prozessors. Über die generische Schnittstelle gab es dabei keinerlei Kommunikation mit den anderen Erweiterungen, so als ob diese nicht vorhanden wären. Auch der Programmablauf und die Verarbeitungszeit des Filters zeigten keine Veränderungen. Deshalb wird von einem Problem bezüglich des Synthesewerkzeugs ausgegangen. Denn beim Austausch von Hardwareerweiterungen wird bei der Implementierung die Platzierung der Komponenten und das Routen im FPGA verändert, was natürlich Einfluss auf die Signallaufzeiten im Chip hat. Das Problem konnte jedoch schließlich durch Anwendung

anderer Synthese und Implementierungseinstellungen umgangen werden. Mit diesen Einstellungen ergibt sich laut Berechnung zwar ein schlechteres Timing, das aber immer noch im Rahmen der gewählten Vorgaben ist und der Prozessor funktioniert für alle Hardwarekonfigurationen fehlerfrei.

5.2 Vergleich Software-/Hardwarelösung

Dieser Abschnitt widmet sich explizit den erzielbaren Leistungssteigerungen durch den Einsatz der präsentierten Implementierungen der rekonfigurierbaren Erweiterungseinheiten. Dabei wird jedoch nicht nur die Leistungssteigerung betrachtet, sondern auch der zusätzliche Hardwareaufwand in Form von Ressourcenverwendung im FPGA. Dieser soll aufgrund fehlender Messungen auch Aufschluss über die Leistungsaufnahme des Prozessors geben, sofern angenommen wird, dass sich Hardwareaufwand und Leistungsaufnahme in etwa proportional verhalten. Über das Verhältnis von Leistungssteigerung zu Hardwareressourcen lassen sich damit vage Rückschlüsse auf die Energieeffizienz ziehen. Im Anschluss an die theoretischen Resultate werden die zwei implementierten Beispielprogramme besprochen und direkte Signalmessungen der Hardware präsentiert.

5.2.1 CORDIC

Die CORDIC-Erweiterungseinheit ist die aufwendigste der drei Implementierungen, was sich auch beim späteren Vergleich der benötigten FPGA Ressourcen zeigt. Weil bei allen CORDIC-Funktionen unterschiedliche Initialisierungen zum Einsatz kommen, wirkt sich das auch auf den Initialisierungsaufwand der Software aus. Daher benötigt jede Funktion eine unterschiedliche Anzahl von Befehlen, was gleichbedeutend mit der Laufzeit ist. Aus diesem Grund wird nur beispielhaft die Kosinusfunktion in Software mit der Hardwareimplementierung verglichen. Dabei wird der gesamte CORDIC-Algorithmus ohne Schleifen und Ausnahmebehandlungen realisiert, um die geringste Laufzeit zu erhalten. Treten also Spezialfälle wie Überlauf, NAN und denormalisierte Zahlen auf, ist das Ergebnis falsch, ohne dass dieses als ungültig gekennzeichnet wird. Für die Eingangsoperanden und das Ergebnis wird angenommen, dass diese im gleichen Format wie bei der Hardwareimplementierung vorliegen. Außerdem wird von einer Gleichverteilung bei den Argumenten des Kosinus ausgegangen, da sich die Größe des Ergebnisses auf die Anzahl an Softwarebefehlen auswirkt. Denn um das Ergebnis wieder auf die Gleitkommarepräsentation zu bringen, muss das erste gesetzte Bit im CORDIC-Ergebnis gesucht werden. Durch die Form des Kosinus und dessen numerische Aussteuerung ist dazu im Schnitt eine Schiebeoperation um 1,65 Positionen erforderlich, was mit dem Integral aus Gleichung 5.1 berechnet werden kann. Als obere Integrationsgrenze dient dabei der größte von der Hardware nicht mehr von null unterscheidbare Wert des Kosinus, wobei sich das Ergebnis praktisch nicht ändert, wenn stattdessen der ganze Wertebereich bis $\frac{\pi}{2}$ verwendet wird.

$$pos = -\frac{2}{\pi} \int_0^{max} \lfloor \mathrm{ld}(\cos(\theta)) \rfloor \, d\theta \tag{5.1}$$
$$max = \arccos(2^{-25})$$

In Tabelle 5.1 werden die Anzahl der benötigten Taktzyklen für die Softwareimplementierung der Hardwareimplementierung gegenübergestellt. Die in der Tabelle verwendeten Teiloperationen ergeben sich dabei aufgrund von bedingten Sprüngen, die mit gewisser Wahrscheinlichkeit ausgeführt werden. Deshalb stellen diese Werte auch nur Durchschnittswerte für die Latenz der Softwareimplementierung dar, während die Hardwarewerte immer gleich und exakt sind.

Tabelle 5.1: Vergleich von Software und Hardware - Kosinusfunktion CORDIC

Funktionsabschnitt	Taktzyklen Hardware	Taktzyklen Software
Initialisierung	4	20
erste CORDIC-Iteration	1	4,5
mittlere CORDIC-Iterationen	21·1	21·7,5
letzte CORDIC-Iteration	1	3,5
Nachbearbeitung	4	24,96
Ergebnisregistertransfer	1	0
Gesamt	**32**	**210,46**

Aus diesen Werten ergibt sich somit eine durchschnittliche Leistungssteigerung um den Faktor 6,58 für die Kosinusfunktion. Da allerdings die anderen Funktionen eine ähnliche Anzahl an Softwareoperationen benötigen, wird auch hier die Leistungssteigerung einen ähnlichen Wert besitzen. Bei dieser Berechnung wurde jedoch nur die Latenz der beiden Implementierungen betrachtet. Wird nun berücksichtigt, dass es sich bei der Hardwareimplementierung um eine Pipeline-Verarbeitung handelt und die Latenz keine Rolle spielt, so wird für jede CORDIC-Funktion nur ein Taktzyklus benötigt. Damit ergibt sich eine gemittelte maximale Beschleunigung der CORDIC-Kosinusfunktion um den Faktor 210,46. Doch selbst wenn immer nur ein CORDIC-Befehl zu einem Zeitpunkt bearbeitet wird, ist zu berücksichtigen, dass der Zeitaufwand für den Prozessor lediglich ein bis maximal zwei Taktzyklen ist. Denn der Prozessor kann parallel dazu andere Befehle abarbeiten und muss nur bei Bedarf zum Rückschreiben des CORDIC-Ergebnisses angehalten werden. Somit können auch ohne der Pipeline-Nutzung der CORDIC-Erweiterung Leistungssteigerungen von über 6,58-fach erreicht werden.

Ein gravierender Nachteil der CORDIC-Erweiterung mit Pipeline-Betrieb ist dessen Hardwareaufwand. Dieser Aufwand beträgt für die präsentierte Implementierung fast drei Viertel des normalen Prozessors bezogen auf die verwendeten FPGA-Slices. Daher macht eine solche Erweiterung nur bei signifikanter Nutzung wirklich Sinn, wo sie während des Betriebs auch entsprechende Beschleunigungen bringt. Eine Kompromisslösung wäre eine Version der Hardware ohne Pipeline-Verarbeitung. Dadurch würde nur eine CORDIC-Stufe benötigt werden, welche dann allerdings über variable Schiebelogik verfügen müsste. Somit würde zwar die Latenz für eine Operation steigen und generell nur eine Operation zu einem Zeitpunkt möglich sein, allerdings wäre auch der Hardwareaufwand wesentlich geringer. Die Parallelverarbeitung von Prozessor und Erweiterungseinheit bliebe jedoch weiterhin bestehen, sodass nach wie vor erhebliche Leistungssteigerungen möglich wären.

5.2.2 Konvertierung des Zahlenformats

Bei der Konvertierung zwischen Gleitkomma- und Fixkommazahlenformat hängt die Anzahl der Taktzyklen der Softwareimplementierung ebenfalls vom Operationsmodus ab. Da es hier jedoch

nur zwei Funktionen gibt, nämlich die Konvertierung in jeweils eines der vorgestellten Zahlenformate, werden beide der Hardwareimplementiertung gegenübergestellt. Die Anzahl der Softwaretaktzyklen gibt ebenfalls wieder nur den Durchschnittswert wieder. Dabei muss ähnlich wie beim CORDIC für die Konvertierung ins Gleitkommaformat die Position des ersten gesetzten Bits ermittelt werden. Wenn auch hier wieder eine Gleichverteilung der Eingangsoperanden angenommen wird, ergibt sich eine durchschnittliche Schiebeweite von etwa einem Bit. In Tabelle 5.2 ist wieder der Verglich zwischen Software- und Hardwareimplementierung dargestellt.

Tabelle 5.2: Vergleich von Software und Hardware - Konvertierungsfunktion

	Hardware	**Software Fix- → Gleitkomma**	**Software Gleit- → Fixkomma**
Taktzyklen	3	17,5	14,5
Faktor Hardware-beschleunigung	1	5,83	4,83

Dabei gilt bezüglich Pipeline- und Parallelverarbeitung das Gleiche wie beim CORDIC, sodass auch hier maximale Leistungssteigerungen um den Faktor 17,5 bzw. 14,5 möglich sind. Aufgrund der Synchronisation mit der Prozessor-Pipeline ist es jedoch ausgeschlossen, dass der Prozessor durch die Konvertierungserweiterung angehalten wird. Im Gegensatz zur CORDIC-Erweiterung ist der Bedarf an Hardwareressourcen vergleichsweise gering. So beträgt die benötigte Anzahl an FPGA-Slices nur ca. ein Zehntel der des normalen Prozessors.

5.2.3 Digitales Filter

Im Unterschied zu den beiden anderen Erweiterungen, besitzt das digitale Filter keine Pipeline, wodurch sich auch keine zusätzliche Leistungssteigerung für hintereinander ausgeführte Filterungen ergibt. Aufgrund der eindeutigen Abarbeitung der mathematischen Operationen ist die Dauer des Softwarealgorithmus konstant. Da der MIPS R2000 Prozessor die Multiplikationen auch in einer separaten Ausführungseinheit durchführt, kann wie bei der Filtererweiterung parallel zur Multiplikation die normale Ausführungseinheit genutzt werden. Aus diesem Grund können die für die Filterung benötigten Speicherzugriffe, Adressmanipulationen und Akkuberechnungen parallel zur Multiplikation erfolgen.

Da die Genauigkeit der Filtererweiterung mit 18 Bit Eingangswerten zwischen 16 und 32 Bit liegt, wird sie mit zwei unterschiedlichen Softwareimplementierungen verglichen. Die erste Implementierung arbeitet mit 16 Bit Eingangswerten und einem 32 Bit Akku, was der Wortbreite des Prozessors entspricht. Dadurch ist die Genauigkeit der Filterung etwas schlechter als in der Erweiterungshardware und Akkuüberläufe sind sehr wahrscheinlich. Allerdings ist in diesem Fall teilweise eine Verarbeitung anderer Befehle während jeder Multiplikation möglich, was unter Umständen auch ausgenutzt werden kann. Um genau zu sein verbleiben bei dieser Implementierung fünf anderweitig nutzbare Befehle pro Multiplikation. Die 32 Bit Implementierung nutzt das gesamte Multiplikationsergebnis, was die Berechnung des Akkumulators erschwert, da es mehrere 32 Bit Operationen erfordert. Der Akku wird bei dieser Implementierung mit 64 Bit angenommen, was bei 32 Bit Eingangswerten aufgrund von Überläufen eigentlich wieder zu wenig ist. Da jedoch die Filtererweiterung nur 18 Bit Eingangswerte mit einem 48 Bit Akkumulator unterstützt, eignet sich diese Softwareimplementierung besser für einen Vergleich. Denn ein noch größerer Akku

würde einen erheblichen Mehraufwand der Software bezüglich der Filterzeit bedeuten. In Tabelle 5.3 sind beide Softwareimplementierungen und die Hardwareerweiterung für ein Filter der Länge 1025 gegenübergestellt. Bei näherer Betrachtung fällt dabei auf, dass die Anzahl der Taktzyklen in Hardware teilweise Kommazahlen sind. Das resultiert aus der doppelten Taktfrequenz mit der das Filter arbeitet.

Tabelle 5.3: Vergleich von Software und Hardware - digitales Filter

Funktionsabschnitt	**Taktzyklen**		
	Hardware	**Software 16 Bit**	**Software 32 Bit**
Initialisierung	0	3	3
Filterung	1025·0,5	1025·13	1025·14
Nachbearbeitung	1,5	1	10
Ergebnisregistertransfer	1	0	0
Gesamt	**515**	**13329**	**14363**

Mit den Werten aus der Tabelle ergeben sich so Leistungssteigerungen bei Nutzung der Hardwareerweiterung um den Faktor 25,88 bzw. 27,89. Trotz dieser großen Leistungssteigerung ist der Aufwand an FPGA-Ressourcen mit ca. 15% im Vergleich zum Prozessor relativ gering. Das liegt zum Teil auch daran, dass eine im FPGA befindliche fest verdrahtete MAC-Einheit benutzt wird, die ja die Hauptkomponente des Filters darstellt. Natürlich ist es auch hier wieder möglich die Parallelität von Erweiterung und Prozessor zu nutzen und andere Befehle während der Filterung auszuführen.

5.2.4 Zusammenfassung der Ergebnisse

In Tabelle 5.4 sind die Details von Leistungssteigerungen und benötigten Hardwareressourcen der Erweiterungen nochmals zusammengefasst. Was auf jeden Fall ersichtlich ist, sind die enormen Leistungssteigerungen und die Effizienz bei allen Spezialerweiterungen. Die Effizienz ergibt sich dabei aus dem Verhältnis von maximaler Rechenleistung zur Summe von FPGA Logikressourcen im Vergleich zum Basisprozessor und soll ein ungefähres Maß für die Energieeffizienz darstellen. Was jedoch bei dieser einfachen Berechnungen vernachlässigt wird, ist die Mitbenutzung von Prozessoreinheiten wie z. B. das Registerfile durch die Erweiterungsblöcke. Für eine echte Anwendung ist zudem neben den gezeigten Leistungsgrößen vor allem die Auslastung der Erweiterungen interessant. Denn die dargestellten Leistungserhöhungen gelten nur für die ununterbrochene Nutzung der Erweiterungen einmal mit und einmal ohne Berücksichtigung einer eventuell vorhandenen Pipeline-Verarbeitung. Reduziert sich der Nutzungsanteil einer Erweiterung, so reduzieren sich auch Leistungs- und Effizienzgewinn und es muss abgewogen werden, ob sich der zusätzliche Hardwareaufwand überhaupt lohnt. Nichts desto trotz stellen die präsentierten und untersuchten Implementierungen nur Beispiele dar und sollen lediglich das Prinzip zeigen. Außerdem ist es das Hauptziel der Rekonfigurierbarkeit während der Laufzeit die Auslastung der Erweiterungen zu maximieren und daher die vorhandenen Hardwareressourcen optimal zu nutzen.

5.2.5 Demonstrationsprogramme

Im Folgenden werden die beiden Demonstrationsprogramme beschrieben, die für die Evaluierung des Prozessordesigns konzipiert wurden. Die Programme sind für Echtzeit-Audioverarbeitung

Tabelle 5.4: Vergleich von Hardwareerweiterungen relativ zum Basisprozessor

	CORDIC	**Konvertierung**	**digitales Filter**
Leistungssteigerung normal	558%	483% / 383%	2488% / 2689%
Leistungssteigerung Vollauslastung	20946%	1650% / 1350%	2488% / 2689%
FPGA-Slices	70,8%	10,1%	15,1%
FPGA-LUTs	118,5%	16,4%	16,4%
FPGA-FlipFlops	106,9%	5,5%	18,9%
FPGA Logik-ressourcen gesamt	97,8%	11,5%	16,4%
Effizienz	215	153 / 127	157 / 170

ausgelegt und benutzen den AC97 Audiochip auf dem FPGA-Entwicklungsboard. Die Funktionssteuerung findet jeweils über die fünf ebenfalls auf dem Board befindlichen Taster statt.

Audiofilter

Das erste Beispielprogramm des Audiofilters zielt auf die extensive Benutzung und Rekonfigurierung der Filtererweiterung ab. Ziel des Programms ist es Bandpassfilter bereitzustellen, die mit den Tastern am Entwicklungsboard aus- und eingeschaltet werden können. Der Frequenzgang soll dabei so bestimmt werden, dass er annähernd flach ist, wenn alle Filter aktiv sind, um ein eingelesenes Audiosignal möglichst unverfälscht wieder auszugeben. Weiters sollen die Bandpässe logarithmisch auf der Frequenzachse verteilt werden, da das menschliche Empfinden der Tonhöhe ebenfalls logarithmisch ist. Dazu werden in MATLAB FIR-Filter mit einer Länge von 513 Koeffizienten entworfen, deren Frequenzgänge in Abbildung 5.1 dargestellt sind. Da die Abtastrate des AC97 Audiochips 48 kHz entspricht, zeigt die Abbildung einen Frequenzbereich von 24 Hz bis 24 kHz. Filter #6 gibt dabei die kombinierte Frequenzkennlinie an, wenn alle Filter aktiv sind.

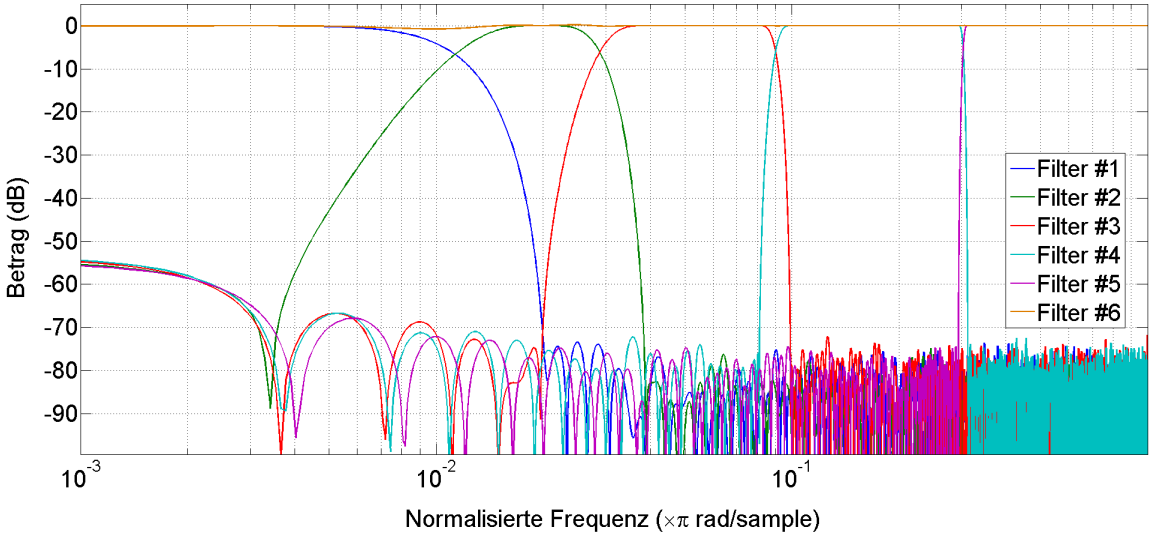

Abbildung 5.1: Logarithmischer Frequenzgang der Audiofilter

Um das Tiefpassfilter (Filter #1) zu noch tieferen Frequenzen zu verschieben, müssten entweder Filterparameter wie Steilheit, Sperrdämpfung, Durchlasswelligkeit gelockert werden oder ein längeres Filter verwendet werden. Ein längeres Filter wurde deshalb nicht verwendet, da ansonsten eine zweite Filtererweiterung für die Zweikanalverarbeitung benötigt würde. Da jedoch die Rekonfigurierbarkeit gezeigt werden soll, sollen beide Kanalfilter auf einer Erweiterung laufen, was die maximale Filterlänge pro Kanal auf 513 beschränkt.

Der Programmablauf sieht nun folgendermaßen aus. Beim Starten werden zuerst die Filter der beiden Stereo-Audiokanäle mit einer flachen Frequenzkennlinie und anschließend der Audiochip konfiguriert. Der weitere Ablauf ist vollständig Interrupt-gesteuert. Sobald acht Audiowerte abgetastet wurden (jeweils abwechselnd vier pro Audiokanal), wird ein externer Interrupt ausgelöst. In diesem Interrupt werden die Werte vom Prozessor in den Datenspeicher eingelesen, wobei der erste Wert schon per Spezialbefehl dem Filter bereitgestellt wird. Direkt nach dem Filterbefehl wird das Filter auf den zweiten Audiokanal konfiguriert, wobei diese Konfiguration erst nach der aktuellen Filterung aktiviert wird. Wenn das Filter fertig ist, wird erneut ein Interrupt ausgelöst, in dem der gefilterte Wert dem Audiocontroller zur Ausgabe übergeben wird. Zusätzlich wird die Filterung des nächsten Wertes gestartet und das Filter wieder auf den ersten Kanal konfiguriert. Dies geschieht nun so lange, bis alle acht Abtastwerte gefiltert wurden. Danach wird wieder auf den Interrupt des Audiocontrollers gewartet. Sofern einer der fünf Taster gedrückt wird, wird ebenfalls ein Interrupt ausgelöst, wobei die Taster zuvor von einer speziellen Zählerlogik in Hardware entprellt werden. In der Interrupt-Routine wird je nachdem, ob das entsprechende Bandpassfilter aktiv war oder nicht eine Variable für das Hauptprogramm entsprechend gesetzt und die zum Taster gehörende LED getoggelt. Im Hauptprogramm wird diese Variable abgefragt und je nach Aktivität des Bandpasses dessen Koeffizienten zum Gesamtfilter addiert oder subtrahiert. Nach der Berechnung der Filteränderung werden die neuen Koeffizienten in beide Kanäle der Filtererweiterung konfiguriert, was parallel zur normalen Filterung erfolgt. Somit lassen sich über sämtliche Tasterkombinationen 32 verschiedene Filterkennlinien erzeugen, wobei mit anderen Eingabemethoden natürlich auch Zwischenstufen möglich wären.

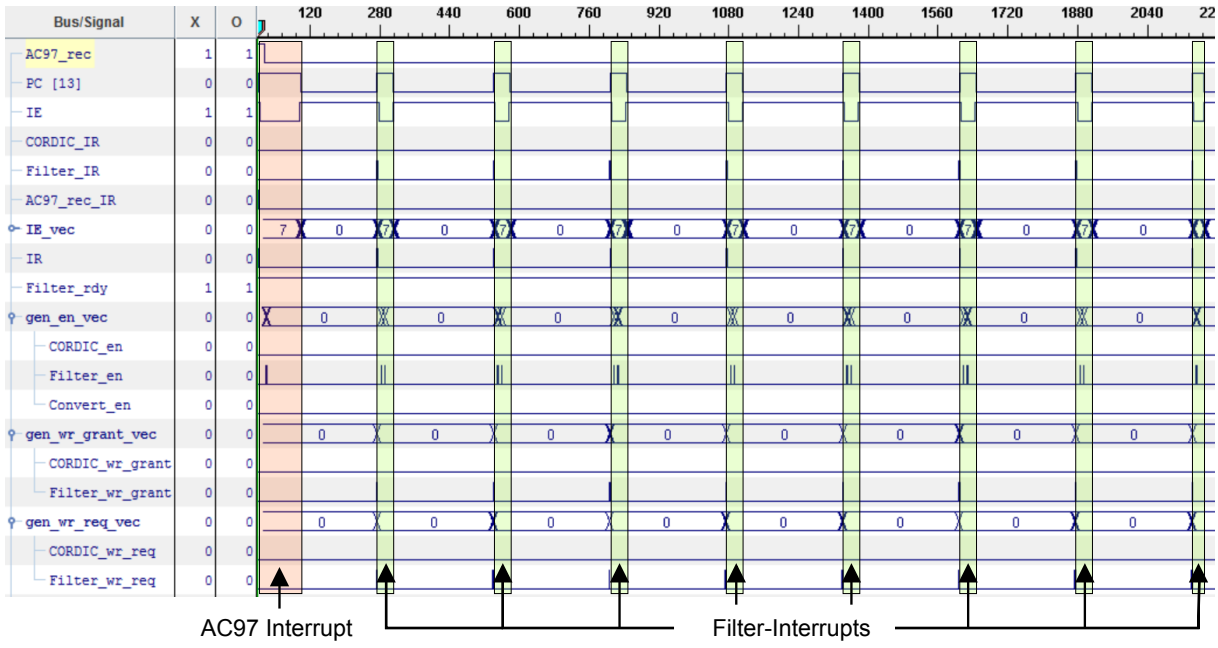

Abbildung 5.2: Hardwaresignale bei der Filterung

Durch die Interrupt-gesteuerte Programmabarbeitung mit der Filtererweiterung wird so eine Echtzeit-Audiofilterung erreicht, die mit dem Standardprozessor aus Leistungsgründen gar nicht möglich wäre. In Abbildung 5.2 sind echte Signale des Prozessors aufgezeichnet, die zeigen, dass der Prozessor die überwiegende Zeit im Leerlauf ist. Zu beachten ist dabei, dass hier nur ein Teilstück des verfügbaren Zeitintervalls für die Abarbeitung der acht Abtastwerte gezeigt ist. Das gesamte Intervall ist statt den gezeigten 2200 Taktzyklen mit 8333 fast vier Mal so lange. Somit ergibt sich eine Prozessorauslastung von 4,7% und eine Filterauslastung von 25% für zwei Kanäle. Dadurch wäre für energiesparsame Anwendungen auch eine Reduktion der Taktfrequenz möglich. Zum Vergleich: mit einem MIPS R2000 Prozessor wäre bei gleicher Taktfrequenz und Vollauslastung nur eine maximale Filterlänge von 73 möglich.

E-Gitarrenverzerrer

Das Hauptziel des zweiten Beispielprogramms ist die Nutzung aller drei Hardwareerweiterungen CORDIC, Zahlenformatkonvertierung und digitales Filter. Eine Anwendung, die sowohl ein digitales Filter, als auch nichtlineare mathematische Funktionen benötigt, ist z. B. ein Verzerrer für Elektrogitarren. Die Verzerrung wird dabei durch eine nichtlineare Funktion herbeigeführt, die dem Eingangsspektrum im Gegensatz zu linearen Systemen zusätzliche Frequenzkomponenten hinzufügt. Als Verzerrungsfunktionen eignen sich dabei nichtlineare, monotone Funktionen mit limitierendem Charakter, wie z. B. die CORDIC-Funktion Arcustangens [Yeh09]. Das heißt, egal wie groß das Eingangssignal ist, das Ausgangssignal bewegt sich bei dieser Funktion immer im Intervall $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Somit ergibt sich bei extremer Aussteuerung ein Rechtecksignal. Um jedoch die Rekonfigurierbarkeit der CORDIC-Erweiterung zu zeigen, wird auch die Sinus-Funktion für die Verzerrung verwendet, die allerdings nicht monoton ist. Daher sind Töne bei zu hoher Aussteuerung nicht mehr zu erkennen. Abbildung 5.3 zeigt die Auswirkungen der beiden verwendeten Verzerrungsfunktionen Arcustangens und Sinus auf das Spektrum eines Eingangssignals. Das Eingangssignal ist ein von einer Elektrogitarre aufgezeichneter Ton (das große A) bei ca. 110 Hz mit einer Abtastrate von 48 kHz und ist 16-fach übersteuert, was einer maximalen Amplitude des Funktionsarguments der Verzerrungsfunktion von 16 entspricht.

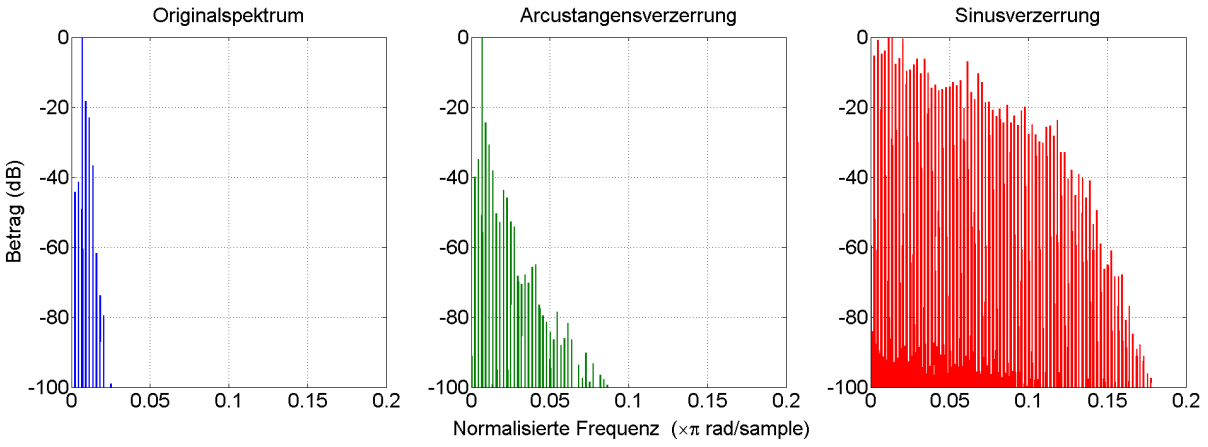

Abbildung 5.3: Originales und verzerrte Spektren eines E-Gitarren Tons

Um auch den Einfluss eines Gitarrenlautsprechers zu simulieren, erfolgt nach der Verzerrung noch eine Tiefpassfilterung mit einer Filterlänge von neun. Im Idealfall wird vor der Verzerrung Oversampling im Bereich von acht bis zehn angewendet um Aliasing durch die bei der Verzerrung entstehenden hohen Frequenzen zu verhindern [Yeh09]. Da jedoch bereits die fehlende

Gitarrenvorverstärkung in Kombination mit der geringen Auflösung des ADCs (**A**nalog/**D**igital **C**onverter) die Audioqualität sehr stark mindern, wurde dieser zusätzliche Programmieraufwand nicht durchgeführt. Bezüglich der Rechenleistung des Systems wäre das allerdings kein Problem gewesen.

Der Programmablauf ist ähnlich wie beim vorigen Demonstrationsprogramm. Beim Programmstart werden der AC97 Audiocodec, das Filter, die Konvertierungseinheit und die CORDIC-Erweiterung konfiguriert. In einer Interrupt-Routine werden acht Abtastwerte eingelesen und per Spezialbefehle nacheinander zuerst in das Gleitkommaformat konvertiert und an die CORDIC-Erweiterung übergeben. Nach Fertigstellen der CORDIC-Berechnungen erfolgt die Rückkonvertierung in das Fixkommazahlenformat, bevor Filterung und Ausgabe der Abtastwerte analog zum ersten Beispielprogramm erfolgen. Per Taster-Interrupt werden die CORDIC-Erweiterung und die Konvertierungserweiterung konfiguriert. Mit zwei Tastern kann zwischen zwischen den Verzerrungsfunktionen hin- und hergeschaltet und mit zwei weiteren Tastern die Aussteuerung des Eingangssignals bestimmt werden. Die Aussteuerung bezieht sich dabei nicht auf das analoge Signal, sondern auf das im Gleitkommaformat vorhandene digitale abgetastete Signal und kann mit einem Zweierpotenz-Faktor skaliert werden. Beide Einstellparameter beziehen sich nur auf die Konfigurationen der Erweiterungseinheiten und bedeuten daher keinen zusätzlichen Aufwand für den Prozessor. In Abbildung 5.4 sind wieder einige Hardwaresignale dargestellt, anhand derer die Auslastungen von Prozessor und Erweiterungen berechnet werden können.

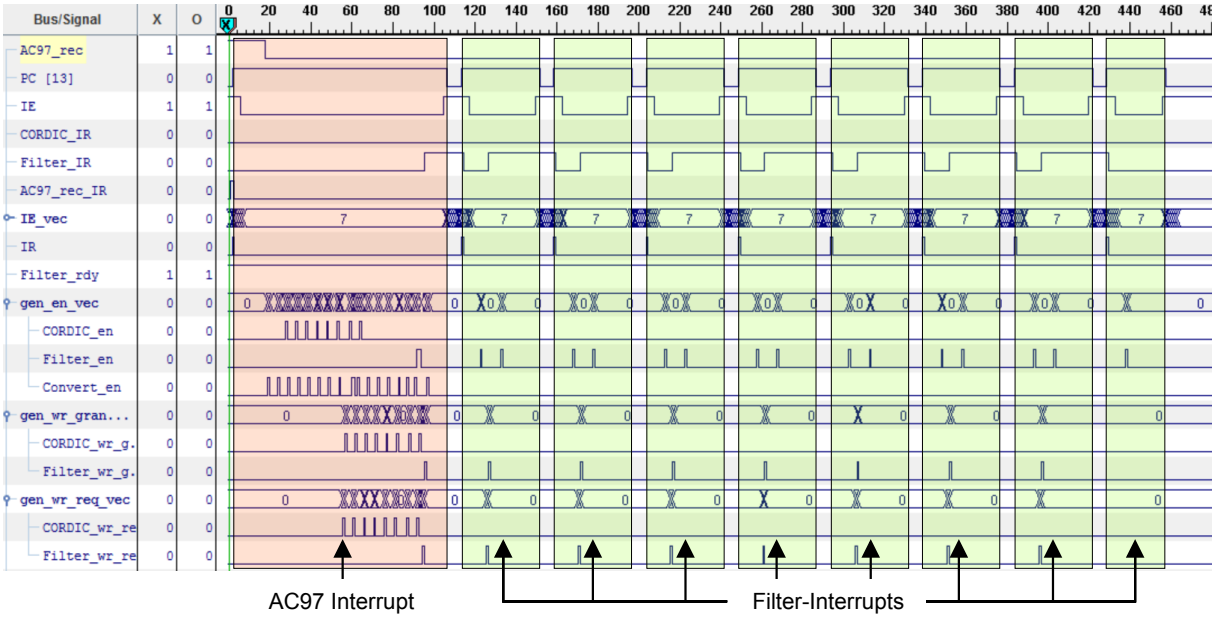

Abbildung 5.4: Hardwaresignale beim Verzerrerprogramm

Das Zeitintervall zur Bearbeitung der acht Abtastwerte ist wieder 8333 Taktzyklen lang, allerdings werden diesmal für die Verarbeitung nicht einmal ganz 460 Takte benötigt. Es ergibt sich insgesamt eine Prozessorauslastung von 4,8%, wobei die Erweiterungseinheiten fast im Leerlauf sind mit unter 0,5% Auslastung für das Filter, 0,1% für die CORDIC-Erweiterung und 0,2% für die Konvertierung. Trotz dieser sehr geringen Auslastung der Erweiterungen ergibt sich ein großer Leistungsgewinn, da der Prozessor ohne Erweiterungen zu ca. 35% ausgelastet wäre.

6 Zusammenfassung

Dieses Kapitel bildet den Abschluss der Diplomarbeit und dient dazu die wesentlichen Inhalte noch einmal in verkürzter Form zusammenzufassen. Dabei orientiert sich der Aufbau an den vorangegangenen Kapiteln. Nach dieser abschließenden Übersicht wird auch noch auf offene Punkte eingegangen, bevor das Kapitel mit einem Ausblick auf mögliche Weiterentwicklungen abgeschlossen wird.

6.1 Ausgeführte Arbeit und Erkenntnisse

Rechenleistung und Energieeffizienz stellen beim Einsatz eines Mikroprozessors oftmals die wesentlichsten physikalischen Parameter dar. Deshalb war es Ziel dieser Diplomarbeit einen Universalprozessor durch Erweiterung des Befehlssatzes daraufhin zu optimieren. Der zweite wichtige Punkt der Problemstellung war die flexible Anpassung des Prozessors an unterschiedliche Aufgabenstellungen und die Ermöglichung einer einfachen Erweiterbarkeit für zukünftige Anwendungen.

Eine Möglichkeit diese Ziele zu erreichen stellt die Erweiterung des Standardbefehlssatzes des Prozessors über rekonfigurierbare Spezialbefehle dar. Diese können durch Rekonfiguration auf neue Gegebenheiten angepasst werden und bieten daher Vorteile in Bezug auf Rechenleistung und Energieeffizienz. Während das Hauptaugenmerk der präsentierten vorangegangenen Arbeiten in Kapitel 2 auf der Laufzeitkonfigurierung der erweiterten Funktionseinheiten liegt, hat sich diese Diplomarbeit auch mit dem universellen Austausch von Erweiterungseinheiten beschäftigt. Dadurch wird mehr Flexibilität für zukünftige Anwendungen geschaffen (Auswahl geeigneter Erweiterungsbefehle bei der Implementierung des Prozessors), um eine einfache Wiederverwendung bestehender Hardware zu ermöglichen und den Prozessor zielgerichtet für die Anwendungsklasse zu optimieren. So kann der Entwicklungsaufwand oft erheblich reduziert werden, was einen wichtigen Kosten- und Zeitfaktor darstellt. Ein weiterer Vorteil der freien Wahl von Erweiterungseinheiten ist, dass diese nicht mehr unbedingt universell einsetzbar sein müssen, sondern bei Wissen über das spätere Einsatzgebiet entsprechend ausgewählt werden können um Energieeffizienz und Rechenleistung noch weiter zu steigern.

Um das zu ermöglichen wurde in Kapitel 3 ein selbst erstelltes Konzept vorgestellt, das über eine generische Schnittstelle zur Erweiterung des Basisprozessors verfügt. Dabei kann der Prozessor nicht nur mit einer, sondern mit mehreren Erweiterungseinheiten ausgestattet werden, wobei es

eine klare Rangfolge bei Ressourcenkonflikten gibt. Die Schnittstelle gestattet es zudem Operationen mit beliebiger zeitlicher Länge auszuführen, wobei trotzdem eine enge Kopplung zum Prozessor erhalten bleibt, was für eine effiziente Kommunikation wichtig ist. So befinden sich die erweiterten Funktionseinheiten parallel zur normalen Ausführungseinheit des Prozessors im Datenpfad und führen ihre Operationen auf Prozessorregistern aus.

Für die rekonfgurierbaren Erweiterungseinheiten wurde eine coarse-grain Logik gewählt, da diese schneller und energieeffizienter als fine-grain Logik ist. Darauf aufbauend wurden in Kapitel 4 drei Implementierungen von rekonfigurierbaren Erweiterungseinheiten beschrieben, die im Rahmen dieser Diplomarbeit entwickelt wurden. Die CORDIC-Erweiterung stellt häufig verwendete transzendente mathematische Funktionen auf Basis des Gleitkommazahlenformats bereit. Durch Rekonfigurierung können mit dem in Hardware implementierten Algorithmus 15 verschiedene Funktionen realisiert werden. Da es sich dabei um ein Näherungsverfahren handelt, wurde auch die erzielbare Berechnungsgenauigkeit behandelt, welche durch selbst erstellte Simulationen in Abhängigkeit vom Hardwareaufbau ermittelt wurde. Eine zweite Erweiterungseinheit, bei der nicht die Rekonfiguration im Vordergrund steht, stellt Konvertierungsfunktionen zwischen Fixkomma- und Gleitkommazahlenformat bereit. Diese sind zur Nutzung der CORDIC-Erweiterung erforderlich, da der verwendete Prozessor nur über eine Festkommaarithmetik verfügt. Außerdem wurde mit dieser zweiten Erweiterung ein alternativer Operationsmodus der generischen Schnittstelle demonstriert, der eine noch effizientere Kommunikation zwischen Prozessor und Erweiterung ermöglicht. Die letzte implementierte und vorgestellte Hardwareerweiterung ist das digitale Filter. Dieses zielt auf die Schwächen des verwendeten Prozessors bei Multiplikationsoperationen ab, die für die digitale Signalverarbeitung jedoch äußerst wichtig sind. Um Filteranwendungen zu beschleunigen, werden fertige rekonfigurierbare Filterstrukturen bereitgestellt, die eine effiziente Filterung von Signalen ermöglichen.

In Kapitel 5 wurden die Tests zur Überprüfung der korrekten Funktionsweise der Hardwarebeschreibung erörtert. Um die Testsignale zu erzeugen waren selbst erstellte Simulationen der Hardware erforderlich, welche im Kapitel vorgestellt wurden. Weiters wurden Probleme bei der Erstellung und Implementierung des Prototypen aufgegriffen und erörtert. Im letzten Abschnitt wurden Aussagen über die zu erreichenden Leistungssteigerungen getroffen, indem ausgehend von theoretischen Überlegungen bis hin zu konkreten Realisierungen Vergleiche zwischen Implementierungen mit und ohne Befehlssatzerweiterung präsentiert wurden. Bei den konkreten Realisierungen handelt es sich um Demonstrationsprogramme, die von den Befehlssatzerweiterungen und deren Rekonfigurierbarkeit Gebrauch machen, wobei auch Aufbau und Erstellung dieser Programme Thema waren. Die präsentierten Leistungsgrößen zeigen dabei in allen Fällen deutliche Gewinne, sofern die Befehlssatzerweiterungen zum Einsatz kommen.

6.2 Nicht behandelte Punkte

Da für die Diplomarbeit nur eine begrenzte Zeit zur Verfügung stand, wurden aus Aufwandsgründen relevante Punkte bewusst weggelassen, welche nicht unerwähnt bleiben sollen. Deshalb ist der folgende Abschnitt diesem Thema gewidmet, wobei auch gewisse Nachteile der erstellten Implementierung angesprochen werden.

Worauf in den vorangegangenen Kapiteln bereits hingewiesen wurde, ist die Wichtigkeit einer Entwicklungsumgebung zur Erstellung von Programmen und Konfigurationen für einen Prozessor mit rekonfigurierbaren Befehlssatzerweiterungen. Zwar können vorhandene Compiler weiterhin benutzt werden, doch hilft das wenig, wenn von Hardwareerweiterungen Gebrauch gemacht

werden soll. Für die Erstellung der kleinen Demonstrationsprogramme hat die manuelle Erstellung einer Nachbearbeitungsroutine ausgereicht, jedoch ist das in der Praxis nicht zielführend. So soll der Compiler im Stande sein die zur Verfügung gestellten Hardwareerweiterungen optimal auszunutzen. Dabei bezieht sich die Ausnutzung nicht nur auf die automatische Selektion von Erweiterungsbefehlen, sondern auch auf die Erstellung von Hardwarekonfigurationen. Im Idealfall soll dem Compiler nur mitgeteilt werden, welche Hardwareerweiterungen verfügbar sind und alles weitere funktioniert auch für eventuelle fine-grain Erweiterungen vollautomatisch. Nur so wäre gewährleistet, dass der Programmierer keine Kenntnis von der zugrundeliegenden Hardware benötigt. Doch schon das alleinige Erstellen eines solchen Compilers würde vermutlich den Umfang einer Diplomarbeit sprengen.

Ein weiterer Punkt, der sich auf die Programmierung des Prozessors bezieht, ist die Behandlung des Zielregisters bei länger dauernden Spezialbefehlen. Auf dieses Register kann nämlich zu jeder Zeit auch während der Ausführung eines Spezialbefehls schreibend und lesend zugegriffen werden. Um hier keine Fehler bei der Programmabarbeitung zu verursachen sollten Maßnahmen getroffen werden, um zu frühe oder falsche Zugriffe zu verhindern. Das kann entweder auf Softwareseite durch den Compiler, als auch seitens der Hardware erfolgen. Entscheidet man sich für Letzteres ist allerdings entweder eine zusätzliche Steuerleitung in der Schnittstellendefinition oder ein vordefiniertes Bit im Spezialbefehl notwendig, um Operationen zu kennzeichnen, die überhaupt ein Ergebnis liefern. Anhand dieser Informationen könnten so die entsprechenden Zielregister bis zur Verfügbarkeit des Ergebnisses einer Spezialoperationen gesperrt werden.

Neben den eben vorgestellten Programmierherausforderungen und des wenig leistungsstarken Basisprozessors hat der vorgestellte Prototyp auch weitere Nachteile, wovon zwei vorgestellt werden. Es wird zwar in der Diplomarbeit mehrfach erwähnt, dass Spezialoperationen mit einer Länge von zwei oder mehr Taktzyklen unterstützt werden, jedoch kann das für sehr einfache Operationen wie z. B. Bitmanipulationen in Kombination mit Datenabhängigkeiten einen Nachteil darstellen. So muss egal wie kurz die Dauer des Spezialbefehls ist mindestens zwei Takte nach dem Befehl gewartet werden, bis das Ergebnis weiterverarbeitet werden kann. Dabei könnte das Ergebnis auch schon früher wieder in den Datenpfad des Prozessors zurückgeführt werden, was etwa bei der in Kapitel 2.4 präsentierten Architektur genutzt wird.

Ein weiterer Nachteil ergibt sich durch die Beschränkung der Anzahl an Operanden für einen Spezialbefehl und den möglichen Ressourcenkonflikten zwischen den Funktionseinheiten. So stellen die Registerzugriffe einen Flaschenhals dar, durch den mächtigere Befehle zur Verarbeitung mehrerer Daten oder sogar Datenströmen ineffizient werden. Hier könnte zumindest für eine Prozessorerweiterung eine zusätzliche Speicherschnittstelle Abhilfe schaffen. Zwar würde sich das vom Prinzip der erweiterten Funktionseinheiten etwas abheben, jedoch könnten so weitere Leistungssteigerungen erzielt und die Flexibilität erhöht werden. Die dafür zusätzlich erforderliche Hardware zur Speicheranbindung und Gewährleistung der Kohärenz von Datenspeicher und eventuell vorhandenen Caches ist jedoch auch zu beachten.

Ein noch nicht zufriedenstellender Punkt der Diplomarbeit ist, dass die Leistungsevaluierungen und Demonstrationsprogramme aus Kapitel 5.2.5 wenig Aufschluss über Einsatz des Prozessors in unterschiedlichsten Applikationen geben. So wurden die Demonstrationsprogramme auf eine möglichst gute Ausnutzung der implementierten Erweiterungen ausgelegt. Echte Benchmarks würden daher nicht so überzeugende Ergebnisse liefern, sind aber aufgrund der fehlenden Entwicklungssoftware nicht mit vernünftigen Aufwand realisierbar. Bezüglich Energieeffizienz würde es neben ordentlichen Benchmarks auch genaue Messungen der Leistungsaufnahme erfordern.

Zusätzlich müsste auch der Einfluss der verwendeten DSP48-Funktionsblöcke (siehe [11]) berücksichtigt werden, um repräsentative Vergleiche zu erhalten. Erst nach Durchführung solcher Leistungsevaluierungen können Erkenntnisse für den Praxiseinsatz gewonnen werden.

6.3 Ausblick

Neben den im vorigen Abschnitt erwähnten Punkten gibt es auch noch andere Möglichkeiten für weiterführende Arbeiten. Einige davon werden im Folgenden aufgezählt.

Es könnten weitere rekonfigurierbare Erweiterungseinheiten mit einem breiteren Einsatzgebiet und mehr Rekonfigurationsmöglichkeiten entwickelt werden. Ein Beispiel wäre eine Erweiterung, die ähnlich wie bei der ReRISC-Architektur (siehe Kapitel 2.4) über coarse-grain PEs und rekonfigurierbaren Verbindungen verfügt. So könnte hier durch die richtige Verschaltung der PEs ein breites Spektrum komplexer und weniger komplexer Funktionen realisiert werden. Mit der ausreichenden Flexibilität und Anzahl an PEs ließe sich auf diese Weise sogar eine Konfiguration ähnlich der CORDIC-Erweiterung erstellen.

Ein weites Gebiet für Weiterentwicklungen wäre die Anwendung des Konzepts auf Multikernprozessorsysteme, bei denen verschiedene Prozessorkerne mit unterschiedlichen Erweiterungen ausgestattet werden, um die Effizienz eines Prozessors weiter zu steigern. So könnten diese Kerne für unterschiedliche Aufgaben, ähnlich wie in [BC11] vorgeschlagen ausgelegt und entsprechend ihrer Aufgabe erweitert werden. Dabei wäre auch der Extremfall denkbar, in dem die Standardausführungseinheit eines Prozessorkerns durch Erweiterungseinheiten ersetzt wird und somit nur die absolut notwendigen Funktionen unterstützt werden. Ohne Standardausführungseinheit wäre somit die Prozessorflexibilisierung maximal, wodurch eine bestmögliche Anpassung ohne viel Entwicklungsaufwand erreicht werden kann. Es ginge zwar die universelle Einsetzbarkeit eines Prozessorkerns verloren, doch durch den Einsatz mehrerer unterschiedlich spezialisierter Kerne könnten die Auswirkungen minimiert werden.

Um die Energieeffizienz weiter zu steigern wäre es sinnvoll gerade nicht benötigte Funktionseinheiten abzuschalten, wie es bei modernen Mikroprozessoren bereits der Fall ist [RRK09]. Somit würden sich die zusätzlichen Funktionseinheiten nur dann im Energieverbrauch bemerkbar machen, wenn sie gerade aktiv sind. Dadurch wäre die Energieeffizienz auch im schlimmsten Fall bei kompletter Nichtbenutzung der Erweiterungen nicht schlechter als ohne Erweiterungen. Bei zumindest einer teilweisen Benutzung der Erweiterungseinheiten ergäbe sich somit in jedem Fall eine Steigerung der Energieeffizienz, da die Erweiterung für eine effiziente Prozessierung ausgelegt sind.

A Anhang

Das folgende Kapitel beinhaltet Zusatzinformationen, die für die Diplomarbeit selbst zwar nicht so sehr von Bedeutung sind, allerdings zur Programmierung des implementierten Prozessors und der Verwendung der Erweiterungseinheiten wichtig sind. Eine Ausnahme bildet des Gleitkommazahlenformat, auf das in mehreren vorangegangenen Kapiteln Bezug genommen wurde.

A.1 IEEE754 Gleitkommazahlenformat

Das IEEE Gleitkommazahlenformat [IEE08] dient zur Darstellung von Zahlen sehr unterschiedlicher Größenordnungen. Im Vergleich zum Fixkommaformat werden Zahlenüberläufe vermieden und die Genauigkeit der Zahlendarstellung ist durch die automatische Skalierung konstant. Diese Aussagen gelten natürlich nur für den endlichen normalen Darstellungsbereich von Gleitkommazahlen, welcher für einfache Genauigkeit betragsmäßig von ca. $1{,}17 \cdot 10^{-38}$ bis $3{,}4 \cdot 10^{38}$ reicht. Neben dem normalen Darstellungsbereich gibt es aber auch noch Bitkombinationen um spezielle Werte wie unendlich und NAN zu kennzeichnen. Zusätzlich können am unteren Ende des normalen Zahlenbereichs weitere Zahlen mit abnehmender Genauigkeit dargestellt werden. Diese Zahlen werden als denormalisierte Zahlen bezeichnet, zu denen auch die Null gehört. Nach dieser ist der kleinste darstellbare Betrag ca. $1{,}4 \cdot 10^{-45}$. Der binäre Aufbau des Zahlenformats ist in Tabelle A.1 dargestellt, wobei 'V' als Abkürzung für Vorzeichen steht.

Tabelle A.1: Aufbau des Gleitkommazahlenformats einfacher Genauigkeit

Bit			
31	30 23	22	0
V	Exponent (8 Bit)	Mantisse (23 Bit)	

Durch das extra Vorzeichenbit existieren zwei Darstellungen des Wertes null. Wenn dieses Bit gesetzt ist, bedeutet das eine negative Zahl. Der Exponent ist ohne Vorzeichen codiert, hat aber einen Offset von -127 und bezieht sich auf die Basis zwei. Die Mantisse ist ebenfalls ohne Vorzeichen im I.Q-Format 0.23 codiert, wobei im normalen Darstellungsbereich ein Offset von eins angenommen wird. In Tabelle A.2 werden die Umrechnungen dieses Binärformats in die entsprechenden Dezimalwerte inklusive der oben genannten Sonderfälle gezeigt.

Tabelle A.2: Interpretationen des Gleitkommazahlenformats

Exponent	Mantisse	Dezimalwert	Kathegorie
255	egal	$(-1)^{\mathrm{V}} \cdot \mathrm{Mantisse} \cdot 2^{-126}$	denormalisierte Zahl
[1, 254]	egal	$(-1)^{\mathrm{V}} \cdot (1 + \mathrm{Mantisse}) \cdot 2^{\mathrm{Exponent}-127}$	normalisierte Zahl
255	0	$(-1)^{\mathrm{V}} \cdot \infty$	unendlich
255	>0	NAN	ungültige Zahl

A.2 Belegung des Special Function Registers

Um die E/A-Ports des implementierten Prozessors anzusprechen und Interrupts konfigurieren zu können, benötigt man bei der Programmierung des Prozessors Kenntnis über die Adressbelegung des SFR (**S**pecial **F**unction **R**egister). Diese Belegung ist in Tabelle A.3 dargestellt, wobei die Speicheradressen Absolutadressen darstellen. Alle Speicherinhalte des SFR werden beim Start mit null initialisiert, um einen definierten Start zu ermöglichen. Somit werden auch alle E/A-Ports als Eingang geschaltet. Die in Klammer stehenden Abkürzungen geben die Art der möglichen Speicherzugriffe an. R (**R**ead) steht für Lesezugriff, W (**W**rite) für Schreibzugriff.

Tabelle A.3: Belegung des SFR

Basis-adresse	**Byte 3** 31 ⟷ 24	**Byte 2** 23 ⟷ 16	**Byte 1** 15 ⟷ 8	**Byte 0** 7 ⟷ 0
0x0000	Port A (R/W)			
0x0004	Port B (R/W)			
0x0008	Port C (R/W)			
0x000C	Port D (R/W)			
0x0010	Port A Datenrichtung: 0...Eingang (R/W)			
0x0014	Port B Datenrichtung: 0...Eingang (R/W)			
0x0018	Port C Datenrichtung: 0...Eingang (R/W)			
0x001C	Port D Datenrichtung: 0...Eingang (R/W)			
0x0020	interne IRQs (R)	externe IRQs (R)	Maske interne Interrupts (R/W)	Maske interne Interrupts (R/W)
0x0024	interne IRQs löschen (R/W)	externe IRQs löschen (R/W)	externe Interrupts Trigger auf beide Flanken (R/W)	externe Interrupts Triggerflanke: 0...positiv (R/W)
0x0028	Bit 31: Interrupts aktiviert (R/W)	Bit 23: automatische Stackpointer-Verwaltung aktiviert (R/W)	Bit 8: Timer aktivieren (R/W) Bit 9: Timer Vergleichswert setzen (R/W)	Erweiterungseinheiten Einsatzbereit (R), Bitnummer = Erweiterungsnummer
0x0038	Timer Zählerresetwert (R) / Zähler(reset)wert (W)			
0x003C	Timer Vergleichswert (R/W)			

Der Adressbereich von 0x0029 bis 0x0037 ist nicht belegt und kann daher wie der normale Datenspeicher verwendet werden. Ab der Adresse 0x0040 werden die 16 frei wählbaren Interrupt-Sprungadressen abgelegt, wobei der Interrupt für die Zahlenüberlaufsbehandlung die höchste Nummer besitzt und daher auf der Basisadresse 0x007C liegt.

A.3 Erweiterungsbefehlssätze

In den Tabellen A.4 bis A.7 werden die Befehlssätze der implementierten Erweiterungseinheiten dargestellt. Dabei wird vom 11 Bit langen Spezialbefehl ausgegangen, der von der generischen Schnittstelle kommt. Der gesamte Aufbau eines gültigen Maschinenbefehls kann mit Hilfe von Tabelle 3.2 aus Kapitel 3.3.5 ermittelt werden. Die Befehle sind jeweils bitweise dargestellt und ein 'X' gibt an, dass das jeweilige Bit irrelevant ist. Alle nicht dargestellten Befehle sind ungültig und dürfen nicht verwendet werden, da ihr Verhalten nicht definiert ist.

Tabelle A.4: Befehlssatz der CORDIC-Erweiterung

Spezialbefehl	Operation	Bemerkung
`0bXXXXX0X0000`	keine Operation	liefert kein Ergebnis
`0bXXXXX000010`	$\sqrt{in_1^2 - in_2^2}$	
`0bXXXXX000011`	$in_1 \cosh(in_2)$	stark eingeschränkter Konvergenzbereich
`0bXXXXX000101`	$in_1 \mathrm{e}^{in_2}$	
`0bXXXXX001010`	$\sqrt{in_1^2 + in_2^2}$	
`0bXXXXX001011`	$in_1 \cos(in_2)$	
`0bXXXXX001110`	$\sqrt{\lvert in_1 \rvert}$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX010010`	$\operatorname{artanh}(\frac{in_2}{in_1})$	eingeschränkter Konvergenzbereich
`0bXXXXX010011`	$in_1 \sinh(in_2)$	stark eingeschränkter Konvergenzbereich
`0bXXXXX010100`	$\log(\lvert in_1 \rvert)$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX010110`	$\mathrm{ld}(\lvert in_1 \rvert)$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX011000`	$\frac{in_2}{in_1}$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX011001`	$in_1 in_2$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX011010`	$\arctan2(in_2, in_1)$	
`0bXXXXX011011`	$in_1 \sin(in_2)$	
`0bXXXXX011100`	$\ln(\lvert in_1 \rvert)$	Konvergenz nur bei normalisierten Zahlen
`0bXXXXX10XXXX`	konfigurierte Operation	auszuführende Operation ist in lokalem Operationsregister gespeichert
`0bXXXXX11XXXX`	keine Operation / Konfiguration	Operationsregister wird mit den unteren fünf Bit von in_2 beschrieben

Tabelle A.5: Befehlsaufbau der Konvertierungserweiterung

Spezialbefehl	Bedeutung
Bits 11 bis 3	Position des Fixkommas wenn Konfiguration nicht verwendet wird, andernfalls irrelevant
Bit 2	steuert Konfiguration des Registers für Fixkommaposition 0: keine Konfiguration 1: Register für Fixkommaposition wird mit unteren 8 Bit von in_2 konfiguriert
Bit 1	selektiert Operand für Position des Fixkommas 0: Operand sind die oberen 8 Bit des Spezialbefehls 1: Operand ist in lokalem Register für Fixkommaposition gespeichert
Bit 0	steuert Konvertierungsrichtung 0: Gleit- → Fixkomma 1: Fix- → Gleitkomma

Tabelle A.6: Befehlsaufbau der Filterbefehle für die Filtererweiterung

Spezialbefehl	**Bedeutung**
Bits 11 bis 9	keine Auswirkung
Bits 8 bis 6	legt Ausrichtung des Filterergebnisses fest 000: Ergebnis wird im I.Q-Format 8.24 zurückgegeben bis 111: Ergebnis wird im I.Q-Format 1.31 zurückgegeben
Bit 5	für Filterbefehl immer eins
Bit 4*	gibt an ob Filterstufe nach Filterung gesetzt werden soll 0: Filterstufe wird nicht verändert 1: Filterstufe wird mit unteren 8 Bit von in_2 gesetzt
Bit 3*	gibt an ob Filterstufe nach Filterung inkrementiert werden soll 0: Filterstufe wird nicht verändert 1: Filterstufe wird inkrementiert
Bit 2	selektiert zwischen FIR und IIR Filterkonfiguration 0: FIR Filterkonfiguration 1: IIR Filterkonfiguration
Bit 1	gibt an ob aktuelle Filterstufe oder konfigurierter Filter zum Einsatz kommt 0: aktuelle Filterstufe 1: konfigurierter Filter
Bit 0	gibt an ob letzte Filterung mit aktuellem Filter fortgesetzt wird 0: neue Filterung mit neuem Eingangswert wird gestartet 1: aktueller Filter wird mit letztem Filter verknüpft und führt Filterung fort

*werden beide Bits gleichzeitig gesetzt, so werden statt einer Filterung die Filterzustände gelöscht; Bit 4 wird auf den Wert von Bit 2 gesetzt und Bit 3 auf den von Bit 0; es wird kein Ergebnis zurückgeliefert

Tabelle A.7: Befehlsaufbau der Konfigurationsbefehle für die Filtererweiterung

Spezialbefehl	**Bedeutung**
Bits 11 bis 6	keine Auswirkung
Bit 5	für Konfigurationsbefehl immer null
Bit 4	gibt an ob Filterstufe nach Filterung gesetzt werden soll 0: Filterstufe wird nicht verändert 1: Filterstufe wird mit unteren acht Bit von in_2 gesetzt
Bit 3	gibt an ob Filterstufe nach Konfiguration von zwei Filterkoeffizienten inkrementiert werden soll 0: Filterstufe wird nicht verändert 1: Filterstufe wird inkrementiert
Bits 2 bis 1	selektiert Konfigurationsbefehl 00: keine Operation 01: zwei Filterkoeffizienten werden mit in_1 und in_2 konfiguriert 10: Filter wird aus den Filterstufen in_1 bis in_2 zusammengesetzt 11: ein Filterkoeffizient wird mit in_1 konfiguriert Position des Koeffizienten wird mit Bits 10 bis 8 von in_2 selektiert Filterstufe für Konfigurierung wird mit unteren 8 Bit von in_2 gesetzt
Bit 0	gibt bei Befehlen zur Konfiguration von zwei Filterkoeffizienten die Position innerhalb einer Filterstufe an, ansonsten irrelevant 0: Filterkoeffizienten null und eins werden beschrieben 1: Filterkoeffizienten zwei und drei werden beschrieben

Wissenschaftliche Literatur

[And98] Andraka, Ray: A Survey of CORDIC Algorithms for FPGA Based Computers. In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 1998 (FPGA '98). – ISBN 0–89791–978–5, S. 191–200

[BC11] Borkar, Shekhar ; Chien, Andrew A.: The Future of Microprocessors. In: *Commun. ACM* 54 (2011), Mai, Nr. 5, S. 67–77. – ISSN 0001–0782

[BMS13] Blem, E. ; Menon, J. ; Sankaralingam, K.: Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013. – ISSN 1530–0897, S. 1–12

[Bon98] Bondyopadhyay, P.K.: Moore's law governs the silicon revolution. In: *Proceedings of the IEEE* 86 (1998), Jan, Nr. 1, S. 78–81. – ISSN 0018–9219

[Dag59] Daggett, D. H.: Decimal-Binary Conversions in CORDIC. In: *Electronic Computers, IRE Transactions on* EC-8 (1959), Nr. 3, S. 335–339. – ISSN 0367–9950

[Dob04] Doblinger, Gerhard: *Signalprozessoren*. Wilburgstetten, Deutschland : J.Schlembach Fachverlag, 2004

[Dob07] Doblinger, Gerhard: *Zeitdiskrete Signale und Systeme - Eine Einführung in die grundlegenden Methoden der digitalen Signalverarbeitung*. Wilburgstetten : J.Schlembach Fachverlag, 2007

[GB11] Galuzzi, Carlo ; Bertels, Koen: The Instruction-Set Extension Problem: A Survey. In: *ACM Trans. Reconfigurable Technol. Syst.* 4 (2011), Mai, Nr. 2, S. 18:1–18:28. – ISSN 1936–7406

[GDHG10] Glaser, Johann ; Damm, Markus ; Haase, Jan ; Grimm, Christoph: A Dedicated Reconfigurable Architecture for Finite State Machines. In: Sirisuk, Phaophak (Hrsg.) ; Morgan, Fearghal (Hrsg.) ; El-Ghazawi, Tarek A. (Hrsg.) ; Amano, Hideharu (Hrsg.): *ARC* Bd. 5992, Springer, 2010. – ISBN 978–3–642–12132–6, S. 122–133

[HC83] Huntsman, C. ; Cawthron, D.: The MC68881 Floating-point Coprocessor. In: *Micro, IEEE* 3 (1983), Nr. 6, S. 44–54. – ISSN 0272–1732

[HD07] Hauck, Scott ; DeHon, Andre: *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. – ISBN 0123705223, 9780080556017, 9780123705228

[HHB91] Hu, X. ; Harber, R.G. ; Bass, S.C.: Expanding the range of convergence of the

CORDIC algorithm. In: *Computers, IEEE Transactions on* 40 (1991), Nr. 1, S. 13–21. – ISSN 0018–9340

[IEE08] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2008* (2008), S. 1–70

[Jam95] Jamil, T.: RISC versus CISC. In: *Potentials, IEEE* 14 (1995), Nr. 3, S. 13–16. – ISSN 0278–6648

[JC99] Jacob, Jeffrey A. ; Chow, Paul: Memory Interfacing and Instruction Specification for Reconfigurable Processors. In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 1999 (FPGA '99). – ISBN 1–58113–088–0, S. 145–154

[KH92] Kane, Gerry ; Heinrich, Joe: *MIPS RISC Architectures*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1992. – ISBN 0–13–590472–2

[RPJ+84] Rowen, C. ; Przbylski, S. ; Jouppi, N. ; Gross, T. ; Shott, J. ; Hennessy, J.: A pipelined 32b NMOS microprocessor. In: *Solid-State Circuits Conference. Digest of Technical Papers. 1984 IEEE International* Bd. XXVII, 1984, S. 180–181

[RRK09] Roy, S. ; Ranganathan, N. ; Katkoori, S.: A Framework for Power-Gating Functional Units in Embedded Microprocessors. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 17 (2009), Nr. 11, S. 1640–1649. – ISSN 1063–8210

[RS94] Razdan, R. ; Smith, M.D.: A high-performance microarchitecture with hardware-programmable functional units. In: *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, 1994. – ISSN 1072–4451, S. 172–180

[TCE+95] Tau, E. ; Chen, D. ; Eslick, I. ; Brown, J. ; Dehon, A.: A First Generation DPGA Implementation. In: *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, 1995, S. 138–143

[VKTN06] Vassiliadis, N. ; Kavvadias, N. ; Theodoridis, G. ; Nikolaidis, S.: A RISC architecture extended by an efficient tightly coupled reconfigurable unit. In: *International Journal of Electronics* 93 (2006), Juni, Nr. 6, S. 421–438. – ISSN 0020–7217

[Vol59] Volder, Jack E.: The CORDIC Trigonometric Computing Technique. In: *Electronic Computers, IRE Transactions on* EC-8 (1959), Nr. 3, S. 330–334. – ISSN 0367–9950

[VS06] Vollmar, Kenneth ; Sanderson, Pete: MARS: An Education-oriented MIPS Assembly Language Simulator. In: *SIGCSE Bull.* 38 (2006), März, Nr. 1, S. 239–243. – ISSN 0097–8418

[VS07] Vassiliadis, Stamatis ; Soudris, Dimitrios: *Fine- and Coarse-Grain Reconfigurable Computing*. 1st. Springer Publishing Company, Incorporated, 2007. – ISBN 1402065043, 9781402065040

[Wal71] Walther, J. S.: A Unified Algorithm for Elementary Functions. In: *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1971 (AFIPS '71 (Spring)), S. 379–385

[WC96] Wittig, R.D. ; Chow, P.: OneChip: an FPGA processor with reconfigurable logic. In: *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, 1996, S. 126–135

[Yeh09] Yeh, David Te-Mao: *DIGITAL IMPLEMENTATION OF MUSICAL DISTORTION CIRCUITS BY ANALYSIS AND SIMULATION*, Diss., June 2009

Internet Referenzen

[1] Analog Devices. Blackfin Embedded Processor, 2011. http://www.analog.com/static/imported-files/data_sheets/ADSP-BF504_BF504F_BF506F.pdf [abgerufen am: 14.01.2014].

[2] Canalys. Mobile device market to reach 2.6 billion units by 2016, 2013. http://www.canalys.com/static/press_release/2013/canalys-press-release-220213-mobile-device-market-reach-26-billion-units-2016_0.pdf [abgerufen am: 18.02.2014].

[3] Intel. Audio Codec '97, 2002. http://www-inst.eecs.berkeley.edu/~cs150/Documents/ac97_r23.pdf [abgerufen am: 16.01.2014].

[4] INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. 2011 EDITION SYSTEM DRIVERS, 2011. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011SysDrivers.pdf [abgerufen am: 13.02.2014].

[5] K. Karimi and G. Atkinson. What the Internet of Things (IoT) Needs to Become a Reality, 2013. http://www.freescale.com/files/32bit/doc/white_paper/INTOTHNGSWP.pdf [abgerufen am: 13.02.2014].

[6] C. Leung and W. Ng. AC97 SOUND CONTROLLER WITH DEVICE DRIVER, 2004. http://www.eecg.toronto.edu/~pc/courses/432/2004/projects/ac97controller.doc [abgerufen am: 16.01.2014].

[7] National Instruments. Technologieausblick 2013 – Embedded-Systeme, 2013. ftp://ftp.ni.com/pub/branches/germany/2013/brochures/embedded-systems-outlook-ger.pdf [abgerufen am: 17.02.2014].

[8] Texas Instuments. TMS320C5515 Fixed-Point Digital Signal Processor, 2013. http://www.ti.com/lit/ds/symlink/tms320c5515.pdf [abgerufen am: 14.01.2014].

[9] Transparency Market Research. Global Embedded System Market is Expected to Reach USD 194.27 Billion in 2018: Transparency Market Research, 2013. http://www.prnewswire.com/news-releases/global-embedded-system-market-is-expected-to-reach-usd-19427-billion-in-2018-transparency-market-research-217360981.html [abgerufen am: 17.02.2014].

[10] Xilinx. ML405 Evaluation Platform, 2008. http://www.xilinx.com/support/documentation/boards_and_kits/ug210.pdf [abgerufen am: 08.01.2014].

[11] Xilinx. XtremeDSP for Virtex-4 FPGAs, 2008. http://www.xilinx.com/support/documentation/user_guides/ug073.pdf [abgerufen am: 19.02.2014].

[12] Xilinx. Virtex-4 FPGA Data Sheet: DC and Switching Characteristics, 2009. `http://www.xilinx.com/support/documentation/data_sheets/ds302.pdf` [abgerufen am: 16.01.2014].

[13] Xilinx. Zynq-7000 All Programmable SoC, 2013. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf` [abgerufen am: 10.01.2014].

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 08.04.2014 _____

Günther Mader