Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Dipl.-Ing. Thorsten Jungeblut

Referent: Prof. Dr.-Ing. Ulrich Rückert
Korreferent: Prof. Dr. rer. nat. Uwe Kastens
Korreferent: Dr.-Ing. Mario Porrmann

Tag der mündlichen Prüfung: 25.10.2011

Bielefeld / Mai 2011
DISS KS / 01

Danksagung

Ich möchte mich an dieser Stelle bei all jenen Personen bedanken, die mich während meiner Arbeit auf unterschiedliche Weise unterstützt haben.

Besonderer Dank gilt meinem Doktorvater Professor Ulrich Rückert dafür, dass er mich in seine Fachgruppe aufgenommen hat und mir die Möglichkeit zur Promotion gegeben hat. Seine konstruktive Kritik hat mir beim Abschluss meiner Arbeit sehr geholfen. Des Weiteren danke ich Professor Uwe Kastens für die Übernahme des Korreferats und der immer sehr angenehmen und guten Zusammenarbeit mit der von ihm geleiteten Fachgruppe „Programmiersprachen und Übersetzer“. Ich wünsche mir, dass diese Kooperation noch lange bestehen wird.

Besonderer Dank gilt auch Dr. Mario Porrmann für seine Hilfsbereitschaft, die fachlichen Diskussionen und das Korrekturlesen, das sehr zum Inhalt dieser Arbeit beigetragen hat.

Dr. Michael Thies und Ralf Dreesen danke ich für die sehr gute Zusammenarbeit im Rahmen verschiedener Projekte. Beide standen mir stets für Fragen zur Verfügung.

Gregor Sievers, Johannes Ax, Sven Lütkemeier und Boris Hübener danke ich für die äußerst gute Zusammenarbeit im Rahmen der Entwicklung des CoreVA-Prozessors – gerade dann, wenn die Zeit wieder einmal sehr knapp war. Die Arbeit mit ihnen hat viel Spaß gemacht! Jens Hagemeyer danke ich für sein unerschöpfliches Wissen, nicht nur im Bereich von FPGAs. Per Wilhelm danke ich für die schöne gemeinschaftliche Zeit in unserem Büro und die vielen (auch nicht-fachlichen) Diskussionen.

Abschließend möchte ich noch meinen Eltern für ihren Rückhalt und ihre Unterstützung danken. Sie haben mir mein Studium ermöglicht und damit überhaupt den Grundstein für meine Promotion gelegt. Ihr abschließendes Korrekturlesen dieser Arbeit hat mir sehr geholfen. Meiner Lebenspartnerin Corinna Peters danke ich für ihre liebevolle Unterstützung. Sie stand mir auch in den zeitweise sehr stressigen Phasen immer zur Seite. Danke, dass ich dich habe!

Zusammenfassung

Die zunehmende Miniaturisierung digitaler Schaltkreise durch moderne Fertigungsverfahren und die damit verbundene steigende Integrationsdichte von mikroelektronischen Schaltkreisen erlaubt die Realisierung von immer komplexeren und leistungsfähigeren Prozessoren. Die Steigerung der Performanz durch eine reine Erhöhung der Taktfrequenz wirkt sich jedoch nachteilig auf die Leistungsaufnahme des Systems aus. Neue Architekturen stellen die geforderte Leistungsfähigkeit durch eine höhere Parallelität zur Verfügung. Diese ermöglicht eine höhere Energieeffizienz, da die Taktfrequenz eines Parallelprozessors vergleichsweise niedrig gehalten werden kann. Es gilt, eine hohe Ressourceneffizienz, d.h. einen guten Kompromiss zwischen Performanz und Bedarf an Ressourcen, wie Fläche oder Leistungsaufnahme, zu erreichen. Die eng gekoppelten Funktionseinheiten skalierbarer Very-Long-Instruction-Word (VLIW)-Prozessoren eignen sich insbesondere für Anwendungsszenarien, in denen eine hohe Ressourceneffizienz gefordert ist.

Diese Arbeit dokumentiert die Entwurfsraumexploration einer skalierbaren und ressourceneffizienten VLIW-Architektur – dem CoreVA-Prozessor. Als Grundlage der Entwicklung dient ein, in Kooperation mit der Fachgruppe „Programmiersprachen und Übersetzer“ der Universität Paderborn entwickelter, dualer Entwurfsablauf, der auf einer zentralen Prozessorspezifikation basiert. Der hohe Automatismus dieses Entwurfsablaufs ermöglicht kürzere Iterationszyklen während der Entwicklung und somit die Abdeckung größerer Entwurfsräume, als es bisher möglich war. Ziel der Entwicklung war die Implementierung und Realisierung einer anwendungsspezifischen Architektur, die möglichst gut an das jeweilige Anwendungsszenario angepasst ist. Die Nutzbarkeit des in dieser Arbeit entwickelten Entwurfsablaufes wird anhand der Entwurfsraumexploration des CoreVA-Prozessors gezeigt. Neben der Exploration der funktionalen Parallelität des Prozessorkerns wird auch eine Analyse der Forwarding-Architektur und des Speicher-Subsystems vorgestellt. Zur weiteren Steigerung der Ressourceneffizienz können Hardware-Beschleuniger an das CoreVA-System gekoppelt werden. Verschiedene Anbindungsvarianten erlauben sowohl die eng gekoppelte Integration direkt an den Prozessorkern als auch die flexible Anbindung von externen Hardware-Erweiterungen auf einem dedizierten rekonfigurierbaren Baustein.

Die Vorstellung der prototypischen Implementierungen sowohl als FPGA-Prototyp als auch als ASIC-Realisierung bildet den Abschluss dieser Dissertation. In einer 65 nm Low-Power-Standardzellentechnologie von STMicroelectronics belegt der vierfach parallele CoreVA-Prozessor eine Chipfläche von 2,7 mm^2. Bei einer Taktfrequenz von 400 MHz liefert die Architektur einen Durchsatz von bis zu 3,2 Milliarden Operationen pro Sekunde. Die Leistungsaufnahme liegt bei durchschnittlich 169 mW. Damit wird die Ressourceneffizienz der entwickelten skalierbaren VLIW-Architektur deutlich.

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel vermittelt einen inhaltlichen Überblick über die vorliegende Arbeit und motiviert die Notwendigkeit einer umfassenden Entwurfsraumexploration bei der Entwicklung von Prozessorarchitekturen mit dem Ziel einer hohen Ressourceneffizienz. Es zeigt die Einbettung dieser Arbeit in die Projektarbeit in der Fachgruppe Schaltungstechnik der Universität Paderborn. Des Weiteren wird der strukturelle Aufbau der Arbeit dargestellt.

1.1. Inhaltlicher Überblick

Diese Arbeit dokumentiert die Entwurfsraumexploration einer skalierbaren und ressourceneffizienten *Very-Long-Instruction-Word (VLIW)-Architektur* – dem *CoreVA*[1]*-Prozessor*. Als Grundlage der Entwicklung dient ein dualer Entwurfsablauf, der auf einer zentralen Prozessorspezifikation basiert. Der hohe Automatismus dieses Entwurfsablaufs ermöglicht kürzere Iterationszyklen während der Entwicklung und somit die Abdeckung größerer Entwurfsräume, als es bisher möglich war. Ziel der Entwicklung war die Implementierung und Realisierung einer anwendungsspezifischen Architektur, die möglichst gut an ihr spezielles Anwendungsszenario angepasst ist.

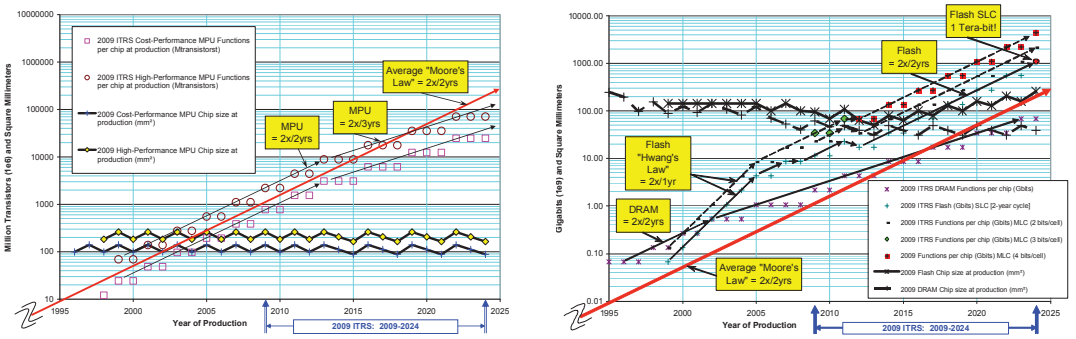
1.2. Motivation

Die zunehmende Miniaturisierung digitaler Schaltkreise durch moderne Fertigungsverfahren und die damit verbundene steigende Integrationsdichte von mikroelektronischen Schaltkreisen erlaubt die Realisierung von immer komplexeren und leistungsfähigeren Schaltungen. Bereits 1965 stellte *Gordon Moore* eine Regel (das sogenannte „Mooresche Gesetz“) auf, welches die zukünftige Entwicklung der Komplexität integrierter Schaltkreise abschätzt [147]. Die Regel basiert auf der historischen Beobachtung der Entwicklung und besagt, dass sich die Marktanforderungen an die Funktionalität pro Chip (Bits, Transistoren) alle 18 bis 24 Monate verdoppeln. Des Weiteren beobachtete er, dass sich auch die Performanz von MPUs[2] in demselben

[1] Configurable Ressource Efficient VLIW Architecture

[2] Micro Processor Unit – dt. Mikroprozessoreinheit

Zeitraum verdoppelt. Die *International Technology Roadmap for Semiconductors (ITRS)* [105, 106], die Daten von führenden Halbleiterherstellern auf der gesamten Welt auswertet, prognostiziert, dass sich die von Moore vorhergesagte Entwicklung auch in den nächsten Jahren fortsetzen wird (vgl. Abbildung 1.1).



(a) Funktionalität pro Chip (Prozessoren) **(b)** Funktionalität pro Chip (Speicher)

Abbildung 1.1.: Historie und Prognose der Entwicklung von mikroelektronischen Systemen (nach ITRS 2009 [105])

Die steigende Integrationsdichte moderner Halbleitertechnologien zur Fertigung mikroelektronischer Systeme ermöglicht sowohl die Realisierung von mehr Funktionalität, als auch den Betrieb bei höheren Taktfrequenzen. Die Steigerung der Performanz durch eine reine Erhöhung der Taktfrequenz wirkt sich jedoch nachteilig auf die Leistungsaufnahme des Systems aus. Neue Architekturen stellen die geforderte Leistungsfähigkeit durch eine höhere Parallelität zur Verfügung. Diese ermöglichen eine höhere Energieeffizienz, da die Taktfrequenz eines Parallelprozessors vergleichsweise niedrig gehalten werden kann. Es gilt einen guten Kompromiss zwischen Performanz und Bedarf an Ressourcen, wie Fläche oder Leistungsaufnahme, zu erzielen.

Insbesondere in mobilen Anwendungsszenarien ist eine hohe Ressourceneffizienz wichtig. Die höhere Leistungsfähigkeit von Mikroprozessoren ermöglicht die Realisierung von mehr Funktionalität, wie neuen Funkstandards aber auch komplexen Multimediaanwendungen. Programme dieser Anwendungsklassen lassen sich gut auf die feingranulare Parallelität universeller VLIW-Prozessoren abbilden. Aufgrund ihrer Vielseitigkeit und Skalierbarkeit sind VLIW-Prozessoren aber auch für die Ausführung anderer Programme, wie beispielsweise Betriebssystemen, geeignet.

Während der Entwicklung wird im Rahmen einer umfassenden Entwurfsraumexploration die Ressourceneffizienz einer oder mehrerer Architekturen unter Variation möglichst vieler Design-Parameter evaluiert. Mit steigender Komplexität der Prozessoren wächst auch der Entwurfsraum und damit die sogenannte *Entwurfs-*

produktivitätslücke[3]. Die Entwurfsproduktivitätslücke bezeichnet die Problematik, dass durch die steigende Komplexität mikroelektronischer Bausteine deren Entwicklung mit bisherigen Methoden nicht in der zur Verfügung stehenden Zeit durch die Entwicklerteams realisiert werden kann. Es gilt also neue Verfahren zu entwickeln, um diese Lücke zu schließen. Hierzu zählen Automatismen, die es dem Entwickler ermöglichen, möglichst *große Entwurfsräume* in möglichst *kurzer Zeit* zu analysieren.

1.3. Einbettung

Die vorliegende Arbeit ist thematisch in die BMBF[4] - Projekte „Netz der Zukunft“ – *MxMobile* – Multi-Standard Mobile Plattform [237] und „Enablers for Ambient Services and Systems“ – *Easy-C* – Wide Area Coverage [246] eingebettet. Hierdurch profitierte die vorgeschlagene Entwurfsmethodik zur Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren bereits in einem frühen Stadium von der interdisziplinären Zusammenarbeit der beteiligten Projektpartner insbesondere der Universität Paderborn und Infineon Technologies. Die Synergie der Expertise im Bereich der Konzeption und Entwicklung von Programmiersprachen und Übersetzern (Arbeitsgruppe „Programmiersprachen und Übersetzer“ von Prof. Dr. Uwe Kastens) und der Erfahrung im Bereich der Entwicklung und Realisierung hochintegrierter Schaltkreise (Fachgebiet Schaltungstechnik, Prof. Dr. Ulrich Rückert) ermöglichte eine ganzheitliche Betrachtung sowohl der Hardwarekomponenten eines Prozessorsystems, als auch der zur effizienten Programmierung essentiellen Werkzeugkette. Die enge Zusammenarbeit mit Infineon Technologies verschaffte auch Einblicke in die Abläufe großer Industrieunternehmen und deren Anforderungen an kommerzielle Produkte.

Die wesentlichen Beiträge dieser Arbeit zur Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren sind:

- Ein *Entwurfsablauf zur Entwurfsraumexploration* von Prozessorarchitekturen, insbesondere unter Betrachtung der Ressourceneffizienz,
- die *Implementierung und Optimierung* eines VLIW-Prozessorsystems (CoreVA-Architektur) für den Einsatz in drahtlosen Anwendungsszenarien,

[3] engl. *design productivity gap*

[4] Bundesministerium für Bildung und Forschung

- die *prototypische Umsetzung* des Gesamtsystems als FPGA[5] -Implementierung basierend auf dem Rapid-Prototyping-System RAPTOR und

- die *Realisierung* des CoreVA-Prozessors als *ASIC*[6] *-Prototyp* in einer 65 nm Standardzellentechnologie von STMicroelectronics (ST).

1.4. Aufbau der Arbeit

Kapitel 2 gibt einen Überblick über den aktuellen *Stand der Technik* von VLIW-Prozessoren. Neben der historischen Entwicklung werden auch die Vor- und Nachteile des VLIW-Architekturkonzeptes diskutiert. Anschließend wird ein Vergleich des Ressourcenbedarfs verschiedener kommerzieller VLIW-Prozessoren aufgeführt.

Kapitel 3 definiert Begriffe, wie *Ressource* und *Entwurfsraumexploration*, und stellt einen dualen Entwurfsablauf zur Entwicklung und Exploration von Prozessorarchitekturen vor. Zentrale Bestandteile dieses Entwurfsablaufs sind eine Prozessorspezifikation, aus der eine vollständige Compiler-Werkzeugkette generiert werden kann, sowie eine automatisierte Werkzeugkette zur Entwicklung der Hardware-Architektur. Vorteil des hohen Automatismus ist eine Verkürzung der Iterationszyklen während der Entwurfsraumexploration. Diese Verkürzung ermöglicht es dem Entwickler, größere Entwurfsräume während des Designs einer Architektur einzubeziehen und so bessere Ergebnisse zu erzielen. Zur Bewertung jedes Entwicklungsschrittes muss der Ressourcenbedarf des Systems auf ein Bewertungsmaß abgebildet werden. Hierzu werden zuerst bekannte Bewertungsmaße diskutiert und anschließend ein Bewertungsmaß für die Ressourceneffizienz eines mikroelektronischen Systems vorgestellt. Dieses Bewertungsmaß vereint zum einen die drei Ressourcen *Fläche*, *Leistungsaufnahme* und *Performanz* miteinander, zum anderen kann es aber auch durch geeignete Parameterwahl auf die bekannten Bewertungsmaße zurückgeführt werden.

In Kapitel 4 wird der vorgestellte Entwurfsablauf zur Entwicklung einer modularen und konfigurierbaren VLIW-Prozessorarchitektur – des *CoreVA-Prozessors* – genutzt. Eigenschaften der Architektur, z. B. die Anzahl der Ausführungseinheiten, können über generische Parameter umfassend variiert werden. Es werden sowohl die zugrunde liegende Computerarchitektur als auch die speziellen Komponenten des Prozessors im Detail vorgestellt. Es wird gezeigt, dass der *Pipeline-Bypass* eine für die Ressourceneffizienz entscheidene Komponente darstellt. Daher wird in diesem Kapitel im Besonderen auf die Thematik des sogenannten *Pipeline-Forwarding* eingegangen, bei der der Pipeline-Bypass die zentrale Rolle spielt.

[5] Field Programmable Gate Array

[6] Application Specific Integrated Circuit

Ausgehend von der modularen CoreVA-Architektur wird in Kapitel 5 eine *Entwurfsraumexploration* durchgeführt. Hierfür wird ein Benchmark-Szenario definiert, welches Anwendungen aus verschiedenen Klassen, wie Basisbandalgorithmen oder Multimediaverarbeitung, zusammenfasst. Durch die hohe Konfigurierbarkeit der Architektur in Verbindung mit dem Benchmark-Szenario spannt sich ein großer Entwurfsraum auf, der mit Hilfe des in Kapitel 3 vorgestellten Entwurfsablaufs evaluiert wird. Als Resultat der Entwurfsraumexploration wird eine konkrete Konfiguration der *CoreVA-Architektur als Referenzarchitektur* ausgewählt, die sich bezüglich des Anwendungsszenarios als besonders ressourceneffizient erweist.

Die in Kapitel 5 durchgeführte Entwurfsraumexploration bezieht sich auf die statische Konfiguration der Anzahl an funktionalen Einheiten der CoreVA-Architektur zur Entwurfszeit. Eine weitere Möglichkeit sowohl zur statischen als auch zur dynamischen Konfiguration bietet der Pipeline-Bypass. Durch gezieltes An- und Abschalten von Bypass-Pfaden zur Laufzeit kann die Ressourceneffizienz weiter anwendungsspezifisch gesteigert werden. Kapitel 6 gibt hierzu zuerst einen Überblick über den Stand der Technik von Forwarding- und Bypass-Systemen bei aktuellen Prozessorarchitekturen. Nach einer Analyse der Auslastung des Pipeline-Bypasses wird ein Verfahren zur Optimierung des Pipeline-Bypasses vorgestellt.

Während sich Kapitel 4 bis 6 auf den Prozessorkern konzentrieren, wird die *Entwurfsraumexploration* in Kapitel 7 auf *Systemebene* ausgedehnt. Das Speicher-Subsystem des CoreVA-Systems wird hierzu um Caches und Scratchpad-Speicher erweitert. Neben der Entwurfsraumexploration des weitreichend konfigurierbaren Speicher-Subsystems wird in Kapitel 7 eine flexible Schnittstelle zur Anbindung von dedizierten Hardware-Erweiterungen vorgestellt. Anwendungsspezifische Hardware-Erweiterungen ermöglichen eine Steigerung der Ressourceneffizienz um bis zu mehreren Größenordnungen.

In Kapitel 8 wird die Realisierbarkeit der in den Kapiteln 4 bis 7 erzielten Ergebnisse anhand einer *prototypischen Implementierung* der CoreVA-Architektur gezeigt. Der CoreVA-Prozessor wird hierzu sowohl auf eine rekonfigurierbare FPGA-Architektur als auch als ASIC-Implementierung auf eine 65 nm Standardzellentechnologie abgebildet. Eine echte Hardwarerealisierung ermöglicht wesentlich präzisere Aussagen über die physikalischen Eigenschaften eines mikroelektronischen Systems, als es mit modernen Synthesewerkzeugen möglich ist. Den Abschluss dieses Kapitels bildet ein Demonstrator, basierend auf dem im Fachgebiet Schaltungstechnik entwickelten RAPTOR-Systems, der den Einsatz der beiden Prototypen in realen Einsatzszenarien ermöglicht.

In Kapitel 9 werden die in dieser Arbeit erzielten Ergebnisse schließlich zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

2. Stand der Technik von VLIW-Architekturen

Dieses Kapitel zeigt die historische Entwicklung des VLIW-Architekturkonzepts. Um die vorliegende Arbeit in den aktuellen Stand der Technik einzuordnen, werden kommerzielle VLIW-Architekturen vorgestellt. Abschließend werden die betrachteten Architekturen bezüglich ihres Ressourcenbedarfes miteinander verglichen.

2.1. Entwicklung der VLIW-Architekturen

Der Begriff *Very-Long-Instruction-Word* wurde Anfang der 80er Jahre von John Fisher geprägt. John Fisher beschäftigte sich zu dieser Zeit mit der automatischen Konvertierung von vertikalem in horizontalen Mikrocode. Rein vertikaler Code wird sequentiell ausgeführt. Bei horizontalem Mikrocode kann die Nebenläufigkeit der Operationen ausgenutzt werden, um sie zeitgleich in unterschiedlichen Funktionseinheiten auszuführen. Entscheidend für die Möglichkeit der Parallelisierung des Codes ist die Menge an Instruktionsparallelität (Instruction Level Parallelism (ILP)), die aus dem Code extrahiert werden kann. Ein Problem bei dieser Konvertierung von vertikalen in horizontalen Mikrocode sind Basisblöcke [219]. Ein Basisblock zeichnet sich dadurch aus, dass (bedingte) Sprünge hinein nur am Anfang und Sprünge hinaus nur am Ende auftauchen. In den meisten Programmen sind Basisblöcke sehr kurz, bestehen also nur aus wenigen Instruktionen. Dadurch bedingt beinhalten diese Blöcke nur eine geringe ILP. Damalige Compiler waren in der Lage, eine ILP von 2–3 zu extrahieren. Die Handoptimierung durch Assembler-Code erforderte jedoch einen sehr hohen Aufwand. Eine weitere Beschränkung stellt Amdahl's Law [9] dar: Angenommen, ein Programm besteht aus einer Schleife mit einem Basisblock. Der Basisblock beansprucht 90 % und der Schleifenkopf 10 % der Ausführungszeit. Selbst wenn der Basisblock im Idealfall optimal parallelisiert werden könnte, so ist der Performanzgewinn auf den Faktor 10 beschränkt. Eine weitere Verbesserung durch mehr Nebenläufigkeit ist nicht möglich. (vgl. Abbildung 2.1)

Der Ansatz von Josh Fisher bestand darin, die Basisblöcke über Sprünge hinaus zu erweitern (vgl. Abbildung 2.2). Diese verfügen dann über eine weitaus größere ILP. Durch falsch vorhergesagte bedingte Sprünge entsteht jedoch ein Fehlverhalten, welches durch zusätzlichen Code rückgängig gemacht werden muss. Dieses kann durch einen höheren Anteil an Nebenläufigkeit der Operationen wieder ausgeglichen

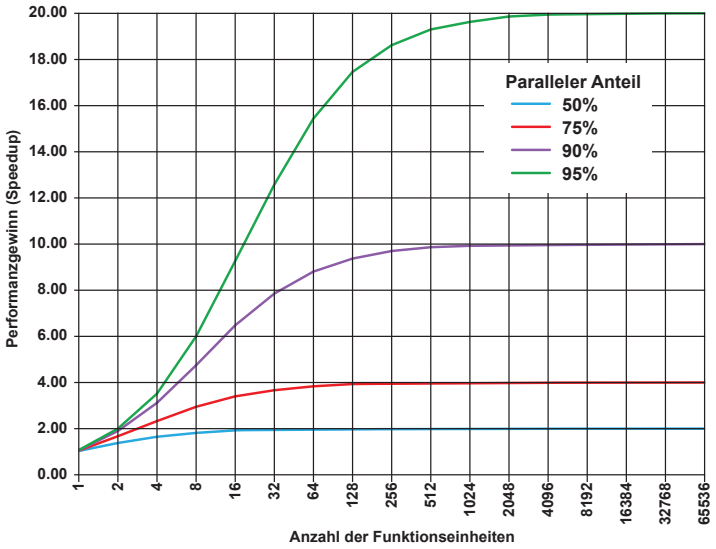


Abbildung 2.1.: Amdahl's Law [9]

werden. Das von John Fisher als *Trace Scheduling* [72, 51] bezeichnete Verfahren extrahiert aus einem Abhängigkeitsgraphen Instruktionsfolgen, welche Sprünge in diesen Block hinein und hinaus enthalten. An diesen Einstiegs- und Ausstiegspunkten wird Code eingefügt, der den Prozessor-Zustand speichert, bzw. wiederherstellt. Diese Instruktionsfolgen können bei der Ablaufplanung (engl. *Scheduling*) nun als Basisblock behandelt werden. Das von John Fisher vorgeschlagene Verfahren stellte das zur damaligen Zeit beste Verfahren dar, um automatisch horizontalen in vertikalen Mikrocode umzuwandeln. Des Weiteren stellte sich heraus, dass Trace Scheduling gut um Optimierungsverfahren, wie z. B. Software-Pipelining, Sprungvorhersage etc., erweitert werden konnte [72, 74].

2.1.1. Allgemeine Eigenschaften von VLIW-Architekturen

John Fisher erkannte schnell, dass sich dieses Verfahren nicht nur auf bestehende Architekturen anwenden ließ. In [72] stellte er die erste VLIW-Architektur (ELI-512) vor. Der zugehörige Compiler (der Bulldog-Compiler [65]) wendete, als erster seiner Art, *Trace Scheduling* zur Parallelisierung an.

Eine VLIW-Architektur zeichnet sich allgemein durch die folgenden Eigenschaften aus [72, 74]:

- Eine zentrale Kontrolleinheit setzt eine lange Instruktion pro Taktzyklus ab.

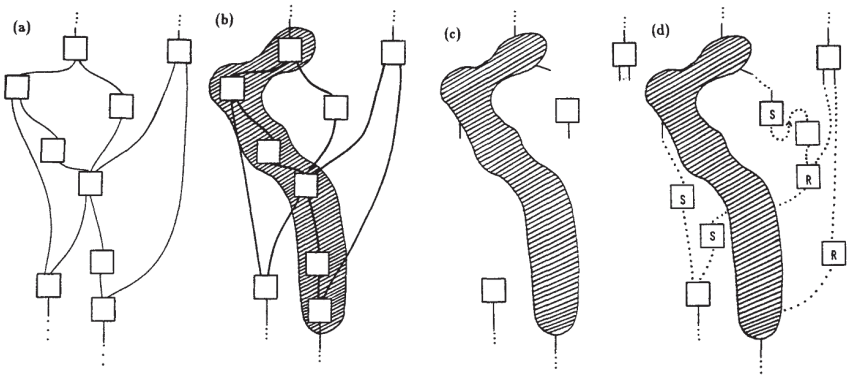


Abbildung 2.2.: Trace Scheduling nach John Fisher [72]: (a) Datenflussgraph mit Basisblöcken. (b) Ausgewählter Trace aus dem Datenflussgraphen. (c) Geplanter Trace der aber noch nicht mit dem Rest des Codes verbunden ist. (d) Ungeplanter Code der neues Verknüpfen erlaubt.

- Jede Instruktion besteht aus vielen feingranularen, eng gekoppelten aber unabhängigen Operationen.
- Jede Instruktion benötigt eine kleine, statisch vorhersagbare Anzahl an Taktzyklen zur Ausführung.
- Operationen können in einer Fließbandverarbeitung (Pipelining) ausgeführt werden.

Diese Eigenschaften stehen im Gegensatz zu asynchron und lose gekoppelten *Multiprozessoren*, wo jeder Prozessorkern über eine eigene Kontrolleinheit verfügt. Sogenannte *Vektorprozessoren* führen eine Operation auf mehrere Datenwörter aus. Im Vergleich zu VLIW-Architekturen sind die Operationen daher feingranular, eng gekoppelt aber logisch abhängig. VLIW-Architekturen sind strukturiert wie horizontaler Mikrocode, sollen es aber erlauben, auch allgemeinen, d.h. nicht spezialisierten Code, wie ein Betriebssystem, auszuführen. Auch Hochsprachen, wie C/C++, lassen sich leicht auf VLIW-Architekturen abbilden [75]. Die Parallelisierung des Programmcodes durch den Compiler erlaubt eine höhere ILP, als beispielsweise *superskalare Architekturen* bei geringeren Hardwarekosten [75, 73]. In superskalaren Prozessoren sortiert eine Hardware-Einheit die Operationen zur Laufzeit um und verteilt diese auf die Funktionseinheiten des Systems. Bei VLIW-Architekturen entfällt dieser Aufwand, da der Compiler die Instruktionen bereits im Vorfeld den sogenannten *VLIW-Slots* zuweist. VLIW-Architekturen sind daher vergleichsweise einfach und modular aufgebaut, was zu einer höheren Skalierbarkeit führt. Im Vergleich zu *digitalen Signal-*

prozessoren (DSPs) benötigt ein VLIW-Prozessor die gleiche Anzahl an Taktzyklen pro Instruktion (Cycles per Instruction (CPI)), erreicht aufgrund der einfacheren Hardware aber üblicherweise eine doppelt so hohe Taktfrequenz [75].

2.1.2. Die ELI-512-VLIW-Architektur

Die ELI-Architektur [72] verarbeitet in einem Taktzyklus bis zu 512 Bit breite Instruktionsgruppen. Die Architektur verfügt über 16 Funktionseinheiten (Arithmetic Logical Unit (ALU)) in denen arithmetisch-logische Operationen durchgeführt werden. Acht dieser Einheiten können Speicherreferenzen berechnen. Das Register-File erlaubt 32 parallele Zugriffe. Multi-Wege-Sprünge können bedingt ausgeführt werden. Die maximale Taktfrequenz beträgt 16 MHz. Der Flächenbedarf liegt bei 100000 ECL[1] -Gattern. Die ELI Architektur wurde im Rahmen der Trace-Reihe [52] von der Firma *Multiflow* kommerziell vertrieben [122]. Hierbei handelt es sich um ein modulares System, welches auf einer siebenfach parallelen VLIW-Architektur basiert. In voller Ausbaustufe werden VLIWs mit bis zu 28 Operationen und 1024 Bit Breite unterstützt. Über einen SIMD (Single Instruction Multiple Data)-Modus konnte, ähnlich wie bei Vektor-Prozessoren, eine Operation auf mehrere Datenwörter angewandt werden kann.

2.1.3. Herausforderungen der VLIW-Architekturen

Als Nachteil der engen Kopplung der Operationen erweist sich das Register-File einer VLIW-Architektur. Um die Operanden aus dem Register-File den Funktionseinheiten der Architektur zur Verfügung zu stellen, muss das Register-File über viele Leseports verfügen. Analog werden viele Schreibports benötigt, um die Ergebnisse der Operationen zurückzuschreiben. Laut [181] ist die Größe eines Register-Files quadratisch proportional zu der Anzahl der Ports. Aus diesem Grund wird in dieser Arbeit dem Ressourcenbedarf des Register-Files eine besondere Bedeutung zugemessen (vgl. Abschnitt 4.3.2). Eine weitere Herausforderung stellen die Ports zum Datenspeicher dar. Zueinander disjunkte Speicheroperationen müssen z. B. auf verschiedene physikalische Bänke des Speichers verteilt werden. Zwei- oder Multi-Port-Speicher benötigen einen um ein Vielfaches höheren Ressourcenbedarf als Ein-Port-Speicher. Kommerzielle Systeme sowie die in dieser Arbeit vorgestellte VLIW-Architektur beschränken sich daher ausschließlich auf Ein- oder Zwei-Port-Speicher. Zur Erhöhung der Speicherbandbreite werden alternative Konzepte, wie beispielsweise sogenannte *Scratchpad-Speicher*, verwendet, die gesondert in Abschnitt 7.2 diskutiert werden. Superskalare Architekturen erleichtern die Binärkompatibilität zwischen

[1] Emitter coupled logic – Emittergekoppelte Logik: Meist auf Bipolartransistoren basierende Schaltungstechnik

unterschiedlichen Architekturen. Änderungen an einer Architektur, insbesondere am Instruktionssatz, erfordern ein Neukompilieren aller Applikationen. Spezielle Hardware-Einheiten, sogenannte Binär-Compiler, sind in der Lage, diese Anpassungen zur Laufzeit durchzuführen. Da bei VLIW-Architekturen die Zuordnung der Operationen zu den Funktionseinheiten bereits durch den Compiler im Vorfeld festgelegt wird und Operationen nicht zur Laufzeit umsortiert werden, wird eine solche Konvertierung von Binärcode erschwert. Die in dieser Arbeit vorgestellte VLIW-Architektur ermöglicht eine Binärkompatibilität durch eine Anlehnung des Instruktionssatzes an den vielfach in eingebetteten Systemen eingesetzten ARM-Prozessor.

2.2. Kommerzielle VLIW-Architekturen

Während der 80er und 90er Jahre waren VLIW-Architekturen nur begrenzt kommerziell erfolgreich. Mit dem Aufkommen der voll integrierten Mikroprozessoren schnellten die Taktfrequenzen von Konkurrenzarchitekturen in die Höhe. Der Anteil des Prozessorkerns am Gesamtsystem wurde von weitaus größeren Caches dominiert. Der zusätzliche Hardware-Aufwand superskalarer Architekturen fiel immer weniger ins Gewicht. Die zunehmende Integrationsdichte durch Verkleinerung der Strukturgrößen aktueller Technologien auf unter 65 nm schafft allerdings neue Voraussetzungen: Die Leitungsverzögerungen dominieren die Verzögerungszeiten der Gatter und wirken einer weiteren Skalierung entgegen [122]. Heutzutage geht der Trend insbesondere bei eingebetteten Systemen wieder in Richtung einfacherer Architekturen. VLIW-Architekturen werden dabei zunehmend auch in kommerziellen Systemen eingesetzt.

2.2.1. NXP TriMedia

Bereits 1987 entwickelte Philips mit dem LIFE-1 seinen ersten Mikroprozessor, der auf VLIW-Technik basierte [122, 75]. 1996 startete die auch heute noch sehr erfolgreiche TriMedia-Reihe. 2006 gliederte Philips seine Halbleitersparte in die rechtlich Eigenständige Gesellschaft NXP aus. Das aktuelle Modell TM3270 [213, 214] beinhaltet eine siebenstufige Pipelinestruktur und fünf Funktionseinheiten.

Mehrzyklenoperationen können zusätzlich bis zu fünf Takte beanspruchen. Auf das 128 Einträge große 32-Bit-Register-File kann über zehn Lese- und fünf Schreibports zugegriffen werden. Zwei VLIW-Slots sind für Speicherzugriffe auf den Datenspeicher vorgesehen. In einer 90 nm Low-Power-Technologie von Philips[2] benötigt der TM3270 inklusive 192 kB statischem Speicher (Static Random Access Memory (SRAM)) eine Fläche 5,61 mm^2 [12]. Bei einer Taktfrequenz von 400 MHz liegt die

[2] Maskenkompatibel zu TSMC (Taiwan Semiconductor Manufacturing Company) 90 nm CLN90LP

Leistungsaufnahme bei 1,71 Watt[3]. Der TM3282 erweitert das Prinzip des TM3270 auf acht Funktionseinheiten und erreicht ebenfalls eine Taktfrequenz von 400 MHz. Einsatzgebiete der TriMedia-Prozessoren sind insbesondere digitale Videoprodukte sowie Mobilfunkgeräte.

2.2.2. STMicroelectronics ST200

Die ST200-Reihe von STMicroelectronics basiert auf der Lx Architektur [69] und wurde im Januar 2001 unter anderem von John Fisher in Kooperation mit Hewlett-Packard entwickelt. Die Lx-Architektur basiert auf sechs Pipelinestufen und vier VLIW-Slots. In [70] wird für einen ST200 in einer 250 nm Technologie ein Flächenbedarf von 32 mm^2 inklusive 64kB Speicher angegeben. Bei einer Versorgungsspannung von 2,5 Volt und 300 MHz liegt die Verlustleistung bei bis zu 1,7 Watt. In [20] ist die Leistungsaufnahme bei 400 MHz mit bis zu 2,2 Watt angegeben. Der ST231 [61] stellt eine Erweiterung der Lx-Architektur um eine weitere Pipelinestufe, 32x32-Bit-Multiplizierer und eine Memory Management Unit (MMU) dar. Die ST200 Architektur zeichnet sich dadurch aus, dass sie aufgrund ihrer einfachen Architektur voll synthetisierbar ist und ein automatisches Layout durchgeführt werden kann [75]. 2009 berichtete STMicroelectronics, über 40 Millionen System-on-Chips (SoCs) verkauft zu haben, was insgesamt über 70 Millionen ST200-VLIW-Kernen entspricht [75]. Einsatzgebiete des ST200 sind Mobiltelefone, digitale Fernseher, Receiver, Rekorder, Digitalkameras und Multifunktionsgeräte.

2.2.3. Fujitsu FR-V

Der Fujitsu FR-V[4] wurde 1999 eingeführt [122]. Die ersten Prozessoren dieser Reihe, der FR-300 und der FR-500 [187, 189], verarbeiten 64 bzw. 128 Bit breite Instruktionsgruppen. Während der FR-300 hauptsächlich für DSP-Anwendungen entwickelt wurde, ist der FR-500 durch seine höhere Performanz für die Ausführung von Multimediaanwendungen konzipiert. Die aktuelle Architektur dieser Prozessorreihe, der FR-550 [155] stellt eine Erweiterung auf 256 Bit breite VLIWs dar. Vier Integer-Operationen können parallel zu vier Fließkommaoperationen oder vier vierfach parallelen SIMD-Operationen ausgeführt werden. Bei einer Taktfrequenz von 533 MHz ergibt sich eine Performanz von 2,1 GFLOPS[5] bzw. 12,8 GOPS[6] . In einer 110 nm-

[3] 1,2 V, MP3 Anwendung

[4] Fujitsu RISC-VLIW

[5] Milliarden Fließkommaoperationen pro Sekunde

[6] Milliarden Operationen pro Sekunde

Technologie[7] beträgt die Leistungsaufnahme 2,5 Watt. Der Flächenbedarf inklusive 64 kByte Speicher beträgt 61 mm^2. Der FR-550 wird auch in dem FR-1000 Multiprozessor [188, 78] von Fujitsu eingesetzt. Der FR-1000 benötigt in einer 90 nm Technologie[8] eine Fläche von 123 mm^2 inklusive 768 kByte SRAM. 19 mm^2 entfallen hierbei auf einen VLIW-Kern inklusive 192 kByte lokalem Speicher. Die Leistungsaufnahme liegt bei 3 Watt[9]. Prozessoren der Fujitsu FR-V-Reihe werden auch aktuell noch in digitalen Spiegelreflexkameras (DSLRs) eingesetzt.

2.2.4. Texas Instruments TMS320C6X

Texas Instruments (TI) erster DSP mit VLIW-Technik, der TMS320C67xx, kam im Jahre 1997 auf den Markt [122, 181]. Im Jahre 2004 startete die TMS320C64xx-Reihe, die einige Verbesserungen und Erweiterungen beinhaltete. Der TMS320C64x+ [207] beinhaltet acht Ausführungseinheiten (6 Arithmetikeinheiten, 2 Multiplizierer) und elf Pipelinestufen.Die Verlustleistung eines TI TMS320DM642 [203] bei 600 MHz[10] inklusive 288 kByte SRAM liegt bei 1,9 Watt [202], die eines TI TMS320C6418 bei 600 MHz bei 1,7 Watt [134] und die eines TI TMS320C6416T bei 1000 MHz[11] bei bis zu 1,6 Watt [100]. Der TMS320C6474 erreicht Taktfrequenzen von bis zu 1,1 GHz und einen Durchsatz von bis zu 8,8 GOPS. Aktuelle Multiprozessoren, wie z. B. die OMAP Plattform [44] oder der TMS3290C6474 [206] von Texas Instruments, vereinen mehrere VLIW-Kerne in einem Multiprozessor SoC. Anwendungsgebiete der Prozessoren der TMS320-Reihe sind u. a. Bildverarbeitung oder Basisbandverarbeitung. Beispielsweise wird der OMAP 3 im aktuellen Palm Pre Phone eingesetzt [122].

2.2.5. Tilera Tile64

Bei dem Tile64 [5, 19, 18, 122] handelt es sich um einen Multiprozessor SoC, der einen Verbund von 8x8 identischen Prozessorkernen auf einem DIE[12] integriert. Der Tile64 wurde von 2005 bis 2007 von der Firma Tilera entwickelt und basiert u. a. auf der 1997 am Massachusetts Institute of Technology (MIT) entwickelten RAW-Architektur [218]. Der Tile64 basiert auf VLIW-Kernen mit einer fünfstufigen Pipelinestruktur und führt 64 Bit breite Instruktionsgruppen aus. Diese Instruktionsgruppen können zwei

[7] 7 Metalllagen, 1,3 V

[8] 9 Metalllagen, 1,2 V

[9] Bei MPEG-2 Dekodierung

[10] 1,4 V, 60 % Auslastung

[11] jeweils 1,4 V

[12] dt. *Würfel*, *Plättchen*, auch „Nacktchip“ genannt

oder drei Instruktionen beinhalten. Des Weiteren ist ein 8 Bit-SIMD-Modus möglich. Jeder VLIW-Kern ist über einen Switch an ein Netzwerk gekoppelt (Tile genannt) das Interaktionen mit anderen Kernen zulässt und eine theoretische Gesamtperformanz des Tile64 von bis zu 433 GOPS[13] ermöglicht. Jeder Tile verfügt über 8 kB Instruktions- bzw. 8 kB Daten-Level-1-Cache und 64 kB Level-2-Cache. Das 32 Bit breite Register-File beinhaltet 64 Einträge von denen 53 Einträge für Instruktionen zur Verfügung stehen. In einer 90 nm Technologie benötigt der Tile64 eine Fläche von 433 mm^2. Die Leistungsaufnahme pro Tile[14] liegt bei 1000 MHz bei 300 mW.

2.2.6. Sonstige Anwendungsgebiete

VLIW-Technologie wird nicht ausschließlich in eingebetteten Systemen eingesetzt, sondern ist auch Bestandteil moderner Prozessorarchitekturen für Hochleistungsserver. Die im Jahre 1995 gegründete Firma Transmeta entwickelte von 2000 bis 2009 die *Crusoe-Prozessorreihe*. Die Prozessoren basieren auf einer vierfach parallelen VLIW-Architektur mit zwei Integer- , einer Sprung-, einer Fließkomma- und einer Load-/Store-Einheit. Eine Besonderheit ist, dass Crusoe-Prozessoren Intel x86-Code ausführen können, der zur Laufzeit in Crusoe-spezifische VLIWs übersetzt wird. Die Pipeline-Architektur besteht aus sieben Stufen. Zwei zusätzliche Ausführungszyklen kommen für Fließkommaoperationen hinzu. Das Register-File umfasst 64 Einträge. Die Leistungsaufnahme eines Crusoe TM5400 [113] bei 600 MHz in einer 180 nm Low-Power-Technologie beträgt 0,326 Watt. Der Flächenbedarf einschließlich 384 kByte SRAM beträgt 73 mm^2. Transmeta konnte sich mit der Crusoe Architektur jedoch nicht gegen Branchenriesen, wie Intel oder AMD, durchsetzen und wurde Anfang 2009 von der Firma Novafora aufgekauft.

Der ursprünglich von Hewlett-Packard (HP) entwickelte *Intel Itanium* basiert ebenfalls auf einem VLIW-ähnlichen Prozessorkern (genannt Explicitly Parallel Instruction Computing (EPIC)). Der Intel Itanium 2 (Madison) [185, 150, 150, 183] ist in einer 130 nm Technologie gefertigt und beinhaltet acht Pipelinestufen sowie bis zu 6 MB Level-3-Cache. Ein 128-Bit-VLIW kann aus bis zu drei Instruktionen bestehen, die auf zwei Fließkomma-, sechs Multimedia-, sechs Integer-, drei Sprung-, und je zwei Load-/Store-Einheiten verteilt werden können. Die Chipfläche beträgt 437 mm^2. Bei 1,5 GHz Taktfrequenz und einer Versorgungsspannung von 1,3 Volt beträgt die Leistungsaufnahme 130 Watt.

Weitere Anwendungen von VLIW-basierten Architekturen finden sich in *Streaming-Prozessoren*, z. B. im Storm-DSP-1 [75] mit zwölffach parallelen VLIW-Kernen, oder in *Grafikprozessoren*, wie z. B. in dem AMD/ATI RV670 [81, 107, 117] (fünffach parallel).

[13] 866 MHz, 8 Bit SIMD Modus

[14] 9 Metalllagen, 1 Volt

2.2.7. Ressourcenvergleich der Architekturen

In den Tabellen 2.1 und 2.2 sind Architekturparameter und der Ressourcenbedarf für die wichtigsten der vorgestellten eingebetteten VLIW-Prozessoren gegenübergestellt. Diese sollen nur eine qualitative Übersicht der Komplexität der Architekturen und deren Ressourcenbedarf darstellen. Für einen präziseren Vergleich müssten genauere Informationen zu verwendeten Standardzellentechnologien (Strukturgröße, Multi-Voltage, Schwellspannungen etc.), Implementierungsdetails der Caches (Größe, Assoziativität etc.) oder Informationen zum Aufbau der Speicher (Ein-Port, Zwei-Port etc.) vorliegen. Hier halten sich die Hersteller allerdings bedeckt. Informationen über die Verteilung der Fläche auf Logik und Speicher liegen meist nicht vor. Die Angaben zum Speicher beinhalten sowohl Level-1 als auch Level-2 Cache für Instruktionen und Daten. Für den Tilera Tile64 wurden die Angaben zur Speichergröße und zum Ressourcenbedarf auf einen Tile skaliert. Die Performanz ist in Gigaoperationen (GOPS)[15] pro Sekunden angegeben und stellt den Maximalwert unter Auslastung aller Funktionseinheiten und Erweiterungen, wie z. B. SIMD, dar. Hier handelt es sich um einen theoretischen Wert, auf den der Compiler und die Anwendung einen sehr großen Einfluss haben. Auf die Compilerwerkzeuge der Hersteller bestand kein Zugriff.

Tabelle 2.1.: *Architekturvergleich ausgewählter VLIW Prozessoren*

Architektur	VLIW-Slots	Pipeline-stufen	Technologie	Speicher [kByte]
NXP TM3270	5	7	90 nm	192
ST Lx	4	6	250 nm	64
Fujitsu FR-550	8	n/a	110 nm	64
TI TMS320DM642	8	11	n/a	288
Tilera Tile64	3	5	90 nm	80

Tabelle 2.2.: *Ressourcenvergleich ausgewählter VLIW Prozessoren*

Architektur	Taktfrequenz [MHz]	Fläche [mm^2]	Leistungs-aufnahme [W]	Performanz [GOPS]
NXP TM3270	400	5,61	1,71	2
ST Lx	400	32	2,2	1,6
Fujitsu FR-550	533	61	2,5	12,8
TI TMS320DM642	600	25	1,9	4,8
Tilera Tile64	1000	6,7	0,3	6,7

[15] Milliarden Operationen

3. Ein dualer Entwurfsablauf zur Entwurfsraumexploration von ressourceneffizienten Prozessorarchitekturen

Wie bereits in Abschnitt 1.2 motiviert, erfordert die Entwicklung ressourceneffizienter Prozessorarchitekturen eine umfassende Entwurfsraumexploration einer oder mehrerer Architekturen unter Variation möglichst vieler Designparameter. Um die mit der wachsenden Komplexität mikroelektronischer Bausteine verbundene Entwurfsproduktivitätslücke zu schließen, sind neue automatisierte Verfahren notwendig, um möglichst große Entwurfsräume in möglichst kurzer Zeit zu analysieren.

In diesem Kapitel wird daher ein dualer Entwurfsablauf zur Entwurfsraumexploration von Prozessorarchitekturen vorgestellt. Das Adjektiv „dual[1]“ bezieht sich auf die zwei enthaltenen Entwurfsabläufe, die die Hardware-Entwicklung von Prozessorarchitekturen und die Implementierung zugehöriger Werkzeugketten zur Entwicklung und Evaluation von Anwendungssoftware in einem übergeordneten Prozess vereinen.

Die Abschnitte 3.1 und 3.2 definieren zunächst die Begriffe *Ressource* und *Entwurfsraumexploration*. In Abschnitt 3.3 werden *bekannte Bewertungsmaße* zur Bewertung der Ressourceneffizienz eingeführt. Abschnitt 3.4 definiert ein *Bewertungsmaß*, welches alle in Abschnitt 3.1 definierten Ressourcen verknüpft, sich aber durch geeignete Parameterwahl auch auf bekannte Bewertungsmaße zurückführen lässt. Abschnitt 3.5 stellt eine *Referenz-Spezifikation* vor, aus der sich die gesamte C-Compiler-basierte Werkzeugkette automatisch generieren lässt. Parallel dazu ermöglicht der in Abschnitt 3.6 beschriebene *automatisierte Hardware-Entwurfsablauf* die frühe Entwicklung und Analyse der Hardware-Implementierung. Um die Konsistenz zwischen Spezifikation und Hardware-Entwicklung auf der einen Seite und der Software-Werkzeugkette auf der anderen Seite sicherzustellen, werden in Abschnitt 3.6.5 Verfahren zur *Verifikation und Validierung* des Systementwurfs untersucht. In Abschnitt 3.7 werden die Ergebnisse dieses Kapitels schließlich zusammengefasst.

[1] von lat. *dualis* = „von zweien“; „zwei enthaltend“

3.1. Der Begriff Ressource

Als *Ressource* definiert man Mittel, die zur Produktion und zum Betrieb des Systems notwendig sind. Die Menge der benötigten Ressourcen ergibt den Ressourcenbedarf. Für die in dieser Arbeit durchgeführte Bewertung der Ressourceneffizienz werden die folgenden Ressourcen betrachtet:

- Fläche (A),
- Leistungsaufnahme (P) und
- Zeit (T).

Fläche. Die *Fläche* einer integrierten Schaltung bestimmt in hohem Maße deren Kosten. Je geringer die Fläche, desto mehr Instanzen des Designs können auf einem DIE untergebracht werden.

Leistungsaufnahme. Die *Leistungsaufnahme*

$$P = P_{stat} + P_{dyn} \tag{3.1}$$

von CMOS-Schaltungen setzt sich aus der *statischen Verlustleistung* P_{stat} und der *dynamischen Verlustleistung* P_{dyn} zusammen. Die statische Verlustleistung ($P_{stat} = P_{quer} + P_{leck}$) entsteht aufgrund von *Querströmen* P_{quer} im Ruhezustand durch den P- und N-Teil einer CMOS-Schaltung (im Wesentlichen durch Subschwellströme der sperrenden Transistoren) und aufgrund von *Leckströmen* P_{leck} beispielsweise an den P/N-Übergängen von Diffusionsgebiet zu Bulk [153]. Die dynamische Verlustleistung $P_{dyn} = P_{last} + P_{schalt}$ setzt sich aus der Lastumladeverlustleistung P_{last} und aus dem *Kurzschlussstrom* P_{schalt} durch den P- und N-Teil einer CMOS-Schaltung während eines Umschaltvorgangs zusammen. Die *Lastumladeverlustleistung* (vgl. Gleichung 3.2) macht derzeit einen Großteil der dynamischen Verlustleistung (80–90 % [114, 153]) aus und ist proportional zum Quadrat der Versorgungsspannung U_{dd}, zur Taktfrequenz f und der Schalthäufigkeit α der Lastkapazitäten C_{last} der CMOS-Gatter des betrachteten Schaltungsteils.

$$P_{last} = \frac{1}{2}\alpha \cdot C_{last} \cdot f \cdot U_{dd}^2 \tag{3.2}$$

Laut [106] gewinnt in modernen CMOS-Schaltungen die statische Verlustleistung mehr und mehr an Einfluss und betrug 2010 bereits ca. 25 % (vgl. Abbildung 3.1). Hersteller moderner Standardzellentechnologien versuchen dem Problem des steigenden Anteils der statischen Verlustleistung an der Gesamtverlustleistung auf technologischer Ebene zu begegnen. So verfügt die in dieser Arbeit verwendete *Low-Power-*

Standardzellentechnologie von STMicroelectronics im Vergleich zur sogenannten *General-Purpose*-Standardzellentechnologie über eine dickere Gate-Oxidschicht um Leckströme zu minimieren.

Trotz des steigenden Einflusses der statischen Verlustleistung gilt [114]:

$$P_{last} \gg (P_{quer} + P_{leck} + P_{schalt}) \Rightarrow P \approx P_{last} \tag{3.3}$$

Für die gesamte Leistungsaufnahme ist somit hauptsächlich die Lastumladeverlustleistung verantwortlich.

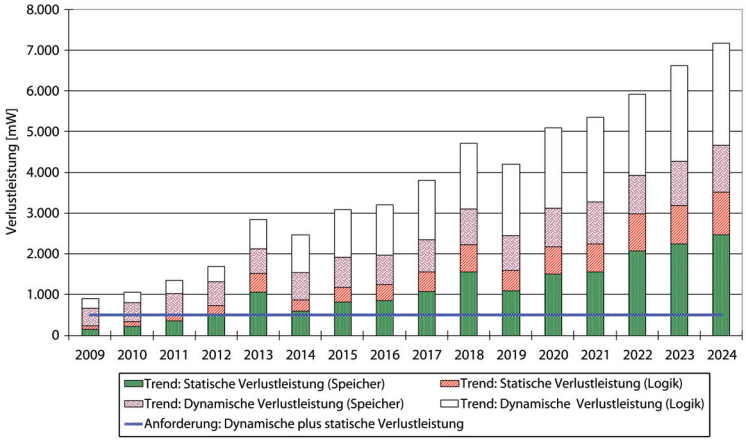


Abbildung 3.1.: Zukünftige Entwicklung der Verlustleistung (nach ITRS 2010 [106])

Durch die Abhängigkeit der dynamischen Leistungsaufnahme von der Schalthäufigkeit der CMOS-Gatter erfordert eine genaue Analyse der Leistungsaufnahme auch die Berücksichtigung der Ausführung der Anwendungen. Im Folgenden wird bei der Bestimmung der dynamischen Verlustleistung zwischen drei Art und Weisen unterschieden:

- Moderne Synthesewerkzeuge bestimmen die Schalthäufigkeiten jedes Gatters über das Propagieren *statistischer Schaltwahrscheinlichkeiten* der primären Eingänge in das Innere der Schaltung (beim Synopsys Design Compiler beispielsweise 10 %). Diese Abschätzung wird im Folgenden als durchschnittliche Leistungsaufnahme bei Annahme statistischer Schaltaktivitäten bezeichnet.

- Wesentlich präziser ist die Bestimmung und Annotation *realer Schaltaktivitäten*. Dieses erfordert bei Mikroprozessoren eine vergleichsweise zeitaufwändige Simulation der betrachteten Anwendungen. Die Simulation und Bestimmung der Schaltaktivitäten kann auf verschiedenen Abstraktionsebenen erfolgen: Bei

einer Simulation auf Register Transfer (RT)-Ebene (engl. Register Transfer Level (RTL)) werden nur die Schaltaktivitäten an primären Eingängen der Teilmodule bestimmt. Die Schaltaktivitäten der primären Eingänge werden dann zu den Gattern der Module propagiert. Bei einer Simulation auf Gatterebene werden die Schaltaktivitäten jeder Standardzelle bestimmt und bei der Bestimmung der Leistungsaufnahme annotiert. Dieses Verfahren stellt die genaueste Vorgehensweise dar. Falls nicht anders angegeben, ist im Folgenden mit „Leistungsaufnahme“ die Leistungsaufnahme bei Annotierung von Schaltaktivitäten nach Simulation auf Gatterebene gemeint.

- Ist von der durchschnittlichen Leistungsaufnahme die Rede, bezieht sich diese auf den *Durchschnitt* der Leistungsaufnahme für die betrachte Menge der Anwendungen.

Zeit. Die *Zeit* ist die Zeit, die benötigt wird, um einen bestimmten Algorithmus zu bearbeiten. Bei einem Prozessor ist diese abhängig von der Anzahl der Taktzyklen N_{cyc}, die der Algorithmus zur Ausführung benötigt, und der Taktfrequenz f, mit der der Prozessor betrieben wird. Somit ergibt sich $T = f/N_{cyc}$.

3.2. Der Begriff Entwurfsraumexploration

Zur Definition der Komposition „Entwurfsraumexploration“ ist zunächst die Definition des Begriffs „Entwurfsraum“ erforderlich. Des Weiteren wird der Begriff der Pareto-Optimalität verwendet, der ebenfalls einer Definition bedarf.

3.2.1. Der Entwurfsraum

Allgemein ausgedrückt ergibt sich der Entwurfsraum ($\mathbb{E}$) als Tripel:

$$\mathbb{E}(\mathbb{S}, \mathbb{Z}, \mathbb{A}) \tag{3.4}$$

$\mathbb{S}$ bezeichnet ein System bzw. eine Architektur. Die Elemente können gänzlich verschieden sein, oder sich nur durch Variation von Konfigurationsparametern unterscheiden. Beispiele sind VLIW- und superskalare Prozessorarchitekturen oder eine Prozessorarchitektur mit konfigurierbarer Anzahl an Pipelinestufen und Funktionseinheiten (vgl. Abbildung 3.2).

Die Zieltechnologien $\mathbb{Z}$ bestimmen den Ressourcenbedarf eines Systems. Die Menge der Anwendungen $\mathbb{A}$ umfasst zum einen die Algorithmen, zum anderen auch die verwendeten Compiler (-optionen). Die Mengen $\mathbb{S}$ und $\mathbb{A}$ spannen somit ihrerseits mehrdimensionale Unterräume innerhalb von $\mathbb{E}$ auf.

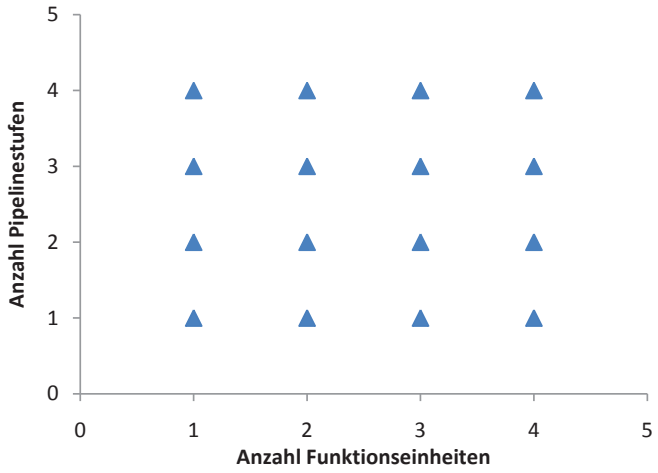


Abbildung 3.2.: Beispiel eines Entwurfsraums einer Prozessorarchitektur mit konfigurierbarer Anzahl an Pipelinestufen und Funktionseinheiten

3.2.2. Das Pareto-Optimum

Der Begriff des *Pareto-Optimums* stammt ursprünglich aus der Volkswirtschaftslehre und wurde erstmals im 19. Jahrhundert von Vilfredo Pareto formuliert. Das Konzept der Pareto-Optimalität bezeichnet einen Zustand, „in dem durch eine Maßnahme kein Wirtschaftssubjekt mehr besser gestellt werden kann, ohne dass ein anderes Wirtschaftssubjekt schlechter gestellt wird“ [41].

Übertragen auf eine Mehrzieloptimierung [98][242], wie sie bei der Entwicklung mikroelektronischer Bauteile Anwendung findet, konkurrieren die Zielgrößen (hier die Ressourcen) miteinander. Beispielsweise resultiert die Erhöhung der Taktfrequenz in der Regel in einer Erhöhung des Flächenbedarfs und der Leistungsaufnahme. Ein Entwurfspunkt (d.h. ein System $\mathbb{S}$) des Entwurfsraums $\mathbb{E}$ ist genau dann Pareto-optimal, wenn die Verbesserung einer beliebigen Zielgröße stets mit der Verschlechterung einer anderen Zielgröße einhergeht [153, 82]. Ein nicht Pareto-optimales System wird von einem Pareto-optimalen System *dominiert*. Die Menge der Pareto-optimalen Punkte im Entwurfsraum wird als *Pareto-Front* bezeichnet [98, 152][242].

3.2.3. Entwurfsraumexploration

Der Begriff *Exploration* kommt aus dem Lateinischen (*exploratio* = „Untersuchung“; „Forschung“). Die Komposition „Entwurfsraumexploration“ bezeichnet also die *Erforschung des Entwurfsraumes*.

Die Exploration des Entwurfsraumes basiert auf der *Abbildung des Entwurfsraums*

($\mathbb{E}$) auf den mehrdimensionalen sogenannten Bildraum ($\mathbb{B}$), der die physikalischen und nicht-physikalischen Eigenschaften der Elemente des Entwurfsraums charakterisiert. Ziel der Entwurfsraumexploration ist die Bestimmung der Menge der (auf den Bildraum bezogenen) Pareto-optimalen Systeme (vgl. Abschnitt 3.2.2) [242].

Die Eigenschaften sind Ziel eines Optimierungsprozesses während der Abbildung des Systems auf den Bildraum. Es können eine oder mehrere Eigenschaften optimiert werden (die sogenannte Mehrzieloptimierung [98][242]).

Ein Beispiel für einen solchen Optimierungsprozess ist beispielsweise die Standardzellensynthese. Ausgangspunkt ist hier üblicherweise die Hardware-Beschreibung (Hardware Description Language (HDL)) einer Architektur. Verschiedene Konfigurationsmöglichkeiten (z. B. generische Parameter) der Architektur bilden den Entwurfsraum. Über die Abbildung der Architektur auf die Zieltechnologie ergeben sich physikalische Eigenschaften, wie die Fläche und die Leistungsaufnahme. In Zusammenhang mit der Ausführung eines Algorithmus ergibt sich aus der Taktfrequenz oder der Latenz als weitere Ressource die Zeit. Die in dieser Arbeit betrachteten (physikalischen) Eigenschaften beschränken sich auf die in Abschnitt 3.1 definierten Ressourcen. Sie können aber auch anderer (nicht-physikalischer) Natur sein, wie beispielsweise Kostenmaße, wie Flexibilität oder Entwicklungszeit.

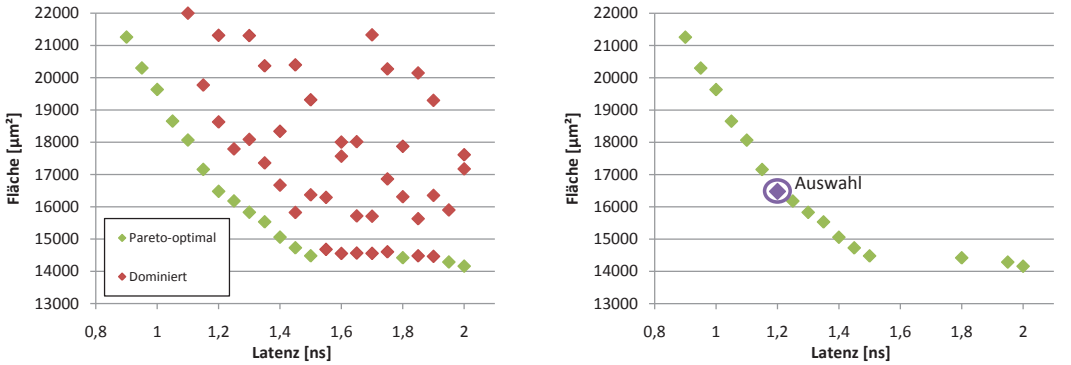
Abbildung 3.3a zeigt das Beispiel eines Bildraumes für eine ALU. Auf der X-Achse ist die Latenz und auf der Y-Achse der Flächenbedarf der Einheit aufgetragen. Mit steigenden Anforderungen an die Latenz steigt auch der Flächenbedarf der kombinatorischen Logik, da Gatter mit einer höheren Treiberstärke verwendet werden müssen. Die grünen Punkte stellen die Pareto-Front dar. Die roten Punkte stellen dominierte Punkte dar. Hier zeigt sich, dass es sich bei der Pareto-Front nur um eine diskrete Näherung handelt. Die Pareto-Optimalität bezieht sich nämlich nur auf die endliche Menge der *untersuchten* Systeme.

Ausgehend von der diskreten Näherung der Menge (vgl. Abschnitt 3.2.2) der Pareto-optimalen Systeme kann vorerst keine Aussage über die Güte der einzelnen Elemente getroffen werden. Erst durch Abbildung der untersuchten Systeme auf ein Bewertungsmaß können die Eigenschaften zueinander in Bezug gesetzt werden. Die Wahl des Bewertungsmaßes ist abhängig von dem jeweiligen Anwendungsszenario. Über das Bewertungsmaß kann nun ein Element aus der Menge der Pareto-optimalen Elemente des Entwurfsraumes ausgewählt werden (vgl. Abbildung 3.3b).

Abschnitt 3.3 stellt bekannte Bewertungsmaße vor und führt ein Bewertungsmaß ein, das die drei in Abschnitt 3.1 definierten Ressourcen miteinander vereint.

3.2.4. Stand der Technik zur Entwurfsraumexploration

Viele Ansätze in der Literatur beschäftigen sich mit der Entwurfsraumexploration mikroelektronischer Systeme und der geeigneten Selektion eines oder mehrerer Systeme aus der Menge der Pareto-optimalen Elemente des Bildraumes. Sie basieren



(a) Abbildung des Entwurfsraumes auf den Bildraum und Bestimmung der Pareto-Front

(b) Anwendung eines Bewertungsmaßes und Selektion eines Elements

Abbildung 3.3.: Beispiel einer Entwurfsraumexploration

auf dem *analytischen Hierarchieprozess* [172], der eine Methode aus der präskriptiven Entscheidungstheorie darstellt und dazu dient bei der Entwurfsraumexploration komplexe Entscheidungen zu vereinfachen.

Sehr große Entwurfsräume und komplexe Abbildungsverfahren führen zu langen Iterationszyklen der Entwurfsraumexploration. Ansätze wie evolutionäre Algorithmen [115] können die Suche von Pareto-optimalen Punkten innerhalb des Entwurfsraum geeignet eingrenzen, konvergieren aber nicht immer in Richtung des Optimums. Satz-orientierte Methoden, wie [28] und [59], stellen deterministische Alternativen dar und konvergieren in Richtung des Optimums.

In dieser Arbeit wird ein hochautomatisierter Entwurfsablauf vorgestellt, der die Iterationszyklen der Entwurfsraumexploration stark verkürzt (vgl. Abschnitt 3.6). In Kapitel 6 wird ein systematisches Verfahren vorgestellt, welches den Entwurfsraum des Pipeline-Bypasses eines Prozessors durch eine selektive Aktivierung/Deaktivierung von einzelnen Komponenten geeignet einschränkt und so den Aufwand einer vollständigen Analyse stark reduziert. Die Selektion eines oder mehrerer Systeme aus der Menge der Pareto-optimalen Elemente des Bildraumes erfolgt durch Anwendung eines Maßes zur Bewertung der Ressourceneffizienz.

3.3. Bekannte Bewertungsmaße

In der Literatur existieren verschiedene Maße, die zur Bewertung und zum Vergleich von Hardware-Architekturen eingesetzt werden. Im Folgenden werden die drei wichtigsten Bewertungsmaße vorgestellt: Das *Power-Delay-Produkt (PDP)*, das *Energy-Delay-Produkt (EDP)* und das *Power-Energy-Produkt (PEP)*.

3.3.1. Power-Delay-Produkt

Das *Power-Delay-Produkt (PDP)* [47, 30, 42, 124, 88, 17, 119] ist ein Gütekriterium, welches besagt, wie effizient elektrische Leistung in Operationsgeschwindigkeit umgesetzt werden kann. Da das PDP das Produkt aus Leistungsaufnahme und Ausführungszeit (d.h. die Energie) darstellt ($PDP = P \cdot T \Rightarrow PDP = E$), kann auch von Energieeffizienz gesprochen werden. Je geringer die Leistungsaufnahme oder je kürzer die Verarbeitungsdauer, desto geringer das PDP und desto effizienter die Schaltung.

3.3.2. Energy-Delay-Produkt

Ähnlich wie das PDP stellt das *Energy-Delay-Produkt (EDP)* [125, 30, 124, 88, 17] eine Beziehung zwischen Energie und Zeit auf. Im Gegensatz zum PDP wird beim EDP jedoch aufgrund von $E = P \cdot T \Rightarrow EDP = P \cdot T^2$ die Verarbeitungsdauer höher gewichtet.

3.3.3. Power-Energy-Produkt

Das selten benutzte *Power-Energy-Produkt (PEP)* [177, 124, 88, 17] ist mit dem EDP vergleichbar, nur wird hier die Leistungsaufnahme höher als die Verarbeitungsdauer gewichtet. ($E = P \cdot T \Rightarrow PEP = P^2 \cdot T$)

Alle hier vorgestellten Gütekriterien haben den Nachteil, dass der Flächenbedarf nicht in die Bewertung eingeht. Daher wird im Folgenden ein Maß zur Bewertung der Effizienz vorgestellt, welches alle drei Ressourcen Fläche, Leistungsaufnahme und Zeit miteinander vereint.

3.4. Ein Maß zur Bewertung der Ressourceneffizienz

Um die Ressourceneffizienz eines Systems zu bewerten, wird ein Maß gesucht, dass den Ressourcenbedarf zweier Implementierungen miteinander vergleicht. Hierzu kann z. B. der Quotient der für zwei Systeme $(1, 2)$ benötigten Ressourcen gebildet werden. Nehmen wir als Beispiel zwei Systeme mit der Fläche $A_1 = 10\,\mu\mathrm{m}^2$ und $A_2 = 20\,\mu\mathrm{m}^2$, der Leistungsaufnahme $P_1 = 25\,\mathrm{mW}$ und $P_2 = 10\,\mathrm{mW}$ und der minimalen Taktperiodendauer von $T_1 = 12\,\mathrm{ns}$ und $T_2 = 10\,\mathrm{ns}$. Durch Betrachtung der einzelnen Ressourcen lassen sich die Systeme vergleichen. Für die relative Flächeneffizienz des Systems 1 gegenüber der des Systems 2 ergibt sich beispielsweise $A_1/A_2 = 2$. System 1 ist also in Bezug auf die Fläche ressourceneffizienter. Betrachtet man allerdings alle Ressourcen (Fläche, Leistungsaufnahme und Zeit) gleichzeitig, lässt keine Aussage

Tabelle 3.1.: *Beziehung zwischen der Ressourceneffizienz und bekannten Effizienzmaßen*

	i	j	k	RE
Power-Delay-Produkt	0	-1	-1	$P \cdot T$
Energy-Delay-Produkt	0	-1	-2	$P \cdot T \cdot T = E \cdot T$
Power-Energy-Produkt	0	-2	-1	$P \cdot E = P \cdot P \cdot T$

mehr treffen, welches System effizienter ist. Als Lösung wird der *Ressourceneffizienz-Index* RE als Produkt aller Ressourcen, d.h.

$$RE = A^{-i} \cdot P^{-j} \cdot T^{-k} \qquad i, j, k \in \mathbb{Z}; A < A_{max}, P < P_{max}, T < T_{max}, \tag{3.5}$$

definiert. Die negativen Exponenten bewirken, dass bei steigendem Ressourcenbedarf die Ressourceneffizienz sinkt und umgekehrt. Für $i = j = k = 1$, d.h. für eine gleiche Gewichtung aller Ressourcen, ergibt sich die Ressourceneffizienz sich für die im Beispiel definierten Systeme zu $RE_1 = 3,3 \cdot 10^{-3}\,\mu\mathrm{m}^2 \cdot \mathrm{mW} \cdot \mathrm{ns}$ und $RE_2 = 5 \cdot 10^{-3}\,\mu\mathrm{m}^2 \cdot \mathrm{mW} \cdot \mathrm{ns}$. Das Verhältnis von RE_1 zu RE_2 ist $0,66$. In Bezug auf die Kombination aller Ressourcen ist System 2 also effizienter.

Durch geeignete Wahl der Exponenten kann der Entwickler je nach Anwendungsszenario die Ressourcen gewichten und somit besonderen Wert auf die Ressourceneffizienz bezüglich einer bestimmten Größe legen. Beispielsweise ist beim mobilen Einsatz eine geringe Leistungsaufnahme unerlässlich. Prozessoren für den Massenmarkt müssen möglichst kostengünstig, d.h. flächeneffizient, sein. Für Hochleistungsserver wird unterdessen höchste Geschwindigkeit gefordert. Preis und Leistungsaufnahme spielen nur eine untergeordnete Rolle. Des Weiteren können Ressourcen durch eine obere Schranke begrenzt sein. Die Definition einer oberen Schranke für eine Ressource entspricht einer Beschränkung des Bildraumes (vgl. Abschnitt 3.2). Durch geeignete Wahl der Exponenten lässt sich der Ressourceneffizienz-Index in die in Abschnitt 3.3 beschriebenen Effizienzmaße überführen (vgl. Tabelle 3.1).

Über den Quotienten zweier Ressourceneffizienz-Indizes (RE_1, RE_2) kann die Ressourceneffizienz zweier Systeme in Bezug zueinander gestellt werden. Die relative Ressourceneffizienz ergibt sich nun zu

$$\frac{RE_1}{RE_2} = \left(\frac{A_1}{A_2}\right)^{-i} \cdot \left(\frac{P_1}{P_2}\right)^{-j} \cdot \left(\frac{T_1}{T_2}\right)^{-k} \qquad i, j, k \in \mathbb{Z} \tag{3.6}$$

und somit aus dem Produkt der relativen Effizienz der Ressourcen Fläche, Leistungsaufnahme und Zeit zueinander. Ein ähnlicher Ansatz zur Selektion eines oder mehrerer Systeme aus der Menge der Pareto-optimalen Elemente des Bildraumes wird in [153] verwendet. Die Berechnung der hierfür verwendeten Kostenmaße mithilfe einer Kostenfunktion erfolgt aber durch *Summation* der gewichteten Ressourcen. Im

Gegensatz zu dem in dieser Arbeit vorgestellten Bewertungsmaß können hier jedoch nicht die *einzelnen Ressourcen*, wie in Gleichung 3.6 gezeigt, durch den Quotienten zweier Kostenmaße in Bezug zueinander gesetzt werden.

3.5. Die UPSLA-Beschreibung als Referenzspezifikation

In klassischen Entwurfsabläufen dient der Instruktionssatz als Basis für die weitere Entwicklung von Hardware und Software. Die Ausprägung des Instruktionssatzes resultiert üblicherweise aus Erfahrung oder in Anlehnung an bestehende Prozessorarchitekturen um Rückwärtskompatibilität zu erleichtern[2]. Als erster Schritt wird ein Instruktionssatzsimulator implementiert, der die Ausführung von Binärcode und deren Auswertung in Hinblick auf die Laufzeit ermöglicht. In weiteren Entwurfsschritten folgt die Implementierung von Assembler, Compiler und Linker. Das Profiling kann nun auf solche Anwendungen ausgedehnt werden, die in einer Hochsprache, wie beispielsweise C oder C++, implementiert sind. Als letzter Schritt wird schließlich die Hardware entwickelt. Eine Bewertung der Effizienz der Hardware kann erst unter Betrachtung der Zielanwendungen erfolgen, die üblicherweise in einer Hochsprache implementiert sind. Für eine realistische Bewertung der Ressourceneffizienz ist daher das Vorhandensein eines Compilers unabdingbar.

In enger Zusammenarbeit mit der Fachgruppe „Programmiersprachen & Übersetzer“ von Prof. Dr. Uwe Kastens wurde ein dualer Entwurfsablauf entwickelt, der zur Entwicklung und zur Bewertung der Ressourceneffizienz von Prozessorarchitekturen dient. Abbildung 3.4 zeigt den dualen Entwurfsablauf, der zur Entwicklung und zur Bewertung der Ressourceneffizienz von Prozessorarchitekturen genutzt wird. Zentraler Bestandteil ist die in der Fachgruppe von Prof. Dr. Kastens entwickelte Prozessorspezifikation *Unified Processor Specification Language (UPSLA)* [112, 205] und die darauf aufbauende automatische Generierung einer Compiler-Werkzeugkette. Bei UPSLA handelt es sich um eine deklarative Sprache zur Beschreibung von Prozessorarchitekturen. Das Prozessormodell beschreibt beispielsweise verfügbare Funktionseinheiten und Registerbänke sowie deren Kapazitäten und ihre Konnektivität (vgl. Abbildung 3.5).

Die Sprache verwendet objektorientierte Techniken, was zu einer kompakten Spezifikation führt. Für die Deklarationen können Vererbungstechniken angewandt werden. Instanzen, die auf gleichem Verhalten beruhen basieren auf Äquivalenzklassen. Dieses verhindert Redundanz und ermöglicht konsistente, globale Veränderungen

[2] Der Instruktionssatz des in dieser Arbeit implementierten VLIW-Prozessors ist beispielsweise an den ARM-Prozessor angelehnt, was die spätere Implementierung eines Binär-Compilers ermöglichen soll. Hierdurch soll es möglich sein, für einen ARM-Prozessor kompilierten Code ohne Änderungen direkt auf dem VLIW-Prozessor auszuführen.

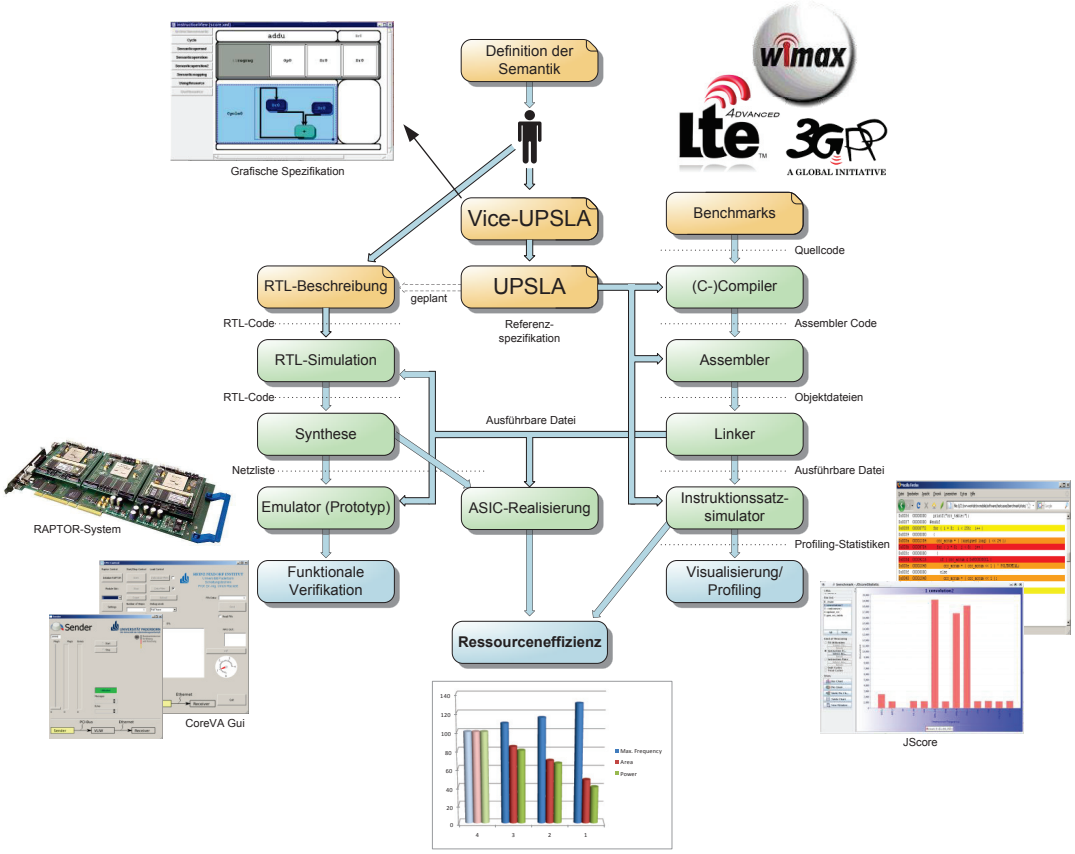


Abbildung 3.4.: Dualer Entwurfsablauf zur Entwurfsraumexploration von Prozessorarchitekturen

an der Spezifikation. Somit kann die Spezifikation leicht an Änderungen der Prozessorarchitektur angepasst werden. Das Werkzeug ViceUPSLA erlaubt die grafische Spezifikation der UPSLA-Spezifikation und erleichtert die intuitive Beschreibung oder Änderung einer Prozessorarchitektur.

Aus der UPSLA-basierten Referenzspezifikation können C-Compiler, Linker, Assembler, ein zyklenakkurater Instruktionssatzsimulator (ISS) und verschiedene Debugging- und Profiling-Werkzeuge automatisch generiert werden. Das Vorhandensein eines C-Compilers bereits mit der ersten Spezifikation eines Prozessors erlaubt sowohl das Profiling der Anwendung als auch die Entwicklung von Hardware und deren Bewertung der Ressourceneffizienz bereits in sehr frühen Entwicklungsstadien. Bei Erweiterungen oder Änderung der Spezifikation kann die Compiler-Werkzeugkette einfach neu generiert werden.

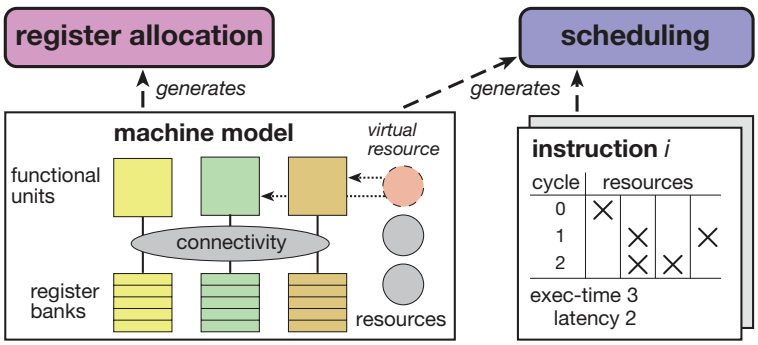


Abbildung 3.5.: Komponenten eines Prozessormodells in der UPSLA-Spezifikation nach [112]

Die automatische Generierung von Hardware ist Gegenstand der aktuellen Arbeiten in der Fachgruppe „Programmiersprachen & Übersetzer“ und ist in der verwendeten Version nicht realisiert. Es stellt sich die Frage, inwieweit sich die Ressourceneffizienz automatisch generierter Hardware von derer handoptimierten RTL-Codes unterscheidet. Diese Fragestellung erfordert weiterführende Forschung und soll nicht Gegenstand dieser Arbeit sein. Daher dient eine manuell erstellte RTL-Beschreibung als Grundlage für die weitere Hardware-Entwicklung. Im Folgenden wird ein Hardware-Entwurfsablauf vorgestellt, der die Abbildung einer RTL-Beschreibung auf eine Zieltechnologie (beispielsweise eine Standardzellentechnologie) ermöglicht. Wichtig bei der Entwicklung ist die regelmäßige Bewertung der Ressourceneffizienz der Schaltung. Ein hoher Grad an Automatismus ist daher für kurze Iterationszyklen bei der Entwurfsraumexploration essenziell.

Neben UPSLA existieren weitere Ansätze zur abstrakten Spezifikation eines Prozessors und der automatischen Generierung sowohl der Werkzeugkette als auch der Hardware. Mit dem Ansatz *Xtensa* von *Tensilica* [87, 121] ist es möglich, einen Basisinstruktionssatz (Xtensa LX Prozessor) um anwendungsspezifische Instruktionen zu erweitern. Der UPSLA-Ansatz bietet hier eine höhere Flexibilität.

Mit dem *CoWare Processor Designer* können sowohl die Entwicklungswerkzeuge als auch die Hardware-Beschreibung aus einer Prozessorspezifikation erzeugt werden [215, 170, 176]. Für die Generierung von Hardware ist allerdings die reine Spezifikation der Befehlssatzarchitektur nicht ausreichend, sondern sie erfordert eine Erweiterung der Spezifikation.

Mit der *CoSy* Compiler-Entwicklungsumgebung von *ACE* [8] lassen sich Compiler für ein weites Spektrum von Zielarchitekturen (8-bit Mikrokontroller, CISC, RISC, DSP, VLIW etc.) erzeugen. CoSy's C-Spracherweiterung *C/DSP-C* ermöglicht es dem Entwickler, den C-Code besser an die Architektur anzupassen.

Der *HiveCC* Compiler von *Silicon Hive* [37, 135] bildet C-Programme auf eine skalierbare VLIW-Architektur ab. Die Architektur kann auf verschiedenen Abstraktionsebenen simuliert werden: Auf C-Code-Ebene, in einer Zwischensprache des HiveCC und auf ISA[3] -Ebene. Wegen der starr vorgegebene Architekturvorlage ist dieser Ansatz allerdings nicht so flexibel wie der von UPSLA oder des CoWare Processor Designers.

3.6. Automatisierter Hardware-Entwurfsablauf

Im folgenden Abschnitt werden zunächst Grundlagen vorgestellt, die für einen allgemeinen Hardware-Entwurfsablauf berücksichtigt werden müssen. In den Abschnitten 3.6.1–3.6.5 wird dieses Vorgehen erweitert, um einen hohen Automatismus beim Hardware-Entwurf zu erzielen. Ziel dieses Automatismus ist die Reduzierung der Entwurfszeit während der einzelnen Iterationen der Entwurfsraumexploration.

Mit zunehmender Verkleinerung der Strukturgrößen und der damit verbundenen Erweiterung der Funktionalität integrierter Schaltungen vergrößert sich auch die Heterogenität der untersuchten Architekturen. Viele verschiedene Komponenten, wie Prozessorkerne, Speicher, On-Chip-Netzwerke (Network-on-Chip (NoC)) oder IP[4] -Blöcke, können in einem System ($\mathbb{S}$) integriert werden. In Kombination mit einer Vielzahl an verfügbaren Zieltechnologien ($\mathbb{Z}$) ergeben sich ständig wachsende Entwurfsräume (vgl. Abschnitt 3.2.1). Zusätzlich bedingt die Wahl der Architektur des Designs die Anwendungen. Beispielsweise hat die Anzahl und Art der Funktionseinheiten einer parallelen Prozessorarchitektur Einfluss auf den Compiler oder die Software-Partitionierung einer Anwendung und muss an die Topologie des NoCs angepasst werden. Die Bestimmung des Ressourcenbedarfs (insbesondere Ausführungszeit und Leistungsaufnahme) kann also nur in Verbindung mit der Anwendung ($\mathbb{A}$) stattfinden.

Die Verkleinerung der Strukturgrößen hat allerdings noch einen weiteren, entscheidenden Einfluss auf die Analyse des Ressourcenbedarfs integrierter Schaltungen: Die maximale Taktfrequenz einer synchronen Schaltung f_{max} ergibt sich aus dem Kehrwert der Verzögerungszeit t_{krit} des kritischen Pfades p_{krit}, welches das Maximum der Länge $t_{delay}(p)$ aller Pfade p eines synchronen Systems $\mathbb{S}$ darstellt:

$$f_{max} = \frac{1}{t_{krit}} = \max\left[t_{delay}(p_{krit})^{-1}\right], \quad p \in \mathbb{S} \tag{3.7}$$

Der kritische Pfad bezieht sich im Allgemeinen auf die Verzögerungszeit eines Register-zu-Register-Pfades t_{delay} und setzt sich aus der Clock-to-Output-Verzöge-

[3] Instruction Set Architecture

[4] Intellectual Property

rungszeit[5] des Quellregistes $t_{cto,reg,source}$, der Summe der Verzögerungszeiten der Gatter $t_{pd,gate}$, der Signalverzögerungen auf den Leitungen $t_{pd,wire}$ und der Setup-Zeit[6] $t_{setup,dest}$ des Zielregisters zusammen:

$$t_{delay} = t_{cto,reg,source} + t_{pd,gate} + t_{pd,wire} + t_{setup,dest} \tag{3.8}$$

Während in der Vergangenheit die Gatterverzögerungen die Gesamtverzögerungszeit dominierten, gewinnen bei kleineren Strukturgrößen (ab etwa 90 nm und geringer) die Leitungsverzögerungen mehr und mehr an Einfluss. Ansätze, die den Ressourcenbedarf (wie bei üblicher Logiksynthese) nur anhand abstrakter Schaltungsmodelle (sogenannter Wire-Load-Modelle) abschätzen, sind nicht mehr ausreichend [31]. Vielmehr ist es notwendig, den Verdrahtungsaufwand genauer zu untersuchen. Aktuelle Entwurfswerkzeuge bieten Funktionen, die die Einflüsse der Verdrahtung genauer abschätzen (beispielsweise *Physical Layout Estimation (PLE)* oder *Physical Synthesis* des Cadence Encounter RTL Compiler). Präzise Abschätzungen bezüglich des Ressourcenbedarfs lassen sich allerdings nur mit vollständig platzierten und verdrahteten Designs treffen.

Die Kombination aus einem ständig wachsenden Entwurfsraum und dem immer höheren Aufwand zur Bestimmung des Ressourcenbedarfs erfordert neue Wege zur Bestimmung der Ressourceneffizienz. Insbesondere ein automatisierter Hardware-Entwurfsablauf ist notwendig. Im Folgenden wird ein *hierarchischer, modulbasierter Ansatz* vorgestellt, der die Abbildung einer abstrakten Hardware-Beschreibung auf eine Zieltechnologie und die Auswertung des Ressourcenbedarfs standardisiert und automatisiert. Die Automatisierung ermöglicht eine Verkürzung der Iterationszyklen der einzelnen Entwurfsschritte und somit der gesamten Entwurfsraumexploration. Die Standardisierung und Automatisierung erleichtert die Reproduzierbarkeit und Wiederverwendbarkeit des Ansatzes. Die implementierte Werkzeugkette definiert Referenzparameter und -abläufe und ermöglicht somit die Vergleichbarkeit der Ergebnisse verschiedener Entwickler.

In heutigen Entwicklungsabteilungen beschränkt sich die Software-Struktur selten auf einen einzigen Anbieter. Vielmehr sind heterogene Strukturen vorhanden in denen jedes Werkzeug anhand sehr spezieller Anforderungen ausgesucht wird. Zwar ähneln sich die Anwendungen zur Entwicklung digitaler Systeme in ihren grundlegenden Eigenschaften, oftmals entscheiden aber spezielle Details des jeweiligen Designs bei

[5] Als „Clock-to-Output“-Verzögerungszeit bezeichnet man die Zeit, nach der das Ausgangssignal des Registers nach dem Auftreten einer Taktflanke am Takteingang des Registers einen stabilen Wert erreicht.

[6] Die „Setup“-Zeit bezeichnet die Zeit, die das Eingangssignal eines Registers vor Auftreten einer Taktflanke am Takteingang des Registers stabil anliegen muss, damit das Register den Wert übernimmt.

der Auswahl der EDA[7] -Werkzeuge. Diese Auswahl ist nicht immer statisch. Oftmals ändern sich auch die Anforderungen während des Hardware-Entwurfs.

3.6.1. Ein hierarchischer Ansatz zur dynamischen Adaption der Entwicklungsumgebung

Die in dieser Arbeit vorgestellte Entwurfsmethodik basiert auf einem modularen Ansatz des *Environment Modules Projects* [80, 79, 223]. Das *Environment Modules Project* erlaubt die dynamische Anpassung der Systemumgebung des Entwicklers durch das Laden und Entladen von sogenannten *Modulen*. Jedes Modul beinhaltet Informationen um die Systemumgebung beispielsweise um Umgebungsvariablen, Aliase, Pfade[8] etc. zu erweitern. Sobald das Modulsystem initialisiert ist, können Module dynamisch geladen oder entladen werden. Beim Laden wird die Systemumgebung um die im Modul definierten Eigenschaften erweitert, beim Entladen wird der ursprüngliche Zustand wiederhergestellt. Die Module können über eine entsprechende Verzeichnisstruktur hierarchisch organisiert werden, um die Übersichtlichkeit und Wartbarkeit zu erleichtern. Abbildung 3.6 zeigt ein Beispiel einer solchen hierarchischen Baumstruktur.

Im gezeigten Beispiel ist der Baum in drei Teilzweige (Technologien/Anwendungen/Sammlungen) unterteilt. Für die Zieltechnologien wird zwischen verschiedenen Technologieherstellern, Strukturgrößen und Versionsnummern unterschieden. Die Blätter des Baumes stellen die ladbaren Module dar. Innerhalb einer Standardzellenbibliothek wird zwischen den verschiedenen Charakterisierungen (erhöhte/typische/reduzierte Versorgungsspannung oder Betriebstemperatur) unterschieden. Wird statt des Namens eines Blattes der Name eines Knotens angegeben, so wird eine zuvor definierte Standardversion eines Moduls geladen/entladen. Durch diese Standardversionen kann der Entwickler automatisch von Bugfixes oder neuen Funktionen profitieren. Eine Erweiterung einer solchen Topologie um verschiedene Gruppen auf der obersten Ebene erleichtert die Organisation von Software und Technologien innerhalb einer Firma, um beispielsweise Software-Lizenzen unterschiedlichen Abteilungen zuordnen zu können. Manche Anwendungen erfordern andere Werkzeuge, um alle Funktionen nutzen zu können. Platzierungs- und Verdrahtungswerkzeuge beispielsweise können auch Logiksynthesen durchführen, falls ein entsprechendes Programm ausführbar ist. Programme zur Logiksynthese erfordern ihrerseits die Definition von Zieltechnologien, um Hardware-Beschreibungen beispielsweise auf eine Standardzellenbibliothek abzubilden. Hierzu erlaubt die Modulumgebung die

[7] Electronic Design Automation

[8] Die Pfad-Variable (engl. *path*) beinhaltet eine Liste der Verzeichnisse, die bei Ausführung eines Kommandozeilen-Befehls nach einer ausführbaren Datei durchsucht werden. Hierdurch ist es nicht notwendig, beim Ausführen von Befehlen absolute Pfade anzugeben.

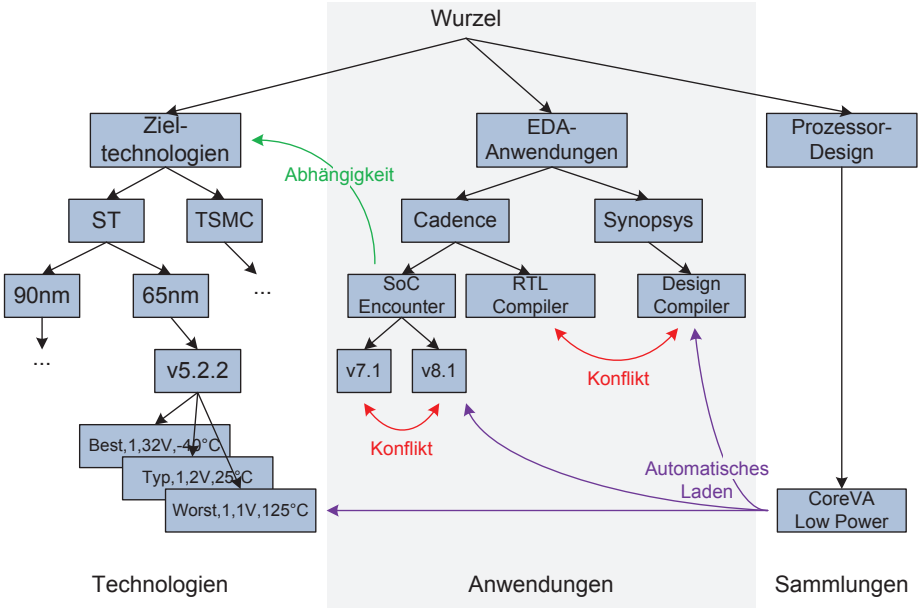


Abbildung 3.6.: Beispiel einer hierarchischen Modulstruktur

Definition von Abhängigkeiten zwischen Modulen. Module mit Abhängigkeiten können nur zeitgleich geladen werden. Andererseits können sich auch Anwendungen gegenseitig ausschließen. Mehrere Werkzeuge einer Anwendungsklasse (z. B. Logiksynthese, vgl. Abschnitt 3.6.2) oder Programme unterschiedlicher Versionen sollten nicht zeitgleich geladen werden. Oftmals liefern EDA-Hersteller eigene Systembibliotheken mit ihrer Software mit, um eine höhere Betriebssystemunabhängigkeit zu gewährleisten. Das Mischen solcher Systembibliotheken kann aber zu Problemen führen. Um solche Probleme zwischen bestimmten Anwendungen zu vermeiden, können Konflikte zwischen Modulen definiert werden. Module mit Konflikten können nicht zeitgleich geladen werden. Die Definition verschiedener Versionen von Technologien und Anwendungen innerhalb der Modultopologie erlaubt einfache Versionswechsel während der Entwurfsraumexploration. Um beispielsweise ein Design auf verschiedene Standardzellenbibliotheken abzubilden, um so den Ressourcenbedarf zu vergleichen, ist lediglich ein Wechsel der Module notwendig. Der in diesem Kapitel vorgestellte automatisierte Entwurfsablauf bleibt hiervon unangetastet und muss lediglich mehrfach ausgeführt werden.

In der hier vorgestellten Entwurfsumgebung standen zum Ende dieser Arbeit mehr als 200 Module für 50 kommerzielle Werkzeuge, acht Zieltechnologien in mehr als 50 Charakterisierungen und zehn Sammlungen für vordefinierte Entwicklungsumgebungen zur Verfügung [243].

3.6.2. Werkzeuge zum automatisierten Hardware-Entwurfsablauf

Um die Iterationszyklen der Entwurfsraumexploration zu verkürzen, wurden verschiedene Werkzeuge implementiert [243]. Mit diesen kann der Entwurf eines integrierten Systems, beginnend bei einer RTL-Beschreibung bis hin zum fertigen Layout, durchgeführt werden. Abbildung 3.7 zeigt eine Übersicht des Hardware-Entwurfsablaufs.

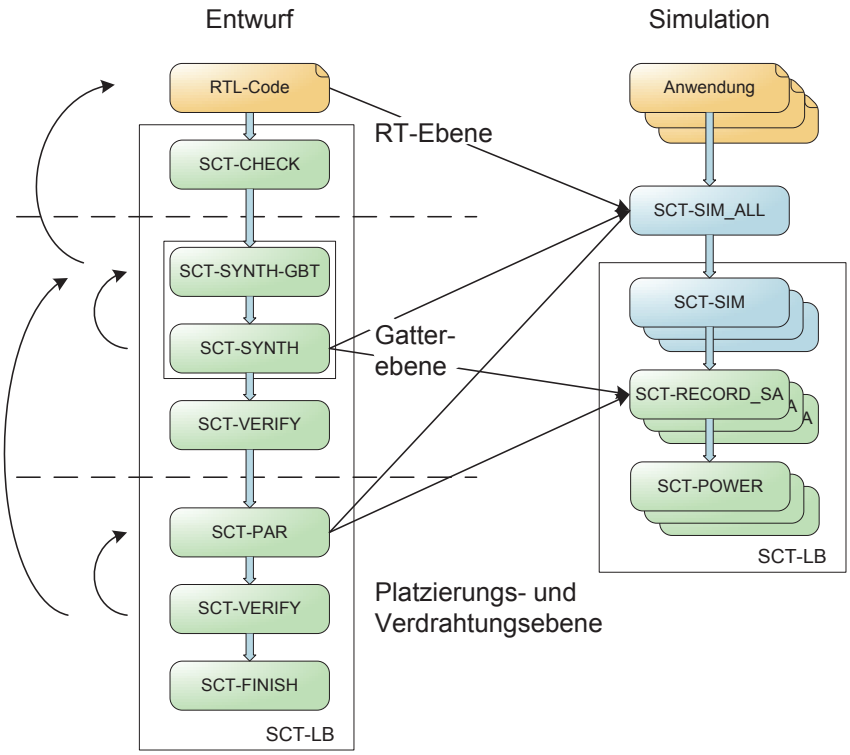


Abbildung 3.7.: Übersicht über den automatisierten Hardware-Entwurfsablauf

Die Abbildung der Hardware-Beschreibung auf RT-Ebene (üblicherweise *VHDL*[9] oder *Verilog*) auf eine Zieltechnologie (z. B. eine Standardzellenbibliothek) unterteilt sich in drei Ebenen:

- RT-Ebene
- Gatter-Ebene

[9] Very High Speed Integrated Circuit Hardware Description Language

- Platzierungs- und Verdrahtungsebene (Layout-Ebene)

Während des Hardware-Entwurfs wird eine *abstrakte Hardware-Beschreibung* der Architektur auf eine *konkrete Technologie* abgebildet. Diese Abbildung wird auch Synthese[10] genannt und untergliedert sich im Allgemeinen in zwei Teilschritte. Laut International Roadmap for Semiconductors (ITRS) 2009 stellen Complementary Metal Oxide Semiconductor (CMOS)-basierte Technologien, insbesondere *Standardzellentechnologien* mit 75 % Marktanteil die gängigste aller Zieltechnologien dar. Im Folgenden werden daher beim Hardware-Entwurf ausschließlich Standardzellen als Zieltechnologien betrachtet. Während der Logiksynthese (auch „Front end“ genannt) wird die RTL-Beschreibung zunächst auf eine technologieunabhängige Netzliste und anschließend auf eine Netzliste, bestehend aus den eigentlichen Standardzellen (Gatterebene), abgebildet[11]. Zu diesem Zeitpunkt werden die Verbindungsleitungen zwischen den Standardzellen über sogenannte „Wire-Load-Modelle“ modelliert. Mit dem Platzieren und Verdrahten (auch physikalische Synthese oder „Back end“ genannt) werden Informationen über die realen Positionen der Zellen und der Verbindungsleitungen hinzugefügt (Platzierungs- und Verdrahtungsebene). Parasitäre Kapazitäten (und somit die Verzögerungszeiten) der Leitungen können genauer als mit den Wire-Load-Modellen abgeschätzt werden.

Die in Abbildung 3.7 dargestellten Werkzeuge des automatisierten Hardware-Entwurfsablaufs werden durch die Module für die jeweiligen EDA-Anwendungen definiert. Nach dem Laden der Module können Kommandos für den jeweiligen Entwurfsschritt ausgeführt werden. Die von den Synthese-Werkzeugen benötigten Eigenschaften der Zieltechnologien werden automatisch durch die Module der Standardzellenbibliotheken definiert. Beispiele für solche Parameter sind:

- Bibliotheken: Die Bibliotheken für die benötigten EDA-Programme müssen angegeben werden. Über die Auswahl der Bibliotheken wird auch der Betriebsbereich (Versorgungsspannung, Betriebstemperatur) definiert.
- Clock-Buffer: Für die Clock-Tree-Synthese werden spezielle Clock-Buffer benötigt.
- Technologiegröße: Cadence SoC Encounter bringt für seine internen Parameter Standarddefinitionen bestimmter Technologiegrößen mit. Parasitäre Kapazitäten können so beispielsweise besser abgeschätzt und Optimierungsschritte genauer auf die jeweilige Technologiegröße abgestimmt werden.

[10] spätlateinisch *synthesis*, von griechisch: σύνθεσις, *sýnthesis* – die Zusammensetzung, Zusammenfassung, Verknüpfung

[11] Im Allgemeinen Sprachgebrauch wird bei der „Logiksynthese“ oft nur von „Synthese“ gesprochen, obwohl genau genommen die Bedeutung des Begriffs „Synthese“ auch die Platzierung und Verdrahtung beinhaltet

- Füllzellen (engl. *filler cells*): Um freie Plätze zwischen Standardzellen aufzufüllen und somit durchgängige Versorgungsleitungen (engl: *power rails*) gewährleisten zu können, werden Füllzellen („leere“ Zellen ohne Funktionalität) eingefügt. Diese müssen zuvor über das Technologiemodul definiert werden. Statt der Füllzellen können auch Zellen mit großen Kapazitäten zwischen Versorgungsspannung und Masse eingefügt werden, die die Versorgungsspannung bei Schwankungen stabilisieren sollen.

- Metalllagen: Die Anzahl und minimalen/maximalen Ausmaße von Metalllagen zum Verdrahten können in dem Technologiemodul angegeben werden.

- Versorgungsnetz (engl.: Power plan): Über das Technologiemodul wird ein Standard-Versorgungsnetz definiert. Das bedeutet, dass zum einen die Namen der Versorgungsnetze (z. B. „VDD“/„GND“), zum anderen auch deren Ausprägung (Weiten der Leiterbahnen, Abstände zwischen vertikalen Verbindungen angegeben werden müssen (vgl. Abschnitt 3.6.2.2).

Insgesamt müssen in einem Technologiemodul etwa 30 Parameter definiert werden.

Der automatisierte Entwurfsablauf [243] startet auf RT-Ebene mit dem Werkzeug *sct-check*[12]. *sct-check* basiert auf dem *Encounter Conformal Equivalence Checker* und führt erste Überprüfungen des RTL-Codes durch. Diese Überprüfungen beinhalten beispielsweise Tests auf ungenutzte Signale, nicht erreichbare Zustände von Zustandsautomaten, Latches etc. Durch diese Vorabtests können viele offensichtliche Fehler bereits zu einem frühen Entwicklungszeitpunkt erkannt werden.

Auf Gatterebene wird die Logiksynthese mithilfe der Werkzeuge *sct-synth* und *sct-synth-gbt* durchgeführt (Verfügbar nach Laden eines Werkzeugmoduls *Synopsys Design Compiler* bzw. *Cadence Encounter RTL Compiler*). Abbildung 3.8 zeigt den Ablauf der Logiksynthese. Dadurch, dass alle technologiespezifischen Parameter in den Technologiemodulen definiert sind, kann die Logiksynthese transparent und unabhängig vom verwendeten Front-end-Werkzeug durchgeführt werden. Der vorgestellte Entwurfsablauf orientiert sich an den Referenzabläufen [24, 116, 50] der Hersteller Synopsys und Cadence und nutzt deren Werkzeuge.

3.6.2.1. Logiksynthese

Als Eingabedaten benötigt die Logiksynthese neben der RTL-Beschreibung die Angabe der Periodendauer (engl.: period constraints). Anschließend wird der RTL-Code kompiliert (Analyze) und auf eine technologieunabhängige Netzliste abgebildet (Elaborate). Nach dem Kompilieren (auch „Synthese“ genannt), wird automatisch

[12] Da der hier vorgestellte Entwurfsablauf in der Fachgruppe Schaltungstechnik der Universität Paderborn entwickelt wurde, ist den Namen der Werkzeuge das Präfix „sct-“ vorangestellt.

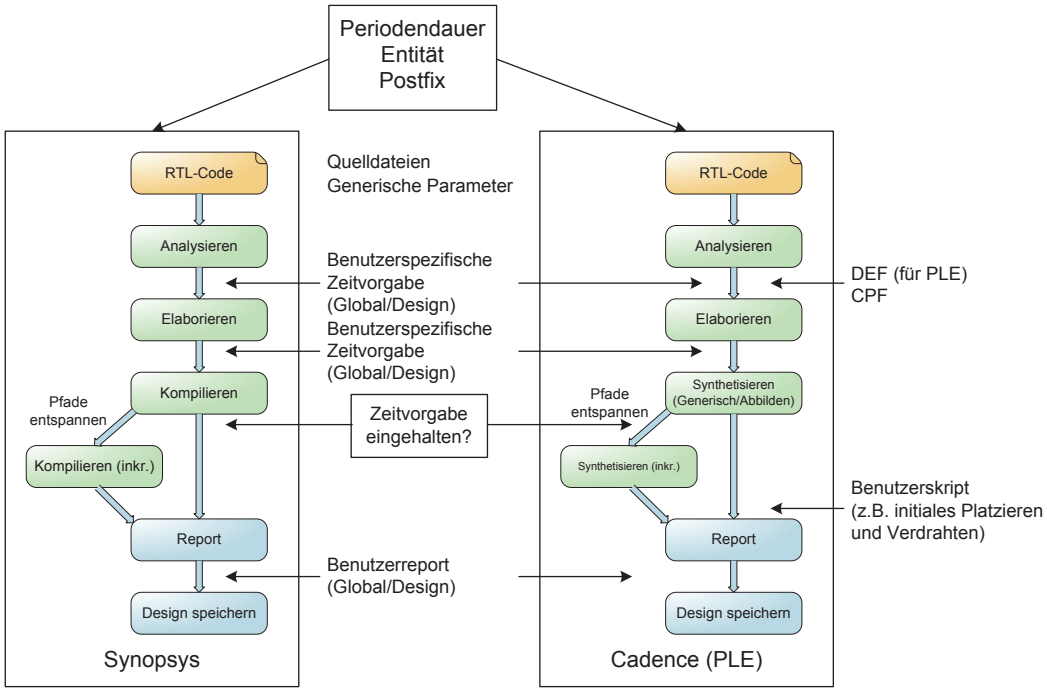


Abbildung 3.8.: Ablaufdiagramm der Logiksynthese

überprüft, ob die Länge des kritischen Pfades (oder auch mehrerer kritischer Pfade gleicher Latenz) die zu Beginn vorgegebene Periodendauer überschreitet. Aus der maximalen Differenz (Worst Negative Slack (WNS)) zwischen der vorgegebenen Periodendauer ($t_{period,constr}$) und der Latenz des kritischen Pfades (t_{krit}) kann die maximale Taktfrequenz abgeschätzt werden (vgl. Gleichung 3.9).

$$t_{wns} = t_{period,constr} - t_{krit} \Leftrightarrow f_{max} = \frac{1}{t_{krit}} = \frac{1}{t_{period,constr} - t_{wns}} \tag{3.9}$$

Erfahrungswerte in der Fachgruppe Schaltungstechnik haben gezeigt, dass sich die besten Ergebnisse (bezüglich der maximal erreichbaren Taktfrequenz) ergeben, wenn die vorgegebenen Periodendauer etwa 10 % unter dem minimal erreichbaren kritischen Pfad liegt. Kann die Vorgabe der Periodendauer für den kritischen Pfad nicht eingehalten werden, so wird die Optimierung dieses Pfades bei der minimal erreichbaren Verzögerungszeit abgebrochen. Alle weiteren Pfade werden aber solange weiter optimiert, bis die Vorgabe eingehalten wird. Die Optimierung der Pfade erfolgt beispielsweise durch die Auswahl größerer Standardzellen mit einer höheren Treiberstärke. Würden diese Pfade nur für die Periodendauer des kritischen Pfades optimiert, wären kleinere Standardzellen ausreichen. Die Abschätzung des

Flächenbedarfs bezieht sich also auf ein Design, welches nur in Teilen die spezifizierte Taktfrequenz erreicht, in seiner Gesamtheit aber nicht funktioniert. Des Weiteren bezieht sich die Abschätzung der Leistungsaufnahme auf die vorgegebene (und nicht die maximal mögliche) Taktfrequenz. Aus diesen Gründen ist es nicht zulässig, Fläche und Leistungsaufnahme für ein Design mit $t_{wns} < 0$ abzuschätzen. Daher analysiert der hier vorgestellte Entwurfsablauf nach der Logiksynthese t_{wns}. Falls t_{wns} negativ ist, so wird die Periodendauer Gleichung 3.10 entsprechend angepasst (die unkritischen Pfade werden „entspannt“):

$$t_{period,constr,neu} := t_{krit} = t_{period,constr,alt} - t_{wns} \tag{3.10}$$

Anschließend wird eine inkrementelle Synthese durchgeführt. Diese bewirkt, dass die Länge des kritische Pfades genau der vorgegebenen Periodendauer entspricht. Kritsche Pfade werden also nicht weiter optimiert. Pfade kürzerer Latenz hingegen werden insoweit optimiert, dass die vorgegebene Periodendauer durch Nutzung langsamerer und kleinerer Gatter genau eingehalten wird. Somit wird der Gesamtflächenbedarf optimiert. Auch die Abschätzung der Leistungsaufnahme bezieht sich nun auf die real erreichte maximale Taktfrequenz.

Zwischen vielen Zwischenschritten des Entwurfsablaufs (vgl. Abbildung 3.8) können über optionale Skripte benutzerspezifische Kommandos abgesetzt werden. Beispielsweise stellt „Clock Gating [225]“ eine häuft eingesetzte Methode zur Reduzierung der dynamischen Verlustleistung dar. Da Clock Gating aber auch negative Eigenschaften (wie z. B. einen erhöhten Flächenbedarf) auf eine Schaltung haben kann, ist es in dem vorgestellten, automatischen Entwurfsablauf nicht standardmäßig aktiviert. Über die optionalen Skripte kann der Anwender diesen Parameter aktivieren. Beim Cadence Encounter RTL Compiler beispielsweise kann während der Synthese eine zusätzliche (Versuchs-) Platzierung und Verdrahtung durchgeführt werden, um bessere Ergebnisse zu erzielen. Die optionalen Skripte können global oder nur auf definierte Architekturen (abhängig vom Namen der Hauptentität) angewendet werden.

Nach Abschluss der Synthese wird der Ressourcenbedarf der Architektur ausgewertet und gespeichert. Zusätzlich zum Referenzablauf der Synthese vom *Cadence Encounter RTL Compiler* werden auch dessen erweiterte Entwurfsabläufe *PLE* und *Common Power Format (CPF)* [57] zur Implementierung von Multi-Voltage-Designs unterstützt. Sind nötige Informationen (Floorplan[13], Widerstands- und Kapazitätswerte zur Modellierung der Verbindungsleitungen bzw. eine `.cpf`-Datei) vorhanden, wird automatisch der erweiterte Enturfsablauf ausgeführt.

Im Anschluss an die Logiksynthese kann ein Werkzeug (*sct-verify*) genutzt werden, welches unter Zuhilfenahme von *Cadence Encounter Conformal Equivalence Checker*

[13] dt.: Raumplan – Informationen über den Grundriss des ASICs und Positionsdaten von speziellen Bausteinen (Hard-Makros), wie z. B. Speicherblöcke oder I/O-Pads

oder *Synopsys Formality*[14] eine formale Verifikation zwischen RTL-Code und erzeugter Netzliste durchführt (vgl. Abschnitt 3.6.5).

3.6.2.2. Platzierung und Verdrahtung

Basierend auf der Ausgabe der Logiksynthese werden bei der Platzierung und Verdrahtung der Netzliste auf Gatterebene Informationen über die Position der Standardzellen sowie die Art der Verbindung dieser hinzugefügt. Das Werkzeug *sct-par* führt diesen Schritt automatisiert durch. Abbildung 3.9 zeigt die Teilschritte der Platzierung und Verdrahtung.

Als erster Schritt wird das Technologiemodul auf die vollständige Definition aller nötigen Parameter überprüft und die Netzliste eingelesen. Anschließend bietet *sct-par* zwei Möglichkeiten des Entwurfs: Platzierung und Verdrahtung mit und ohne fertigen Floorplan. In einem frühen Entwicklungsstadium eines Designs steht häufig noch kein Floorplan zur Verfügung, bzw. der Entwickler kann noch keine Aussage über die benötigte DIE-Fläche treffen.

Die benötigte Fläche ergibt sich nicht nur aus der Summe der Grundfläche der benötigten Standardzellen. Zusätzlich muss noch ein sogenannter Routing-Overhead (Verdrahtungsüberschuss) berücksichtigt werden. Der Routing-Overhead bestimmt die Fläche, die zusätzlich für die Verdrahtung der Standardzellen benötigt wird:

$$A_{Kern} = \sum_i A_{Zelle(i)} \cdot O, \qquad O = \frac{1}{U}$$
$$\text{O : Routing Overhead}$$
$$\text{U : Utilization} \tag{3.11}$$

Eine Flächenausnutzung (Utilization) von 0,8 bedeutet beispielsweise, dass 20 % der DIE-Fläche ausschließlich für das Routing verwendet werden kann. Während der Optimierungsschritte beim Platzieren und Verdrahten kann das Werkzeug jedoch noch Gatter, wie z. B. Buffer, einfügen, um den kritischen Pfad zu optimieren. Aus diesem Grund sollte der Entwickler bei der Utilization eine Reserve einplanen. Die Wahl der Utilization stellt bei der Entwurfsraumexploration einen entscheidenden Parameter dar. So führt eine sehr geringe Utilization zu einem relativ hohen Flächenbedarf. Die Länge der Verbindungsleitungen und somit die Leitungsverzögerungen zwischen den Gattern steigt. Mit wachsender Utilization verringert sich die Fläche des Designs. Da jedoch weniger Ressourcen für das Routing vorhanden sind, kann das Layout-Werkzeug möglicherweise nicht immer die bestmögliche Verbindung wählen (sog. *Routing Congestion*). Je weniger Fläche für das Routing genutzt

[14] je nach geladenem Modul

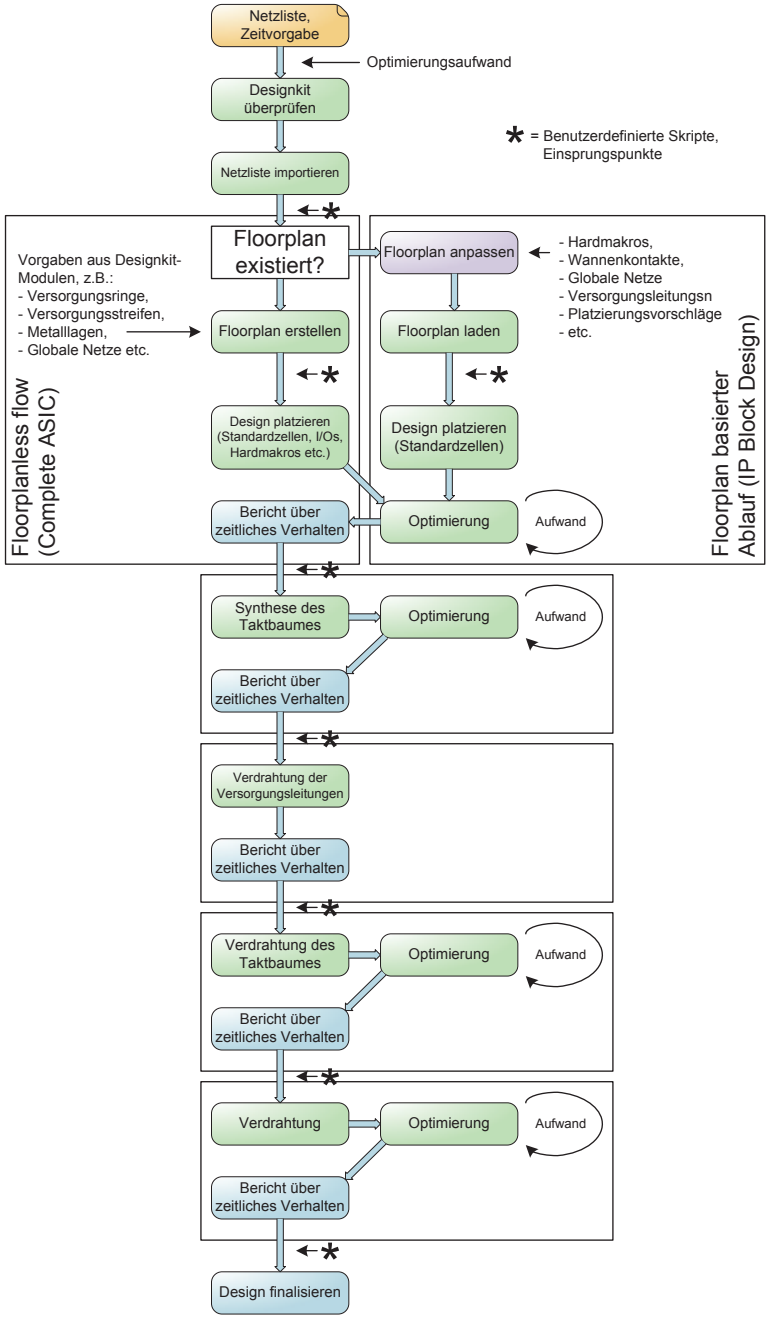


Abbildung 3.9.: Ablaufdiagramm der Platzierung und Verdrahtung

werden kann, um so mehr Wechsel zwischen denn Metalllagen des Prozesses sind notwendig. Die Verbindungen (Vias) zwischen den Metalllagen weisen einen hohen Übergangswiderstand auf. Insgesamt kann sich die Verzögerungszeit zwischen den Gattern mit stark steigender Utilization also auch erhöhen. Hier gilt es, den besten Kompromiss zwischen Chip-Fläche und Verzögerungszeit des kritischen Pfades zu finden. Ist noch kein Floorplan für das Platzieren und Verdrahten angegeben, so kann der Entwickler bei Aufruf des Werkzeugs *sct-par* einen Startwert für die Utilization vorgeben. Aus der Summe der Fläche der Standardzellen und der Utilization wird nun die Chipfläche vorgegeben. Diese wird quadratisch vorgegeben. Nach der Definition der Grundfläche wird das Versorgungs-Netzwerk erzeugt. In ASICs mit ringförmiger Anordnung der Versorgungs-Pads (im Gegensatz zu der sogenannten *Flip-Chip*-Technologie, wobei die Versorgungs-Pads über die gesamte Chip-Fläche verteilt sind), muss die Versorgungsspannung also von außen an die Standardzellen heran geführt werden. Üblicherweise wird hierzu ein sogenannter *Power-Ring* erzeugt, der die eigentliche Standardzellenfläche umschließt. Die Beschaffenheit der Standardzellen ergibt horizontale Versorgungsspannung- und Masse-Verbindungen, die an den beiden äußeren Enden mit dem Power-Ring verbunden werden (Power-Rails). Zusätzlich werden in äquidistanten Abständen vertikale Verbindungen von dem oberen bis zum unteren Ende des Power-Rings gelegt (Power-Strips). Dieses ergibt ein maschenförmiges Power-Netzwerk und soll Spannungsabfälle (den sogenannten IR-Drop, vgl. Abschnitt 3.6.3) innerhalb des ASICs vermeiden. In den Technologie-Modulen sind Parameter (Abstände zwischen den Power-Strips, Breite der Metallleitungen etc.) für ein initiales Power-Netzwerk vorgegeben. Eine spätere Analyse des IR-Drops kann eventuelle Versorgungsprobleme aufzeigen. In einem manuell erstellten Floorplan kann das Power-Netzwerk dann an die Anforderungen des Designs angepasst werden.

Sind in der verwendeten Netzliste I/O-Pad-Zellen vorhanden, so werden diese in äquidistanten Abständen um die Chip-Fläche herum angeordnet. Freiräume zwischen den Pad-Zellen werden durch Füllzellen bzw. Eckzellen (engl. *corner cells*) für die vier Ecken aufgefüllt. Diese werden zuvor im Technologie-Modul definiert. Eine durchgehende Verbindung ist für die Anbindung der Pad-Zellen an die Spannungsversorgung (sowohl I/O-Spannung als auch Kernspannung) notwendig.

Im nächsten Schritt werden die Standardzellen der Netzliste auf der Chip-Fläche platziert. Nach dem Platzieren der Standardzellen erfolgt das Verdrahten der Leitungen für die Versorgungsspannungen, die Taktbaumsynthese, das Verdrahten des Taktbaumes und das eigentliche Verdrahten der Logik. Nach den Schritten Platzieren, Taktbaumsynthese, Verdrahten des Taktbaumes und Verdrahten der Logik erfolgt ein Optimierungsprozess, dessen Aufwand (über die Anzahl der Optimierungsschritte) dem Werkzeug übergeben werden kann. Die Anzahl der Optimierungsschritte hat einen großen Einfluss auf das Ergebnis des Platzierungs- und Verdrahtungsvorganges (Optimierung des kritischen Pfades) aber auch auf die Laufzeit. Bei komplexen

Designs und einer Iteration liegt die Laufzeit für eine vollständige Platzierung und Verdrahtung bei einigen Stunden. Infolgedessen kann bei mehreren Optimierungsdurchläufen die Laufzeit durchaus mehrere Tage beanspruchen.

3.6.3. Analyse der Leistungsaufnahme

Wie Abbildung 3.7 verdeutlicht, kann die *Analyse der Leistungsaufnahme* auf verschiedenen Abstraktionsebenen erfolgen. Möglich sind die Register-Transfer-, die Synthese- und die Platzierungs- und Verdrahtungsebene. Mit Hilfe des Werkzeugs *sct-record_sa* können während der Simulation der Ausführung von Anwendungen die Schaltaktivitäten der Bauteile aufgenommen werden. Die Schaltaktivitäten werden von dem Werkzeug *sct-power* ausgewertet und damit die anwendungsspezifische Leistungsaufnahme abgeschätzt. Weiterhin kann nach der Platzierung und Verdrahtung eine Analyse des Versorgungsnetzwerkes durchgeführt werden. *sct-irdrop* basiert auf den in SoC Encounter integrierten Mechanismen zur Analyse der Potentialverteilung über die Chipfläche. Es bestimmt den sogenannten IR-Drop, d.h. den maximalen Spannungsabfall an den Versorgungs-Pins der Standardzellen, und ermittelt mögliche Probleme durch Elektromigration [27]. IR-Drop und Elektromigration können, wie in Abbildung 3.10 dargestellt, visualisiert werden.

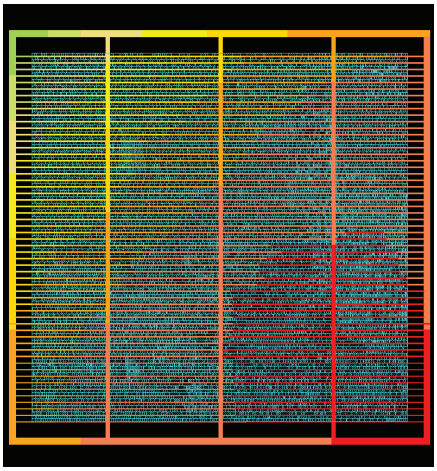


Abbildung 3.10.: Ergebnis einer IR-Drop-Analyse. Rote Bereiche kennzeichnen Regionen mit einem kritischen Spannungsabfall am jeweiligen Bauelement

3.6.4. Verteilte Ausführung von Entwurfsschritten

Die Komplexität des automatisierten Entwurfsablaufs erfordert eine hohe Rechenleistung. Zur Exploration sehr großer Entwurfsräume sind jedoch kurze Iterationszyklen wichtig. Viele Teilschritte (z. B. die Synthese für verschiedene Prozess-Parameter, die Bestimmung der Schaltaktivitäten bei Simulation der Ausführung der Anwendungen etc.) können jedoch parallel erfolgen. Steht eine Vielzahl von Rechnerknoten zur Verfügung, können mehrere Schritte gleichzeitig auf diesen ausgeführt werden. Das Werkzeug *sct-lb*[15] realisiert eine effiziente *Lastverteilung* der Teilschritte auf einem Rechencluster. Wird ein Auftrag über *sct-lb* ausgeführt, ermittelt das Werkzeug die aktuelle Rechenlast auf dem Rechencluster und platziert den Auftrag auf dem Knoten mit der geringsten Last. Hierbei wird auch die Performanz der jeweiligen Rechnerhardware berücksichtigt. Eine Überlastung des Rechenclusters wird verhindert, indem bei kritischer Last Aufträge verzögert werden. *sct-lb* ist nicht auf spezielle Anwendungen beschränkt, sondern kann mit beliebigen Programmaufrufen kombiniert werden.

3.6.5. Verifikation und Validierung

Ein wichtiger Bestandteil des Entwurfsprozesses von Prozessorarchitekturen besteht in der Verifikation. Wie bereits in Abschnitt 3.6.2.1 beschrieben kann im Anschluss an die Logiksynthese das Werkzeug (*sct-verify*) genutzt werden, welches unter Zuhilfenahme von *Cadence Encounter Conformal Equivalence Checker* oder *Synopsys Formality*[16] die formale Verifikation zwischen RTL-Code und erzeugter Netzliste durchführt. Somit kann die Konsistenz zwischen Prozessorspezifikation auf RT-Ebene und finaler Hardware-Implementierung verifiziert werden.

Zusätzlich muss jedoch auch die Konsistenz zwischen der Hardware-Beschreibung und der Compiler-Werkzeugkette sichergestellt sein. Die Ursache einer Inkonsistenz kann sowohl auf der Hardware-Seite (z. B. durch eine falsche Implementierung einer Instruktion), als auch auf Seiten der Werkzeugkette liegen (z. B. durch eine fehlerhafte Beschreibung einer Instruktion in der UPSLA-Spezifikation). Die Herausforderung bei der formalen Verifikation von Hardware-Implementierung und Compiler-Werkzeugkette liegt in der Beschreibung auf verschiedenen Abstraktionsebenen (vgl. Abbildung 3.11): In der Ausführungsdomäne der Hardware-Implementierung (RTL-Simulation, FPGA-Emulator, ASIC-Prototyp) handelt es sich um eine *Mikroarchitektur*, bei dem Instruktionssimulator dagegen um eine *Befehlssatzarchitektur* (ISA). Der Instruktionssatzsimulator abstrahiert Eigenschaften der Mikroarchitektur stark, d.h. Komponenten wie die Pipelinearchitektur werden im ISS nicht simuliert,

[15] *load balancing*

[16] je nach geladenem Modul

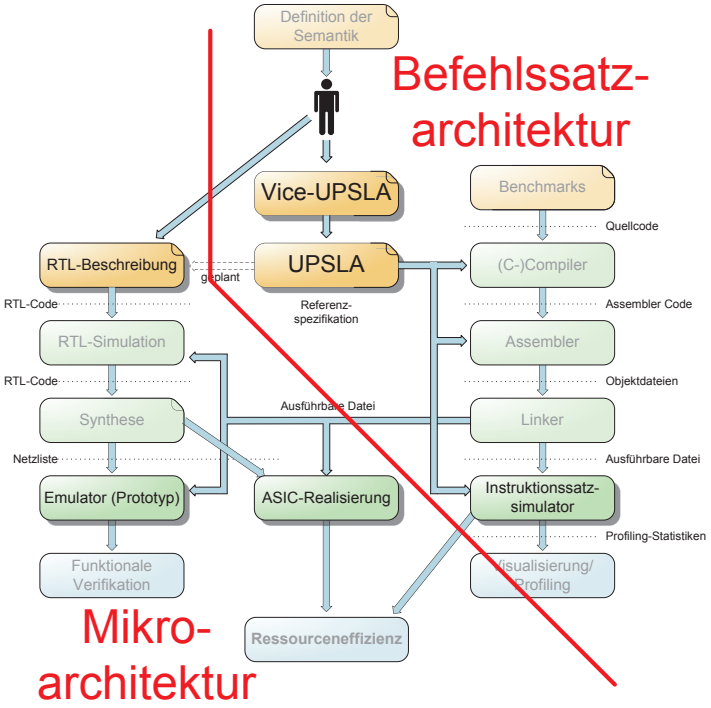


Abbildung 3.11.: Klassifizierung der Ausführungsdomänen in Microarchitektur und Befehlssatzarchitektur

sondern Instruktionen werden als Ganzes ausgeführt. Eine vollständige formale Verifikation zwischen Mikroarchitektur und Befehlssatzarchitektur (d.h. die Überprüfung, ob das System seiner formalen Spezifikation genügt [49, 141]) ist für ein komplexes Prozessorsystem nicht praktikabel (Explosion des Zustandsraumes [48, 94, 10]).

Obwohl sich laufende Arbeiten im Fachgebiet „Programmiersprachen und Übersetzer“ von Prof. Kastens mit der automatischen Generierung der Hardware-Beschreibung aus der Spezifikation einer Prozessorarchitektur beschäftigen, ist es zur Zeit noch nicht möglich, eine RTL-Beschreibung aus der UPSLA-Spezifikation zu erzeugen. In der Literatur existieren zwar Ansätze zur formalen Verifikation von ISA und Mikroarchitektur [36, 174]. Diese beziehen sich allerdings nur auf den Kontrollpfad der Architektur und berücksichtigen beispielsweise kein Pipeline-Forwarding. Andere Ansätze, wie z. B. [131], schränken die Freiheitsgrade der Architektur stark ein.

Daher wird in dieser Arbeit zusätzlich zur formalen Verifikation der Hardware-Beschreibung gegenüber der Hardware-Realisierung sowie der funktionalen Verifikation durch FPGA-basierte Prototypen (vgl. Abschnitt 3.6.5.1) eine *simulationsbasierte*

Validierung durchgeführt (vgl. Abschnitt 3.6.5.2 [232]).

3.6.5.1. Funktionale Verifikation

Zur *funktionalen Verifikation* von Prozessorarchitekturen wird das in der Fachgruppe Schaltungstechnik entwickelte *Rapid-Prototyping-System RAPTOR* [161, 162, 110] eingesetzt. Der modulare Aufbau des als PCI-Teilnehmer realisierten Systems (vgl. Abbildung 3.12) erlaubt den Einsatz verschiedener Erweiterungsmodule, wie beispielsweise FPGA-Module zur sehr schnellen Emulation. Die Emulationsgeschwindigkeit liegt um Größenordnungen (mehrere MHz) über derer der VHDL-Simulation (kHz). Die Rekonfigurierbarkeit der FPGAs stellt eine kostengünstige Alternative zu ASIC-Prototypen dar (vgl. Kapitel 8). Weiterhin stehen auch Erweiterungsmodule zur Verfügung, die physikalische Schnittstellen, wie beispielsweise USB[17] , (Gigabit-) Ethernet oder WLAN[18] nach IEEE 802.11 (vgl. Abschnitt 8.3.1), implementieren. Physikalische Schnittstellen erlauben die Emulation eines Prozessorsystems in realen Einsatzszenarien bei hohen Taktraten. Die Modularität des RAPTOR-Systems erlaubt weiterhin die einfache Entwicklung von ASIC-Evaluations-Platinen. Aufgrund der einheitlichen Schnittstelle lassen sich ASIC-Prototypen so direkt in die RAPTOR-basierte Prototyping-Umgebung integrieren.

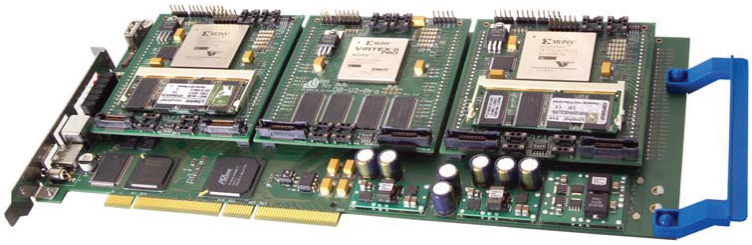


Abbildung 3.12.: Das Rapid-Prototyping-System RAPTOR

3.6.5.2. Multi-Domänen-Validierung

Zur *Validierung der Architektur* wird Programmcode in verschiedenen Ausführungsdomänen (beispielsweise RTL-Simulation und ISS) simuliert. In jedem Taktzyklus wird der aktuelle Prozessorzustand bestimmt. Die Folge von Prozessorzuständen ergibt sogenannte Prozessor-Traces (von engl. *trace* = *Ablaufverfolgung*). Steht der Zustand der Architektur zu Beginn des Programmablaufes fest, so kann durch Bestimmung

[17] Universal Serial Bus

[18] Wireless Local Area Network

der Schreibzugriffe auf interne oder externe Speicher (in der Regel Register-File oder Datenspeicher) in jedem Taktzyklus der aktuelle Zustand bestimmt werden. Die Prozessor-Traces zweier Simulationsdomänen werden miteinander verglichen. Treten Unterschiede zwischen diesen Prozessor-Traces auf, besteht eine Inkonsistenz zwischen diesen beiden Domänen. Die in diesem Abschnitt vorgestellte Methode berücksichtigt Pipelining-Effekte, wie Prozessor-Stalls, Pipeline-Flushes oder nicht-uniforme Latenzen. Existierende Methoden zur Generierung von Prozessor-Traces, wie beispielsweise in [43], ermitteln den Prozessorzustand durch Bestimmung der Schreibzugriffe in der letzten Pipelinestufe (Register-Write). Dieses führt jedoch zu einem zusätzlichen Hardware-Aufwand (z. B. Pipelineregister), da normalerweise nicht alle Signale bis in die letzte Pipelinestufe geführt werden müssen. Des Weiteren eignet sich der Ansatz von [43] nicht für Prozessorarchitekturen mit nicht-uniformen Latenzen bei Schreibzugriffen. Ähnliche Ansätze, wie die in [96] oder [227], verwenden Co-Simulation zur Validierung von Prozessorarchitekturen, präsentieren aber keine Lösungen zur Konvertierung der Prozessortraces von einer Befehlsarchitektur zur Mikroarchitektur oder umgekehrt. In [13] und [23] wird eine Validierung von Prozessoren und Multiprocessor-System-On-Chips (MPSOCs) ausschließlich durch Vergleich der Zugriffsmuster auf den Datenspeicher durchgeführt.

Der in dieser Arbeit vorgestellte Ansatz bestimmt die notwendigen Signalwerte in der geeignetsten Pipelinestufe und synchronisiert die Signale verschiedener Ausführungsdomänen in einer virtuellen Pipeline. Dieses erfordert aber auch die Berücksichtigung von Pipelineeffekten, wie etwa Strafzyklen, Pipeline-Flushes, Pipeline-Forwarding oder nicht-uniformer Latenzen [232]. Die Qualität der Validierung steigt mit der Testüberdeckung [95, 6, 178]. Die Generierung von Testszenarien mit einer hohen Testüberdeckung ist seit langem Gegenstand der Forschung [55, 142, 4, 212, 25, 45, 38] und soll in dieser Arbeit nicht weiter behandelt werden. Im Gegensatz zu anderen Ansätzen können mit dem hier vorgestellten Verfahren nicht nur RTL-Simulation und Instruktionssatzsimulator, sondern auch FPGA-Emulator und ASIC-Prototyp validiert werden. In der RTL-Simulation werden die Signalwerte über das sogenannte *Foreign Language Interface (FLI)* von ModelSim bestimmt. Für die Hardwarerealisierungen, wie FPGA-Emulator oder ASIC-Prototyp, wurde eine generische *Trace-Schnittstelle* entwickelt, die das Auslesen der Prozessorzustände zur Laufzeit ermöglicht. Treten Inkonsistenzen zwischen zwei Ausführungsdomänen auf, so gibt das Werkzeug *Tracediff* [232] dem Entwickler detaillierte Informationen, die die Lokalisierung der Ursache der Inkonsistenz erleichtern. Abbildung 3.13 zeigt die Ausgabe von *Tracediff*. Es wird die Ablaufverfolgung der n letzten Instruktionen dargestellt, die für das Verhalten der aktuellen Instruktion relevant sind. Der Übersichtlichkeit halber wird nicht der gesamte Zustand dargestellt, sondern nur die Register-Einträge gezeigt, die für die Fehlerlokalisierung relevanten sind. Weiterführende Informationen zu der in dieser Arbeit realisierten Multi-Domänen-Validierung finden sich in [232].

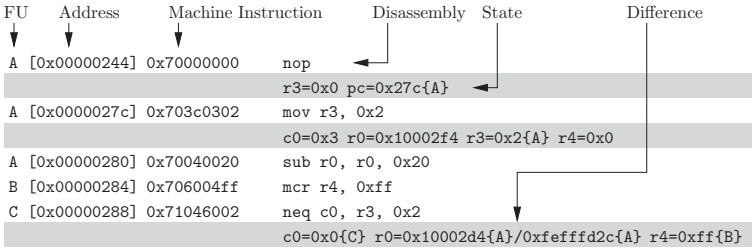


Abbildung 3.13.: Ausgabe von Tracediff zur Lokalisierung von Inkonsistenzen zwischen verschiedenen Ausführungsdomänen

3.7. Zusammenfassung

In diesem Kapitel wurde ein dualer Entwurfsablauf zur Entwurfsraumexploration von Prozessorarchitekturen vorgestellt. Grundlage für den Entwurf ist eine Referenzspezifikation des Prozessors in UPSLA. Aus dieser Spezifikation kann die gesamte Werkzeugkette, bestehend aus einem C-Compiler, einem Assembler, einem Linker, einem zyklenakkuraten Instuktionssatzsimulator und verschiedenen Profiling-Werkzeugen automatisch generiert werden. Die automatische Generierung garantiert die Konsistenz zwischen Spezifikation und Werkzeugen. Für die Entwicklung der Hardware-Architektur wurde ein automatisierter Entwurfsablauf implementiert, der auf der Spezifikation des Systems in einer HDL-Beschreibung beruht. Die Dualität des Entwurfsablaufs erlaubt die parallele Entwicklung von Soft- und Hardware und eine frühe Auswertung des Ressourcenbedarfs des Gesamtsystems. Der Automatismus sowohl der Generierung der Software-Werkzeugkette als auch der Abbildung der Hardware-Architektur auf eine Zieltechnologie, wie beispielsweise Standardzellen, ermöglicht kurze Iterationszyklen. Um die Konsistenz zwischen Hardware-Architektur und Compiler-Werkzeugkette sicherzustellen wurden verschiedene Verfahren zur Verifikation und Validierung realisiert.

Zur Bewertung der Ressourceneffizienz wurde ein Maß definiert, welches die drei Ressourcen *Fläche*, *Leistungsaufnahme* und *Zeit* miteinander verknüpft. Über geeignete Wahl der Parameter kann dieses Maß auch auf bekannte Bewertungsmaße, wie das *Power-Delay-Produkt* oder das *Energy-Delay-Produkt*, zurückgeführt werden. Eine Vergleichbarkeit zu bestehenden Arbeiten bleibt somit bestehen.

Der vorgestellte Entwurfsablauf dient als Grundlage für die Entwicklung und Analyse eines VLIW-Prozessors in Kapitel 4. Der Automatismus ermöglicht die Untersuchung eines sehr großen Entwurfsraumes unter Berücksichtigung einer vielseitig konfigurierbaren Architektur und individueller Anwendungsszenarien. Weitere Informationen zum entwickelten Entwurfsablauf finden sich auch in [243, 232, 242, 238, 236].

4. Die CoreVA-VLIW-Architektur – Entwicklung einer modularen, konfigurierbaren Prozessorarchitektur

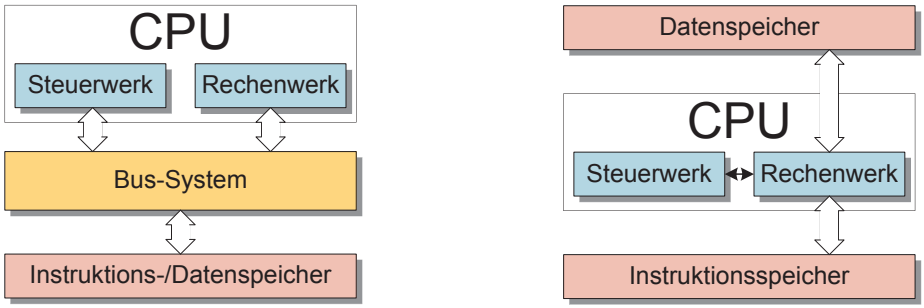
Dieses Kapitel beschreibt die *Implementierung* einer modularen und konfigurierbaren Prozessorarchitektur. Wie bereits in Kapitel 2 beschrieben, stellen VLIW-Architekturen eine vielversprechende Alternative gegenüber superskalaren Architekturen im Bereich eingebetteter Systeme dar [74]. Ziel der Entwicklung ist es, bezüglich der möglichen Hardware-Konfigurationen einen großen Entwurfsraum abzudecken. In Abschnitt 4.1 wird die zugrunde liegende Computerarchitektur eingeführt. In Abschnitt 4.2 wird beschrieben, wie im Vorfeld der Entwicklung die Verzögerungszeiten der einzelnen Teilmodule abgeschätzt wurden. Abschnitt 4.3 zeigt den Aufbau der VLIW-Architektur und die realisierten Konfigurationsmöglichkeiten auf. Abschnitt 4.4 geht auf die Besonderheiten des Pipeline-Bypasses ein. In Abschnitt 4.5 werden die erzielten Ergebnisse zusammengefasst und ein Ausblick auf die nachfolgende Entwurfsraumexploration der Architektur gegeben.

4.1. Zugrunde liegende Computerarchitektur

Zur Entwicklung einer modularen, konfigurierbaren VLIW-Architektur ist es notwendig, zuerst die allgemeinen Eigenschaften einer *Reduced Instruction Set Computer (RISC)-Architektur* zu betrachten. Grundsätzlich benötigt ein Prozessor Programmbefehle sowie Daten, die durch das Programm manipuliert werden können [157]. Dieses kann auf verschiedenste Weise erfolgen. Erste (analoge) Rechenwerke hatten fest verdrahtete Programme, die nur umständlich zu ändern waren. Mit der Entwicklung von dynamischen Speichern war es möglich, Programme auch zur Laufzeit zu verändern. Im Wesentlichen entwickelten sich zwei verschiedene Architekturen zur Anbindung von Speicher an einen Prozessorkern.

Die *Von-Neumann-Architektur* wurde 1945 von John von Neumann entwickelt [86]. Sie zeichnet sich durch ein Speicherwerk aus, in dem sowohl Daten als auch Programmcode gespeichert sind. Rechenwerk und Steuerwerk des Prozessors können über einen gemeinsamen Bus auf den Inhalt des Speichers zugreifen. Zu Beginn der Prozessorentwicklung stellte insbesondere das Rechenwerk den Flaschenhals bezüglich der maximalen Taktfrequenz dar. Speicher konnte verhältnismäßig schnell

angesprochen werden. Mit Aufkommen hochintegrierter, mikroelektronischer Bausteine fand jedoch eine Kehrtwende dieser Abhängigkeiten statt. Während die kombinatorische Logik moderner CMOS[1] -Technologien immer schnellere Taktfrequenzen ermöglichte, wurde nun die Leistungsfähigkeit der Computersysteme durch den gemeinsamen Daten- und Steuerbus begrenzt [151, 99, 14].



(a) Von-Neumann-Architektur **(b)** Harvard-Architektur

Abbildung 4.1.: Vergleich der verschiedenenen Möglichkeiten der Speicheranbindung

Bei der *Harvard-Architektur* [157] gibt es daher separate Speicher für Daten und Instruktionen (vgl. Abbildung 4.1). Hierdurch kann der Hauptprozessor (Central Processing Unit (CPU)) in einem Taktzyklus sowohl Daten als auch Instruktionen laden. Ein Nachteil der Harvard-Architektur ist, dass nicht genutzter Daten- bzw. Instruktionsspeicher nicht für Instruktionen bzw. Daten zur Verfügung steht. Die strikte Trennung von Programm und Daten verhindert jedoch potenzielle Sicherheitslücken, wie Pufferüberläufe. Besteht trotzdem die Notwendigkeit der Vereinigung von Programm- und Datenspeicher, so kann dieses über den Einsatz von Speicherhierarchiemodellen und unter Zuhilfenahme von Caches gelöst werden. Moderne Mikroprozessoren sind fast ausschließlich als Harvard-Architektur implementiert. Aus den zuvor genannten Gründen ist der in diesem Kapitel entwickelte Prozessor als Harvard-Architektur ausgelegt.

Der Instruktionssatz des im Rahmen dieser Arbeit entwickelten VLIW-Prozessors ist an den der ARM-Architektur angelehnt (vgl. Anhang A). Dieses soll die spätere Realisierung eines Binär-Compilers erleichtern. Ein Binär-Compiler kann beispielsweise eine dedizierte Hardware-Einheit sein, die für ARM erzeugten Binärcode in Instruktionen der VLIW-Architektur übersetzt. Ein Beispiel für Binär-Compiler ist die *Code-Morphing*-Technologie der Crusoe-Prozessoren der Firma Transmeta [113, 58].

[1] Complementary Metal Oxide Semiconductor

4.2. Abschätzung der Taktfrequenz

Als Zielfrequenz für die zu entwickelnde Architektur wurden 400 MHz bei typischen Operationsbedingungen (*Typical Case*) in einer 65 nm Standardzellentechnologie von STMicroelectronics anvisiert. „Ungünstige“ Operationsbedingungen (reduzierte Versorgungsspannung, hohe Umgebungstemperatur, (*Worst Case*)) führen in diesem Fall zu einer maximalen Taktfrequenz von etwa 260 MHz. Tabelle 4.1 zeigt die Betriebsbereiche (engl. *corner cases*), für die die verwendete Standardzellentechnologie charakterisiert ist. Um die Robustheit eines ASICs gegen Änderungen der Umgebung zu erhöhen, werden Standardzellensynthesen üblicherweise für ungünstige Umgebungseigenschaften (Worst Case) durchgeführt. Die nachfolgenden Synthese-Ergebnisse sind daher durchgehend mit der Worst-Case-Charakterisierung der Zellen durchgeführt worden.

Tabelle 4.1.: *Charakterisierte Betriebsbereiche für die 65 nm Standardzellentechnologie von STMicroelectronics*

Operationsbedingung	Versorgungsspannung	Umgebungstemperatur
Best Case	1,30 V	-40 °C
Typical Case	1,20 V	25 °C
Worst Case	1,10 V	125 °C

Die Taktfrequenz von 260 MHz (Worst Case) entspricht einer maximalen Verzögerungszeit der kombinatorischen Logik von 3,8 ns. Erfahrungen in der Fachgruppe Schaltungstechnik haben gezeigt, dass für das spätere Platzieren und Verdrahten der Synthesenetzliste eine gewisse „Reserve“ von etwa 33 % eingeplant werden muss. Diese Diskrepanz resultiert zum einen aus den zusätzlichen parasitären Effekten der Verdrahtungsleitungen. Zum anderen wird aus Kostengründen beim ASIC-Design versucht, die benötigte Chip-Fläche zu minimieren. Restriktionen bezüglich der Fläche haben negativen Einfluß auf die maximale Taktfrequenz. Die maximale Verzögerungszeit des VLIW-Prozessors nach der Synthese sollte also maximal 2,8 ns betragen, um die Vorgaben einhalten zu können. Die in dieser Arbeit verwendeten Speichermakros haben eine Latenz von etwa 0,5 ns. In der im Folgenden vorgestellten Architektur verläuft der kritische Pfad durch den Prozessorkern zur Speicherschnittstelle des Instruktionsspeichers. Die Latenz des Speichers ist also additiv. Für den Prozessorkern verbleibt somit eine maximale Verzögerungszeit von etwa 2,3 ns (vgl. Abbildung 4.2).

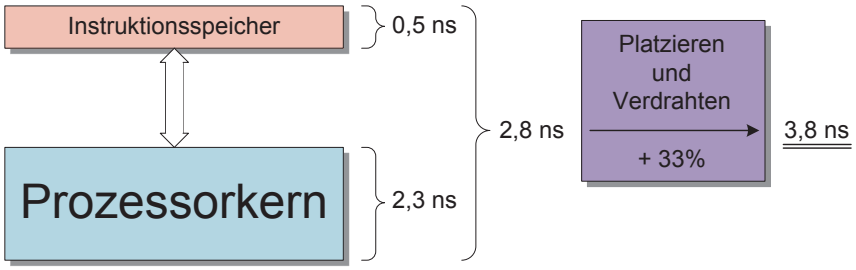


Abbildung 4.2.: Abschätzung der Verzögerungszeit des VLIW-Prozessors

4.3. Die Pipelinearchitektur

Die Erhöhung des Datendurchsatzes einer Prozessorarchitektur durch simples Anheben der Taktfrequenz stößt trotz immer kleiner werdender Strukturgrößen leicht an ihre Grenzen. Die Länge des kritischen Pfades, der maßgeblich zur Taktfrequenz beiträgt, ist abhängig von der zugrunde liegenden Logik. Zwar erlaubt eine verhältnismäßig einfache Logik eine geringe Latenz, gleichzeitig reduziert sich jedoch der Funktionsumfang und somit auch die Leistungsfähigkeit des Gesamtsystems. Eine Architektur, die alle Funktionen der Instruktionen in einem Taktzyklus beschreibt, ist offensichtlich für moderne Architekturen nicht effizient. Ein Ausweg zur Reduktion des kritischen Pfades trotz gleichbleibender Funktionalität des Gesamtsystems ist das *Pipelining*. Beim Pipelining wird die Ausführung der verschiedenen Instruktionen überlappt, indem Register-Stufen in die kombinatorische Logik eingefügt werden. Die maximale Taktfrequenz erhöht sich. Pipelining macht sich die Parallelität der einzelnen Ausführungsbestandteile der Instruktionen zunutze. Während eine Funktionseinheit mit der Berechnung einer Instruktion beschäftigt ist, kann die nachfolgende bereits aus dem Instruktionsspeicher geladen werden. Die Parallelität und damit der Durchsatz steigt (in erster Annäherung[2]) linear mit der Anzahl der Pipelinestufen.

Die grundlegenden Funktionen, die eine (Harvard-Architektur-basierte) Prozessorarchitektur unterstützen muss, sind nach [157]:

1. Laden der Instruktion (Instruction-Fetch (FE))
2. Dekodieren der Instruktion und Laden der Operanden aus dem Register-File (Instruction-Decode (DC))

[2] Eigenschaften der Pipeline-Register, wie Setup-Zeit oder Clock-to-Output-Verzögerung, oder unsymmetrisch auf die Pipelinestufen verteilte kombinatorische Logik begrenzen den Gewinn für eine hohe Anzahl an Pipelinestufen und führen zu einer asymptotischen Begrenzung der möglichen Performanzsteigerung

3. Ausführen der Instruktion (Execute (EX))
4. Zugriff auf den Datenspeicher (Memory (ME))
5. Schreiben der Ergebnisse in das Register-File (Register-Write (WR))

Im Allgemeinen werden die folgenden Funktionseinheiten betrachtet, die für eingebettete Prozessoren relevant sind: Die *ALU* umfasst die Funktionalität der Mehrheit an notwendigen Instruktionen. Diese beinhaltet beispielsweise boolesche Operationen (Und, Oder, Negation etc.), arithmetische Operationen (Addieren, Subtrahieren). Auch Basisoperationen für Spezialinstruktionen (z. B. Addition von Basisadresse und Offset bei Sprüngen), können auf die ALU zurückgeführt werden. In der implementierten Architektur ist eine ALU in allen VLIW-Slots vorgesehen. Ausnahme stellen die in der Basisbandverarbeitung häufig genutzten Operationen *Multiplikation* und *Division* dar. Multiplikation und Division könnten zwar auch in Software (über sogenannte Multi-Cycle-Instruktionen (vgl. Abschnitt 4.3.7.4) emuliert werden, für eine effiziente Ausführung ist eine Hardwareunterstützung aber essentiell. Da (Hardware-) Multiplizierer und Dividierer im Vergleich zu den anderen Operationen einen hohen Ressourcenbedarf verursachen, sind beide in der vorgestellten Architektur als dedizierte Hardware-Einheiten implementiert. Wie in Abschnitt 4.3.7 detailliert beschrieben, kann die Anzahl dedizierter Multiplizierer und Dividierer in der implementierten Architektur parametrisiert werden.

4.3.1. Allgemeine Eigenschaften der Architektur

In den nachfolgenden Abschnitten werden verschiedene Bezeichner für Adressen verwendet: Als *Byteadresse* wird eine 32 Bit breite Adresse bezeichnet, mit dem jedes Byte im 2^{32} Byte großen Adressraum des Instruktions- und Datenspeichers adressiert werden kann. Eine *Wortadresse* zeigt auf ein ganzes 32-Bit-Wort. Daher können im Vergleich zur Byteadresse die zwei niederwertigen Bits vernachlässigt werden. Eine *Gruppenadresse* zeigt auf eine Instruktionsgruppe (auch VLIW genannt) bestehend aus 1 bis n Instruktionen bei einer n-fach parallelen VLIW-Architektur. Da in der nachfolgend beschriebenen Architektur Instruktionsgruppen nicht auf $(n \cdot 32)$ Bits sondern nur auf 32-Bit-Grenzen ausgerichtet sein müssen, sind genau wie bei den Wortadressen die letzten zwei Bits im Vergleich zu den Byteadressen redundant. *Blockadressen* adressieren $(n \cdot 32)$-Bit-Speicherblöcke im Instruktionsspeicher. Hier sind die letzten vier Bits im Vergleich zur Byteadresse redundant. Daten werden analog mit *Instruktions-/Datenbyte*, *Instruktions-/Datenwort*, *Instruktionsgruppe* und *Instruktionsblock* bezeichnet. Instruktionswörter werden durch *Instruktionen* abgekürzt. Abbildung 4.3 zeigt die Nomenklatur der Speicheradressen.

Die Pipelinearchitektur des Prozessors ist als sogenannte *non-interlocked pipeline* realisiert [231], d.h. Datenkonflikte zwischen aufeinanderfolgenden Instruktionen

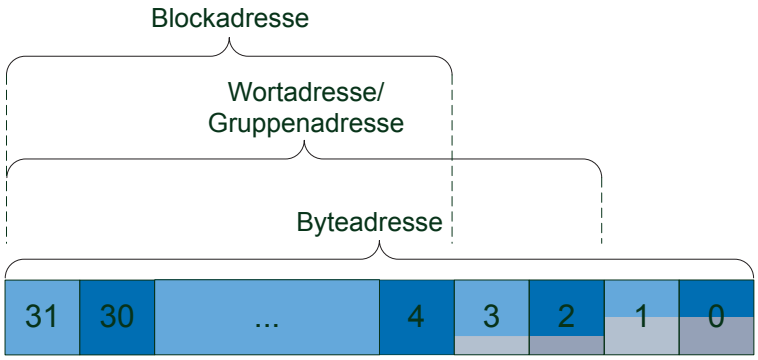


Abbildung 4.3.: Nomenklatur der Speicheradressen

können durch einen sogenannten *Pipeline-Bypass* aufgelöst werden. Falls Datenabhängigkeiten zwischen aufeinanderfolgenden Instruktionen bestehen, werden Zwischenergebnisse vorangegangener Instruktionen in die Decode-Pipelinestufe zurückgeführt. Das Prinzip des Pipeline-Bypasses wird genauer in Abschnitt 4.4 erläutert. Wie sich zeigen wird, ist die kombinatorische Logik des Pipeline-Bypasses kritisch und kann die maximale Taktfrequenz einer Prozessorarchitektur begrenzen. Daher wurde bereits im Vorfeld die im vorigen Abschnitt zugrunde gelegte Pipelinearchitektur um die Register-Read-Pipelinestufe (RD) erweitert. Hieraus ergibt sich die folgende, sechsstufige Pipelinestruktur [236]:

1. Laden der Instruktion (Instruction-Fetch, (FE))
2. Dekodieren der Instruktion (Instruction-Decode, (DC))
3. Laden der Operanden aus dem Register-File (Register-Read (RD))
4. Ausführen der Instruktion (Execute, (EX))
5. Zugriff auf den Datenspeicher (Memory, (ME))
6. Schreiben der Ergebnisse in die Register (Register-Write, (WR))

Ziel der weiteren Entwicklung ist die Implementierung einer n-fach-parallelen VLIW-Architektur. Im Folgenden werden die einzelnen Pipelinestufen und Funktionseinheiten der Prozessorarchitektur beschrieben und im Besonderen auf die speziellen Anforderungen durch die geforderte Modularität eingegangen [249].

4.3.2. Das Register-File

Als lokaler Speicher kommt ein *Register-File* mit 31 32-Bit-Einträgen zum Einsatz. Bei VLIW-Architekturen können mehrere VLIW-Slots zur gleichen Zeit ein und dasselbe Register verändern (Ressourcenkonflikt). Dieses wird bei der entwickelten Architektur durch eine Veroderung der Eingangsdaten behandelt. Hierdurch lassen sich auch mehrere 8-Bit-Operationen auf ein 32-Bit-Wort effizient parallelisieren. Zur bedingten Ausführung von Instruktionen stehen zusätzlich zwei *8-Bit-Condition-Register* zur Verfügung. Ein Condition-Register steht für skalare Operationen zur Verfügung. Das zweite Condition-Register kommt bei SIMD-Instruktionen (vgl. Abschnitt 4.3.7.2) zum Einsatz. Hierdurch können zwei 16-Bit-SIMD-Instruktionsströme unabhängig voneinander ausgeführt werden. Unbedingte Instruktionen referenzieren auf Bit 7 des Condition-Registers. Dieses liegt fest auf dem Wert 1 und kann nicht verändert werden (read only).

4.3.3. Instruktionskompression

Die *Chipfläche* stellt in eingebetteten Systemen eine kritische Größe dar. Sie bestimmt in erster Linie die *Kosten* eines Systems. In modernen Prozessorarchitekturen wird neben der reinen kombinatorischen Logik ein Großteil der Chipfläche durch On-Chip-Speicher belegt. In einer n-fach parallelen VLIW-basierten Architektur bestehen die Instruktionsgruppen aus maximal n Instruktionswörtern. Verfügt das auszuführende Programm aber nicht über genügend ILP, so ist der Compiler nicht in der Lage, alle n Funktionseinheiten der Architektur auszunutzen. In Folge dessen werden in ungenutzten VLIW-Slots sogenannte *No-Operation (NOP)*-Instruktionen ausgeführt. Dieses bläht den Instruktionscode unnötig auf. Als Lösung dieses Problems verwenden VLIW-Architekturen in der Regel Kompressionsmethoden, um den Einfluss von ungenutzten Funktionseinheiten auf die Codegröße zu verringern [69, 140, 104, 226]. In der hier vorgestellten Architektur signalisieren sogenannte *Stop-Bits (st)* in den Instruktionen das Ende einer Instruktionsgruppe (vgl. Abbildung 4.4) [75]. Abbildung 4.5 zeigt einen Vergleich zweier Instruktionsspeicherabbilder mit und ohne Instruktionskompression.

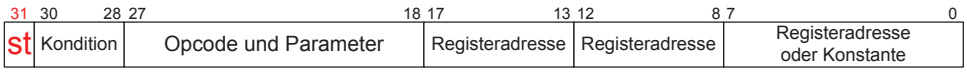
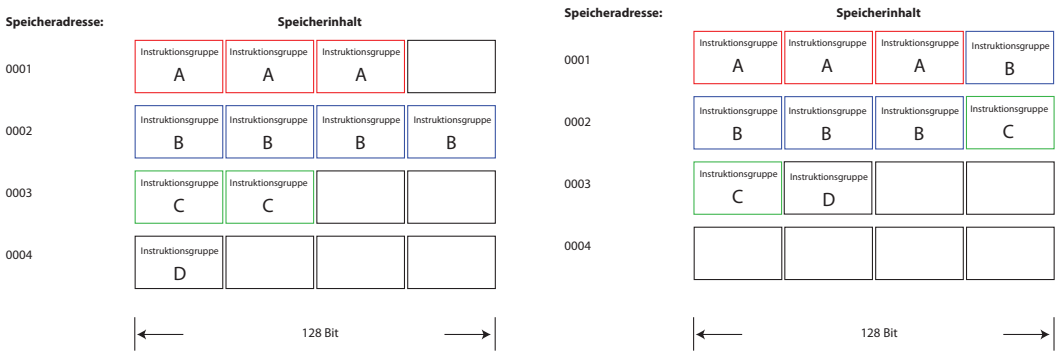


Abbildung 4.4.: Kodierung des Stop-Bits innerhalb einer Instruktion



(a) Ohne Kompression **(b)** Mit Kompression durch Stop-Bits

Abbildung 4.5.: Vergleich zweier Instruktionsspeicherabbilder

4.3.4. Instruction-Fetch

Durch die verwendete Kompression für den Instruktionsspeicher ergeben sich insbesondere durch eine variable Anzahl von Ausführungskanälen einige Einschränkungen. Für ein besseres Verständnis wird daher im Folgenden die genaue Funktionsweise des Lesezugriffs auf den Instruktionsspeicher skizziert. Zur Dekodierung der Instruktionen müssen alle Instruktionen einer Instruktionsgruppe zur gleichen Zeit in den Eingangsregistern der Decode-Stufe vorliegen. Wie in Abbildung 4.5 zu sehen, können Instruktionsgruppen aber auch die Speicheradressen von Instruktionsblöcken überlappen. Das führt dazu, dass bis zu zwei Taktzyklen benötigt werden, um eine komplette Instruktionsgruppen aus dem Instruktionsspeicher zu laden. Hierzu wird in der vorgestellten VLIW-Architektur ein sogenanntes *Alignment-Register* zur Ausrichtung der Instruktionsgruppen eingesetzt. Das Alignment-Register kann mindestens zwei Instruktionsblöcke zwischenspeichern. Dieses garantiert, dass immer mindestens zwei Instruktionsgruppen gleichzeitig gespeichert werden können. Die Größe des Alignment-Registers ist also im Allgemeinen $2 \cdot n \cdot 32$ Bits. Aufgrund der Zwischenspeicherung von Instruktionen wird ein Alignment-Register oftmals auch als *Level-0*-Cache bezeichnet. Abbildung 4.6 zeigt ein Beispiel für den Ausschnitt eines Instruktionsspeichers und der zeitlichen Abfolge des Inhaltes des Alignment-Registers für eine 4-fach-parallele VLIW-Architektur.

Das Alignment-Register ist in zwei Hälften aufgeteilt: Jede Hälfte kann einen Instruktionsblock aufnehmen. Das Alignment-Register kann logisch gesehen auch als Ring betrachtet werden. Instruktionsgruppen werden (im sequentiellen Fall) immer abwechselnd in der oberen und unteren Hälfte des Alignment-Registers abgelegt. Im folgenden Beispiel symbolisieren die Buchstaben (A-G) die Instruktionsgruppenzugehörigkeit. Der Zugriff auf den Instruktionsspeicher läuft hier in folgenden Schritten ab:

Abbildung 4.6.: Funktionsweise des Alignment-Registers

- Taktzyklus (1)
  Die erste Instruktionsgruppe (A) ist bereits ins Alignment-Register geladen worden[3] und kann im nächsten Takt an die nächste Pipelinestufe (Decode) übergeben werden. Der Inhalt der Speicheradresse 0x0001[4] wird in den oberen Teil des Alignment-Registers gelesen.

- Taktzyklus (2)
  Die zweite Instruktionsgruppe (B) wurde eingelesen. Ein Teil der dritten Instruktionsgruppe (C) befindet sich bereits im oberen Teil des geladenen Instruktionsblocks, da die zweite Instruktionsgruppe (B) nicht die volle Parallelität

[3] Hier wird zur Vereinfachung angenommen, dass die erste Speicheradresse bereits in das Alignment-Register geladen wurde. Normalerweise ist hierfür ein zusätzlicher Takt notwendig.

[4] Instruktionsblockadresse

der Architektur nutzt. Speicheradresse 0x0002 wird in den unteren Teil des Alignment-Register eingelesen.

- Taktzyklus (3)
  Die ersten Instruktionen der dritten Instruktionsgruppe (C) befinden sich nun im oberen Teil, die anderen beiden Instruktionen im unteren Teil. Eine Zuweisung der Instruktionen an die Ausführungskanäle kann recht einfach mithilfe einer Modulo-2 Operation erfolgen. Dazu ist es lediglich erforderlich, die Startposition der Instruktionsgruppe innerhalb des Alignment-Registers zu speichern (hier: Registeradresse 7). Weitere Registerzugriffe werden einfach inkrementiert und mit einer Modulo-8-Operation verknüpft. Dadurch wird der Zugriff auf das 8. Register auf das Register an Speicherstelle Null abgebildet, der Zugriff auf das 9. entsprechend auf das 1. Register etc. Hierdurch ergibt sich die zuvor beschriebene Ringtopologie. Speicheradresse 0x0003 wird in den oberen Teil des Alignment-Register eingelesen.

- Taktzyklus (4)
  Die Instruktionsgruppen (D), (E), (F) und ein Teil von Instruktionsgruppe (G) sind bereits im Alignment-Register gespeichert. Das Einlesen des nächsten Instruktionsblocks von Adresse 0x0004 aus dem Instruktionsspeicher würde einen Teil der noch nicht bearbeiteten Instruktionen überschreiben. Daher muss der Lesezugriff verzögert werden, bis die Instruktionsgruppen (D), (E) und (F) an die nächste Pipelinestufe übergeben wurden.

Für das Beispiel einer 4-fach-parallelen VLIW-Architektur kann die Berechnung der Startadresse einer Instruktionsgruppe innerhalb des Alignment-Registers über eine Modulo-4-Operation (dieses entspricht dem einfachen Selektieren der zwei niederwertigsten Bits) auf die Instruktionswortadresse (d.h. dem Programmzähler) berechnet werden. Im Allgemeinen kann die Startadresse innerhalb des Alignment-Registers für eine n-fach-parallele Architektur über eine Modulo-n-Operation erfolgen. Gilt

$$\exists\, m \in \mathbb{N} : 2^m = n, \tag{4.1}$$

d.h. n ist eine Zweierpotenz, so kann die Berechnung der Startadresse über Auswahl der m niederwertigsten Bits erfolgen. Für alle anderen $n \in \mathbb{N}$, d.h. für

$$\nexists\, m \in \mathbb{N} : 2^m = n, \tag{4.2}$$

ist jedoch eine verhältnismäßig aufwendige Division notwendig.

Um diesem Problem zu begegnen, sind drei Lösungswege denkbar:

1. Für kleinere n (z. B. für $n < 4$) kann unter Umständen eine *Tabelle* eingesetzt werden, die bereits vorberechnete Werte enthält.

2. *Verzicht auf die Kompression:*
   Im einfachsten Fall kann auf eine Kompression des Instruktionsspeichers verzichtet werden. Instruktionsgruppen, die nicht die gesamte Bandbreite nutzen, werden mit Leerbefehlen (NOP) aufgefüllt. Dadurch vereinfacht sich auch der Aufbau des Alignment-Registers. Ein großer Nachteil dieser Lösung ist, dass die Codedichte innerhalb des Instruktionsspeichers sinkt. Auch wenn der Instruktionsspeicher im Vergleich zum Datenspeicher häufig kleiner ist, ist nicht nutzbarer Speicher, insbesondere im Bereich eingebetteter Systeme, häufig keine Option.

3. *Ausrichtung von Sprungbefehlen:*
   Bei der sequenziellen Ausführung von Instruktionscode kann auf eine Modulo-Operation verzichtet werden. In diesem Fall kann anhand der Instruktionsgruppen im Alignment-Register entschieden werden, ob die nächste Speicheradresse einzulesen ist. Bei Sprungbefehlen muss diese Berechnung stattfinden. Eine Option ist hier, Instruktionsgruppen, die das Ziel eines Sprungs sind vom Compiler an Instruktionsblockadressen ausrichten zu lassen[5]. Bei kurzen Sprüngen kann die Zielberechnung eventuell anhand einer Tabelle erfolgen. Ein Problem stellt bei dieser Lösung der Instruktionszeiger dar. Nach einem Sprung muss der Instruktionszeiger aufwendig neu berechnet werden. Auch sind relative Sprünge zum Instruktionszeiger nicht mehr ohne weiteres möglich.

4. *Speicherschnittstelle mit der Größe einer Zweierpotenz:*
   Wie gezeigt, ist die Berechnung einer Modulo-Operation im Binärsystem immer dann besonders einfach, wenn der Operator eine Potenz von zwei ist. In diesem Fall vereinfacht sich der Divisionsprozess zu einer einfachen Auswahl der niederwertigsten Bits. Daher bietet sich die Möglichkeit an, die Speicherschnittstelle bzw. die Instruktionsblockgröße im Instruktionsspeicher unabhängig von der Anzahl der Ausführungskanäle zu bestimmen. Für einen Ausführungskanal kann eine 32 Bit, für zwei Ausführungskanäle eine 64 Bit und für drei und vier Ausführungskanäle eine 128 Bit breite Speicherschnittstelle genutzt werden. Im Allgemeinen kann die Breite der Speicherschnittstelle über

$$N_{Speicherschnittstelle} = 2^{\lceil log_2 n \rceil}, \quad n \in \mathbb{N} \tag{4.3}$$

   berechnet werden. Andererseits erfordert diese Implementierung eine größere Speicherschnittstelle, als eigentlich benötigt. Auch das Alignment-Register, das

[5] Die aktuelle Version des für diese Arbeit implementierten VLIW-Compilers nutzt ein solches Verfahren bereits, um stets eine gesamte Instruktionsgruppe einlesen zu können. Dieses vermeidet zusätzliche Wartetakte.

direkt von der Größe der Speicherschnittstelle abhängt, muss größer dimensioniert werden und benötigt daher mehr Ressourcen innerhalb des Prozessors. Ein größeres Alignment-Register ist jedoch nicht nur ein Nachteil. Beispielsweise können sehr kurze Schleifen in einem großzügig ausgelegten Alignment-Register komplett zwischengespeichert werden. Betrachtet man nun das Alignment-Register wie zuvor als Level-0-Cache, so können Zugriffe auf (ggf. langsameren) externen Speicher vermieden werden. Zur Bewertung der zusätzlich benötigten Logikressourcen wurden Synthesen des Alignment-Registers für verschiedene Wortbreiten der Speicherschnittstelle durchgeführt (vgl. Tabelle 4.2). Wie die spätere Analyse des Gesamtsystems zeigen wird, liegt der zusätzliche Ressourcenbedarf bezogen auf die Gesamtgröße des Prozessorkerns im Durchschnitt bei unter 2 %. Jedoch müssen auch die zusätzlichen Ports für die Anbindung an den Speicher sowie ein Speicher mit einer größeren Speicherschnittstelle berücksichtigt werden.

Tabelle 4.2.: *Größenvergleich des Alignment-Registers bei unterschiedlicher Wortbreite der Speicherschnittstelle in einer 65 nm Standardzellentechnologie von STMicroelectronics*

Breite der Speicherschnittstelle	Größe in μm^2
64 Bit	3808
128 Bit	4799
256 Bit	6004

Abschließend lässt sich feststellen, dass es keine optimale Lösung für dieses Problem[6] gibt. Da sich Möglichkeit 4 mit einem vertretbarem Ressourcenaufwand realisieren ließ, wurde diese Implementierung für die in diesem Kapitel vorgestellte Architektur ausgewählt.

4.3.5. Instruction-Decode

Anhand des *Programmzählers* kann die Position der aktuellen Instruktion innerhalb des Alignment-Registers bestimmt werden. Aus den Stop-Bits der Instruktionen wird die Länge der aktuellen Instruktionsgruppe dekodiert. Die Instruktionswörter der Instruktionsgruppe werden auf n Dekodiereinheiten verteilt. Dort werden

[6] Die Verwendung von Ausführungskanälen, die von der Anzahl her keine Zweierpotenz sind, stellt generell ein Problem bei VLIW-Prozessorarchitekturen für den Zugriff auf den Instruktionsspeicher dar. Dieses zeigt sich auch am Beispiel des Intel Itanium Prozessors. Obwohl dieser nur drei Ausführungskanäle verwendet, nutzt er dennoch eine 128 Bit breite Speicherschnittstelle.

die Operationen, Parameter, Quell- und Zieladressen der Register, Steuersignale für die Funktionseinheiten etc. dekodiert. Insgesamt besteht der Instruktionssatz des VLIW-Prozessors aus 41 Basisinstruktionen und 15 SIMD-Instruktionen (vgl. Abschnitt 4.3.7.2).

Bei Sprüngen auf nicht auf Blockadressen ausgerichtete Gruppenadressen, die sich über zwei Blockadressen erstrecken, werden zwei CPU-Takte benötigt, um die gesamte Instruktionsgruppe zu laden. Daher müssen die Pipelinestufen RD–WR des Prozessors angehalten werden. Da die Auswertung der Condition-Register erst in der Register-Read-Pipelinestufe erfolgt, kann erst einen Taktzyklus später ermittelt werden, ob ein bedingter Sprung ausgeführt wird oder nicht. Die nachfolgende Instruktionsadresse (d.h. das Sprungziel) könnte also erst einen Takt später ermittelt werden. Um diesen zusätzlichen Wartezyklus zu vermeiden, wird eine für RISC-Architekturen übliche *Sprungvorhersage* durchgeführt. Relative Sprünge treten z. B. bei Schleifendurchläufen auf. Wird eine Schleife mehr als einmal durchlaufen, so dominieren *Rücksprünge*, d.h. das absolute Sprungziel ist kleiner als die Wortadresse der aktuellen Instruktion. *Vorwärtssprünge* treten in den meisten Programmen seltener auf. Daher wird bei einem Rücksprung im Allgemeinen angenommen, dass dieser ausgeführt wird. Der Programmzähleroffset wird durch Summation von dem aktuellen Programmzähler und dem in der Instruktion enthaltenen Offset gebildet. Die Adresse der (bei sequentieller Ausführung) nachfolgenden Instruktion[7] wird in einem Register zwischengespeichert. Einen Takt später wird in der Register-Read-Stufe das zu dem Sprung gehörige Condition Bit ausgewertet, um festzustellen, ob die Sprungvorhersage richtig durchgeführt wurde. Ist dieses nicht der Fall, werden die Pipelineregister zwischen der Register-Read- und der Execute-Pipelinestufe zurückgesetzt und die zuvor für die sequentielle Ausführung gesicherte Adresse an den Instruktionscache angelegt (*Pipeline Flush*). Der fälschlich ausgeführte Sprung wird also „rückabgewickelt“. Analog erfolgt die Behandlung bei einem Vorwärtssprung. Zunächst wird die Folgeadresse für die sequentielle Ausführung an den Instruktionscache angelegt. Der berechnete Programmoffset wird in einem Register zwischengespeichert. Im Falle einer falschen Vorhersage wird die Folgeinstruktion gelöscht und die Instruktion am Sprungziel geladen. Die verwendete Pipelinestruktur sorgt für eine weitere Eigenschaft der VLIW-Architektur: Wird ein Sprung ausgeführt, so vergehen mindestens zwei Taktzyklen, bis die Instruktion am Sprungziel dekodiert werden kann. Eine auf einen Sprung folgende Instruktion ist also bereits geladen, wenn der Sprung erkannt wurde. Um diese Eigenschaft effizient zu nutzen, wurde die Architektur so implementiert, dass eine auf einen Sprung nachfolgende Instruktion (im Nachfolgenden als *Delay-Slot* bezeichnet) grundsätzlich *zusätzlich* ausgeführt wird.

Um eine möglichst homogene Architektur zu erhalten, wurden identische Deko-

[7] Die Instruktion, die bei einem nicht ausgeführten Sprung eigentlich ausgeführt werden müsste.

diereinheiten implementiert. Eigene Tests ergaben, dass der Einfluss anwendungsspezifischer Dekodiereinheiten mit unterschiedlichem Instruktionssatz vernachlässigbar ist. Des Weiteren sind moderne Synthesewerkzeuge in der Lage, Redundanz innerhalb der Dekodiereinheiten der Execute-Pipelinestufe bei nicht implementierten Funktionseinheiten weitgehend zu entfernen.

4.3.6. Register-Read

In der *Register-Read-Pipelinestufe* werden die in der vorherigen Pipelinestufe dekodierten Registeradressen an das *Register-File* angelegt. Pro ALU müssen je zwei Operanden bereitgestellt werden. Ein dedizierter Multiplikationsakkumulator (Multiply-Accumulate (MLA)) benötigt drei Operanden ($Y = A \cdot B + C$). Die benötigte Anzahl der Lese-Ports des Register-Files ergibt sich also zu:

$$N_{\text{Register-Lese-Ports}} = N_{\text{MLA}} \cdot 3 + (N_{\text{ALU}} - N_{\text{MLA}}) \cdot 2 \tag{4.4}$$

Für eine Beispielkonfiguration mit vier ALUs und zwei MLA-Einheiten benötigt das Register-File also zehn Lese-Ports. Die Adressen der Lese-Ports des Register-Files sind kombinatorisch mit den Ausgängen verknüpft, so dass die Ergebnisse direkt an den Datenpfad des jeweiligen VLIW-Slots angelegt werden. Analog wird bei dem Condition-Register vorgegangen. Die ausgelesenen Konditionen werden in die Instruktions-Decode-Stufe zur Evaluierung der Sprungvorhersage zurückgeführt und Löschung von bedingten (und nicht auszuführenden) Instruktionen aus den Pipelineregistern genutzt.

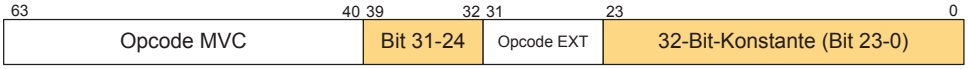


Abbildung 4.7.: Der Sonderfall der MVC-Instruktion belegt 64 Bit im Instruktionsspeicher

Die MVC Anweisung. Die *Move Constant (MVC)*-Anweisung (vgl. Abbildung 4.7) stellt einen Sonderfall im Instruktionssatz dar, da sie mit 64 Bit als einzige Instruktion *zwei* Ausführungskanäle belegt. Während es mit einer *MOV*-Instruktion nur möglich ist eine 8-Bit-Konstante zu laden, lädt *MVC* in einem Takt eine 32-Bit-Konstante in ein Register. Dafür nutzt sie eine Pseudoinstruktion (*Extension (EXT)*), die in den nächsten Instruktionsspeicherplatz ausgelagert ist. Die eigentliche Instruktion enthält die höherwertigsten 8 Bits der Konstanten, die Pseudoinstruktion die 24 niederwertigsten Bits. Befindet sich die *MVC*-Instruktion in VLIW-Slot i, so wird der Operand *EXT*-Instruktion in der Register-Read-Pipelinestufe von VLIW-Slot

$i + 1$ über einen Multiplexer auf den Datenpfad des Ausführungskanal der *MVC*-Instruktion $(i + 1)$ gelegt. Für eine n-fach parallele VLIW-Architektur impliziert diese Bedingung, dass *MVC* Instruktionen vom Compiler nur in VLIW-Slot 1 bis $n - 1$ platziert werden dürfen. In der Execute-Pipelinestufe wird später über die ALU die Gesamtkonstante durch eine Verkettung der beiden Werte bestimmt. Für den Fall, dass nur ein Ausführungskanal verfügbar ist, ist daher eine Sonderbehandlung für diese Instruktion erforderlich. Hier gibt es zwei mögliche Lösungsansätze:

1. *Einlesen in zwei Takten:*
   Wird eine *MVC*-Instruktion dekodiert, muss der Prozessor angehalten werden. Der erste Teil des Operanden wird in einem Register zwischengespeichert. Im nachfolgenden Takt kann der zweite Teil des Operanden gelesen und in der Pipeline weiterverarbeitet werden.

2. *Vergrößerung der Speicherschnittstelle:*
   Für nur einen Ausführungskanal ist mindestens eine Speicherschnittstelle mit der Größe von 32 Bits erforderlich. Verwendet man stattdessen eine Speicherschnittstelle mit 64 Bits, kann auch die *MVC* Anweisung in demselben Taktzyklus verarbeitet werden. In diesem Fall kann das Alignment-Register mit einer Breite von 128 Bit, wie bei der 2-Slot-Konfiguration, genutzt werden. Die Modulo-Operation zur Bestimmung der Startadresse der aktuellen Instruktion innerhalb des Alignment-Registers vereinfacht sich zu einer Exklusiv-Oder (XOR)-Operation.

Hinsichtlich des Ressourcenbedarfs innerhalb des Prozessors sind beide Lösungen in etwa gleichwertig. Für die erste Implementierung wird ein zusätzliches 24-Bit-Register für die Zwischenspeicherung des *EXT*-Operanden benötigt. Für die zweite Implementierung kommen 64 Bits für das größere Alignment-Register hinzu. Beides ist allerdings im Vergleich zur Gesamtlogik des Prozessorkerns vernachlässigbar.

Für die in diesem Kapitel vorgestellte VLIW-Architektur wurde die zweite Lösung implementiert, da diese die *MVC*-Anweisung in einem Takt verarbeitet. Zusätzliche Funktionalität zum Anhalten der nachfolgenden Pipelinestufen ist nicht notwendig. Die 24 Bits der *EXT*-Instruktion werden als zusätzliches Signal aus der Decode-Pipelinestufe herausgeführt und über ein Register um einen Takt verzögert an die Register-Read-Pipelinestufe übergeben. Um diesen Spezialfall möglichst transparent in die Prozessorentwickelung zu integrieren, ist die Implementierung auch in Prozessorkonfigurationen mit mehr als einem VLIW-Slot enthalten. Da es in VHDL nicht möglich ist, Eingangs- und Ausgangs-Ports in Abhängigkeit von einer generischen Konstante zu definieren, wird dieses zusätzliche Signal mit der Konstanten 0 vorbelegt, falls mehr als eine Ausführungseinheit vorhanden sind. Die Synthesewerkzeuge entfernen dieses Signal im Rahmen der Optimierung. Das zusätzliche Register

und der Multiplexer in der Register-Read-Pipelinestufe wird nur für den Fall einer Ausführungseinheit erzeugt, so dass keine redundante Hardware erzeugt wird.

4.3.7. Execute-Pipelinestufe

In der Ausführungsstufe werden die dekodierte Operation und die zugehörigen Operanden an die Funktionseinheiten angelegt. In jedem VLIW-Slot ist eine ALU vorhanden ($N_{\text{ALU}} = N_{\text{VLIW-Slots}}$). Die wichtigsten Komponenten innerhalb der Ausführungsstufe sind auf Synopsys *DesignWare-Komponenten* zurückgeführt [190]. Synopsys stellt mit der DesignWare-Bibliothek Strukturbeschreibungen von Basiskomponenten, wie beispielsweise Addierer oder Multiplizierer, zur Verfügung.

4.3.7.1. Arithmetisch logische Einheit (ALU)

Eine Vielzahl der Operationen wird in der arithmetisch-logischen Einheit verarbeitet. Als Eingangssignale werden an die ALU unter anderem die in der Decode-Stufe dekodierte Instruktion, für die jeweilige Instruktion benötigte Steuersignale und zwei Operanden angelegt. Die ALU erzeugt daraus kombinatorisch ein Ausgangssignal als Resultat. Für einige Instruktionen, wie *Pre-Modify-* und *Post-Modify*-Ladeoperationen, wird ein zweites Ergebnis erzeugt. *Pre-Modify-* und *Post-Modify*-Ladeoperationen ermöglichen das Laden eines Datenwortes aus dem Datenspeicher in ein Register und die gleichzeitige Änderung eines zweiten Registers, in welchem die Speicheradresse für den Ladevorgang abgelegt wurde. Beispielsweise kann die Summe aus Basisadresse und Offset in das zweite Register zurückgeschrieben werden. Hierdurch lassen sich sequentielle Zugriffe auf Speicherblöcke effizient realisieren. Die Präfixe *Pre-* und *Post-* beziehen sich auf die zeitliche Abfolge. Bei *Pre-Modify*-Instruktionen wird zuerst die Zieladresse aus Basisadresse und Offset berechnet und dann der Speicherzugriff durchgeführt. Bei *Post-Modify*-Instruktionen wird von der Basisadresse gelesen, danach die Basisadresse und Offset addiert und das Ergebnis dann in das zweite Zielregister zurückgeschrieben.

Die in Anhang A beschriebenen Instruktionen des an den ARM-Prozessor angelehnten Instruktionssatzes des VLIW-Prozessors basieren in erster Linie auf Additionen bzw. Substraktionen und auf Schiebeoperationen. Hierzu verfügt jede ALU über je eine dedizierte Schiebe- und eine Addier-/Subtrahiereinheit. Das Verhalten der Schiebeeinheit (`DW01_ash` [192]) kann so konfiguriert werden, dass entweder logisch oder arithmetisch geschoben wird. Der Unterschied liegt darin, dass beim arithmetischen Schieben nach rechts das Vorzeichenbit mitgeführt wird. Beim logischen Schieben wird mit Nullen aufgefüllt. Der kombinierte Addier/Subtrahierer (`DW01_addsub` [191]) kann über einen Steuereingang konfiguriert werden.

Aus dem Ergebnis der Operation werden verschiedene Steuersignale berechnet (`Zero`, `Carry`, `Overflow` und `Negative`). Diese werden in einer nachgeschalteten

Komponente zur Auswertung von Bedingungen zur Bestimmung der Condition Flags genutzt. Abbildung 4.8 zeigt die Architektur der ALU.

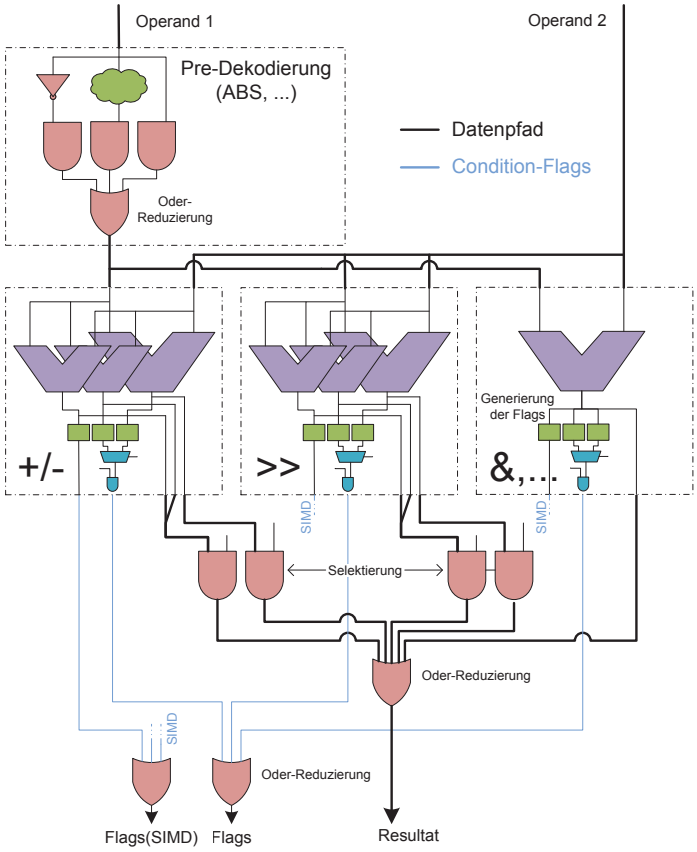


Abbildung 4.8.: Architektur der arithmetischen logischen Einheit (ALU)

MLA- und Division (DIV)-Instruktionen sind als dedizierte Einheiten implementiert. Ihre Anzahl ist weitgehend von der Anzahl der VLIW-Slots unabhängig, jedoch gilt folgende Bedingung:

$$N_{\text{MLA/DIV}} > 1 \qquad \wedge \qquad N_{\text{MLA/DIV}} \leq N_{\text{VLIW-Slots}} \tag{4.5}$$

Das bedeutet, dass mindestens eine MLA- und eine DIV-Einheit vorhanden sein muss. Die maximale Anzahl an MLA-/DIV-Einheiten ist jeweils durch die Anzahl der VLIW-Slots nach oben beschränkt. MLA- oder DIV-Einheiten können exklusiv in einem VLIW-Slot genutzt werden, oder mehreren VLIW-Slots zugeordnet werden. Über eine Konfigurationsmatrix in der VHDL-Implementierung kann festgelegt

werden, welcher VLIW-Slot Zugriff auf welche dedizierten MLA- oder DIV-Einheiten hat. Zwei Beispiele solcher Konfigurationen sind in den Abbildungen 4.9 und 4.10 gezeigt. Die zugehörige Architektur ist in Abbildung 4.11 dargestellt.

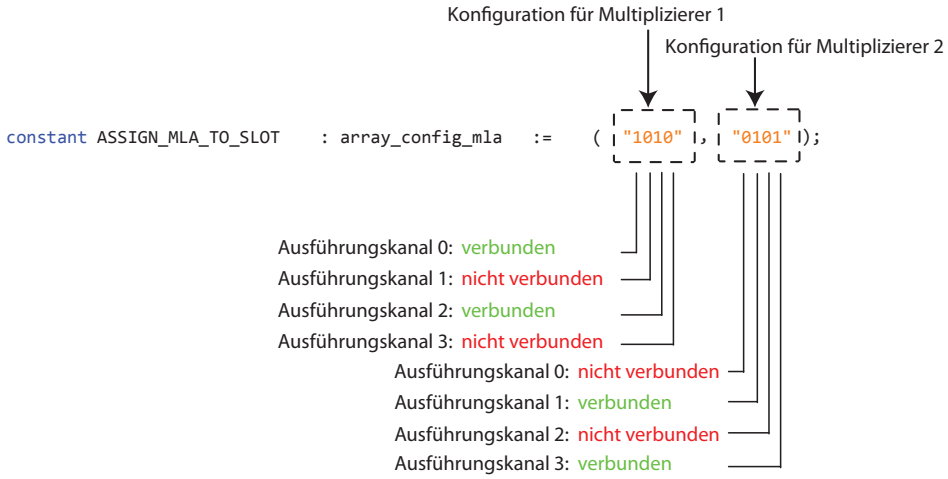


Abbildung 4.9.: Konfiguration der gemeinsamen Ressourcennutzung für die MLA-Instruktionen

Das erste Bit steht für den ersten Ausführungskanal (VLIW-Slot 0), dass n-te Bit für den Letzten in einer n-fach parallelen VLIW-Architektur. Im Beispiel aus Abbildung 4.9 ist eine mögliche Konfiguration für zwei Multiplizierer und vier Ausführungskanäle dargestellt. Der erste Multiplizierer ist hier mit den Ausführungskanälen 0 und 2 verbunden. Der zweite Multiplizierer ist mit den Ausführungskanälen 1 und 3 verbunden. Das Beispiel aus Abbildung 4.10 behandelt eine Zuweisung von zwei Dividierern an vier Ausführungskanäle. Hier ist der erste Dividierer mit dem Ausführungskanal 0 und 2 verbunden. Ausführungskanal 1 belegt den zweiten Dividierer exklusiv. Die Verwendung solcher Konfigurationsvektoren hat den Vorteil, dass zum einen alle Freiheitsgrade für die Zuweisung an die Ausführungskanäle frei genutzt werden können, zum anderen eine solche Matrix relativ einfach direkt im VHDL-Programmquelltext verwendet werden kann. Der VLIW-Compiler muss die in mehreren VLIW-Slots gemeinsam genutzten Ressourcen berücksichtigen und dafür Sorge tragen, dass nicht mehrere MLA- oder mehrere DIV-Instruktionen auf ein und dieselbe dedizierte Einheit verteilt werden.

4.3.7.2. SIMD-Parallelität

Fast alle Operationen der ALU sowie die MLA-Instruktion können in einem *16-Bit-SIMD-Modus* ausgeführt werden. Hierbei werden die Instruktionen auf die jeweils

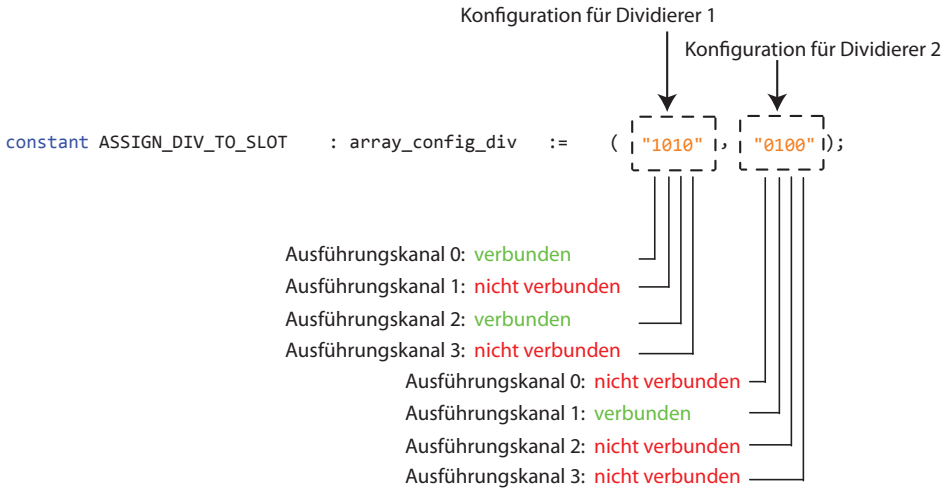


Abbildung 4.10.: Konfiguration der gemeinsamen Ressourcennutzung für die DIV-Instruktionen

oberen und unteren Halbwörter der Operanden getrennt angewendet. Für die Addition bedeutet dieses beispielsweise, dass das Carry-Bit des 16. Bits nicht an die höherwertigen Bits propagiert werden darf. Für die additions- und schiebe-basierten Instruktionen enthält die ALU einen zusätzlichen 16 Bit Addierer und einen 16 Bit arithmetischen-Schieber. Für die Multiplikation sind zwei zusätzliche 16-Bit-Multiplizierer implementiert.

4.3.7.3. Multiply-Accumulate

Zur Ausführung von Multiplikationen stehen zwei dedizierte MLA-Einheiten[8] zur Verfügung. Zusätzlich kann ein Akkumulator auf das Produkt aufaddiert werden ($Y = A \cdot B + C$). Die Multiplizierer stellen die Komponenten mit der größten Latenz innerhalb des VLIW-Prozessors dar und müssen gesondert behandelt werden, um den kritischen Pfad des Gesamtsystems nicht unnötig zu verlängern. Tabelle 4.3 zeigt den Ressourcenbedarf der wichtigsten Komponenten innerhalb der Execute-Pipelinestufe und deren Latenz. Die 16-Bit-Addierer/Subtrahierer sowie arithmetische Schieber sind bezüglich der Latenz vernachlässigbar.

Auch wenn im Instruktionssatz für die MLA-Instruktionen keine nachfolgende Berechnung von Konditionen (Die zusätzliche Latenz für die Berechnung der Konditionen liegt bei maximal 0,16 ns für die Berechnung des *Zero*-Flags.) vorgesehen ist,

[8] Im Folgenden der Einfachheit halber auch nur Multiplizierer (MUL) genannt.

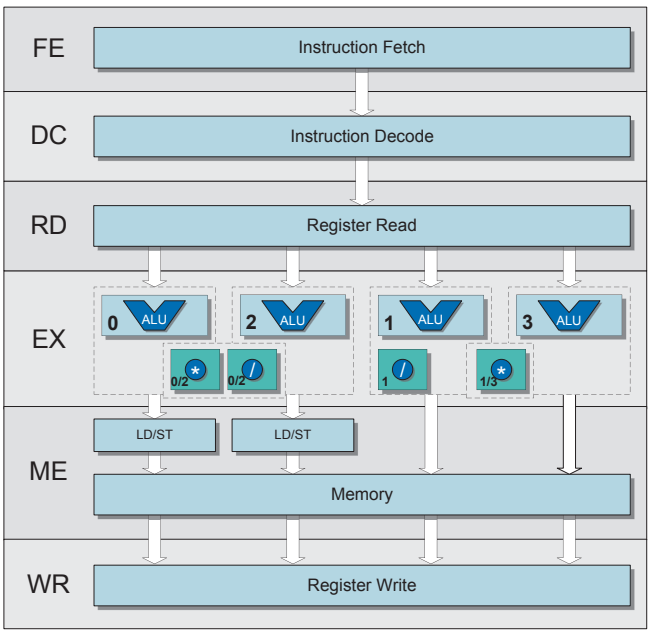


Abbildung 4.11.: Beispielarchitektur für gemeinsam genutzte DIV-/MLA-Ressourcen

verursacht die dedizierte MLA-Einheit mit Abstand die höchste Latenz und würde den kritischen Pfad der Execute-Pipelinestufe (und wie später in Kapitel 5 gezeigt, auch des gesamten Prozessorkerns) bestimmen. Zur Reduzierung und Balancierung der Länge des kritischen Pfades wurde ein manuelles *Re-Timing* der MLA-Einheit, d.h. ein Verschieben von Logik über Registergrenzen hinweg, durchgeführt. Die MLA-Funktionalität kann auch durch eine Kombination eines *partiellen Multiplizierers*, *eines Carry-Save-Addierers* und *eines Volladdierers* implementiert werden. In der Execute-Pipelinestufe ist nur der partielle Multiplizierer implementiert. Der Carry-Save-Addierer und der Volladdierer werden in die nachfolgende Memory-Pipelinestufe verschoben. Der über die DesignWare-Komponente *DW02_multp* realisierte partielle Multiplizierer implementiert einen Multiplizierer, bei dem der abschließende Carry-Propagate-Addierer entfällt. Das Resultat sind zwei 32-Bit-Vektoren im Carry-Save-Format. Das eigentliche Resultat der Multiplikation kann über eine Addition berechnet werden. Um den Akkumulator zu addieren, werden diese beiden Zwischenergebnisse zuvor über einen Carry-Save-Addierer verknüpft und im abschließenden Volladdierer schließlich das Resultat der MLA-Instruktion gebildet. Die Latenz des partiellen Multiplizierers liegt bei 0,93 ns. Die kombinatorische Logik lässt sich hiermit also weitestgehend gleich auf beide Pipelinestufen verteilen und die Latenz liegt in der Größenordnung der restlichen Komponenten der Execute-Pipeli-

Tabelle 4.3.: *Vergleich der MLA-Einheiten zu den wichtigsten Komponenten der ALU in einer 65 nm Standardzellentechnologie von STMicroelectronics*

Komponente	Fläche [μm^2]	Latenz [ns]
Addierer/Subtrahierer[9]	2253	0,45
Arithmetischer Schieber[10]	4085	0,31
16-Bit-Multiplizierer[11]	12853	1,01
Multiplizierer (MLA)[12]	42468	2,21

nestufe. Die Latenz des Carry-Save-Addierers liegt bei 0,11 ns. Die Implementierung der MLA-Instruktion ist in Abbildung 4.12 dargestellt [236].

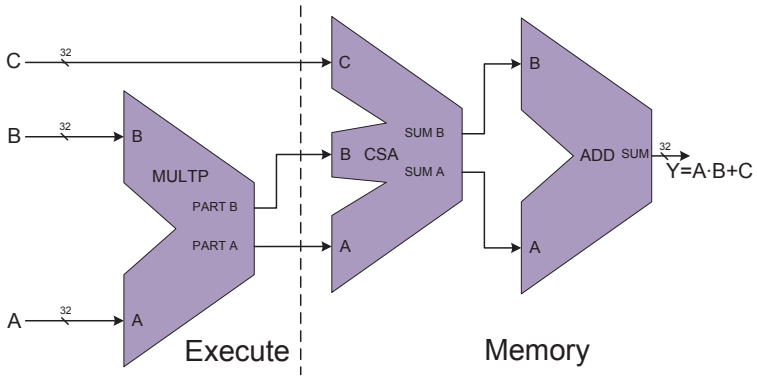


Abbildung 4.12.: Re-Timing der MLA-Einheit durch Partitionierung auf zwei Pipelinestufen

Die Ausdehnung der MLA-Instruktion auf zwei Pipelinestufen führt aber auch zu einer erhöhten Latenz von zwei Taktzyklen. Dieses muss durch den VLIW-Compiler bei der Ablaufplanung (engl. *Scheduling*) berücksichtigt werden. Eine Reduktion der Gesamtperformanz durch das geänderte Scheduling war jedoch nicht messbar. Der Vorteil durch die erhöhte Taktfrequenz überwiegt hier also.

[9] Synopsys DesignWare-Komponente DW01_addsub [191]

[10] Synopsys DesignWare-Komponente DW01_ash [192]

[11] Synopsys DesignWare-Komponente DW02_mult [194]

[12] Synopsys DesignWare-Komponente DW02_mac [193]

Tabelle 4.4.: *Vergleich der Anzahl an Taktzyklen für eine ganzzahlige 32-Bit-Division bei verschiedenen Prozessorarchitekturen*

Architektur	Taktzyklen
VLIW (Divisionsschritteinheit)	**32**
VLIW (Software)	500
Intel Itanium	35
N-Core	37
Infineon X-GOLD™ SDR 20	224

4.3.7.4. Division

Für ganzzahlige Divisions-Operationen wird kein dedizierter Dividierer eingesetzt. Die kombinatorische Verzögerungszeit eines Dividierers aus der Synopsys DesignWare-Bibliothek *DW_div* beträgt in der 65 nm Standardzellentechnologie von STMicroelectronics 17,56 ns und liegt damit weit über der notwendigen Latenz für eine Zielfrequenz von 400 MHz. Bei einer Divisionseinheit (*DW_div_pipe*) mit zehn Pipelinestufen verkürzt sich zwar die Verzögerungszeit der kombinatorischen Logik auf 2,56 ns, aber auch dieser Wert ist nach ersten Abschätzungen viel zu groß. Des Weiteren würde sich die Verarbeitungslatenz der Divisionsinstruktion auf zehn Taktzyklen erhöhen. Aufgrund des hohen Ressourcenbedarfs, wird auf komplette Dividierer in eingebetteten Systemen üblicherweise verzichtet. Eine einfache Software-Implementierungen der Division (Quotient und Rest) auf einem RISC-Prozessor[13] benötigt knapp 500 Taktzyklen. Daher kommen in Prozessorarchitekturen zur effizienten Division Hardware-Einheiten zur Unterstützung der Division zum Einsatz. Der N-Core-Prozessor [118, 120, 89] nutzt beispielsweise die ALU zur Unterstützung der Divisionsberechnung. Eine Division benötigt bis zu 37 Taktzyklen. Beim Infineon X-GOLD SDR 20 [102, 29, 169] ermöglicht eine Divisionsschritt-Instruktion[14] eine 32-Bit-Division in 224 Taktzyklen. Beim Intel Itanium benötigt die Berechnung des Quotienten einer 32-Bit-Division 35 Taktzyklen (Rest: 39 Taktzyklen) [103]. Tabelle 4.4 zeigt einen Vergleich der Anzahl an Taktzyklen für eine ganzzahlige 32-Bit-Division bei den verschiedenen Prozessorarchitkturen.

Bei der in diesem Kapitel vorgestellten VLIW-Architektur kommt daher eine *Divisionsschritteinheit* (der Einfachheit halber im Folgenden auch nur *Dividierer* genannt) zum Einsatz, die im Gegensatz zu reinen Software-Realisierung, die Division unter-

[13] Als Referenz wurde eine 1-Slot-Implementierung der VLIW-Architektur angenommen.

[14] Für Divisionsschritt-Instruktion werden beim SDR-20 Standardoperationen der ALU verschaltet. Zusätzliche Hardware ist nicht notwendig.

stützet. Die Struktur der so genannten Divisionsschritt-Einheit ist in Abbildung 4.13 dargestellt.

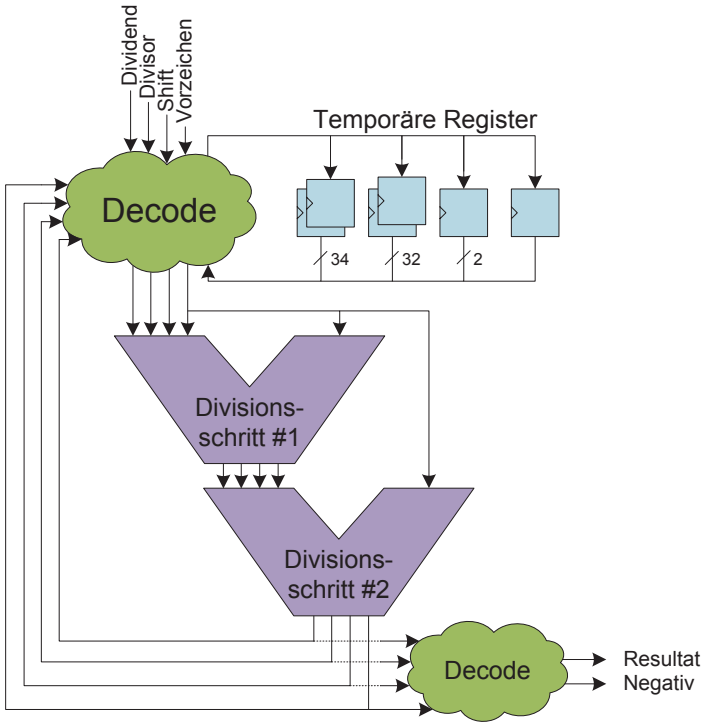


Abbildung 4.13.: Divisionsschritt-Einheit

Die Divisionsschritteinheit arbeitet nach dem in [157, Appendix H] beschriebenen Prinzip der *Radix-2-Division*. Vier Instruktionen unterstützen die Division:

- Division Initialization (DVI)
- Division Step (DVS)
- Division Quotient (DVQ)
- Division Remainder (DVR)

Eine Division wird über die DVI-Instruktion initialisiert. In einem iterativen Prozess werden über die DVS-Instruktion in zwei Taktzyklen je zwei Bits des Ergebnisses berechnet. Zwischenergebnisse und das Endergebnis werden in internen Registern der Divisionsschritteinheit gespeichert. Eine 32-Bit-Division benötigt durch dieses

Verfahren 32 Takte. Sowohl Quotient als auch Rest werden parallel berechnet. Über die Instruktionen DVQ und DVR können der Quotient und der Rest ausgelesen werden. Ähnlich wie die ALU erzeugt die Divisionsschritteinheit Steuersignale, aus denen Konditionen für die bedingte Ausführung berechnet werden können. Die minimal mögliche Latenz der kombinatorischen Logik der Divisionsschritteinheit beträgt 0,71 ns und liegt nicht im kritischen Pfad der gesamten Architektur. Der Register-zu-Register-Pfad innerhalb der Divisionsschritteinheit beträgt 1,19 ns und liegt somit auch weit unter den in Kapitel 5 erzielten minimal erreichbaren Ergebnissen für die gesamte Prozessorarchitektur. Der Ressourcenbedarf der Divisionsschritteinheit im Vergleich zu einem vollständigen Dividierer ist in Tabelle 4.5 aufgeführt. Die Ergebnisse der Funktionseinheiten ALU und Divisionsschritteinheit werden noch in der Execute-Pipelinestufe über einen Multiplexer zusammengefasst. Das Resultat der MLA-Einheit wird in der Memory-Stufe in den Datenpfad eingefügt. Die Verschaltung der Funktionseinheiten innerhalb eines VLIW-Slots ist in Abbildung 4.14 dargestellt.

Tabelle 4.5.: *Vergleich des Ressourcenbedarfs für die Divisionsschritteinheit im Vergleich zu einem vollständigen Dividierer in einer 65 nm Standardzellentechnologie von STMicroelectronics*

Komponente	Fläche [μm^2]	Latenz [ns]
Divisionsschritteinheit[15]	**8049**	**1,19**
Dividierer[16]	40634	17,56
Dividierer (10 Pipelinestufen)[17]	34988	2,56

Speicherzugriffe. Zum Zugriff auf den Datenspeicher wird die Speicheradresse aus einer *Basisadresse* und einem *Offset* berechnet. Die hierfür notwendige Addition wird in der ALU durchgeführt. In einer der ALU nachgeschalteten Einheit (LD/ST-Einheit) wird die berechnete Adresse an den Datenspeicher angelegt. Bei *Schreibzugriffen (Store (ST))* wird zusätzlich das zu schreibende Datenwort übergeben.

4.3.8. Memory-Access

Bei *Leseanfragen (Load (LD))* auf den Datenspeicher werden die Daten in der Memory-Pipelinestufe auf den Datenpfad des jeweiligen VLIW-Slots gelegt. Auch das Ergebnis

[15] Eigene Implementierung

[16] Synopsys DesignWare-Komponente DW_div [195]

[17] Synopsys DesignWare-Komponente DW_div_pipe [196]

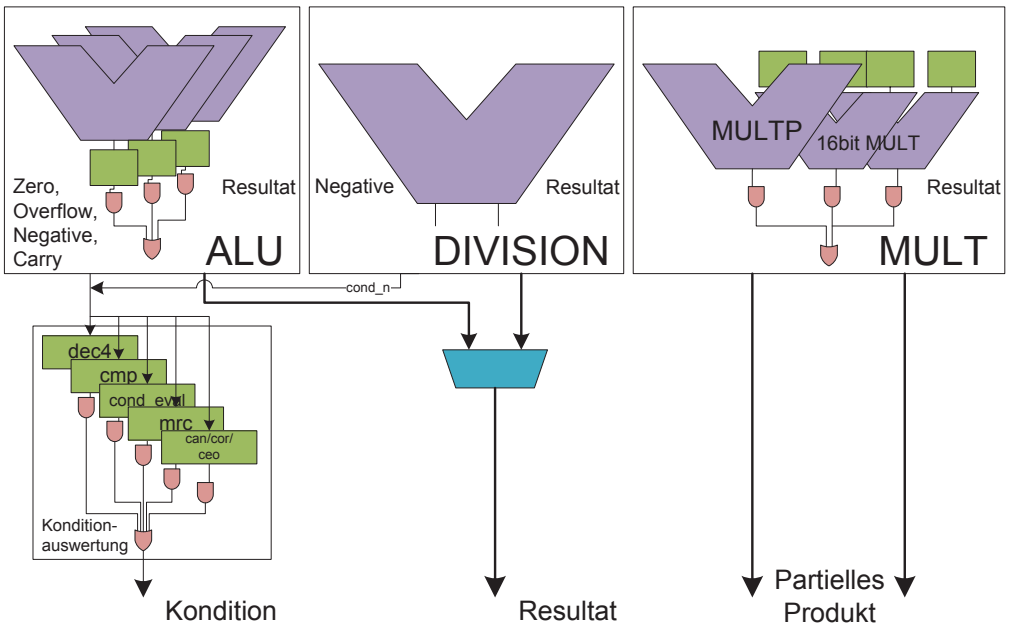


Abbildung 4.14.: Verschaltung der Funktionseinheiten innerhalb eines VLIW-Slots

der MLA-Instruktionen steht in dieser Pipelinestufe zu Verfügung. LD- und MLA-Instruktionen haben im Gegensatz zu allen anderen Instruktionen eine Latenz von 2 Taktzyklen. Da es keinen Datenpfad vom Datenspeicher oder der MLA-Einheit zu den Condition-Registern gibt, können die Condition-Register bereits in der Memory-Pipelinestufe geschrieben werden.

4.3.9. Register-Write

Die Ergebnisse der Funktionseinheiten und die Daten, die aus dem Datenspeicher gelesen wurden, werden in der Register-Write-Pipelinestufe in das Register-File zurückgeschrieben. Jede ALU erzeugt für Leseoperationen aus dem Datenspeicher maximal zwei Ergebnisse: Das gelesene Datenwort und die modifizierte Speicheradresse für *Pre-* und *Post-Modify*-Instruktionen. Für alle anderen Operationen wird nur ein Ergebnis erzeugt. Die benötigte Anzahl der Schreib-Ports des Register-Files ergibt sich also zu:

$$N_{\text{Register-Schreib-Ports}} = N_{\text{LD/ST}} + N_{\text{ALU}} \tag{4.6}$$

Für eine Beispielkonfiguration vier ALUs und zwei LD/ST-Einheiten benötigt das Register-File also sechs Schreib-Ports.

4.4. Der Pipeline-Bypass

Aufgrund der mehrstufigen Pipelinestruktur des in diesem Kapitel beschriebenen Prozessors kann es zu *Pipeline-Konflikten* kommen. Solche Pipelinekonflikte werden durch Daten- und Steuerflussabhängigkeiten sowie durch die Nichtverfügbarkeit von Ressourcen hervorgerufen. Werden solche Abhängigkeiten nicht erkannt und behandelt, kann es zu fehlerhaften Datenzuweisungen kommen. Drei Arten von Pipelinekonflikten werden unterschieden [157]:

1. *Steuerflusskonflikte (Control Hazards)* treten auf, wenn die Zieladresse des nächsten auszuführenden Befehls noch nicht berechnet ist oder wenn im Falle eines bedingten Sprungs noch nicht feststeht, ob dieser ausgeführt wird.

2. *Struktur-/Ressourcenkonflikte (Structural Hazards)* treten auf, wenn mehr Pipelinestufen oder VLIW-Slots eine Ressource benötigen, als diese Zugriffe erlaubt.

3. *Datenkonflikte (Data Hazards)* treten auf, wenn der Operand für einen Befehl in der Pipeline noch nicht verfügbar ist oder wenn das Resultat einer Operation noch nicht in ein Register bzw. einen Speicherplatz geschrieben werden kann.

Die in dieser Arbeit entwickelte VLIW-Architektur behandelt mögliche *Steuerflusskonflikte* durch eine in der Decode-Pipelinestufe implementierte *Sprungvorhersage*. Dazu gehört ein Mechanismus zum Leeren der Pipeline, der Ergebnisse von Instruktionen, deren Ausführung fälschlicherweise bereits begonnen hat, korrigiert *(Pipeline-Flushing)*.

Ressourcenkonflikte, z. B. beim gleichzeitigen Zugriff von zwei VLIW-Slots auf eine Funktionseinheit (z. B. eine MLA- oder DIV-Einheit), werden durch eine Priorisierung gelöst: Generell wird bei gleichzeitigem Zugriff die Instruktion im niederwertigeren VLIW-Slot priorisiert. Die andere Instruktion wird verworfen. Im Allgemeinen muss ein solcher Zugriff durch den Compiler verhindert werden. Ressourcenkonflikte beim gleichzeitigen Zugriff auf das Register-File oder das Condition-Register wird durch eine Oder-Verknüpfung der zu schreibenden Daten gelöst. Gleichzeitiger (Schreib-)Zugriff auf ein und dieselbe Adresse des Datenspeichers muss durch den VLIW-Compiler verhindert werden.

Eine große Rolle bei der Untersuchung der Ressourceneffizienz des Prozessors spielen *Datenkonflikte*. Diese treten bei der gewählten VLIW-Architektur auf, wenn Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen bestehen. Liest eine Instruktion in der Register-Read-Stufe einen Operanden aus dem Register-File, während eine vorangegangene Instruktion, die sich zeitgleich in der Execute-, Memory-, oder Register-Write-Pipelinestufe befindet, das Ergebnis ihrer Instruktion noch nicht in das Register-File zurückgeschrieben hat, so tritt eine Datenabhängigkeit auf [231].

Datenkonflikte des Prozessors können vom VLIW-Compiler durch das Einfügen von Leeroperationen (No Operation, NOP) vermieden werden. Im folgenden Beispiel muss der Compiler drei Warteinstruktionen einfügen, da in der Register-Read-Stufe ein Operand der Additions-Instruktion aus dem Daten-Register R1 gelesen werden muss, welches durch die vorangegangene Move-Instruktion erst in der Write-Register-Pipelinestufe beschrieben wird.

```
c7 mov r1, 1 lsl 0 ;Initialisierung r1=1
c7 nop             ;Leerinstruktion
c7 nop             ;Leerinstruktion
c7 nop             ;Leerinstruktion
c7 add r0, r1, r2  ;addiere r1 und r2, speichere Summe in r0
```

Die Behandlung von Datenkonflikten im Voraus durch den Compiler würde zum einen die Codegröße durch die zusätzlichen NOP-Instruktionen vergrößern, zum anderen müsste die Bestimmung möglicher Datenkonflikte konservativ ausfallen, da im Vorfeld nicht entschieden werden kann, ob bedingte Instruktionen wirklich ausgeführt werden und somit eine Datenabhängigkeit entsteht.

Eine weitere Möglichkeit zur Vermeidung von Datenkonflikten ist das sogenannte *Interlocking* (auch *Pipeline-Stalling* genannt) [33]. Hierbei werden alle Pipelinestufen, die nicht zur Fertigstellung der vorangegangenen Instruktion benötigt werden, durch das Einfügen eines Strafzyklusses angehalten. Im Falle der in diesem Kapitel beschriebenen sechstufigen Pipelinestruktur müssten beim Auftreten eines Datenkonflikts die Instruction-Fetch, die Decode- und die Register-Read-Pipelinestufe angehalten werden, um die vorzeitige Ausführung der nachfolgenden Instruktion in der Execute-Pipelinestufe zu verhindern. Sowohl Interlocking als auch das Einfügen von Leerinstruktionen führt zu einer Erhöhung der Verarbeitungszeit. Da die Ergebnisse der meisten Instruktionen aber schon am Ende der Execute-Pipelinestufe[18] vollständig berechnet sind, sind Verzögerungen nicht unbedingt notwendig. Aus diesem Grund wurden zur Vermeidung der Datenkonflikte eine Vielzahl von Bypass- beziehungsweise Forwarding-Mechanismen implementiert, die einen direkten und verzögerungsfreien Zugriff auf die Rechenergebnisse der Vorgängerinstruktionen aus den Pipelinestufen Execute bis Register-Write ermöglichen.

4.4.1. Der Register-Bypass

Um die zuvor beschriebenen *Datenkonflikte und Wartezyklen* bei aufeinander folgenden Instruktionen zu vermeiden, wurde die Register-Read-Stufe durch einen Register-

[18] Für die MLA und LD-Instruktionen am Ende der Memory-Pipelinestufe

Bypass erweitert. Dieser Bypass ermöglicht es dem System, die bereits gültigen Rechenergebnisse der Verarbeitungseinheiten (VE) direkt aus den unteren Pipelinestufen in die Register-Read-Pipelinestufe zurückzuführen um sie den nachfolgenden Instruktionen als Operanden zur Verfügung stellen zu können. Hierdurch können die im obigen Beispiel benötigten Warteinstruktionen ersatzlos entfallen und somit drei Verarbeitungstakte eingespart werden. Der Register-Bypass behandelt sowohl Zugriffe auf das Register-File als auch Zugriffe auf das Condition-Register. Der implementierte Instruktionssatz erlaubt den Datenaustausch zwischen diesen beiden Registern. Hierdurch ergeben sich insgesamt vier verschiedene Datenpfade zwischen dem Register-File und dem Condition-Register (vgl. Abbildung 4.15).

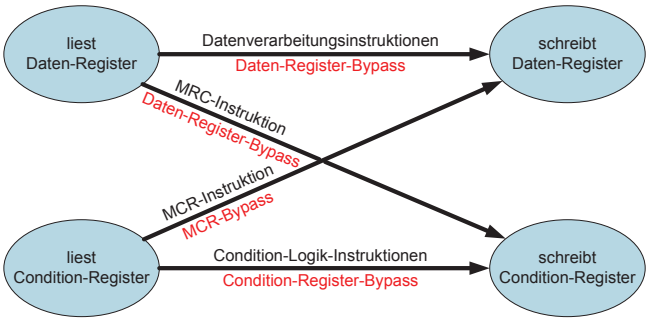


Abbildung 4.15.: Die verschiedenen Datenpfade zwischen Register-File und Condition-Register und die dafür benötigten Bypass-Systeme[19]

Da Datenkonflikte nicht nur bei direkt aufeinander folgenden Instruktionen auftreten können, sondern auch Datenabhängigkeiten zwischen der ersten und dritten oder der ersten und vierten Instruktionsgruppe zu Konflikten führen, benötigt der Daten-Register-Bypass Zugriff auf die Execute-, die Memory-Access- und die Register-Write-Pipelinestufe (vgl. Abbildung 4.16).

Die Ergebnisse der Berechnung der Konditionen stehen bereits in der Memory-Pipelinestufe zur Verfügung. Pfade für die Konditionen aus der Register-Write-Pipelinestufe zurück in die Register-Read-Pipelinestufe können daher entfallen.

Die Auswahl und Aktivierung eines Bypasspfades wird durch die Analyse der übergebenen Registeradressen gesteuert. Die Multiplexer prüfen, ob die Lese-Registeradressen der Instruktionen in der Register-Read-Pipelinestufe mit den Schreib-Registeradressen der Instruktionen in den nachfolgenden Pipelinestufen übereinstimmt.

[19] *Move-Register-to-Condition (MRC)*- und *Move-Condition-to-Register (MCR)*-Instruktionen erlauben das Kopieren von Inhalten vom Register-File zum Condition-Register und umgekehrt. Dieses ist z. B. bei Funktionsaufrufen notwendig. Condition-Logic-Instruktionen erlauben boolesche Operationen zur direkten Modifikation des Condition-Registers

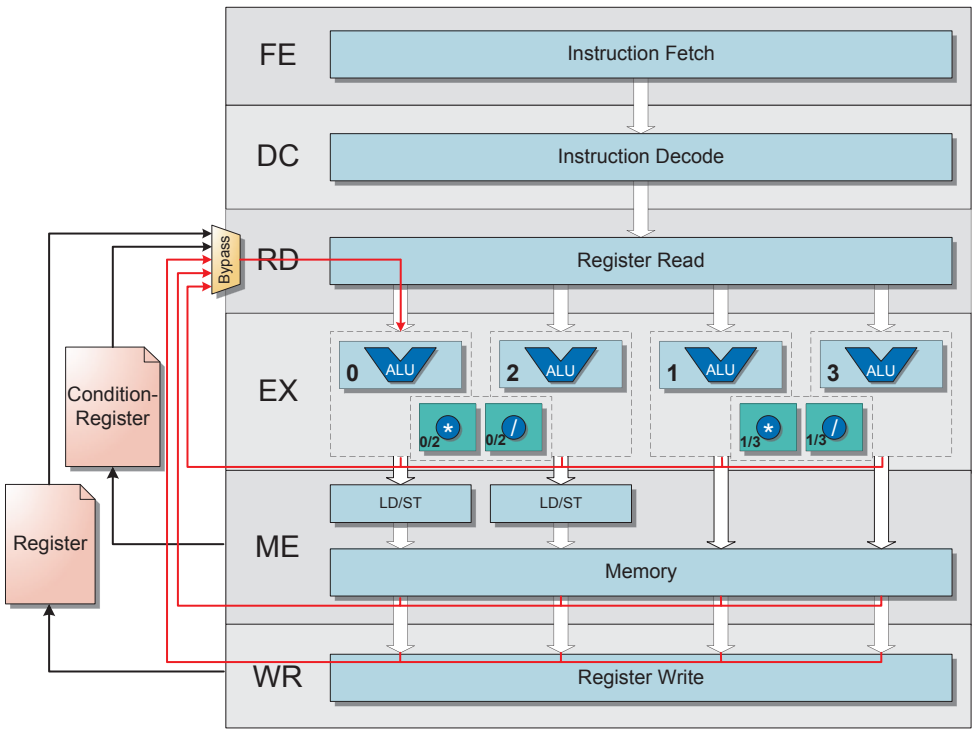


Abbildung 4.16.: Rückführung der Rechenergebnisse durch den Register-Bypass

Stimmen die Adressen überein, werden in den nachfolgenden Pipelinestufen Instruktionen verarbeitet, die den Inhalt der betreffenden Register aktualisieren werden. In diesem Fall übernimmt das Bypass-System anstelle der veralteten Registerinhalte die zu schreibenden Werte aus den nachfolgenden Stufen. Falls die Registeradressen in mehreren unteren Pipelinestufen übereinstimmen, wird das aktuellste Rechenergebnis (also die zeitlich gesehen letzte Instruktion) priorisiert. Für jeden in der Register-Read-Pipelinestufe zu lesenden Operanden muss ein Register-Bypass implementiert werden. Hieraus ergibt sich die Anzahl der zu implementierenden Register-Bypass-Komponenten für eine n-fach parallele VLIW-Architektur zu:

$$N_{\text{Registerbypässe}} = 3 \cdot n \tag{4.7}$$

Für eine Beispielkonfiguration von 4 VLIW-Slots ergeben sich also zwölf benötigte Register-Bypass-Komponenten. Da das Scheduling des VLIW-Compilers alle

Instruktionen, die nicht an exklusive Ressourcen gebunden sind[20], in beliebige VLIW-Slots platzieren kann, muss der Pipeline-Bypass nicht nur Lese-Registeradressen mit Schreib-Registeradressen innerhalb *eines* VLIW-Slots, sondern über *alle* VLIW-Slots hinweg vergleichen. Dieses führt zu einer stark steigenden Komplexität des Register-Bypasses mit der Anzahl der VLIW-Slots. Tabelle 4.6 und Gleichung 4.11 zeigen die Anzahl der benötigten Bits, die über einen Register-Bypass-Pfad zurückgeführt werden müssen.

Tabelle 4.6.: *Art und Breite der verschiedenen Bypass-Pfade einer Register-Bypass-Komponente*

Register-Bypass-Pfad	**Datenbreite [Bit]**
Daten-Register-Bypass(EX)	$N_{\text{Register-Schreib-Ports}} \cdot 32$
Daten-Register-Bypass(ME)	$N_{\text{Register-Schreib-Ports}} \cdot 32$
Daten-Register-Bypass(WR)	$N_{\text{Register-Schreib-Ports}} \cdot 32$
Condition-Register-Bypass(EX)	$N_{\text{ALUs}} \cdot 16$
Condition-Register-Bypass(ME)	$N_{\text{ALUs}} \cdot 16$
MCR-Bypass(EX)	$N_{\text{ALUs}} \cdot 16$
MCR-Bypass(ME)	$N_{\text{ALUs}} \cdot 16$

$$
\begin{aligned}
N_{\text{Gesamtbitbreite}} &= 3 \cdot 32 \cdot N_{\text{Register-Schreib-Ports}} + 2 \cdot 16 \cdot N_{\text{ALUs}} + 2 \cdot 16 \cdot N_{\text{ALUs}} && (4.8)\\
&= 96 \cdot N_{\text{Register-Schreib-Ports}} + 64 \cdot N_{\text{ALUs}} && (4.9)\\
&= 96 \cdot (N_{\text{LD/ST}} + N_{\text{ALU}}) + 64 \cdot N_{\text{ALUs}} && (4.10)\\
&= 96 \cdot N_{\text{LD/ST}} + 160 \cdot N_{\text{ALU}} && (4.11)
\end{aligned}
$$

Für eine Beispielkonfiguration einer 4-fach-parallelen VLIW-Architektur mit zwei LD/ST-Einheiten ergibt sich die Wortbreite eines Register-Bypasses zu 832 Bit. Die Gesamtwortbreite für alle zwölf Bypass-Komponenten summiert sich so auf 9984 Bit. Der große Einfluss der Bypass-Pfade auf den Ressourcenbedarf lässt sich hier bereits erkennen und wird in den Kapiteln 5 und 6 genauer untersucht.

Einige Datenkonflikte, wie z. B. beim direkten Zugriff auf das Ergebnis einer LD- oder MLA-Instruktion im nachfolgenden Taktzyklus, können nicht durch Register-Bypass-Pfade aufgelöst werden. Hierfür müssen wie zuvor beschrieben, Teile der Pipelinestufen angehalten werden.

[20] Beispielsweise können MLA- und DIV-Instruktionen nur in VLIW-Slots mit entsprechender Einheit platziert werden

4.4.2. Der MLA-Bypass

Wie bereits in Abschnitt 4.3.7.3 beschrieben wurde, erstreckt sich die Verarbeitung der *Multiply-Accumulate-Instruktionen* über zwei Pipelinestufen. Die Multiplikation erfolgt in zwei dedizierten Einheiten (Partieller Multiplizierer und Carry-Save-Addierer) der Execute-Pipelinestufe und der Memory-Pipelinestufe. Die Verbindung der Multiplikationsergebnisse mit dem Akkumulator erfolgt erst in der Memory-Access-Pipelinestufe. Falls eine direkte Abhängigkeit mit Latenz Eins zwischen einer arithmetischen oder logischen Operation und einer vorangegangenen MLA-Instruktion besteht, so kann diese Abhängigkeit nicht durch den Register-Bypass aufgelöst werden. Die nachfolgende Instruktion muss verzögert werden. Viele Algorithmen der digitalen Signalverarbeitung, wie beispielsweise der in Kapitel 5.2.4 beschriebene Algorithmus zur diskreten Faltung, basieren auf der Berechnung von Skalarprodukten. Skalarprodukte werden durch eine Additionsreihe von Multiplikationen dargestellt [35].

$$Skalarprodukt : \vec{x} \cdot \vec{y} = x_1 \cdot y_1 + x_2 \cdot y_2 + \ \ldots \ + x_n \cdot y_n \tag{4.12}$$

Eine solcher Algorithmus kann durch direkt hintereinander ausgeführte MLA-Instruktionen abgebildet werden. Hierbei bildet jeweils das Ergebnis einer MLA-Instruktion den Akkumulator der folgenden MLA-Berechnung. Da hier eine direkte Abhängigkeit mit Latenz Eins zwischen den beiden MLA-Instruktionen besteht, müsste die Prozessor-Pipeline nach jedem Berechungsschritt bis zur Register-Read-Pipelinestufe für einen Takt angehalten werden. Um die Verarbeitung dieser Instruktionskombinationen zu beschleunigen, wurde in der in diesem Kapitel vorgestellten VLIW-Architektur ein MLA-Bypass implementiert. Da das Aufaddieren des Akkumulators der nachfolgenden MLA-Instruktion erst in der Memory-Access-Stufe erfolgt und die Multiplikation in der Execute-Pipelinestufe davon nicht beeinflusst wird, kann das Endergebnis der vorangegangenen MLA-Instruktion über den MLA-Bypass ohne Verzögerung in den Carry-Save-Addierer zurückgeführt werden (siehe Abbildung 4.17).

Im Vergleich zum Register-Bypass werden für dieses Bypass-System eine deutlich geringere Anzahl an Bypass-Pfaden benötigt. Der MLA-Bypass wird nur für den dritten Operanden der MLA-Einheiten benötigt. Für jeden VLIW-Slot, in den MLA-Instruktionen platziert werden können[21], ist eine Bypass-Instanz notwendig. Jede Instanz muss die Ergebnisse vorangegangener Instruktionen aller n VLIW-Slots zurückführen, so dass sich die Anzahl der MLA-Bypass-Pfade auf $N_{\text{MLA-Slots}} \cdot n \cdot 32$ summiert. Für eine Beispielkonfiguration einer vierfach parallelen VLIW-Architektur,

[21] Dieses ist unabhängig von der Anzahl der MLA-Einheiten, da über die zuvor beschriebene Ressourcenteilung der Datenpfad beliebiger VLIW-Slots mit den Ressourcen der MLA-Einheiten verbunden werden kann.

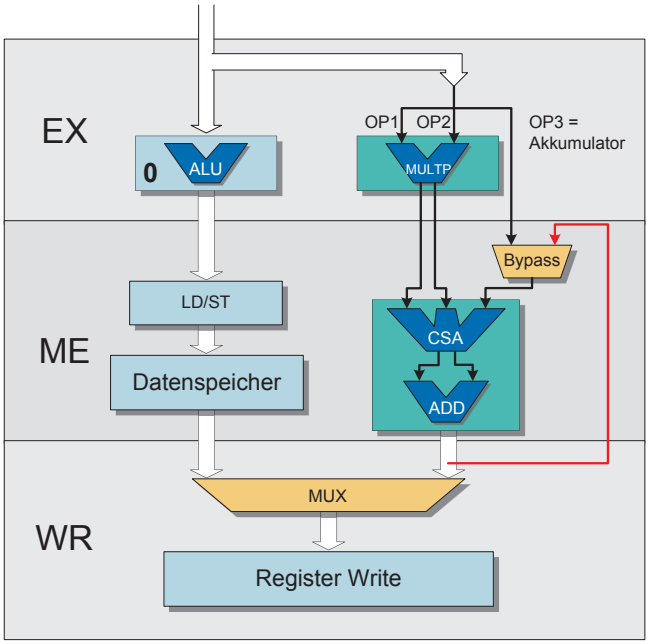


Abbildung 4.17.: MLA-Bypass zur direkten Akkumulatorrückführung

in der in jedem VLIW-Slot MLA-Instruktionen ausgeführt werden können, summiert sich die Anzahl der Bypass-Pfade auf 512 Bits, was etwa 6 % der Anzahl der Register-Bypass-Pfade entspricht.

4.4.3. Der Kontroll-Bypass

Um eine bedingte Ausführung von Instruktionen und Sprüngen zu ermöglichen, werden diesen Instruktionen durch den Compiler einzelne Bits des Condition-Registers zugeordnet. Falls die zugehörigen Bits nicht gesetzt sind, werden die Instruktionen beziehungsweise die hierdurch abgebildeten Sprünge nicht ausgeführt (vgl. Abschnitt 4.3.2). Da die Dekodierung der Instruktionen bereits in der Instruction-Decode-Stufe stattfindet, die Inhalte der Condition-Register aber erst in der Register-Read-Pipelinestufe ausgelesen werden, müssen Instruktionen und insbesondere Sprünge, die bereits in der Prozessor-Pipeline verarbeitet werden, nachträglich rückgängig gemacht werden (auch *Instruction Canceling* oder *Pipeline Flushing* genannt).

Wie beim Register-Bypass kann es bei den Condition-Registern ebenfalls zu Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen kommen. Hierfür wurde in der in diesem Kapitel vorgestellten VLIW-Architektur ein Kontroll-Bypass implementiert (vgl. Abbildung 4.18). Durch Vergleich der Leseadresse in der Register-

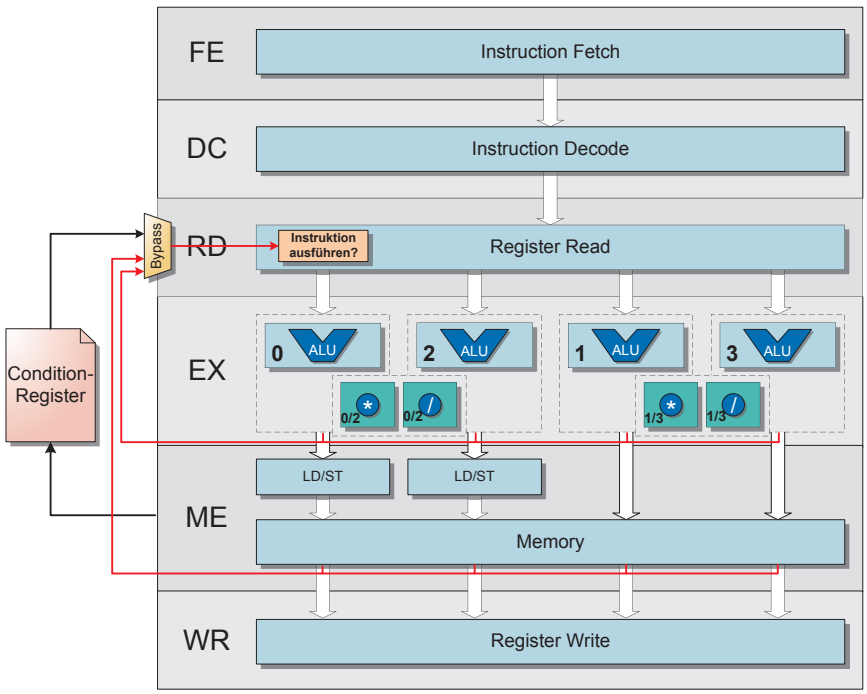


Abbildung 4.18.: Architektur des Kontroll-Bypasses

Read-Pipelinestufe mit den Schreibadressen innerhalb der Execute- und Memory-Access-Pipelinestufen bestimmt der Bypass, ob bereits berechnete Konditionen aus diesen Pipelinestufen in die Register-Read-Pipelinestufe zurückgeführt werden müssen.

Im Unterschied zur Implementierung des Condition-Register-Bypasses (als Bestandteil des Register-Bypasses) wird die gewählte Kondition nicht an einen Operanden der Verarbeitungseinheiten weitergeleitet, sondern zur Steuerung des Abbruchs und Löschen von Instruktionen aus der Prozessor-Pipeline genutzt. In jedem VLIW-Slot kann im SIMD-Modus auf zwei unterschiedliche Condition-Registereinträge zugegriffen werden. Für eine n-fach parallele Architektur werden daher $2 \cdot n$ Kontroll-Bypass-Instanzen benötigt. Jeder Kontroll-Bypass muss Zwischenergebnisse der Konditionen aus der Execute- und der Memory-Pipelinestufen zurückführen können. Aus diesem Grund ergibt sich eine Gesamtwortbreite von $2 \cdot 16 \cdot 4 \cdot n$ Bit. Für eine vierfach parallele Architektur ergibt sich diese Breite zu 512 Bit. Die Gesamtwortbreite für alle Bypass-Pfade der beschriebenen Beispielkonfiguration des VLIW-Prozessors beträgt somit $(9984 + 512 + 512)$ Bit $= 11008$ Bit.

4.5. Zusammenfassung

Dieses Kapitel hat eine modularen VLIW-Architektur beschrieben, die als Grundlage für eine weiterführende Entwurfsraumexploration dient. Ziel war es, mit dieser Architektur einen möglichst großen Entwurfsraum bezüglich der funktionalen Parallelität abzudecken. Die Variation der generischen Parameter erlaubt die Konfiguration der Anzahl der Funktionseinheiten, wie ALUs, Multiplizierer und Dividierer sowie die Anzahl der Dekodierer und LD/ST-Einheiten. Die Konfiguration dieser Komponenten bedingt durch die Anzahl der VLIW-Slots direkt die Datenparallelität und hat somit Einfluss auf weitere Kernkomponenten, wie die Pipelineregister, das Register-File, den Pipeline-Bypass oder das Instruktions-Alignment. Über eine geeignete Wahl der Parameter kann der Prozessor neben einer VLIW-Architektur auch als skalare RISC-Architektur mit nur einem Datenpfad konfiguriert werden. Abbildung 4.19 zeigt das Blockschaltbild der Pipelinestruktur.

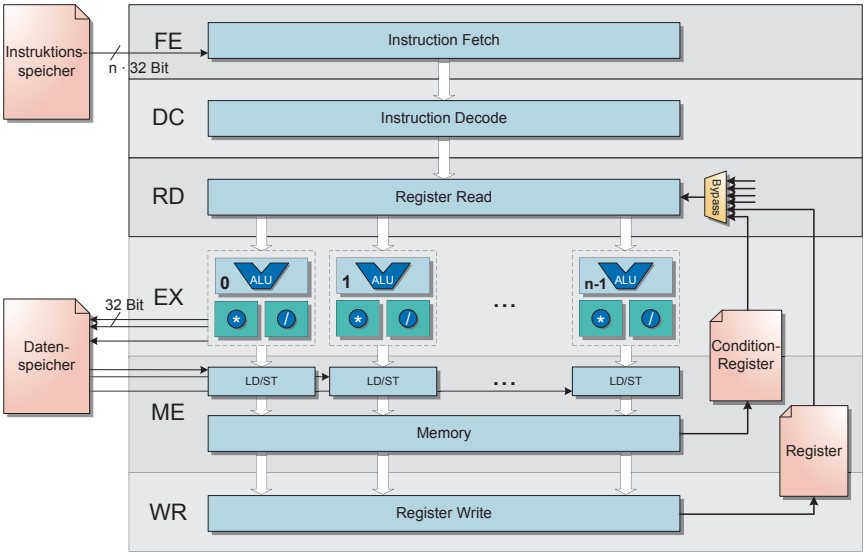


Abbildung 4.19.: Pipeline-Architektur des modularen VLIW-Prozessors

Die folgende Auflistung fasst die wichtigsten Eigenschaften der modularen VLIW-Architektur zusammen [234]:

- Modulare VLIW-Architektur
- Harvard-Computerarchitektur
- Little-Endian-Byte-Reihenfolge

- Sechsstufige Pipelinestruktur
- 31 universell einsetzbare Register
- Zwei 8-Bit-Condition-Register für die bedingte Ausführung von Instruktionen
- Instruktionskompression durch Stop-Bits
- Parametrisierbares Alignment-Register als *Level-0-Cache* und zur Ausrichtung der Instruktionen
- Sprungvorhersage
- Instruktionssatz an ARM angelehnt (Binärkompatibilität möglich)
- 41 Basisinstrutionen, 15 SIMD-Instruktionen, orthogonaler Instuktionssatz erleichtert die Dekodierung
- Laden von 32-Bit-Konstanten in einem Taktzyklus
- *Pre-* und *Post-Modify*-Instruktionen
- Einsatz heterogener ALU-Implementierungen möglich
- 16-Bit-SIMD-Modus
- 1,6 GOP/s Durchsatz (3,2 GOP/s im SIMD-Modus) bei 400 MHz
- Fast duchgängig 1-Zyklen-Instruktionen mit 1 Taktzyklus Latenz
- *Multiply-Accumulate*-Instruktionen mit 2 Taktzyklen Latenz
- Divisionsschritt-Instruktionen (32 Taktzyklen Latenz pro Division)
- Vollständiger Pipeline-Bypass

Die modulare Architektur dient als Grundlage für die im folgenden Kapitel durchgeführte Entwurfsraumexploration. Hier wird der Ressourcenbedarf verschiedener Prozessorkonfigurationen in Bezug zur intrinsischen Parallelität von Algorithmen aus verschiedenen Anwendungsbereichen gesetzt. Das Ergebnis ist ein anwendungsspezifischer Vergleich der Ressourceneffizienz verschiedener Hardware-Konfigurationen.

5. Entwurfsraumexploration des CoreVA-VLIW-Prozessors

Dieses Kapitel zeigt die *Entwurfsraumexploration* der in Kapitel 4 vorgestellten modularen VLIW-Architektur in Hinblick auf ihre *funktionale Parallelität*. In Abhängigkeit der Anzahl an verfügbaren Funktionseinheiten werden verschiedene Anwendungen mit dem in Kapitel 3 vorgestellten Entwurfsablauf auf die Architektur abgebildet. Als Anwendungsszenario wird für die entwickelte VLIW-Architektur in erster Linie der drahtlose Einsatz angesehen. Hierdurch ergeben sich besondere Anforderungen an die Ressourceneffizienz, insbesondere an die Energieeffizienz. Sogenanntes *Software-defined Radio* stellt hier eine flexible Alternative zu bisherigen Ansätzen zur drahtlosen Kommunikation dar. Abschnitt 5.1 erläutert den Begriff Software-defined Radio und zeigt die besonderen Anforderungen an die Architektur auf. Abschnitt 5.2 führt kurz die zur Entwurfsraumexploration genutzten Anwendungen ein. In Abschnitt 5.3 wird die Skalierung der Ausführungszeiten der Anwendungen mit der Parallelität untersucht. Abschnitt 5.4 untersucht den Ressourcenbedarf verschiedener Architekturvarianten. Zur Bewertung der Ressourceneffizienz wird in Abschnitt 5.5 das in Abschnitt 3.4 eingeführte Maß angewandt. Aus diesen Ergebnissen wird eine für die untersuchte Anwendungsklasse geeignete Konfiguration (die CoreVA-Architektur) ausgewählt (vgl. Abschnitt 5.6). Abschnitt 5.7 vergleicht die CoreVA-Architektur mit kommerziellen Prozessoren. Abschnitt 5.8 fasst die Erkenntnisse aus diesem Kapitel zusammen.

5.1. Anwendungsszenario Software-defined Radio

Drahtlose Kommunikation hält mehr und mehr Einzug in unser tägliches Leben. Immer komplexere Übertragungsverfahren (WLAN, UMTS[1], LTE[2] etc.) erlauben stetig *steigende Datenraten* bei *geringer Latenz*. Dieses ermöglicht neue Anwendungen, wie hochauflösendes Fernsehen (HDTV[3]), Videokonferenzen oder 3D-Online-Spiele. Grundsätzlich muss ein mobiles System alle diese Standards unterstützen, wofür

[1] Universal Mobile Telecommunications System

[2] Long Term Evolution

[3] High Definition Television

bisher *heterogenen Hardware-Plattformen* mit dedizierten Beschleunigerkomponenten verwendet wurden. Im realen Betrieb werden aber meist nur ein oder wenige Übertragungsverfahren *zeitgleich* genutzt. Die vielen verschiedenen Bausteine tragen jedoch alle zum Flächenbedarf und damit zu den Kosten sowie zur Verlustleistung des Gesamtsystems bei, unabhängig davon, ob sie gerade genutzt werden, oder nicht. Aus diesem Grund setzen moderne Systeme zunehmend auf flexible Architekturen, welche auf hochperformanten aber gleichzeitig universellen Prozessoren basieren [133, 132, 126][241, 238, 234]. Dieses „Software-defined Radio“ genannte Prinzip erlaubt einen Austausch von Anwendungen zur Laufzeit. Im Idealfall wird die Architektur optimal ausgenutzt. Zusätzlich erlaubt eine programmierbare Architektur auch die Implementierung neuer Standards. Hochsprachen, wie C/C++, erleichtern die Entwicklung der Anwendungen und sind weitgehend hardwareunabhängig. Die steigende Integrationsrate moderner Fertigungsprozesse erlaubt die Realisierung von solchen hochperformanten Prozessoren bei gleichzeitig niedriger Leistungsaufnahme. Neben der Ausführung von drahtlosen Algorithmen können universelle Prozessoren zudem auch zur Ausführung von Betriebssystemen und sonstigen Anwendungen, wie Multimedia-Anwendungen, genutzt werden. Zusätzliche, feingranular rekonfigurierbare Bausteine, wie FPGAs, erlauben die Implementierung von hoch parallelisierbaren Signalverarbeitungskomponenten.

5.1.1. Stand der Technik von SDR-basierten Architekturen

Bereits Anfang der 1990er Jahre beschreibt Joe Mitola in [143, 144] das Potential einer sogenannten *Software Radio Architektur* und die Notwendigkeit offener, standardisierter Architekturschnittstellen, wie beispielsweise *Common Object Resource Broker (CORBA)* [149]. Ein Software-Radio ist definiert als eine Menge von DSP-Primitiven, ein Metasystem zur Kombination der Primitiven in Kommunikationssystemkomponenten (Sender, Empfänger etc.) und eine Menge an Prozessoren zur Ausführung der Echtzeitkommunikation. Joe Mitola unterteilt die Verarbeitung eines Kanalstroms in sechs Segmente (vgl. Abbildung 5.1):

- Antenna Segment
- RF Conversion Segment
- IF Processing Segment
- Baseband Processing Segment
- Bitstream Segment
- Source Segment

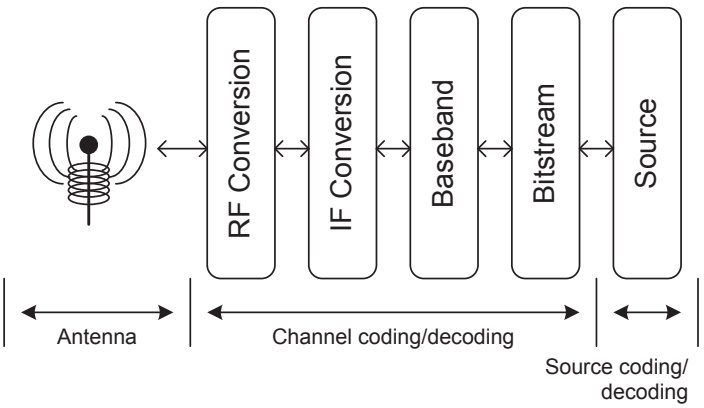


Abbildung 5.1.: Unterteilung eines Kanalstroms in sechs Segmente nach [143]

Eine Antenne (*Antenna Segment*) kann beispielsweise ein Teil eines Beamforming[4]-Netzwerks zur Reduktion von Interferenzen sein oder zur Implementierung von Raummultiplexverfahren (Space division multiple access (SDMA)) genutzt werden. Im *RF*[5] *Conversion Segment* werden Leistungsgenerierung, Vorverstärkung und die Konvertierung von RF-Signalen in und aus einer Zwischenfrequenz (Intermediate Frequency (IF)) durchgeführt, um eine spätere breitbandige A/D/A-Wandlung zu ermöglichen. Sei B_{IF} die Bandbreite der Zwischenfrequenz, ist in der Praxis eine leichte Überabtastung notwendig:

$$f_S = 2.5 B_{IF} \tag{5.1}$$

Das *IF Processing Segment* konvertiert die zu übertragenden und empfangenden Signale vom Basisband zur Zwischenfrequenz und umgekehrt. Außerdem enthält dieses Segment digitale Filter, Frequenzumsetzer sowie Komponente zum Filtern, zur Dezimierung oder zum Spreizen der Basisbandsignale. Im *Baseband Processing Segment* wird die Basisbandverarbeitung, wie zum Beispiel Trelliskodierung oder Soft-Decision[6]-Parameterabschätzung, durchgeführt [210, 211]. Im *Bitstream Segment* werden die Bitströme verschiedener Nutzer gebündelt. Auf den gebündelten Bitstrom werden Vorwärtsfehlerkorrekturverfahren oder Block- bzw. Faltungscodes angewendet, Verschränkung durchgeführt, Automatische Wiederholungsanfragen (Automatic Repeat Requests (ARQs)) erkannt und beantwortet, Datenrahmen ausgerichtet, Bitmuster

[4] dt. *Strahlformung*. Im deutschen Sprachgebrauch ist allerdings die Verwendung des englischen Begriffs *Beamforming* gebräuchlich.

[5] Radio Frequency

[6] Weiche Entscheidung

aufgefüllt oder Daten verschlüsselt. Das *Source Segment* letztendlich unterscheidet zwischen dem mobilen Endgerät und der Basisstation. Weiterhin präsentiert [143] eine Abschätzung des benötigten Ressourcenbedarfs der einzelnen Segmente. Der größte Rechenbedarf (>1 Milliarden Operationen pro Sekunde) entfällt auf das *IF Processing Segment*. Joe Mitola nimmt als Beispiel eine Bandbreite von 10 MHz bei einer Mehrfachabtastung von 2,5 an. Mit einem Rechenaufwand von ca. 100 Operationen pro Sample ergibt sich ein benötigte Leistungsfähigkeit von bis zu 2,5 Milliarden Operationen pro Sekunde. Um diese Leistungsfähigkeit bereitzustellen, schlägt Mitola vor, das *IF Processing Element* auf heterogene Multiprozessorsysteme oder dedizierte Hardwarekomponenten abzubilden.

In [15] wird ein einstellbarer Software-defined Radio Empfänger präsentiert, der Frequenzen von 800 MHz bis 5000 MHz abdeckt. Der Chip ist in einer 90 nm Technologie implementiert und hat einen Flächenbedarf von 7 mm^2. Der Empfänger ist in der Lage, Mobilfunkstandards von GSM[7] bis hin zu IEEE[8] 802.11g-WLAN-Algorithmen (maximal 40 MHz Abtastrate) zu verarbeiten.

Eine programmierbare Basisbandplattform für Mobilfunk- und WLAN-Standards wird in [29] beschrieben. Die Architektur beinhaltet vier SIMD-DSP-Kerne, dedizierte programmierbare Prozessoren zur Kanal En-/Decodierung sowie für Filteroperationen und einen ARM-Prozessor zur Ausführung des Protokollstacks. Um die Effizienz ihres Ansatzes zu zeigen wurde ein IEEE-802.11b-Algorithmus auf das System abgebildet. Das System arbeitet bei einer Taktfrequenz von 300 MHz.

In [179, 180] wird ein SDR-Prototyp vorgestellt, der auf einer Multiprozessor-Architektur basiert. Verschiedene Mobilfunkalgorithmen, wie z. B. WLAN nach IEEE 802.11b, wurden implementiert. Neben Pre-/Post-Prozessoren mit programmierbarer Datenrate kommen vier TI TMS320C6201 DSPs (200 MHz) und ein PowerPC (400 MHz) zum Einsatz, um Bandbreiten von bis zu 20 MHz zu verarbeiten. Hierbei konnten Datenraten von bis zu 2 MBit/s (IEEE 802.11b, DQPSK[9]) erreicht werden.

In [93] schlägt Intel ihre rekonfigurierbare Kommunikationsarchitektur bestehend aus einer vollvermaschten Netzwerktopologie von eingebetteten Prozessoren zur Verarbeitung von WLAN-Standards der IEEE-802.11-Reihe vor. Diese Elemente sind in einer CMOS-Technologie implementiert und direkt an analoge Front-Ends gekoppelt.

Alle Ansätze beschreiben den immensen Rechenbedarf der für die Basisbandverarbeitung notwendig ist. Des Weiteren wird gezeigt, dass die Erweiterung des Entwurfsraums um Software-basierte Ansätze bei der Verarbeitung von Funkstandards neue, effiziente Verfahren zur schnellen und genauen Evaluierung von SDR-basierten Anwendungen erfordert.

[7] Global System for Mobile Communication

[8] Institute of Electrical and Electronics Engineers

[9] Differential Quaternary Phase Shift Keying

Die im folgenden Abschnitt zur Analyse vorgestellten Anwendungen stellen eine Auswahl typischer Algorithmen aus dem Bereich der digitalen Signalverarbeitung und dem Bereich der Multimediaanwendungen dar. Hierbei wurden bewusst Algorithmen gewählt, die unterschiedlich gut an die Architektur des CoreVA-Prozessors angepasst sind und deshalb bei der Verarbeitung unterschiedliche Parallelitätsgrade aufweisen.

5.2. Analysierte Anwendungen

Zur Analyse und Bewertung der Ressourceneffizienz wurde eine *heterogene Menge von Algorithmen aus verschiedenen Anwendungsklassen* ausgewählt. Diese beinhalten sowohl Basisband-relevante Algorithmen sowie Algorithmen oberhalb des Medienzugriffs beispielsweise aus der Anwendungsschicht. Eine heterogene Gruppe aus Anwendungen sowohl aus dem Basisbandbereich als auch Benutzeranwendungen und der Wechsel zwischen diesen repräsentiert die Grundidee von Software-defined Radio [238]. Die Anwendungen wurden mit dem in Kapitel 3 vorgestellten Entwurfsablauf für verschiedene Prozessorkonfigurationen kompiliert und mit Hilfe des zyklenakkuraten Instruktionssatzsimulator simuliert. Die analysierten Algorithmen sind in Tabelle 5.1 aufgeführt und in die Kategorien *Benchmark*, *Codierung*, *Fehlererkennung*, *Signalverarbeitung*, *Kryptographie* und *Bildverarbeitung* klassifiziert. In diesem Abschnitt wird die Funktionsweise der Anwendungen kurz beschrieben und ihre Relevanz bezüglich des Einsatzgebietes in eingebetteten Systemen erläutert.

5.2.1. Dhrystone

Das synthetische Benchmarkprogramm Dhrystone [220, 221] wurde 1984 von Reinhold P. Weicker entwickelt, um die Leistungsfähigkeit von Prozessoren und Compilern zu vergleichen. Bei VLIW-Systemen kann dieses Programm zusätzlich eingesetzt werden, um die *Güte der Parallelverarbeitung* verschiedener Compiler-Versionen zu bestimmen. Der Dhrystone-Code besteht im Wesentlichen aus einfacher Ganzzahlarithmetik, aus String-Operationen, logischen Entscheidungen und Speicherzugriffen, wodurch die meisten universellen Computer-Anwendungen abgedeckt werden. Im Vergleich zu anderen Benchmarkprogrammen enthält dieser Programmcode nur Ganzzahloperationen. Die C-Programmcodeversion dieses Benchmarks wird als industrieller Standard eingesetzt. Das Testergebnis bildet die mittlere Zeit, die der Prozessor zur Verarbeitung der einzelnen Iterationen benötigt. Hierbei wird die Dhrystone-Leistung in *Dhrystone Million Instructions Per Second (DMIPS)* oder *Dhrystone-MIPS/MHz* angegeben und anschließend mit einer Referenzmaschine verglichen.

Tabelle 5.1.: *Referenzalgorithmen zur Bestimmung der Ressourceneffizienz des VLIW-Prozessorsystems*

Name	Kategorie	Kurzbeschreibung
Dhrystone	Benchmark	Synthetischer Benchmark zur Bewertung von Prozessor und Compiler
Coremark	Benchmark	Synthetischer Benchmark zur Bewertung von Prozessor und Compiler
Viterbi	Codierung	Erkennung und Korrektur von Bitfehlern
Faltung	Signalverarbeitung	Bitweise Faltungsoperation
FFT	Signalverarbeitung	Digitales Verfahren zur Berechnung einer Fourier-Transformation
IEEE 802.11b	Signalverarbeitung	Teilimplementierung eines WLAN Senders
CRC	Fehlererkennung	Zyklische Redundanzprüfung
ECC	Kryptographie	Asymmetrische Krytographie
AES	Kryptographie	Schlüsselberechnung eines symmetrischen Verschlüsslungsverfahrens
SNOW	Kryptographie	Stromverschlüsselung (SNOW 3G)
SATD	Bildverarbeitung	Videokompression des H.264-Standards beziehungsweise des MPEG-4/AVC-Standards

5.2.2. Coremark

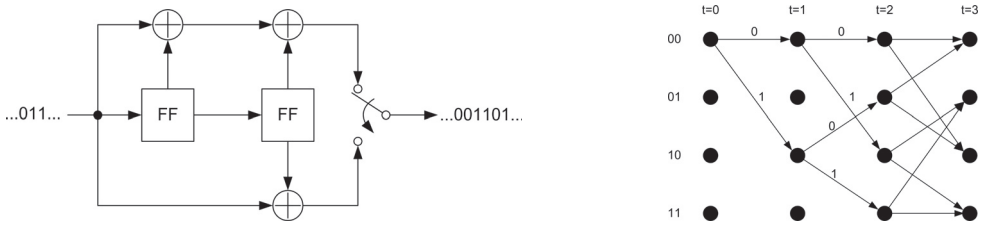
Coremark ist ein vom *Embedded Microprocessor Benchmark Consortium (EEMBC)* [204, 160, 123, 111, 91] herausgegebener synthetischer Benchmark. EEMBC ist eine gemeinnützige Organisation, die 1997 gegründet wurde. Coremark wurde 2009 von Shay Gal-On herausgegeben und ist, ähnlich wie Dhrystone (vgl. Abschnitt 5.2.1), ein kostenloser Benchmark zur Bewertung von Prozessorkernen. In jedem Iterationsschritt führt der Benchmark folgende Algorithmen durch: Verarbeitung von Listen, Matrixmanipulationen, Operationen auf Zustandsautomaten und zyklische Redundanzprüfungen (vgl. Abschnitt 5.2.7). Der Vergleich der Performanz von verschiedenen Architekturen erfolgt über die Angabe von Coremark-Iterationen pro Sekunde (I/s) oder von Coremark-Iterationen pro Sekunde und Megahertz ($I/s/f$). Im Gegensatz zu Dhrystone verwendet Coremark keine synthetischen Algorithmen, sondern Programmausschnitte aus realen Anwendungen.

5.2.3. Viterbi

Der Viterbi-Algorithmus ist eine von Andrew Viterbi entwickelte Vorschrift zur Dekodierung von Faltungscodes [217]. Diese Codes ermöglichen es, Redundanzen in Bitströme einzuarbeiten, welche im Falle eines Bitfehlers die Rekonstruktion der ursprünglichen Sequenz erlauben. Der Viterbi Algorithmus ermöglicht dieses bei vertretbarem Aufwand und wird in vielen Bereichen eingesetzt, wo eine robuste

Kanalkodierung nötig ist. Auch bei der Analyse von Gensequenzen und in der computergestützten Spracherkennung findet er Anwendung.

Ohne eine effektive Fehlererkennung und Korrektur wäre in der mobilen Kommunikation keine zuverlässige Datenübertragung möglich. Eine Art der Kodierung ist die Verwendung von Blockcodes. Hierbei werden die Bits einer bestimmten Blockgröße m zusammengefasst und durch ein festgelegten Codewort der Länge n ersetzt, wobei $n > m$ gilt. Die zusätzlichen Bits enthalten Informationen, z. B. ob die Anzahl von Einsen gerade oder ungerade ist (*Paritätsbit*). Diese Informationen können dann zur Fehlererkennung oder sogar Fehlerkorrektur genutzt werden. Für den Fall, dass das empfangene Codewort nicht im verwendeten Codealphabet bekannten ist versagt die Dekodierung. Es handelt sich hierbei um eine sogenannte Hard-Decision-Dekodierung. Ein oft in der Literatur zu findendes Beispiel ist in der Abbildung 5.2a zu sehen [77]. In Hardware lässt sich ein Kodierer sehr leicht durch die Verwendung eines Schieberegisters oder eines Zustandsautomaten realisieren. Die Schaltung übernimmt die aktuellen Bits am Eingang und leitet sie mit den Registerbits verknüpft zum Ausgang weiter. Als Mealy-Automat dargestellt, bilden die möglichen Werte der Register die Zustände, die Ausgabe ist vom aktuellen Zustand und dem Eingangssignal abhängig, und der nächste Zustand ergibt sich aus dem Einangssignal und dem alten Zustand. Visualisiert man die Faltungskodierung mit Hilfe eines endlichen Automaten werden zwar die Abhängigkeiten zwischen den einzelnen Zuständen dargestellt, auf die zeitliche Abfolge lassen sich dabei allerdings keine Rückschlüsse ziehen. Diese werden erst deutlich bei der Verwendung eines Trellisdiagramms. Diese Diagrammform erweitert das Zustandsdiagramm des Automatens gewissermaßen um die zeitliche Dimension. Jeder Knotenpunkt einer Spalte (vgl. Abbildung 5.2b) stellt dabei einen möglichen Zustand des Zustandsautomaten dar, eine Verbindung zur nächsten Spalte steht für einen möglichen Übergang. Die meisten Diagramme beginnen in einem definierten Zustand und kehren zum Ende wieder in diesen zurück. Meist handelt es sich dabei um den Zustand an dem alle Speicherbits den Wert "0" haben.



(a) Faltungskodierer **(b)** Trellis-Diagramm

Abbildung 5.2.: Aufbau eines Faltungskodierers und Beispiel eines Trellis-Diagramms

Die Dekodierung erfolgt dann nicht durch das Ersetzen ganzer Codewörter, durch festgelegte Blöcke, wie es bei den Hard-Decision-Dekodierern der Blockcodes der Fall ist, sondern es werden sogenannte Soft-Decision-Verfahren verwendet. Ein Beispiel für solch ein Verfahren ist der Viterbi-Algorithmus. Dieser betrachtet die Übergänge zwischen den Zuständen und vergleicht die daraus emittierten Bits mit den tatsächlich empfangenen Bit-Sequenzen. Er berechnet den Hamming-Abstand zwischen ihnen bewertet so jeden Übergang mit seinen individuellen Weg-Kosten. Für jeden Knoten wird ein Vorgänger gespeichert und der Weg, welcher bis zu ihm zurückgelegt wurde. Für den Fall, dass der Knoten bereits über einen anderen Pfad erreicht wurde, vergleicht der Algorithmus die benötigten Weg-Kosten und speichert nur den Vorgänger mit den geringeren Kosten. Bei gleichen Kosten wird nach einer beliebigen festen Vorschrift einer, zum Beispiel der vom niederwertigen Zustand, ausgewählt. Der am Ende überlebende Pfad stimmt mit dem ursprünglichen, bei der Kodierung durchlaufenen, am wahrscheinlichsten überein. So lässt sich aus ihm die wahrscheinlichste gesendete Codefolge rekonstruieren.

5.2.4. Faltung

Eine Faltung (engl. *Convolution*) ist ein mathematisches Modell in Form des Faltungsoperators. Dieser gewichtet, entsprechend seiner Vorschrift, eine Funktion mit einer sogenannten *Filterfunktion* und findet Anwendung bei der Beschreibung zahlreicher physikalischer Vorgänge. Allgemein gilt für den Faltungsoperator:

$$\begin{aligned} y\left[t\right] &= h\left[t\right] * x\left[t\right] \\ &= \int_{\tau=-\infty}^{\infty} h\left[\tau\right]\ x\left[t-\tau\right] \\ &= \int_{\tau=-\infty}^{\infty} h\left[\tau\right]\ x\left[-\left(t+\tau\right)\right] \end{aligned}$$

Die diskrete Faltung findet ihre Anwendung in der digitalen Signalverarbeitung und ist definiert durch den diskreten Faltungsoperator.

$$\begin{aligned} y\left[n\right] &= h\left[n\right] * x\left[n\right] \\ &= \sum_{k=0}^{M-1} h\left[k\right]\ x\left[n-k\right] \\ &= \sum_{k=0}^{M-1} h\left[k\right]\ x\left[-\left(n+k\right)\right] \end{aligned}$$

$x[n]$ bezeichnet hier das digitalisierte (zeitdiskrete) Eingangssignal, $h[n]$ die zeitdiskrete Impulsantwort und $y[n]$ das zeitdiskrete Ausgangssignal eines linear zeitinvarianten Systems (LZI) [138]. Die diskrete Faltung ist eine der wichtigesten Operationen in der digitalen Signalverarbeitung. Durch eine Faltung ist es beispielsweise möglich,

die Signalverarbeitung eines LZI-Systems vollständig durch seine Impulsantwort zu beschreiben.

5.2.5. Schnelle Fouriertransformation (FFT)

Die schnelle Fouriertransformation (Fast Fourier Transformation (FFT)) ist ein 1965 von James Cooley und John W. Turkey entwickeltes Verfahren zur schnellen Berechnung der diskreten Fouriertransformation [53].

Es gilt:

$$f_k = \sum_{n=0}^{2N-1} x_n e^{-j\frac{2\pi n}{2N}k} \quad k = 0, ..., 2N-1$$

f_k bezeichnet hier das finite Fourierspektrum der diskretisierten Zeitfunktion x_n (mit $2N$ Abtastungen). Die diskrete Fouriertransformation dient dem Lösen der zeitkontinuierlichen Fouriertransformation eines zeitdiskreten, periodischen Signals und ist wesentliches Instrument der Signalverarbeitung. Mit Hilfe der Fouriertransformation (zeitkontinuierlich) der diskreten Fouriertransformation (zeitdiskret) und deren Implementierung FFT, wurde es erstmals möglich eine Reihe von vorher nur per Analog-Bausteinen in Hardware ausführbaren Signalmanipulationen auf Digitalrechner auszulagern. So konnten z. B. Filterbänke fast vollständig durch digitale Verarbeitungen abgelöst werden. Der *Cooley and Turkey* Algorithmus, also die Radix-2-FFT, ist ein klassisches „Teile-und-Herrsche-Verfahren“, welches rekursiv implementiert ist. Hierbei wird die zu berechnende Diskrete Fourier-Transformation (DFT) über $2N$ Punkte in gerade ($f_0^{'} \dots f_{n-1}^{'}$) und ungerade ($f_0^{''} \dots f_{n-1}^{''}$) Anteile zu je N Punkten zerlegt. Daraus ergibt sich

$$\begin{aligned} f_k &= \textstyle\sum_{n=0}^{N-1} x_{2n} e^{-j\frac{2\pi}{2N}k(2n)} + \sum_{n=0}^{N-1} x_{2n+1} e^{-j\frac{2\pi}{2N}k(2n+1)} \\ &= \textstyle\sum_{n=0}^{N-1} x'_n e^{-j\frac{2\pi}{N}kn} + e^{-j\frac{\pi}{N}k} \sum_{n=0}^{N-1} x''_n e^{-j\frac{2\pi}{N}kn} \\ &= \begin{cases} f'_k + e^{-j\frac{\pi}{N}k} f''_k & \text{falls } k < N \\ f'_{k-n} - e^{-j\frac{\pi}{N}(k-N)} f''_{k-N} & \text{falls } k \geq N \end{cases} \end{aligned}$$

Durch diese effiziente Aufteilung in zwei Gruppen von DFTs (gerade und ungerade) und anschließendem Vereinen zu einem Gesamtergebnis, kann die Berechnung in $\mathcal{O}(n \log n)$ Zeit geschehen [53, 32, 154].

Der in dieser Arbeit implementierte Algorithmus bildet die Funktionalität einer Radix-2-FFT-Architektur durch einen rekursiven Programmaufbau ab und basiert auf einer Radix-2 FFT mit 2048 komplexen 16-Bit-Eingangswerten.

5.2.6. Wireless LAN nach IEEE 802.11b

Der *IEEE-802.11b-Algorithmus* beschreibt die Basisbandsignalverarbeitung eines Algorithmus zur drahtlosen Kommunikation (WLAN) in Senderichtung [60]. Der implementierte C-Code implementiert die Verarbeitung der Bitübertragungsschicht des ISO[10] / OSI[11] -Schichtenmodells.

In mehreren Verarbeitungsschritten werden die digitalen PLCP[12] -Datenpakete in die diskreten Werte des späteren analogen Signals überführt. Während der Verarbeitung der übergebenen Daten werden die einzelnen Nutzbits in den komplexen Symbolraum abgebildet. Hierzu schreibt der IEEE-802.11b-Standard vor, dass bei der Abbildung in den *BPSK*[13] (1 MBit/s), beziehungsweise den *QPSK*[14] -Symbolraum (2 MBit/s und höher) eine Phasenverschiebung (Offset) von $\frac{\pi}{4}$ hinzugefügt werden muss. Die anschließende Datenübertragung zwischen Sende- und Empfangsstationen, die auf dem IEEE-802.11b-Standard basieren, erfolgt im lizenzfreien ISM[15] -Frequenzband. Innerhalb des hierfür reservierten Frequenzbereichs zwischen 2,4 GHz und 2,48 GHz werden die einzelnen Frequenzen in Europa zu 13 Kanälen zusammengefasst, von denen aufgrund von Überschneidungen aber nur drei Kanäle zeitgleich genutzt werden können. Um das Übertragungssignal unanfällig gegenüber schmalbandigem Rauschen zu machen, benutzt der IEEE-802.11b-Standard, im Unterschied zu vielen anderen WLAN-Standards, das Frequenzspreizungsverfahren *Direct Sequence Spread Spectrum (DSSS)*. Hierbei werden die Nutzdaten durch Multiplikation mit einer speziellen Bitfolge, im technisch einfachsten Fall mit einem 11 Bit langen *Barker-Code*[16], auf ein breiteres Frequenzspektrum aufgespreizt. Die einzelnen Bits des gespreizten Datenstroms werden in diesem Zusammenhang *Chips* genannt. Durch das Spreizen mit diesem Barker-Code erhöht sich die Anzahl der zu sendenden Symbole, da nun nicht mehr ein Nutzbit, sondern elf Chips gesendet werden, die anschließend in elf Symbole des BPSK-Symbolraums überführt werden. Der IEEE-802.11b-Standard schreibt vor, dass die Chips mit 11 MChip/s über den Kanal übertragen werden müssen. Hieraus resultiert, dass der entwickelte IEEE-802.11b-Algorithmus zur Erfüllung dieser Vorgabe die Datenpakete der Sicherungsschicht bei Verwendung der implementierten BSPK-Kodierung mit einem Durchsatz von

[10] International Organization for Standardization

[11] Open Systems Interconnection

[12] Physical Layer Convergence Protocol

[13] Binary Phase Shift Keying

[14] Quadrature Phase Shift Keying

[15] Industrial Scientific Medial

[16] (+1 +1 +1 -1 -1 -1 +1 -1 -1 +1 -1)

mindestens 1 MBit/s verarbeiten muss, um die geforderte Datenrate am Ende der Basisbandverarbeitung einhalten zu können [101] [108].

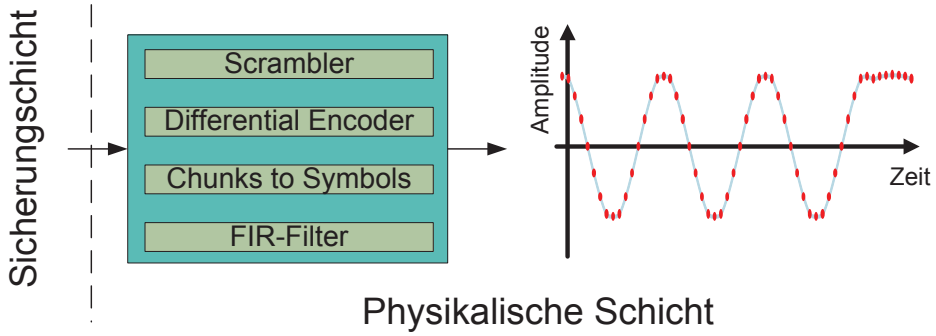


Abbildung 5.3.: Funktionsblöcke des IEEE-802.11b-Algorithmus

Die Implementierung des Algorithmus ist in vier Funktionsblöcke unterteilt (vgl. Abbildung 5.3):

- *Scrambler:*
  Um die Datenübertragung gewährleisten zu können, müssen sich der Sender und die Empfangsstation auf einen gemeinsamen Takt synchronisieren. Hierzu wird bei dem IEEE-802.11b-Algorithmus die Taktübertragung in das gesendete Signal integriert, indem der Empfänger die Flanken des Signals analysiert und den Takt des Senders hieraus zurückgewinnt. Die Taktrückgewinnung ist aber nur gesichert möglich, wenn das gesendete Signal keine großen Gleichanteile, sprich keine langen 0- bzw. 1-Folgen besitzt. Um dieses zu vermeiden, wird mit Hilfe eines selbstsynchronisierenden linear rückgekoppelten Schieberegisters (Linear Feedback Shift Register (LFSR)) der Scrambler („Verwürfler“) realisiert. IEEE 802.11b schreibt vor, dass dieses 7 Bit große Schieberegister zu Beginn der Sendung mit der Startbelegung *1101100* initialisiert werden muss. Das LFSR repräsentiert das einheitliche Polynom $G(z) = z^{-7} + z^{-4} + 1$.

- *Differential Encoder:*
  Während der Übertragung der Datenpakete kann sich die Phase des Signals durch Störungen auf dem Kanal oder durch Synchronisationsverluste zwischen Sender und Empfänger verschieben. Wenn diese Verschiebung einen gewissen Wert übersteigt, entsteht eine Phasenmehrdeutigkeit[17], wodurch der Empfänger die erhaltenen Symbole falsch interpretiert. Bei einer BPSK-Symbolraumkodierung tritt dieser Effekt bei Verschiebungen auf, die größer als +90° und kleiner

[17] Phase ambiguity

als -90° sind. Bei QPSK-kodierten Symbolen kommt es, aufgrund der näher aneinander liegenden Symbole, sogar schon bei Verschiebungen von $\pm 45°$ zu Fehlinterpretationen. Der *Differential Encoder* vermeidet diese Fehlinterpretationsprobleme bereits vor dem Abbilden der einzelnen Bits in den Symbolraum, indem er Abhängigkeiten zwischen dem aktuell zu übertragenen und dem vorangegangenen Bit herstellt. Hierzu werden die vom Scrambler erstellten Daten bitweise an den Differential Encoder übergeben und dort durch eine Modulo-2-Operation mit dem zuletzt errechneten Bit verbunden. Dieses impliziert bei der anschließenden Weiterverarbeitung und Übertragung, dass nur noch die Unterschiede zwischen den Datenbits übertragen werden. Tritt nun während der Übertragung eine dauerhafte Phasenverschiebung auf, ändert sich im Empfänger zwar gegebenenfalls die Symbolinterpretation, die Abhängigkeiten können aber weiterhin vom Empfänger korrekt analysiert und dekodiert werden [46].

- *Chunks-to-Symbols:*
  In der Implementierung dieses Funktionsblocks werden die Ausgangsdaten des Differential Encoders auf den BPSK-Symbolraum mit einem Phasenoffset von $\frac{\pi}{4}$ abgebildet.

- *Interpolation FIR-Filter:*
  Der Interpolationsfilter formt aus den Symbolen des vorherigen Funktionsblocks die diskreten Werte für die spätere (analoge) Signalübertragung. Da dieses Signal ohne Interpolation im Zeitbereich steilen Flanken und somit ein sehr großes Frequenzspektrum besitzen würde, wäre es in dieser Form nicht effizient zum Empfänger übertragbar. Aus diesem Grund wird das Frequenzspektrum durch eine Pulsformung in eine Form gebracht, die gut in dem ISM-Frequenzband darstellbar ist. Im Zeitbereich führt diese Pulsformung zu einer Glättung der steilen Flanken, indem mehrere Zwischenwerte zwischen die Amplitudenwerte eingefügt werden. Die Realisierung dieser Pulsformung erfolgt über eine Root-Raised-Cosine-Funktion mit Hilfe eines FIR-Filters erster Ordnung. Zusätzlich werden die Daten auf den Eingangsbereich des AD/DA-Wandlers des in Abschnitt 8.3.1 beschriebenen DB-SDR-Erweiterungsmoduls für das RAPTOR-System skaliert.

Die Größe eines vom IEEE-802.11b-Algorithmus verarbeiteten Datenpaketes ist frei wählbar. Für die Entwurfsraumexploration werden 6 Byte große Datenpakete verarbeitet.

5.2.7. Zyklische Redundanzprüfung (CRC)

Die *zyklische Redundanzprüfung (Cyclic Redundancy Check (CRC))* ist ein 1961 von WW. Peterson [159] entwickeltes Verfahren zur Sicherstellung einer fehlerfreien Datenübertragung [182]. Der Sender berechnet mittels einer Polynomdivision mit einem vorher vereinbarten Generatorpolynom eine Prüfsumme aus den zu übertragenden Nutzdaten. Anschließend vereint er die Nutzdaten und die Prüfsumme in einem Rahmen und überträgt ihn zum Empfänger. Da dem Empfänger das verwendete Generatorpolynom ebenfalls bekannt ist, kann dieser durch eine ähnliche Polynomdivison eine fehlerhafte Datenübertragung erkennen und diesen Rahmen neu anfordern.

5.2.8. Kryptographie mit elliptischen Kurven (ECC)

Dieser Kryptographiealgorithmus kann zur Verschlüsselung [184] und zur digitalen Signatur bei der Übertragung von Daten verwendet werden. Die Verschlüsselung erfolgt in Operationen auf Punkten einer elliptischen Kurve, deren Koordinaten Elemente eines endlichen Feldes sind.

Durch seine hohe Komplexität kann dieser Algorithmus bei vergleichbarer Sicherheit einen kleineren Schlüssel als herkömmliche Verschlüsselungsalgorithmen, wie beispielsweise RSA[18] , verwenden. Hieraus resultiert, dass bei der Datenübertragung deutlich weniger Informationen gesendet und gespeichert werden, wodurch eine energieeffizientere Übertragung ermöglicht wird.

Die Kryptographie mit elliptischen Kurven basiert auf der Binärfeldarithmetik. Allgemein gilt für eine supersinguläre elliptische Kurve EC über einen binären endlichen Körper $\mathbb{F}_{2^m}$:

$$EC : y^2 + xy = x^3 + ax^2 + b \tag{5.2}$$

für geeignete Parameter $a, b \in \mathbb{F}_{2^m}$. Die Menge $(x, y) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$, die Gleichung 5.2 genügt, zusammen mit dem „neutralen Element“ $\mathcal{O}$, bilden eine additive Abelsche Gruppe. Die Grundoperationen der Kryptographie mit elliptischen Kurven (Addition, Quadrierung und Multiplikation auf einem endlichen Körper) können auf die Punktaddition und Punktverdopplung eines Punktes auf einer elliptischen Kurve zurückgeführt werden. Punktaddition und Punktverdopplung können schließlich wieder auf eine Skalarmultiplikation zurückgeführt werden. Zur effizienten Berechnung der Punktmultiplikation kommt in der Realisierung der Anwendung der Montgomery Ladder Algorithmus [145, 127] zum Einsatz. Zur Reduktion der Komplexität der Polynommultiplikation wird das Karatsuba-Verfahren genutzt. Abbildung 5.4 zeigt die Hierarchie der Arithmetik bei der Kryptographie mit elliptischen Kurven (nach [167, 166]).

[18] Rivest, Shamir, Adleman

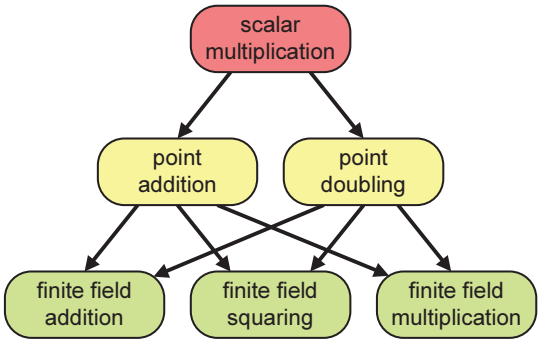


Abbildung 5.4.: Hierarchie der Arithmetik bei der Kryptographie mit elliptischen Kurven

5.2.9. Advanced Encryption Standard (AES)

Bei dem *Advanced Encryption Standard (AES)* [56] handelt es sich um ein symmetrisches Verschlüsselungsverfahren. Es wird als Nachfolger des DES[19] - bzw. 3DES[20] - Verschlüsselungsverfahrens betrachtet und ist nach seinen Erfindern Joan Daemen und Vincent Rijmen benannt. Der AES-Algorithmus besitzt eine Schlüssellänge von 128, 192 oder 256 Bit und eine feste Blöckgröße von 128 Bit[21]. Trotz einiger Kritikpunkte am Algorithmus gilt AES zur Zeit als sicher. Die betrachtete Implementierung des AES-Standards wird als Verschlüsselungsverfahren beim Protokol-Stack des 3GPP[22] Mobilfunkstandards LTE eingesetzt.

5.2.10. SNOW

Bei dem *SNOW-Algorithmus* [63, 64] handelt es sich um ein Verfahren zur wortbasierten Stromverschlüsselung [184]. Die Schlüssellänge kann 128 oder 256 Bit betragen. Die Verschlüsselung basiert auf 16 linear rückgekoppelten Schieberegistern und einem Zustandsautomaten. Ein wichtiger Aspekt für Verschlüsselungsverfahren ist die Resistenz gegen verschiedene Angriffsverfahren. Der SNOW-Algorithmus bietet zum gegenwärtigen Zeitpunkt keine praktischen Ansatzpunkte für bekannte, algebraische Angriffsverfahren abgesehen von der immer möglichen *Exhaustionsmethode* [109], bei

[19] Data Encryption Standard

[20] Triple Data Encryption Standard

[21] Beim ursprünglichen „Rijndael"-Algorithmus ist eine variable Blockgröße von 128, 192 oder 256 Bit spezifiziert.

[22] 3rd Generation Partnership Project

der alle möglichen Schlüssel „ausprobiert“ werden. Daher kann SNOW als *sicher* angesehen werden. SNOW ist als *SNOW 3G* [3, 2] Bestandteil der Verschlüsselungsalgorithmen UEA2[23] and UIA2[24] [1] und wird, wie der betrachtete AES-Algorithmus, beim Protokoll-Stack des 3GPP Mobilfunkstandards LTE eingesetzt.

5.2.11. Sum of absolute transformed differences (SATD)

Der *Sum of Absolute Transformed Differences (SATD)-Algorithmus* [171] ist ein wesentlicher Bestandteil der Videokompression des ITU[25] -Standards H.264 beziehungsweise des ISO/ IEC[26] -Standards MPEG[27] -4/ AVC[28] .

Er unterstützt eine blockbasierte Bewegungskompensation (Block-Matching), wodurch die Kompression bei bewegten Bildern nicht die Änderungen der einzelnen Pixel verarbeiten muss, sondern lediglich die Verschiebung unveränderter Pixel-Blöcke speichert. Während der Kompression erfolgt die Analyse der Verschiebungen in Blöcken von 16 x 16 Pixeln. Bleibt der Inhalt eines Blocks nahezu unverändert, wird angenommen, dass keine Bewegung vorliegt. Übersteigen die Änderungen aber einen vorgegebenen Schwellenwert, muss durch verschiedene Suchstrategien bestimmt werden, wohin sich dieser Block verschoben hat. Hierbei müssen gegebenenfalls Skalierungen, Rotationen oder andere Translationen berücksichtigt werden. Wird ein passender Nachfolgeblock gefunden, speichert die Kompression den Verschiebungsvektor zwischen altem und neuem Standort des Blocks [186].

Die Bewegungsabschätzung, das heißt die Bestimmung der Unterschiede zwischen dem vorangegangenen und dem aktuellen Block, erfolgt durch den SATD-Algorithmus. Hierbei werden die Differenzen zwischen den einzelnen Pixeln durch eine Hadamard-Transformation in den Frequenzbereich übertragen und anschließend addiert. Im Vergleich zu anderen Verfahren, die die Differenzen nicht im Frequenzbereich addieren, ist dieser Algorithmus zwar rechenintensiver, ermöglicht aber eine exaktere Bewegungsabschätzung.

[23] UMTS Encryption Algorithm 2

[24] UMTS Integrity Algorithm 2

[25] International Telecommunication Union

[26] Internationale Organisation für Normung, International Electrotechnical Commission

[27] Moving Picture Experts Group

[28] Advanced Video Coding

5.3. Skalierung der Ausführungszeiten der Anwendungen mit der Parallelität der VLIW-Architektur

Oftmals wird in der Literatur zum Vergleich von Prozessorarchitekturen der maximale Durchsatz an Instruktionen (Instruktionen pro Sekunde (IPS), (Fließkomma-) Operationen pro Sekunde ((FL)OPS) etc.) herangezogen. Der maximale Durchsatz eines VLIW-Prozessors ist abhängig von der Anzahl der Funktionseinheiten und der Taktfrequenz der Schaltung. Der maximale Durchsatz kann aber nur unter optimalen Bedingungen erreicht werden, falls der Compiler in der Lage ist, genügend ILP aus einer Anwendung zu extrahieren, so dass alle Funktionseinheiten ausgenutzt werden. Auch spielt der Instruktionssatz eine große Rolle. Prinzipbedingt kann eine Anwendung auf eine CISC[29] -Architektur im Vergleich zu einer RISC-Architekturen mit „reduziertem“ Instruktionssatz so abgebildet werden, dass die Gesamtanzahl an benötigten Taktzyklen geringer ist. RISC-Architekturen bieten dagegen den Vorteil, dass sie aufgrund des geringerem Funktionsumfangs mit einfacherer Logik auskommen und somit weitaus höhere Taktfrequenzen erreichen. Zur objektiven Bewertung der Rechenleistung einer (VLIW-)Prozessorarchitektur ist es daher erforderlich, neben der Taktfrequenz auch die Effizienz eines optimierenden Compilers zur Bestimmung der Leistungsfähigkeit zu untersuchen. Insbesondere bei hochparallelen Architekturen ist dieses wichtig, da die zunehmende Parallelität sehr hohe Anforderungen an das Scheduling stellt. Ohne Power-Management tragen nur selten verwendete Funktionseinheiten zur Leistungsaufnahme bei und können die Energieeffizienz eines Systems reduzieren.

Aus diesem Grund wurden zur Bewertung des in Kapitel 4 entwickelten, modularen VLIW-Prozessorsystems die in Abschnitt 5.2 beschriebenen Anwendungen für verschiedene Konfigurationen kompiliert und in Hinblick auf die Anzahl benötigter Taktzyklen analysiert. In diesem Abschnitt wird die Anzahl benötigter Taktzyklen in Bezug zur erreichbaren Taktfrequenz der verschiedenen Prozessorkonfigurationen gesetzt.

Die untersuchten Anwendungen wurden mit der in Kapitel 3 beschriebenen Werkzeugkette übersetzt und analysiert. Dem VLIW-Compiler kann die Anzahl verfügbarer VLIW-Slots als Parameter übergeben werden. Zum Zeitpunkt dieser Arbeit wurde allerdings nur eine maximale Anzahl von vier VLIW-Slots unterstützt. Wie die Analysen zeigen werden, stößt der VLIW-Compiler ab einer Parallelität von vier an seine Grenzen, weswegen die Untersuchung einer höheren Parallelität nicht notwendig erscheint. Des Weiteren erlaubt die Analyse auch die Klassifizierung von Anwendungsgruppen in gut oder schlecht parallelisierbare Kategorien.

Für nur einen verfügbaren VLIW-Slot berücksichtigt der Compiler den Sonderfall für die MVC-Instruktion (vgl. Abschnitt 4.3.6). Die Anzahl der MLA-, DIV- und

Tabelle 5.2.: *Relative Verkürzung der Benchmarklaufzeit mit 1–4 Ausführungseinheiten*

Anwendung	1→2	2→3	3→4	1 →4
Dhrystone	12%	2%	1%	14%
Coremark	19%	3%	0%	22%
Viterbi	31%	20%	7%	49%
Faltung	51%	10%	0%	61%
FFT	48%	14%	7%	68%
IEEE 802.11b	39%	9%	2%	50%
CRC	11%	6%	3%	20%
ECC	40%	10%	2%	52%
AES	26%	1%	0%	27%
SNOW	24%	6%	0%	30%
SATD	45%	14%	8%	67%

LD/ST-Einheiten ist für den Fall eines VLIW-Slots auf 1 und für den Fall von 2 oder mehr VLIW-Slots fest auf 2 gesetzt.

5.3.1. Auswertung

Abbildung 5.5 zeigt die Anzahl der benötigten Taktzyklen der Anwendungen für eine unterschiedliche Anzahl an VLIW-Slots. Die Daten sind auf die Ausführungszeit der 1-Slot-Konfiguration normiert [249]. Tabelle 5.2 zeigt die relativen Änderungen

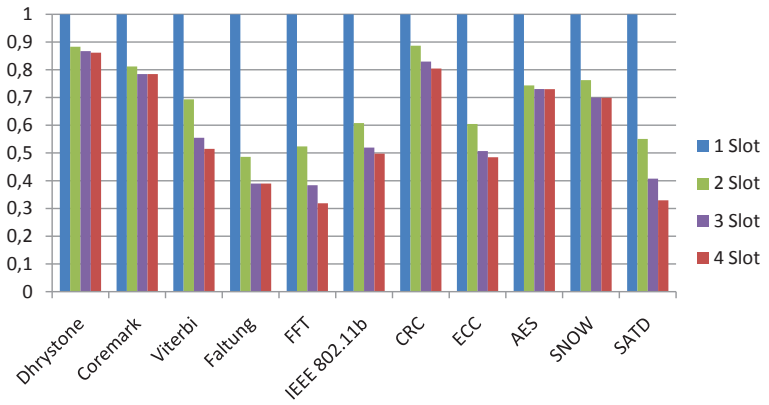


Abbildung 5.5.: Anzahl der benötigten Taktzyklen für eine unterschiedliche Anzahl an VLIW-Slots normiert auf 1 Slot

der Ausführungszeit mit zunehmender Anzahl an Ausführungszeiten bezogen auf die 1-Slot-RISC Konfiguration. Es zeigt sich deutlich, dass alle Anwendungen von einer zweiten Funktionseinheit profitieren. Insbesondere bei Anwendungen aus der

digitalen Signalverarbeitung der Kryptographie und der Bildverarbeitung von der VLIW-Architektur erscheint eine Parallelität von mehr als drei VLIW-Slots lohnenswert. Eine dritte Funktionseinheit reduziert hier die Ausführungszeit um 9–14 %. Eine vierte Funktionseinheit kann die Performanz weiterhin um bis zu 8 % (FFT, SATD) steigern. Eine detaillierte Analyse der Auslastung der dedizierten Multiplizierer und Divisionsschritteinheiten hat gezeigt, dass für die untersuchten Anwendungen eine maximale Anzahl von jeweils zwei dedizierten Einheiten ausreichend ist. Insbesondere der Faltungsalgorithmus profitiert von einem zweiten Multiplizierer. Algorithmen für die Durchführung der ganzzahligen Division sind als Makrofunktionen implementiert. Diese lassen sich effizient auf zwei VLIW-Slots aufteilen, woher auch hier die maximale Anzahl von zwei parallel verarbeiteten Divisionsschritt-Instruktionen resultiert. Aus Algorithmen wie CRC, die strukturell eher sequentiell aufgebaut sind, lässt sich wenig ILP extrahieren. Hier sind zwei (11 % Performanzsteigerung) oder drei (6 % Performanzsteigerung) Ausführungseinheiten ausreichend. Die synthetischen Benchmarks Coremark und Dhrystone profitieren hauptsächlich von nur einer weiteren Ausführungseinheit. Die relativ schlechte Parallelisierbarkeit des Dhrystone ist nach [221] darauf zurückzuführen, dass dieser Benchmark im Wesentlichen seit 1988 nicht weiterentwickelt worden ist und nicht auf feinkörnige, parallele Architekturen, wie VLIW-Architekturen, sowie Architekturen mit SIMD-Vektorverarbeitung angepasst ist.

5.3.2. C-Code-basierte Optimierung

Die Analyse der Anwendungen hat gezeigt, dass auch die Art und Weise der Programmierung einen großen Einfluss auf die Parallelisierbarkeit der Algorithmen hat. Relativ einfache Programmiertechniken ermöglichen es dem Compiler weit mehr ILP aus C-Code zu extrahieren. Trotz der Tatsache, dass ein (theoretisch) „idealer“ Compiler in der Lage sein sollte, verschiedene Beschreibungen der Semantik eines Algorithmus auf identischen (und optimalen) Maschinencode abzubilden, gibt es verschiedene Techniken, um dem Compiler die Extraktion von ILP aus der Anwendung zu erleichtern. Ein Grund hierfür ist bei der Compiler-Entwicklung die Abwägung von (Maschinen-)Code-Qualität und Arbeitsaufwand (und somit Zeit beim Compiliervorgang/der Software-Entwicklung). Ein Beispiel ist die Subjunktion von *Division* durch eine Zweierpotenz zu *logischem Linksschieben*[30] im Binärsystem. Unter Verwendung vorzeichenbehafteter Variablen (welche der Einfachhalber häufig bei der C-Programmierung Verwendung finden) wird bei der Division ein Algorithmus für die vorzeichenbehaftete Division angewandt. Für die Skalierung von Variablen auf kleinere Wertebereiche muss bei der Division eine Überprüfung auf mögliche

[30] Logisches Linksschieben kann z. B. zum einfachen Skalieren einer Größe auf einen kleineren Wertebereich genutzt werden.

Überläufe durchgeführt werden, welches die Ausführungszeit erhöht. Ein weiteres Beispiel ist die Verwendung von Variablen, die nicht an die *Breite des Datenpfades* (in der Regel 32 Bit) der Architektur angepasst sind. Oftmals möchte der Programmierer beispielsweise durch die Verwendung von 8 Bit-Variablen Speicherplatz sparen. Auch hier müssen nun Tests auf Überläufe durchgeführt werden, die bei der Verwendung von 32 Bit-Variablen unnötig gewesen werden. Eine Möglichkeit zur Optimierung rechenintensiver Schleifen durch den Compiler bietet das sogenannte *Abrollen der Schleifen*. Kann die Anzahl der Schleifendurchläufe im Vorfeld eindeutig bestimmt werden, so kann der Schleifenrumpf mehrfach hintereinander direkt in den Maschinencode gesetzt werden. Dieses ermöglicht wiederum eine bessere Parallelisierbarkeit der Operationen. Kann der Compiler diese Vorhersage des Ablaufs im Vorfeld nicht eindeutig bestimmen, gibt der Algorithmus dieses Verhalten aber implizit vor, so kann der Programmierer die Schleifen manuell im C-Code abrollen. Dieses bietet häufig weiteres Optimierungspotenzial. Die Auswirkungen der Optimierung von Algorithmen auf C-Code-Ebene soll im Nachfolgenden in zwei Beispielen (IEEE 802.11b und ECC) verdeutlicht werden. Die vorgestellten Optimierungsverfahren sind speziell auf die in dieser Arbeit vorgestellte VLIW-Architektur zugeschnitten, lassen sich aber auch verallgemeinert für andere Architekturen anwenden.

5.3.2.1. IEEE 802.11b

Die Implementierung des Schieberegisters des *Scramblers* konnte auf zwei verschiedene Weisen realisiert werden. Durch Speicheroperionen oder durch boolesche Operation. Durch Verwendung der booleschen Operationen konnte die Verarbeitszeit um 47 % gesenkt werden. Beim *Differential Encoder* nimmt die meiste Verarbeitungszeit die Berechnung der Unterschiede aufeinander folgender Symbole ein. Die Tatsache, dass sich der Wertebereich der Eingangs- und Ausgangsdaten auf 1 Bit erstreckt, konnte vom Compiler nicht ausgenutzt werden. Die Reduktion der Operation auf eine simple Exklusiv-Oder-Verknüpfung reduzierte die Ausführungszeit der Funktion um 37 %. Die Ausführungszeit der Implementierung des *Chunks-to-Symbols*-Funktionsblocks konnte durch eine leicht geänderte Beschreibung der Semantik im Kontrollfluss um 11 % verringert werden. Die größte absolute Anzahl an Taktzyklen ließ sich bei der Implementierung der *FIR-Funktion* einsparen. Durch Re-Organisation der verwendeten Speicherkonstrukte konnte die Ausführungszeit um 4672 Taktzyklen (24 %) reduziert werden. Ein manuelles Abrollen von Schleifen beschleunigte die Funktionsblöcke des Algorithmus weiterhin um bis zu 7,7 %. Die Resultate für der Optimierung des IEEE-802.11b-Algorithmus auf C-Code-Ebene sind in Abbildung 5.6 dargestellt. Durch verhältnismäßig einfache Änderungen auf C-Code-Ebene ließ sich die Gesamtausführungszeit des Algorithmus also um mehr als 26 % verringern.

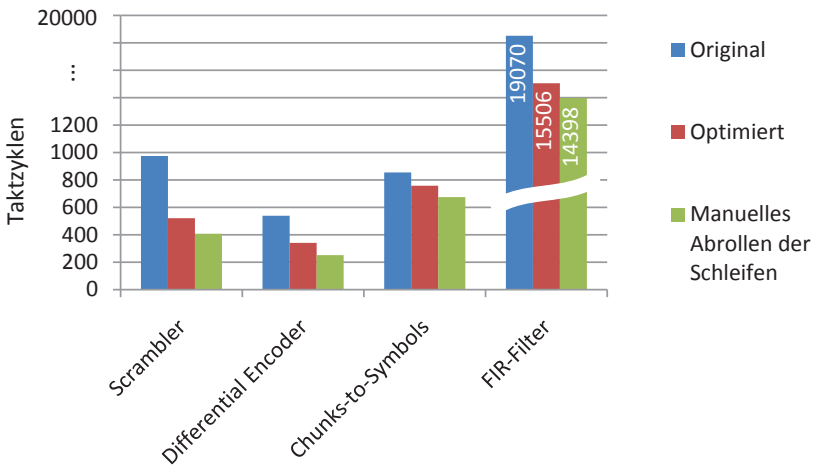


Abbildung 5.6.: Resultate für die Optimierung des 802.11b-Algorithmus auf C-Code-Ebene

5.3.2.2. Kryptographie mit elliptischen Kurven (ECC)

Während die Addition auf endlichen Körpern auf triviale Exklusiv-Oder-Operationen zurückgeführt werden kann, stellen die Quadrierung und Multiplikation auf endlichen Körpern die rechenintensivsten Operationen bei der Kryptographie mit elliptischen Kurven dar. Alleine durch die Einführung von *temporären Variablen* im Algorithmus der Punktmultiplikation konnten 31 % der Taktzyklen eingespart werden. Da die Punktmultiplikation die Grundlage für die Multiplikation auf endlichen Körpern bildet, konnte hierdurch der gesamte Algorithmus um knapp 6 % beschleunigt werden. Durch das gleiche Prinzip konnte die Ausführungszeit der Quadrierung auf endlichen Körpern um knapp 39 % reduziert werden. Die Optimierung beider Funktionen resultiert in einer Beschleunigung des gesamten Algorithmus um 22 %. Der Performanzgewinn ist in Abbildung 5.7 dargestellt. Es zeigt sich, dass durch geschickte Programmierung und leichte Hilfestellungen für den Compiler sehr viel bessere Ergebnisse erzielt werden können.

5.4. Skalierung der Ressourcen mit der Parallelität der VLIW-Architektur

Im vorherigen Abschnitt wurde der Einfluss der funktionalen Parallelität der Architektur auf Parallelisierbarkeit der Anwendung durch den Compiler untersucht. In den folgenden Abschnitten soll nun das Potenzial zusätzlicher Funktionseinheiten in

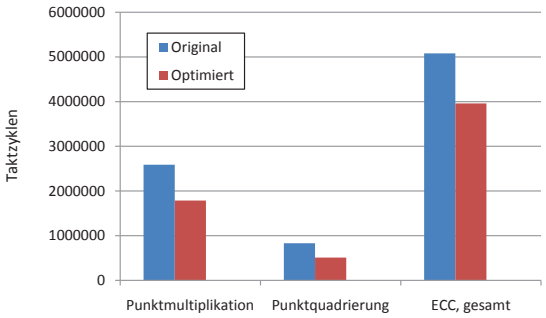


Abbildung 5.7.: Resultate für die Optimierung des ECC-Algorithmus auf C-Code-Ebene

Relation zu dem Mehraufwand an Hardwareressourcen gesetzt werden. Auch soll der Einfluss auf eine mögliche Abnahme der maximalen Taktfrequenz untersucht werden. Das VLIW-Prinzip ermöglicht die Entwicklung gut skalierbarer, feingranularer (und bezüglich der Komplexität der Hardware einfacher) Architekturen. Die Datenpfade einzelnen VLIW-Slots sind weitestgehend disjunkt (vgl. Abbildung 5.8).

Lade-, Dekodier- und Ausführungs-Einheiten sind unabhängig voneinander und werden mehrfach instanziiert. Eine Kombination der Datenpfade besteht in im Wesentlichen im Instruktions-/Datenspeicher, dem Register-File und dem Pipeline-Bypass. Die maximale Länge eines VLIWs hat auf den Instruktionsspeicher geringen Einfluss, da auch dieser lediglich mehrfach instanziiert wird. Die Komplexität des Alignment-Registers ist, wie in Abschnitt 4.3.4 gezeigt, sehr gering und bezüglich des Ressourcenbedarfs vernachlässigbar.

Wesentlichen Einfluss auf die Komplexität des Gesamtsystems haben der Pipeline-Bypass und das Register-File. Insbesondere die Anzahl der Leseports bestimmt den Ressourcenbedarf des Register-Files, da das Lesen aus diesem rein kombinatorisch erfolgt. Im Folgenden wird daher auf diese beiden Komponenten besonderes Augenmerk gelegt. Die Anzahl der Lese-/Schreibports bestimmt sich, wie in Kapitel 4 beschrieben, nach den Gleichungen 4.4 und 4.6. Zur Bestimmung der Skalierung der Ressourcen wurde der in Abschnitt 3.6.2 beschriebene automatisierte Hardware-Entwurfsablauf genutzt, um verschiedene Konfigurationen mit variierender Anzahl der Lese- und Schreibports des Register-Files durchzuführen. Abbildung 5.9 zeigt den Flächenbedarf und die Leistungsaufnahme für bis zu zwölf Lese- und Schreibports. Abbildung 5.10a zeigt die minimal erreichbare Latenz der kombinatorischen Logik. Es zeigt sich, dass die Fläche des Register-Files jeweils linear mit der Anzahl der Lese- und Schreibports anwächst. Bei der minimal erreichbaren Latenz ist ein logarithmischer Anstieg zu erkennen. Die Abhängigkeit besteht in erster Linie zur Anzahl der Leseports. Dieses ist darauf hinauszuführen, dass die Multiplexerstruktur der

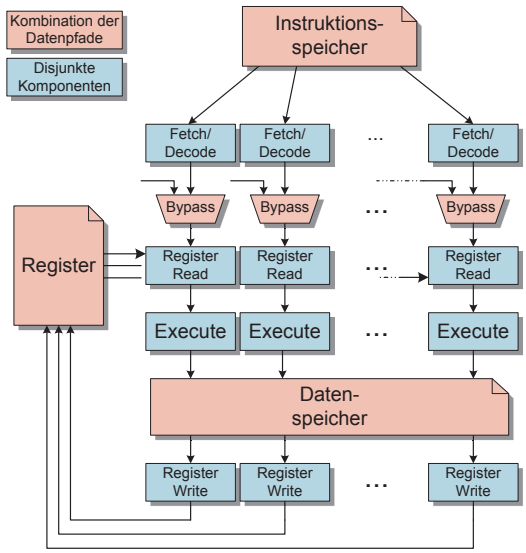
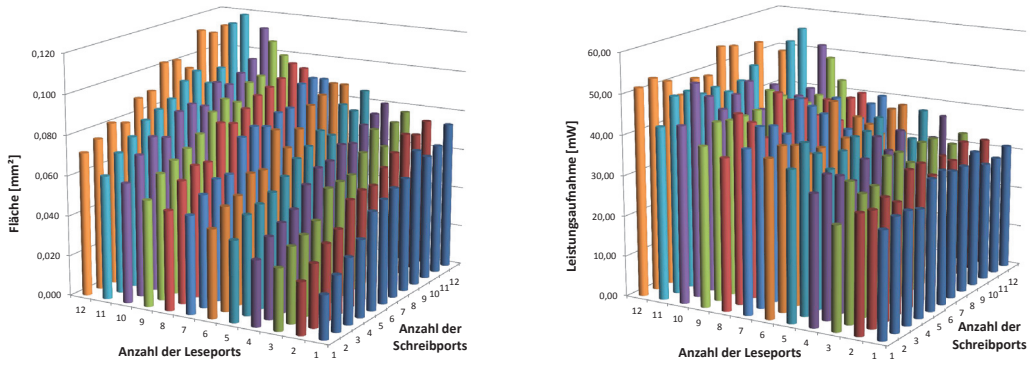


Abbildung 5.8.: Disjunkter Datenpfad einer VLIW-Architektur

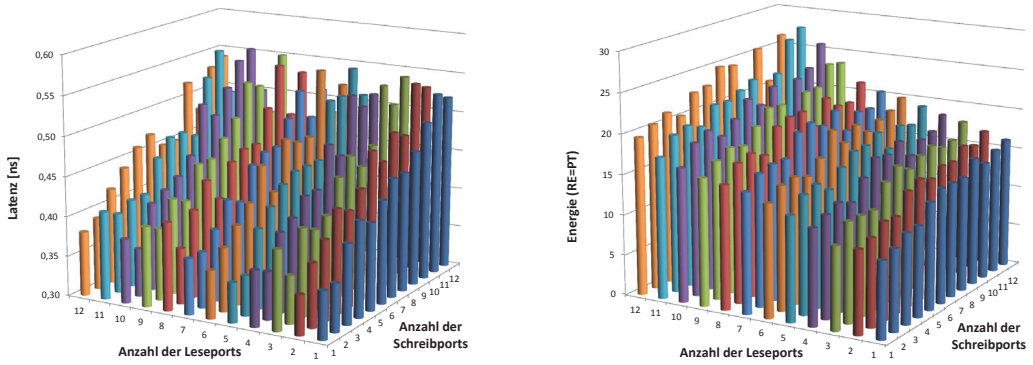
(Registeradress-)Dekoderstufe der Schreib- und Leseports über eine Baumstruktur realisiert ist. Eine Erhöhung des kritischen Pfades erfolgt also proportional zur Tiefe der Multiplexerstruktur. Die leichten Schwankungen der minimalen Latenz resultieren aus pseudozufälligen Sequenzen, die in modernen Synthesewerkzeugen genutzt werden. Die Leistungsaufnahme steigt sowohl mit Anzahl der Lese- als auch der Schreibports. Durch die Abhängigkeit der dynamischen Leistungsaufnahme von der Taktfrequenz (vgl. Gleichung 3.2) wird die Leistungsaufnahme auch von Streuungen der Latenz beeinflusst. Abbildung 5.10b zeigt das Produkt aus Latenz und Leistungsaufnahme. Dieses entspricht bei Anwendung des in Abschnitt 3.4 vorgestellten Maßes zur Bewertung der Ressourceneffizienz einem Index von $RE = 1/(P \cdot T)$ oder auch der Energie pro Taktzyklus. Hieran lässt sich die Skalierung der Energieeffizienz des Register-Files in Abhängigkeit von der Parallelität der VLIW-Architektur ablesen. Mit steigender Anzahl an Lese- und Schreibports ist eine leichte Reduktion der Energieeffizienz zu beobachten.

Zur Bestimmung der *Skalierung des Prozessorkerns* in Abhängigkeit vom Grad der funktionalen Parallelität wurde die Architektur für 1–16 VLIW-Slots auf eine 65 nm Standardzellentechnologie von STMicroelectronics abgebildet. Die Konfigurationen wurden wie folgt gewählt: Für eine 1-Slot-Konfiguration (RISC-Architektur) stehen jeweils eine dedizierte MLA-, DIV- und LD/ST-Einheit zur Verfügung. Ab zwei VLIW-Slots sind jeweils zwei MLA-, DIV- und LD/ST-Einheiten vorhanden. Zum Einen ist dieses konsistent zur Compiler-Konfiguration, die nur die Konfiguration



(a) Fläche **(b)** Leistungsaufnahme

Abbildung 5.9.: Ressourcenbedarf des Register-Files



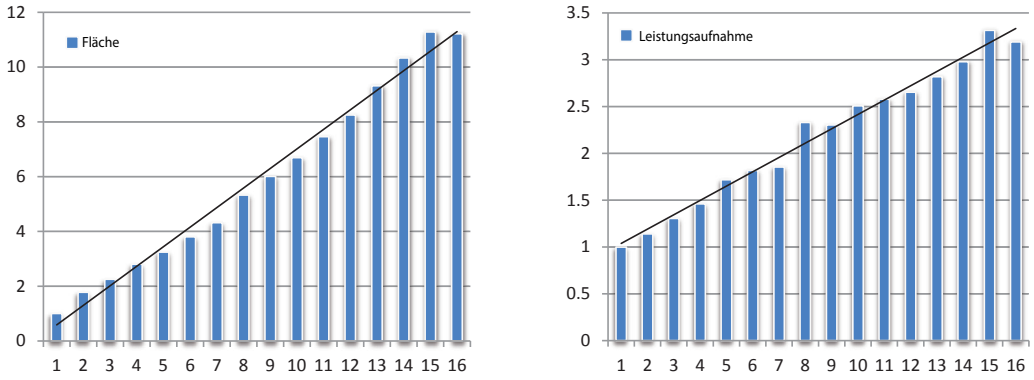
(a) Latenz **(b)** Energiebedarf

Abbildung 5.10.: Minimal erreichbare Latenz und Energiebedarf des Register-Files

der Anzahl der Funktionseinheiten ermöglicht. Zum Anderen sind, wie bereits in Abschnitt 5.3 diskutiert, Konfigurationen mit mehr als zwei dedizierten MLA-, DIV- und LD/ST-Einheiten nur für Spezialfälle sinnvoll. Eine genaue Betrachtung aller untersuchten Algorithmen hat gezeigt, dass mehr als zwei dedizierte Multiplizierer oder Dividierer keinen zusätzlichen Vorteil bringen. Viel wichtiger ist die Anzahl der VLIW-Slots, d.h. wie viele Instruktionen parallel verarbeitet werden können. Anstatt mehrere MLA-Instruktionen parallel zu platzieren, kann der VLIW-Compiler diese Instruktionen auch meist einen Taktzyklus vorziehen oder verzögern, ohne dass sich die Bearbeitungszeit verändert, vorausgesetzt, die vorhandene Parallelität der Architektur ist ausreichend, um ein solches Scheduling zu ermöglichen. Wie später in Abschnitt 7.1 detailliert beschrieben, lassen sich Multi-Port-Speicher mit mehr als zwei Ports im Vergleich zu 1-Port- oder 2-Port-Speichern nicht effizient imple-

mentieren. Abschnitt 7.2 beschreibt einen alternativen Ansatz zur Beschleunigung verschiedener Algorithmen durch den Einsatz von Scratchpad-Speicher, weswegen hier die Analyse auf maximal zwei Speicher-Ports beschränkt wird. Die Auswertung der Ergebnisse zeigt aber auch, das zusätzliche LD/ST-Einheiten bezüglich ihrer Fläche und Leistungsaufnahme vernachlässigbar im Vergleich zur der restlichen Logik der Prozessorarchitektur sind.

Abbildung 5.11 zeigt die *Skalierung der Fläche und der Leistungsaufnahme* des gesamten VLIW-Prozessors für 1–16 VLIW-Slots. Abbildung 5.12 zeigt die minimal



(a) Fläche **(b)** Leistungsaufnahme

Abbildung 5.11.: Skalierung der Ressourcen des Prozessorkerns in Abhängigkeit vom Grad der Parallelität

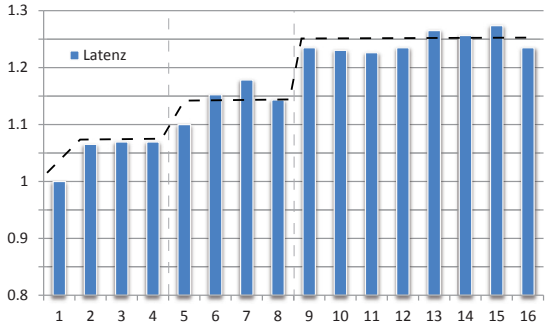


Abbildung 5.12.: Skalierung der minimalen Latenz in Abhängigkeit vom Grad der Parallelität

erreichbare Latenz (und damit implizit als Kehrwert die erreichbare Taktfrequenz). Alle Ergebnisse sind auf die 1-Slot-Konfiguration normiert. Der Flächenbedarf des Prozessorkerns bei einer 1-Slot-Konfiguration beträgt 90313 μm^2 bei einer Verlustleis-

tung von 10 mW und einer maximal möglichen Taktfrequenz von 380 MHz (Worst Case). Sowohl Fläche als auch Leistungsaufname steigen linear mit der Anzahl der VLIW-Slots an. Bei der minimalen Latenz erkennt man eine Besonderheit. Die Taktfrequenz bleibt weder konstant (wie man das bei einer vollständigen Entkopplung der VLIW-Slots erwarten könnte), noch steigt sie linear mit Anzahl der Slots an. Stattdessen ist eine Art „Treppeneffekt" zu beobachten. An dem Übergang jeder Zweierpotenz ($1 \rightarrow 2$, $4 \rightarrow 5$, $8 \rightarrow 9$ etc.) steigt die Latenz relativ stark an, während sie innerhalb dieser Grenzen nahezu konstant bleibt[31]. Dieser Treppeneffekt ist, wie bei dem Register-File, auf die Realisierung der Multiplexerstrukturen des Pipeline-Bypasses und der Ressourcenteilung der dedizierten MLA- und DIV-Einheiten zurückzuführen.

Zur genaueren Analyse wurde zusätzlich der Ressourcenbedarf der Teilkomponenten des VLIW-Prozessor-Kerns für 1–8 VLIW-Slots ausgewertet. Abbildung 5.13 zeigt die *Verteilung der Chip-Fläche auf die Hauptkomponenten* des Prozessorkerns bei 50 MHz (gestrichelte Linien) und maximal möglicher Taktfrequenz (durchgezogene Linien). Die geringe Taktfrequenz wurde gewählt, damit die Skalierung der Ressourcen unabhängig vom kritischen Pfad der Architektur untersucht werden kann. Komponenten im kritischen Pfad würden ansonsten überproportional mit steigender Anzahl der VLIW-Slots anwachsen, da das Synthese-Werkzeug für diese Komponenten Gatter mit größerer Treiberstärke (und damit geringerer Latenz) auswählt. Mit steigender Anzahl an VLIW-Slots steigt die Fläche (und proportional dazu auch die Leistungsaufnahme) der Komponenten Execute, Register-File, Decode und der Pipeline-Register linear an. Beim Pipeline-Bypass ist ein quadratischer Anstieg zu beobachten. Bis zu einer VLIW-Slot-Anzahl von sechs dominieren die Funktionseinheiten des Prozessorkerns den Ressourcenbedarf. Ab sieben VLIW-Slots dominiert der *Pipeline-Bypass* die Gesamtfläche des Prozessorkerns. Um die maximale Taktfrequenz zu erreichen verwendet das Synthese-Werkzeug für Komponenten, die im kritischen Pfad liegen, Gatter mit hoher Treiberstärke. Der Flächenbedarf dieser Bauteile steigt demensprechend. Für 1–8 VLIW-Slots liegt das Register-File nicht im kritischen Pfad. Von den Pipeline-Registern sind nur sehr wenige Gatter von einer Erhöhung der Treiberstärke betroffen. Für beide Komponenten ist der Flächenbedarf bei 50 MHz und bei maximaler Taktfrequenz identisch. Der Pipeline-Bypass wächst bei der Synthese für die maximal erreichbare Taktfrequenz um bis zu 8 % an. Die Komponenten der Fetch/Decode-Pipelinestufe sind für 1–4 VLIW-Slots unkritisch bezüglich der maximalen Taktfrequenz. Erst ab 5 VLIW-Slots fallen diese beiden Pipelinestufen ins Gewicht. Der Flächenbedarf steigt um bis zu 66 %. Aufgrund des geringen Anteils an der Gesamtfläche des Prozessors fällt der Flächenzuwachs bezogen auf das Gesamtsystem nicht ins Gewicht. Die größte Steigerung des Flächenbedarfs bei maximaler Taktfrequenz ist bei der Execute-Pipelinestufe zu verzeichnen. Hier steigt

[31] Die leichten Schwankungen der Ergebnisse resultieren erneut aus pseudozufälligen Sequenzen moderner Synthesewerkzeuge

der Flächenbedarf bereits ab zwei VLIW-Slots mit Erhöhung der Taktfrequenz stark an. Bei acht VLIW-Slots ergibt sich ein Zuwachs von 40 %.

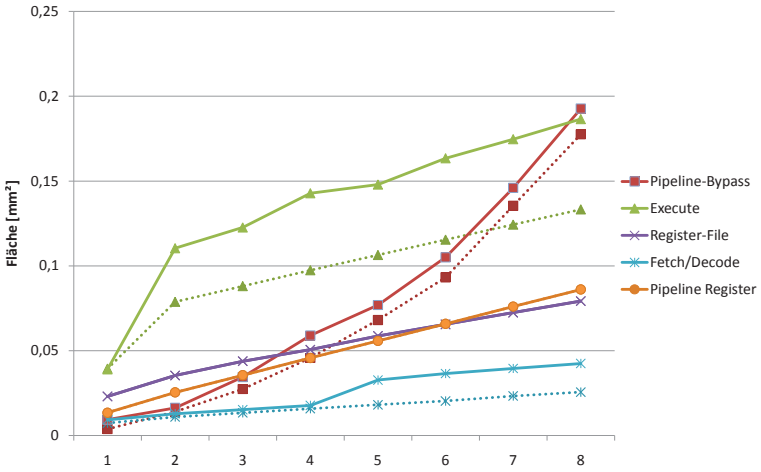
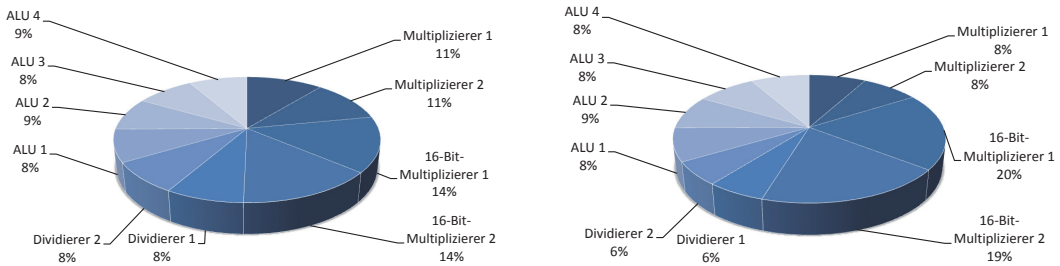


Abbildung 5.13.: Verteilung der Fläche des VLIW-Prozessorkerns auf die Hauptkomponenten

Abbildung 5.14 zeigt schließlich den Vergleich der Verteilung der Fläche der Execute-Pipelinestufe auf die enthaltenen Funktionseinheiten für 50 MHz und die maximale Taktfrequenz (358 MHz) für eine Beispielkonfiguration des VLIW-Prozessors von vier VLIW-Slots mit vier ALUs und je zwei Multiplizierern und Dividierern. Die Fläche der für den kritischen Pfad verantwortlichen Komponenten (in diesem Fall insbesondere die 16-Bit-Multiplizierer für den SIMD-Modus) steigt aus den zuvor genannten Gründen erneut überproportional an.



(a) 50 MHz **(b)** $f_{\max}$ (358 MHz)

Abbildung 5.14.: Flächenbedarf der Execute-Pipelinestufe (4 VLIW-Slots)

5.5. Bewertung der Ressourceneffizienz des Hardware/Software-Systems

Die Analyse der Anwendungen im Abschnitt 5.3 hat gezeigt, dass der Compiler von Prozessorkonfigurationen bis etwa vier VLIW-Slots profitieren kann. Im vorherigen Abschnitt wurde die Skalierung des Ressourcenbedarfs in Abhängigkeit zur Anzahl der VLIW-Slots bestimmt. In diesem Abschnitt soll nun die *Ressourceneffizienz* der Kombination aus Compiler-Werkzeugkette und Hardware bestimmt werden, indem die Ausführungszeiten in Bezug zu dem Ressourcenbedarf gesetzt werden. Hierfür wird das in Abschnitt 3.4 eingeführte Maß zur Bewertung der Ressourceneffizienz auf die Daten angewandt und die Ergebnisse miteinander verglichen.

Abbildung 5.15 zeigt die Ressourceneffizienz bei $RE = 1/(P \cdot T)$ (Energieeffizienz, Power-Delay-Produkt) für des Prozessorkerns für eine VLIW-Slot-Anzahl von 1–4. Die Werte sind auf die 1-Slot-Konfiguration normiert. Es zeigt sich, dass für die Anwendungen Viterbi, Faltung, IEEE 802.11b und ECC der höchste Ressourceneffizienz-Index für 3-VLIW-Slots erreicht wird. Bei den Anwendungen FFT und SATD ist die 4-Slot-Variante am effizientesten. Lediglich die schlecht parallelisierbaren synthetischen Benchmarks Dhrystone und Coremark profitieren nicht von der höheren Anzahl an Funktionseinheiten und erreichen das Maximum von $RE = 1/(P \cdot T)$ bei einem VLIW-Slot.

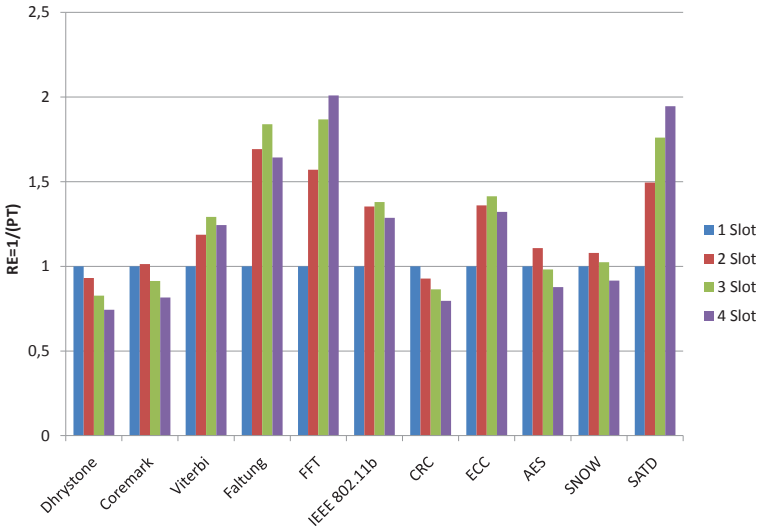
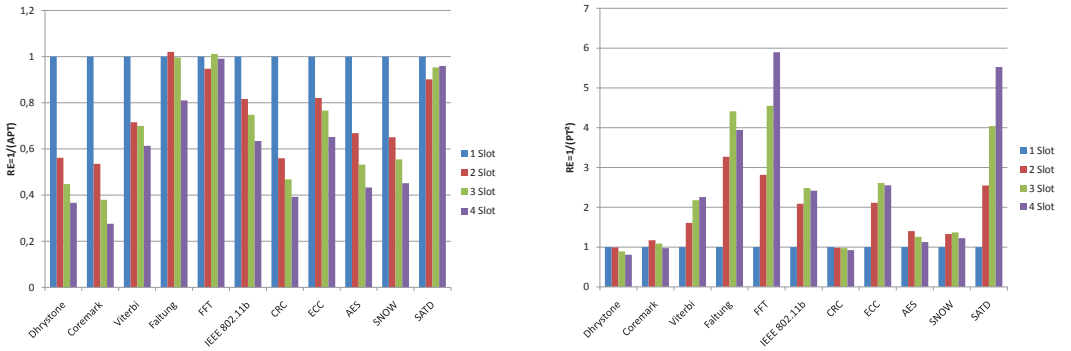


Abbildung 5.15.: Ressourceneffizienz des CoreVA-Prozessors bei Anwendung des Maßes $RE = 1/(P \cdot T)$ (Energie) zur Bewertung der Ressourceneffizienz

Des Weiteren wurde die Ressourceneffizienz für verschiedene Variationen des Ressourceneffizienz-Index (d.h. für verschiedene Anwendungsszenarien) berechnet. Für stark *kostenbeschränkte Architekturen* kann die Chip-Fläche mit in die Bewertung einfließen $RE = 1/(A \cdot P \cdot T)$. In Abbildung 5.16a zeigt sich hier ein starker Vorteil der kleineren Architekturen. Für die gut parallelisierbaren Anwendungen wir FFT und SATD sind die 1–4-Slot-Konfigurationen gleichwertig.

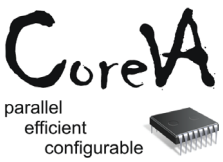
Für Anwendungsszenarien mit *hohen Performanzanforderungen* kann die Zeit beim Ressourceneffizienz-Index höher gewichtet werden ($RE = 1/(P \cdot T^2)$, Power-Energy-Produkt). Die Ergebnisse zeigt Abbildung 5.16b. Hier verschiebt sich der Trend klar in Richtung höherer Parallelität.



(a) $RE = 1/(A \cdot P \cdot T)$ **(b)** $RE = 1/(P \cdot T^2)$, Power-Energy-Produkt

Abbildung 5.16.: Ressourceneffizienz des CoreVA-Prozessors für verschiedene Ressourceneffizienz-Indizes

5.6. Der CoreVA-VLIW-Prozessor



Im Anschluss an die Bewertung der Ressourceneffizenz der modularen VLIW-Architektur in Abhängigkeit von der VLIW-Parallelität soll nun eine (Referenz-)Architektur für die weiteren Analysen der Architektur auf Systemebene und für die Implementierung von FPGA- und ASIC-Prototypen ausgewählt werden. Im Hinblick auf das geplante Anwendungsszenario des VLIW-Prozessors im *mobilen Umfeld* erscheint die Wahl der *Energie* ($RE = P \cdot T$) als geeignetes Bewertungsmaß. Hier zeigte sich eine Konfiguration von drei oder vier VLIW-Slots am effizientesten. Für die Referenz-Architektur wurde somit eine vierfach parallele Konfiguration mit vier ALUs, zwei MLA-, zwei DIV- und zwei LD/ST-Einheiten gewählt. Die Referenz-Architektur wird im Folgenden *CoreVA* (*Co*nfigurable, *re*source efficient, *V*LIW *A*rchitecture) genannt. Abbildung 5.17 zeigt das Blockschaltbild

der gewählten Architektur. In einer 65 nm Standardzellentechnologie von STMicroelectronics erreicht der CoreVA-Prozessorkern eine maximale Taktfrequenz von 358 MHz (Worst Case) bei einer Verlustleistung von 15 mW. Der Flächenbedarf liegt bei 0,27 mm^2.

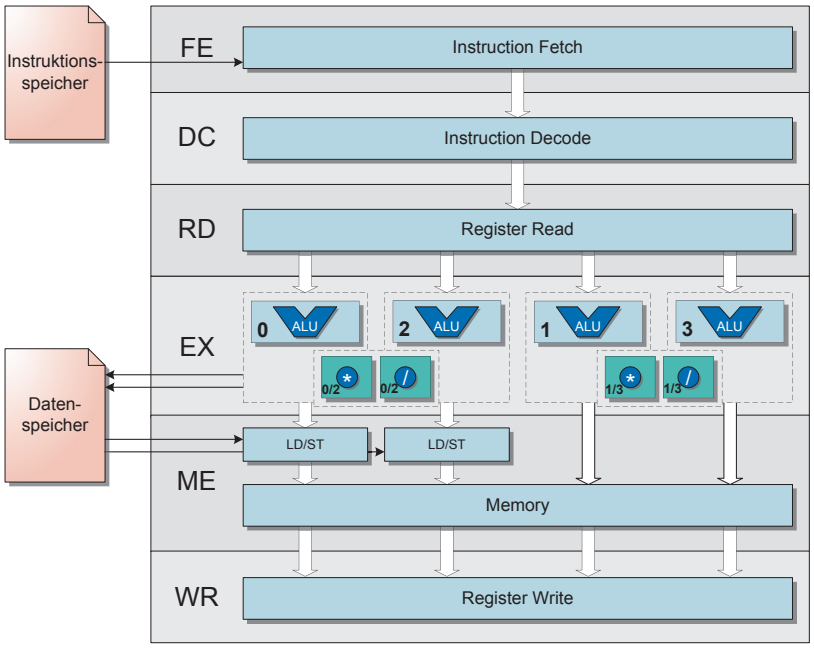


Abbildung 5.17.: Blockschaltbild der CoreVA-Architektur mit sechs Pipelinestufen und vier VLIW-Slots

Tabelle 5.3 vergleicht den Ressourcenbedarf des CoreVA-Prozessors zum *N-Core* [120, 89] und *QuadroCore* [164, 163]. Der N-Core stellt eine Von-Neumann-Architektur mit einer dreistufige Pipeline (FE, DC, EX) und RISC-Instruktionssatz dar. Der QuadroCore ist ein auf dem GigaNetIC-System basierender, dynamisch rekonfigurierbarer Prozessor-Cluster mit vier N-Core-Prozessoren [153, 165][239, 240]. Beide Architekturen wurden im Fachgebiet Schaltungstechnik entwickelt und liegen als HDL-Beschreibung auf RT-Ebene vor. Zur Vergleichbarkeit wurden Standardzellensynthesen bei einer Referenzfrequenz von 200 MHz durchgeführt. Die maximale Taktfrequenz unter Berücksichtigung der Latenzen des On-Chip-Speichers liegen beim N-Core bei 257 MHz, beim QuadroCore bei 240 MHz. Hier zeigt sich der Vorteil der mehrstufigen Pipelinearchitektur des CoreVA-Prozessors. Sowohl beim Flächenbedarf als auch bei der Leistungsaufnahme übertrifft der CoreVA den QuadroCore. Weiterhin wurde in [164] die Ausführungszeit des in Abschnitt 5.2.8 vorgestellten Algorithmus zur Kryptographie mit elliptischen Kurven für den QuadroCore und den

N-Core bestimmt. Auch hier liefert die CoreVA-Architektur in Verbindung mit dem parallelisierenden Compiler mit 1839 Taktzyklen (CoreVA) zu 3193 (QuadroCore) bzw. 9402 Taktzyklen (N-Core) eine um 39 % höhere Performanz (N-Core: 80 %).

Tabelle 5.3.: *Vergleich des Ressourcenbedarfs des CoreVA-Prozessors zum N-Core und zum QuadroCore bei einer Taktfrequenz von 200 MHz*

Architektur	# VEen	Flächenbedarf	Leistungsaufnahme
CoreVA	**4**	**0,20 mm^2**	**3,73 mW**
N-Core	1	0,04 mm^2	1,38 mW
QuadroCore	4	0,23 mm^2	4,23 mW

5.7. Architekturvergleich zu kommerziellen VLIW-Architekturen

Zum Vergleich der CoreVA-Architektur mit anderen Architekturen aus dem Bereich der Multimedia und Signalverbeitung wurden zwei *kommerzielle Architekturen* ausgewählt. Der *TI TMS320C642* (vgl. Abschnitt 2.2.4) verfügt über eine vergleichbare Architektur. Mit Hilfe eines zyklenakkuraten Simulators des Code Composer Studios [201, 168] kann die Anzahl der Taktzyklen der zuvor analysierten Anwendungen für den TMS320C642 ermittelt werden. Des Weiteren werden die Ergebnisse für den CoreVA-Prozessor mit denen für einen *ARM9E-Prozessor* verglichen. Der Instruktionssatz des CoreVA-Prozessors ist stark an den des ARM9E angelehnt, weswegen ein Vergleich hier sinnvoll ist. Abbildung 5.18 zeigt einen Vergleich der Ausführungszeiten der analysierten Anwendungen. Auch bei den kommerziellen Architekturen zeigt sich, dass die synthetischen Benchmarks schlecht parallelisierbar sind. Bei vielen Anwendungen (insbesondere Faltung, FFT und ECC) ist die parallele Architektur des CoreVAs auch dem hochoptimierten ARM9E-Compiler weit überlegen. Bei dem Faltungs-Algorithmus erreicht der CoreVA sogar im Vergleich zum TMS320C642 eine geringere Ausführungszeit. Ansonsten macht sich natürlich die höhere Anzahl an VLIW-Slots des TMS320C642 in der Ausführungszeit bemerkbar. Der klare Vorteil des TMS320C642 beim SATD-Algorithmus ist auf spezielle Instruktionen zur Verarbeitung von Video-Codecs und Sättigungsarithmetik zurückzuführen [207]. Bezüglich der Ressourceneffizienz (insbesondere der Energieeffizienz) ist die CoreVA-Architektur gegenüber der TMS320C642-Architektur im Vorteil, da dessen Verlustleistung weit über der des gesamten CoreVA-Systems liegt (vgl. Abbildung 5.19b). Natürlich bietet sich sowohl bei der Werkzeugkette als auch beim Hardware-Entwurf weiteres Optimierungspotential, welches beispielsweise bei einer Vermarktung des CoreVA-Prozessors unter Einsatz eines weitaus höheren Entwicklungsaufwandes

ausgeschöpft werden könnte. Zusammenfassend kann gesagt werden, dass sich die CoreVA-Architektur und der zugehörige VLIW-Compiler auch gegen die hochoptimierten kommerziellen Produkte von ARM und TI gut behaupten kann. Beim Coremark Benchmark erreicht der CoreVA ein Ergebnis von 1,44 Iterationen pro Sekunde und Megahertz und liegt damit im Mittelfeld vergleichbarer eingebetteter Systeme. Abbildung 5.19a zeigt qualitativ den Ressourcenbedarf des CoreVA-Prozessors und der in Abschnitt 2.2 vorgestellten, kommerziellen VLIW-Prozessoren. Die Fläche der Kreise stellt den Flächenbedarf der jeweiligen Architektur dar. Performanz und Flächenbedarf sind auf die eine Strukturgröße von 65 nm skaliert (Konstantfeldskalierung [222]: $f \rightarrow s \cdot f$, $A \rightarrow s^2 \cdot A$). Die Leistungsaufnahme wurde über die Versorgungsspannung und die Taktfrequenz skaliert (vgl. Gleichung 3.2, $P \propto V_{dd}^2 \cdot f$ [97]). Die CoreVA-Architektur zeichnet sich durch vergleichsweise geringen Ressourcenbedarf aus. Abbildung 5.19b vergleicht die Energieeffizienz der Architekturen. Hier bietet nur die Tile64-Architektur von Tilera eine höhere Effizienz. Die Performanz der Architektur ist allerdings auf einen Prozessorkern der 64-Prozessor-Architektur normiert. Die von Tilera publizierte maximale Performanz ist aber nur bei einer optimalen Ausnutzung aller 64 Prozessorkerne erreichbar und entspricht sicherlich nicht realen Anwendungszenarien.

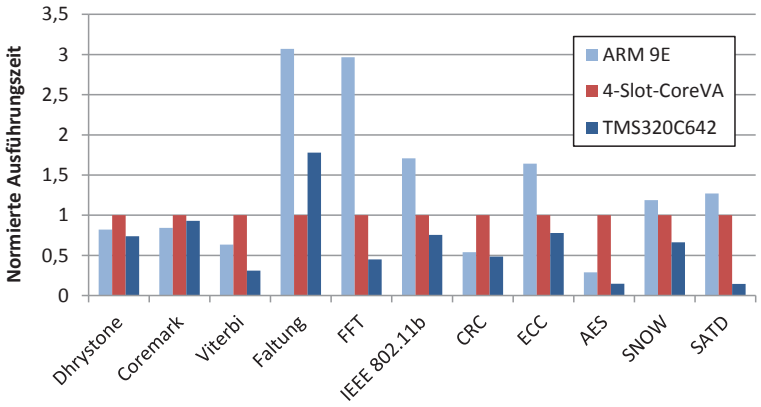
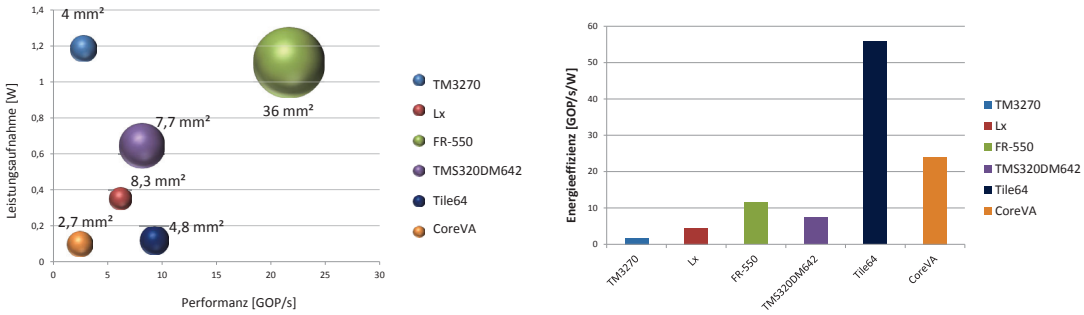


Abbildung 5.18.: Vergleich der Performanz der CoreVA-Architektur mit kommerziellen Prozessorarchitekturen

5.8. Zusammenfassung

In diesem Kapitel wurde eine Entwurfsraumexploration der in Kapitel 4 entwickelten modularen VLIW-Architektur durchgeführt. Die intrinsische Parallelität von Algorithmen aus einer breiten Auswahl an Anwendungsszenarien wurde in Bezug



(a) Skalierter Ressourcenbedarf **(b)** Skalierte Energieeffizienz

Abbildung 5.19.: Vergleich des Ressourcenbedarfs der CoreVA-Architektur mit kommerziellen VLIW-Architekturen (vgl. Abschnitt 2.2)

zum Ressourcenbedarf verschiedener Prozessorkonfigurationen gesetzt [234]. Hierbei wird implizit auch die Qualität des C-Compilers bei der Extraktion von ILP aus der jeweiligen Anwendung berücksichtigt. Die Analyse der Anwendungen zeigte unterschiedliche Ergebnisse. Insbesondere Algorithmen aus dem Bereich der Basisbandverarbeitung und Multimediaanwendungen ermöglichen einen hohen Grad an Parallelität. Anwendungen mit hohem Kontrollanteil und die in den synthetischen Benchmarks verwendeten Programmkerne lassen sich schlecht parallelisieren. Die Analyse der Architektur zeigte die lineare Skalierung des Ressourcenbedarfs mit der Anzahl der Funktionseinheiten. Aufgrund von weitgehend disjunkten Datenpfaden, welche VLIW-Prozessoren prinzipbedingt aufweisen, ist die maximale Taktfrequenz weitgehend unabhängig von der Parallelität. Der kritische Pfad wird im Wesentlichen durch die Ressourcenteilung gemeinsam genutzter Funktionseinheiten und den Pipeline-Bypass beeinflusst. Die Auswertungen haben ergeben, dass eine vierfach parallele VLIW-Konfiguration mit vier ALUs, zwei Multiplizierern, zwei Dividierern und zwei Speicherkanälen zum Datenspeicher einen guten Kompromiss zwischen Performanz und Ressourcenbedarf darstellt. Diese Konfiguration wird im Folgenden als Referenzarchitektur gewählt und als *CoreVA-Prozessor* bezeichnet. Des Weiteren zeigte sich, dass mit steigender Anzahl an VLIW-Slots der Pipeline-Bypass mehr und mehr den Ressourcenbedarf des Gesamtsystems dominiert. Im folgenden Kapitel werden daher Möglichkeiten einer statischen und dynamischen (Re-)Konfiguration des Pipeline-Bypasses untersucht. In Kapitel 7 wird die Entwurfsraumsexploration des CoreVA-Prozessors schließlich auf Systemebene ausgeweitet.

6. Optimierung des Pipeline-Bypasses

In den Kapiteln 4 und 5 wurden die Möglichkeiten einer *statischen* Konfiguration der *Anzahl der implementierten Funktionseinheiten* untersucht. In diesem Kapitel soll nun die *statische und dynamische* Konfiguration des *Pipeline-Bypasses* betrachtet werden.

6.1. Motivation

Die Entwurfsraumexploration der modularen, konfigurierbaren VLIW-Architektur in Kapitel 5 hat gezeigt, dass der Pipeline-Bypass des CoreVA-Prozessors einen sehr großen Einfluss auf die Ressourceneffizienz der Kombination aus Hardware und Software hat. Bei der vierfach-parallelen CoreVA-Referenzkonfiguration belegt der Pipeline-Bypass bereits ein Fünftel der Gesamtfläche des Prozessorkerns. In diesem Kapitel soll das Potential einer statischen und dynamischen (Re-)Konfiguration des Pipeline-Bypasses durch das Abschalten einzelner Bypass-Pfade untersucht werden. Wie gezeigt werden wird, stellt die Deaktivierung von Bypass-Pfaden jedoch ein Kompromiss zwischen der Erhöhung der maximalen Taktfrequenz und dem Hinzufügen von Strafzyklen dar, weswegen eine vollständige Deaktivierung des Bypass-Systems nicht zielführend ist. Aus diesem Grund wird ein Greedy-basierter Algorithmus vorgestellt, der auf der systematischen Deaktivierung einzelner Bypass-Pfade basiert, um eine optimierte Bypass-Konfiguration zu ermitteln.

6.1.1. Stand der Technik von Bypass-Systemen

Bei der Entwicklung von Prozessorarchitekturen spielt der Pipeline-Bypass in der Forschung eine große Rolle.

In [7], [34] und [68] wird beschrieben, dass zur Implementierung einer vollständigen Bypass-Struktur sehr viele Bypass-Pfade eingebunden werden müssen. Diese Pfade ermöglichen zwar eine schnelle Auflösung von Datenkonflikten, sie können durch ihre Kontrolllogik jedoch auch negativen Einfluss auf die maximale Taktfrequenz und den Flächenbedarf eines Prozessors haben.

[7] untersucht die Auswirkungen einer partiellen Bypass-Implementierung. Die Evaluation verschiedener Bypass-Konfigurationen zeigte, dass das Deaktivieren einzelner Pfadgruppen unterschiedliche Auswirkungen auf die Leistungsfähigkeit eines Prozessors haben kann. Abhängig von der ursprünglichen Auslastung dieser Pfade

verschlechterte sich die Verarbeitungsgeschwindigkeit verschiedener Benchmark-Tests in einigen Fällen lediglich um wenige Prozent, bei anderen Bypass-Kombinationen jedoch um bis zu 90 %. Ein angepasstes Scheduling konnte diese Einbußen jedoch deutlich vermindern.

Die Analysen in [68] kamen zu der Erkenntnis, dass ein Großteil der implementierten Bypass-Pfade nur sehr geringe Auslastungen aufweist und deshalb in anwendungsspezifischen Prozessoren deaktiviert werden kann. Aus diesem Grund wurde in einem ersten Optimierungsschritt durch das iterative Deaktivieren einzelner Bypass-Pfade eine Pareto-optimale Bypass-Konfiguration ermittelt. Im Rahmen dieser Veröffentlichung wurde jedoch nicht erläutert, auf welchen Richtwerten die Deaktivierungsentscheidungen basierten. Die Implementierung bezieht sich nur auf die Konfiguration des Daten-Register-Bypasses. Ob dieses System einen Kontroll-Bypass besitzt und ob dieser ebenfalls optimiert werden kann, wird nicht erwähnt. In einem zweiten Optimierungsschritt wurde der prozessorspezifische Compiler an die ermittelten Bypass-Konfigurationen angepasst. Eine neu implementierte Priorisierungsfunktionalität analysiert vor dem Platzieren der Instruktionen, ob Datenabhängigkeiten zwischen bestimmten Instruktionen auftreten und ob Bypass-Pfade zur Auflösung dieser Konflikte zur Verfügung stehen. Zur Reduzierung der Strafzyklen werden diese Instruktionen gegebenenfalls neu angeordnet. Durch das Verarbeiten mehrerer Beispielanwendungen zeigte sich, dass neben den ungenutzten Bypass-Pfaden bis zu 70 % der genutzten Pfade deaktiviert werden konnten. Die Performanz des Prozessors sinkt hierdurch jedoch um 10 %. In dieser Arbeit in Abschnitt 6.3 wird ein ähnlicher Ansatz vorgestellt der jedoch sämtliche Bypass-Systeme bei der Optimierung berücksichtigt.

[85] und [173] beschreiben ebenfalls die Nachteile, die durch den Einsatz eines vollständigen Bypass-Systems entstehen. In dem verwendeten Optimierungsansatz wurden diese Nachteile jedoch bewusst in Kauf genommen, da diese Pfade zur Reduzierung der Leistungsaufnahme der Daten-Register genutzt werden konnten. Da der Großteil der Rechenergebnisse innerhalb von drei Prozessortakten verworfen wird, konnten die Zugriffe auf die Daten-Register reduziert werden, indem diese Ergebnisse über mehrere Takte in den Bypass-Systemen zwischengespeichert wurden. Aus diesem Grund wurden Compiler-basierte Lösungen implementiert, die die Kurzlebigkeit bestimmter Rechenergebnisse erkennen und die Zugriffe auf die Daten-Register durch zusätzlich eingefügte Steuerbits unterbinden. In [173] konnte die Leistungsaufnahme der Daten-Register durch diesen Ansatz um 26 % reduziert werden. Das Reduzierungspotential in [85] lag sogar zwischen 40 % und 60 %. Da durch die Compiler-basierte Steuerung ein Großteil der Bypass-Logik eingespart werden konnte, konnte der Flächenbedarf des Prozessors ebenfalls um bis zu 5 % reduziert werden.

6.1.2. Analyse der Auslastung des Pipeline-Bypasses

Um die Möglichkeit der Effizienzsteigerung durch die Implementierung anwendungsspezifischer Bypass-Konfigurationen zu bewerten, wird in diesem Abschnitt eine detaillierte *Analyse der Auslastung* der einzelnen Bypass-Systeme und deren Pfade durchgeführt. Abbildung 6.1 zeigt die Auslastung verschiedener Bypass-Pfade für den IEEE-802.11b-Algorithmus (vgl. Abschnitt 5.2)[1]. Auf der horizontalen Achse sind die zwölf Bypass-Instanzen des Daten-Register-Bypasses (zu den drei Operanden pro VLIW-Slot) aufgeführt. Auf der vertikalen Achse ist aufgetragen, aus welcher Pipelinestufe (EX, ME, WR) und aus welchem VLIW-Slot (0 (ALU+LD/ST), 1 (ALU+LD/ST), 2 (nur ALU), 3 (nur ALU)) ein Zwischenergebnis bezogen wurde. Auffällig ist, dass in diesem Beispiel 64 % Bypass-Pfade nicht genutzt werden (grün). 22 % der Bypass-Pfade haben bezogen auf die Gesamtanzahl der Bypass-Zugriffe eine Auslastung von weniger als 1 % (gelb). 90 % aller Zugriffe des Daten-Register-Bypasses entfallen auf weniger als 15 % der vorhandenen Bypass-Pfade (rot). 53 % aller Bypass-Zugriffe erfolgen für Operanden des ersten VLIW-Slots. Fasst man die Zugriffe aller Instanzen auf die Execute, Memory-Access- und Register-Write-Pipelinetufe zusammen, erfolgen 51 % der Zugriffe auf Zwischenergebnisse des ersten VLIW-Slots. 62 % aller Bypass-Zugriffe erfolgen auf Zwischenergebnisse der Execute-Pipelinestufe. Im Folgenden soll nun untersucht werden, inwieweit die Anhäufung der Bypass-Zugriffe (beispielsweise aus/nach VLIW-Slot 0 oder aus der Execute-Pipelinestufe) in Korrelation zur Auslastung der Ausführungskanäle der VLIW-Slots steht.

Zur Auswertung des Pipeline-Bypasses wurden die in Abschnitt 5.2 vorgestellten Anwendungen herangezogen. Zusätzlich werden eine Assembler-optimierte Version des Faltungsalgorithmus und ein Algorithmus zur Berechnung einer Fibonacci-Folge hinzugezogen. Die Assembler-optimierte Version des Faltungsalgorithmus (Faltung (ASM[2])) zeichnet sich durch einen außergewöhnlich hohen Anteil an MLA-Instruktionen (und dadurch auch an Zugriffen des MLA-Bypasses) aus. Der rekursive Fibonacci-Algorithmus verfügt über einen sehr hohen Anteil an Sprüngen (und verursacht dadurch eine sehr hohe Auslastung des Kontroll-Bypasses). Als erster Teil der Analyse soll untersucht werden, inwieweit sich die Auslastung der VLIW-Ausführungskanäle auf die Auslastung der Bypass-Systeme auswirkt. In den folgenden Diagrammen sind die Analyseergebnisse beispielsweise in Abhängigkeit von der VLIW-Slot-Nummer oder der jeweiligen Pipelinestufe aufgetragen. Obwohl es sich hierbei um eine diskrete Verteilung handelt, sind die einzelnen Werte zur Veranschaulichung der Tendenz miteinander durch Linien verbunden. Die Abbildun-

[1] Der Übersichtlichkeit halber beschränkt sich diese Darstellung ausschließlich auf den Daten-Register-Bypass.

[2] Assembler

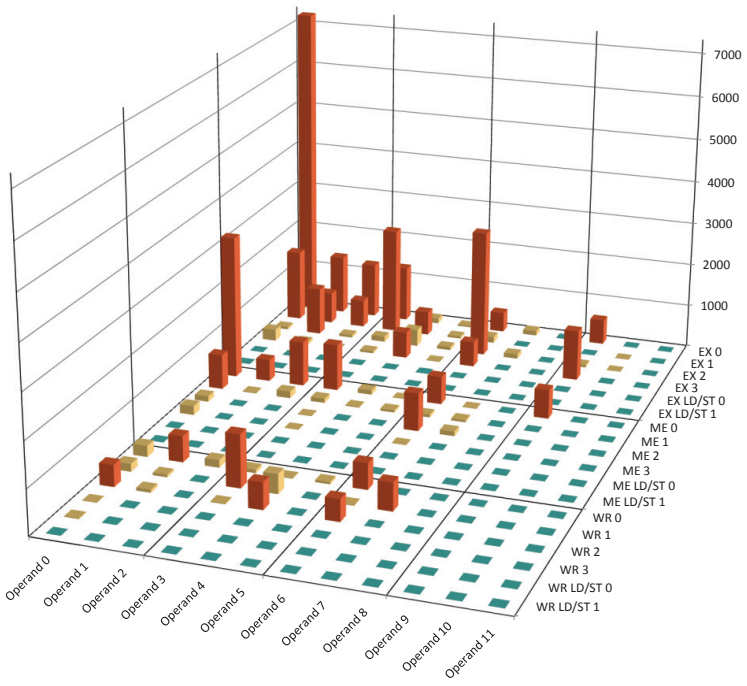


Abbildung 6.1.: Daten-Register-Bypass-Zugriffe des IEEE-802.11b-Algorithmus

gen 6.2 und 6.3 zeigen die Anzahl der Instruktionen und Operanden pro Taktzyklus und VLIW-Slot. Die Anzahl ist abhängig von der Parallelisierbarkeit der Anwendung und nimmt mit steigender VLIW-Slot-Nummer ab [231]. Abbildung 6.4 zeigt die Anzahl der Bypass-Zugriffe pro Taktzyklus und VLIW-Slot. Diese verhält sich für die meisten Anwendungen ähnlich zur Anzahl der Operanden pro Taktzyklus und VLIW-Slot. Hieraus ergibt sich eine verhältnismäßig konstante Verteilung der Bypass-Zugriffe pro Operand und VLIW-Slot (vgl. Abbildung 6.5). Für die meisten Anwendungen besteht also eine Korrelation zwischen Grad der Parallelität und Auslastung des Register-Bypasses. Ausnahmen bilden die beiden Faltungsalgorithmen: Beide führen sehr viele Ladeoperationen durch, deren Instruktionen aufgrund der Beschränkung der LD/ST-Einheiten in den VLIW-Slots 0 und 1 platziert werden müssen. Die MLA-Instruktionen werden daher in den verbleibenden VLIW-Slots 2 und 3 platziert. Die Operanden der Ladeoperationen sind Konstanten, die keine Bypass-Zugriffe erzeugen. Die MLA-Instruktionen benötigen drei Operanden und verursachen dadurch besonders viele Bypass-Zugriffe.

Des Weiteren erfolgen die meisten Zugriffe der Bypass-Instanzen auf Slot 0 (vgl. Abbildung 6.6). Auch hier spiegelt sich die Parallelität der Anwendung wieder. Ab-

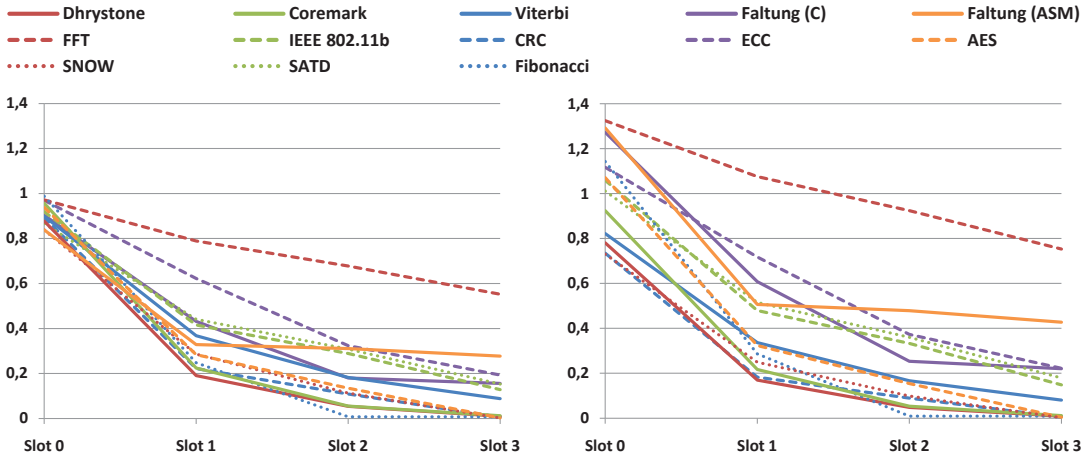


Abbildung 6.2.: Anzahl der Instruktionen pro Taktzyklus

Abbildung 6.3.: Anzahl der Operanden pro Taktzyklus

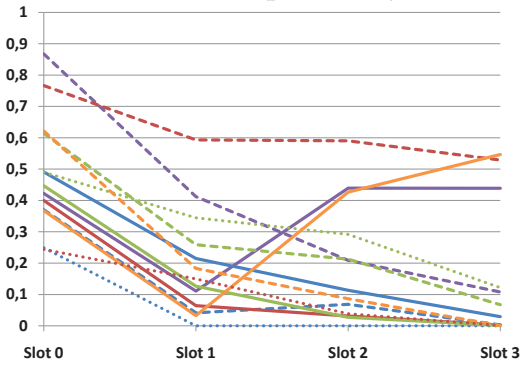


Abbildung 6.4.: Anzahl der Bypass-Zugriffe pro Taktzyklus

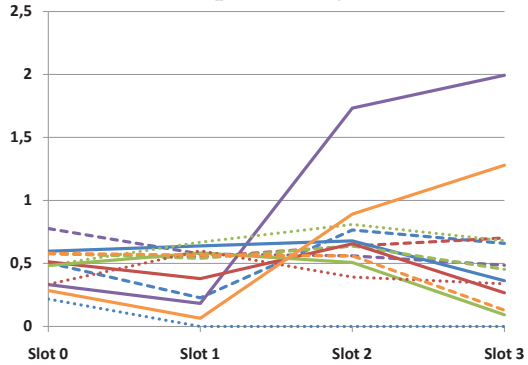


Abbildung 6.5.: Anzahl der Bypass-Zugriffe pro Operand

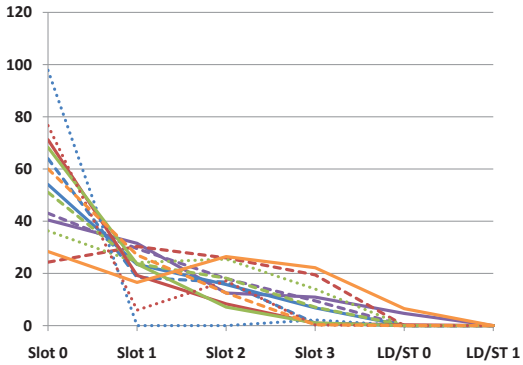


Abbildung 6.6.: Anteil der Bypass-Zugriffe auf einen Slot [%]

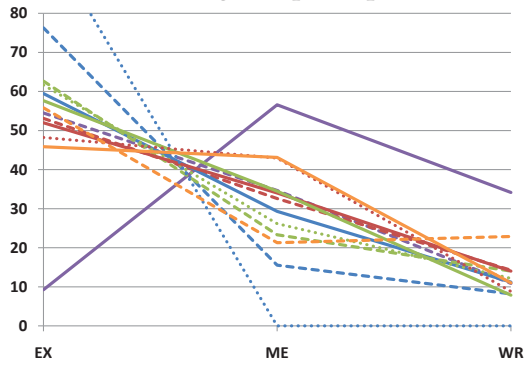


Abbildung 6.7.: Anteil der Bypass-Zugriffe auf eine Stufe [%]

bildung 6.7 zeigt die Verteilung der Pipelinestufen, von denen ein Zwischenergebnis über den Pipeline-Bypass gelesen wird. Die meisten Zugriffe erfolgen auf die Ausführungsstufe, d.h. Datenabhängigkeiten bestehen meist zwischen direkt aufeinander folgenden Instruktionen. Die Faltungsalgorithmen stellen erneut eine Ausnahme dar. Abhängigkeiten bestehen meist zu Ladeinstruktionen, die ihr Ergebnis erst in der Memory-Pipelinestufe bereitstellen.

Eine detailliertere Analyse des Kontroll-Bypasses zeigt, dass dieser ein ähnliches Verhalten wie der Daten-Register-Bypass besitzt. Da durch die Instruktionsverteilung verhältnismäßig viele Instruktionen innerhalb des ersten VLIW-Slots verarbeitet werden, ist auch die Anzahl der bedingten Instruktionen und somit die Anzahl der Kontroll-Bypass-Zugriffe in diesem Slot höher (siehe Abbildung 6.8). Die Bedingungen dieser Instruktionen werden zu einem Großteil erst in direkt vorausgehenden Befehlen bestimmt, weshalb der Kontroll-Bypass mit 60 % bis 100 % am häufigsten Zwischenergebnisse der Execute-Pipelinestufe verarbeitet.

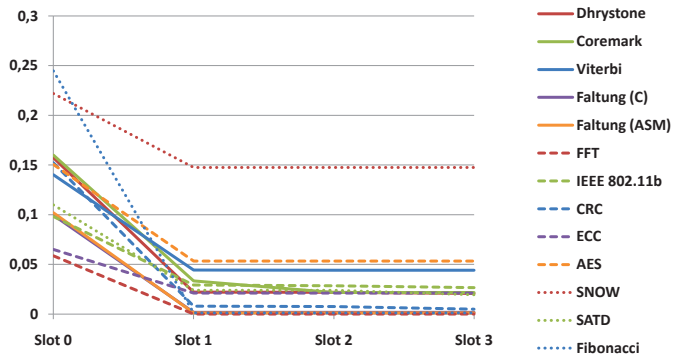


Abbildung 6.8.: Zugriffe pro Takt des Kontroll-Bypasses

Abbildung 6.9 zeigt den Anteil der verschiedenen Bypass-Systeme an der Gesamtzahl aller Bypass-Zugriffe. Der Daten-Register-Bypass stellt im Allgemeinen den größten Anteil an den Zugriffen. Zweitgrößte Komponente ist der Kontroll-Bypass. Auf den *MCR-Bypass* (als Teil des Register-Bypasses) wird am dritthäufigsten zugegriffen. Beim FFT-, ECC- und bei den beiden Faltungsalgorithmen wird der Kontroll-Bypass nur sehr selten genutzt. Der Fibonacci und SNOW-Algorithmus nutzen den Kontroll-Bypass dagegen außergewöhnlich häufig. Dieses ist auf einen hohen Anteil an bedingten Sprüngen zurückzuführen. Die Faltung (ASM) nutzt als einziger Algorithmus in ca. 22 % der Fälle den MLA-Bypass. Beim SNOW-Algorithmus wird der MCR-Bypass in hohem Maße genutzt, was auf einen hohen Anteil an Funktionsaufrufen zurückzuführen ist. Lediglich der Condition-Register-Bypass wird in keinem der Anwendungsfälle genutzt. Dieses ist aber darauf zurückzuführen, dass die derzeitige Version des VLIW-Compilers nur über sehr einfache Algorithmik zur Nutzung der Condition-Instruktionen verfügt. Da dieses in späteren Versionen des

VLIW-Compilers vorgesehen ist, ist der Condition-Register-Bypass in der aktuellen Implementierung des CoreVA-Prozessors trotzdem integriert. Der Anteil des Condition-Register-Bypasses am gesamten Bypass ist jedoch sehr gering und bezüglich des Ressourcenbedarfs vernachlässigbar. Wie bereits in Kapitel 5 gezeigt, hat der Pipeline-Bypass einen großen Einfluss auf die Ressourceneffizienz des CoreVA-Prozessors. Wie im Folgenden gezeigt werden wird, wird insbesondere die maximale Taktfrequenz des Systems durch den Pipeline-Bypass beeinflusst, da der kritische Pfad durch diesen verläuft. Im folgenden Abschnitt wird nun sowohl die *statische* als auch die *dynamische*, anwendungsspezifische *(Re-)Konfiguration* der Bypass-Systeme untersucht.

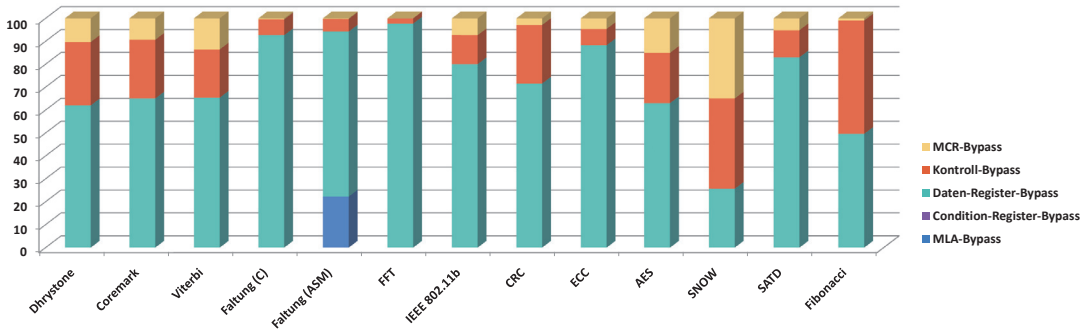


Abbildung 6.9.: Anteil der Bypass-Systeme an der Gesamtzahl aller Bypass-Zugriffe

6.2. Deaktivieren einzelner Bypass-Komponenten

Dieser Abschnitt zeigt, wie die CoreVA-Architektur strukturiert erweitert wurde, um das *gezielte Deaktivieren* einzelner Bypass-Pfade zu ermöglichen. Die Konfiguration der Bypass-Komponenten erfolgt durch generische Variablen. Das Verfahren ähnelt der Beschreibung des CoreVA-Prozessors zur Konfiguration der Anzahl der implementierten Verarbeitungseinheiten (vgl. Kapitel 4). Zum Deaktivieren der verschiedenen Bypass-Komponenten werden mit Hilfe der generischen Variablen Matrizen erzeugt, die eine individuelle Konfiguration jedes Bypass-Pfades zur Entwurfszeit ermöglichen (vgl. Anhang C).

Das Deaktivieren der einzelnen Bypass-Komponenten erfolgt durch das Unterbrechen des zugehörigen Bypass-Pfades. Hierzu wurde die Bypass-Steuerung so erweitert, dass die Pfade nur genutzt werden können, wenn das jeweilige Bit der Konfigurationsmatrix gesetzt ist. Falls das Bit nicht gesetzt ist und das System diesen Bypass-Pfad nicht auswählen kann, wird er und die von ihm abhängige Steuerlogik von der Hardware-Synthese auch nicht implementiert. Um nicht nur das Deaktivieren

vollständig ungenutzter Bypass-Pfade zu ermöglichen, wurde neben der Erweiterung der Fallunterscheidungen eine Verzögerungsfunktionalität implementiert. Falls ein Bypass-Pfad benötigt wird um Datenkonflikte aufzulösen, dieser aber deaktiviert wurde, fügt die Verzögerungsfunktionalität einen Strafzyklus ein (vgl. Abbildung 6.10). Falls die Lese-Registeradresse der Register-Read-Pipelinestufe mit der Register-Schreibadresse der Ausführungsstufe übereinstimmen, wird die Verfügbarkeit des zugehörigen Bypass-Pfades überprüft. Ist dieser Pfad aktiviert, wird er von dem System genutzt und die nachfolgende Instruktion kann verzögerungsfrei ausgeführt werden. Falls dieser Bypass-Pfad jedoch deaktiviert wurde, muss die Bearbeitung der nachfolgenden Instruktion zur Auflösung des Konflikts verzögert werden.

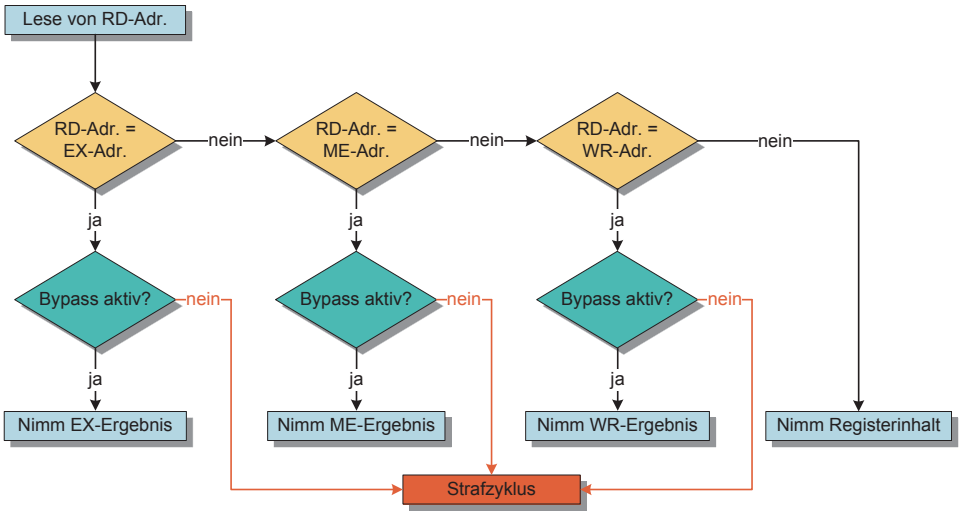


Abbildung 6.10.: Erweiterung der Bypass-Architektur um Funktionalitäten zur Verzögerung von Instruktionen bei deaktivierten Bypass-Konfigurationen

6.3. Auswahl der effizientesten Bypass-Konfiguration

Zur *Auswahl der effizientesten Bypass-Kombination* muss das Zusammenspiel zwischen einer möglichen *Verkürzung des kritischen Pfades*, der Reduzierung der benötigten Fläche und Leistung und der *hinzukommenden Strafzyklen* betrachtet werden. Insgesamt ergeben sich durch die 456 Bypass-Komponenten $2^{456} = 1,86 \cdot 10^{137}$ verschiedene Konfigurationsmöglichkeiten. Da die Analyse einer Konfiguration je nach Anwendung zwischen wenigen Minuten und mehreren Stunden benötigt, können nicht alle möglichen Kombinationen bewertet werden. Aus diesem Grund wird im folgenden Kapitel zuerst die Analyse des kritischen Pfades und anschließend die Entwicklung

eines *Greedy (dt. gierig)*-basierten Algorithmus [54] zur Ermittlung einer optimierten Bypass-Konfiguration vorgestellt. Der Algorithmus basiert auf der systematischen Deaktivierung selten genutzter Bypass-Pfade [251]. Zur Analyse des kritischen Pfades wurde die im vorigen Abschnitt vorgestellte Erweiterung der Bypass-Architektur genutzt, um die Bypass-Pfade, die den kritischen Pfad beeinflussen, gezielt abzuschalten. Die Abbildung 6.11 zeigt hierzu eine Übersicht des Verlaufs des kritischen Pfades für verschiedene Bypass-Konfigurationen.

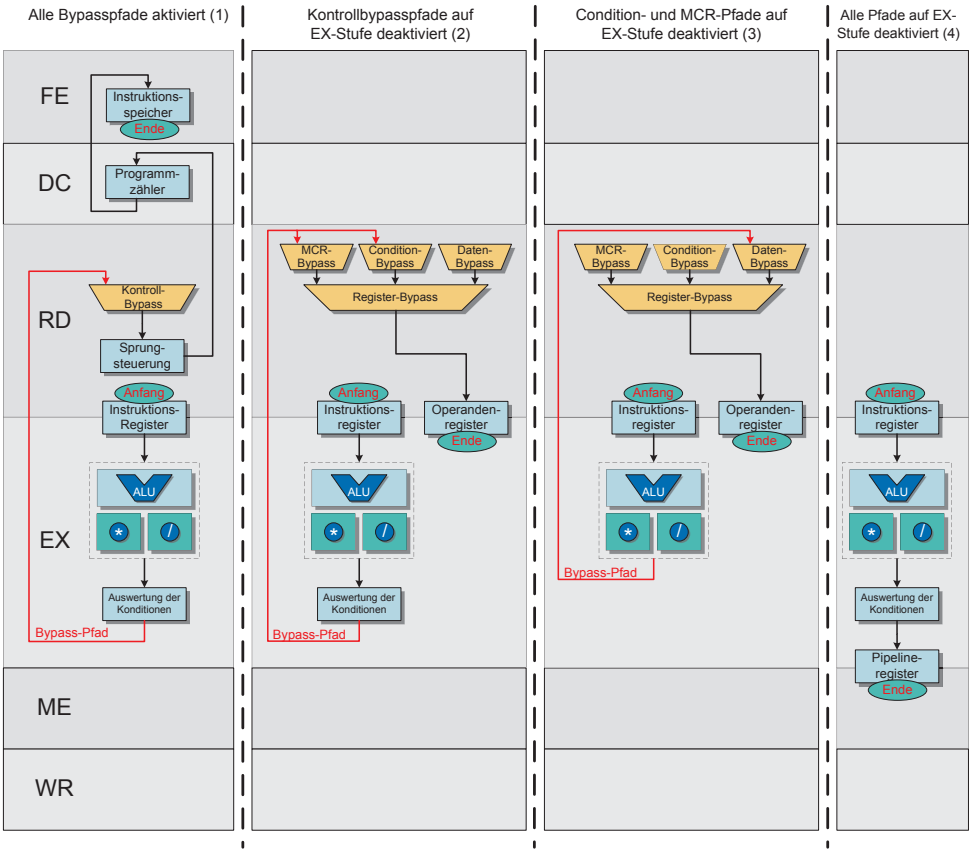


Abbildung 6.11.: Verlauf des kritischen Pfades des CoreVA-Prozessors für verschiedene Bypass-Konfigurationen

Der kritische Pfad eines des CoreVA-Prozessors, bei dem alle Bypass-Komponenten aktiviert wurden, erreicht eine maximale Verzögerungszeit von 2,79 ns, was eine maximale Betriebsfrequenz von 358 MHz ermöglicht. Er beginnt in dem Instruktionsregister zwischen der Register-Read- und der Execute-Pipelinestufe. Vom Ausgang dieses Registers verläuft der kritische Pfad durch eine Vielzahl kombinatorischer

Logikbausteine, die Teil der Verarbeitungseinheiten sind. Das Ergebnis der Verarbeitungseinheiten wird durch die Komponenten zur Auswertung der Konditionen (*Condition Evaluation*) geführt. Über den *Kontroll-Bypass* kann diese Kondition in die Register-Read-Pipelinestufe zurückgeführt werden. Dort wird sie zur Auswertung der Sprungvorhersage benötigt. Die Sprungvorhersage ihrerseits bedingt den Programmzähler in der Instruction-Fetch-Pipelinestufe aus dem die Adresse der nächsten zu ladenden Instruktionsgruppe berechnet wird.

Um den kritischen Pfad zu verkürzen, werden in einem ersten Schritt die Bypass-Pfade des Kontroll-Bypasses auf die Zwischenergebnisse der Execute-Pipelinestufe deaktiviert. Da die Verarbeitungseinheiten des CoreVA-Prozessors parallel aufgebaut sind und somit jeweils einen ähnlichen kritischen Pfad bilden, müssen hierbei die Bypass-Pfade aller Kontroll-Bypass-Instanzen auf die Execute-Zwischenergebnisse aller vier VLIW-Slots deaktiviert werden. Wie zuvor verläuft der kritische Pfad nun vom Instruktionsregister bis zum Ausgang der Komponente zur Auswertung der Konditionen. Die Konditionen werden nun über den MCR-Bypass und den Condition-Register-Bypass auf den Register-Bypass gelegt und damit in die Operandenregister zwischen der Register-Read- und Execute-Pipelinestufe geschrieben. Die Auswertung der Sprungvorhersage wirkt sich nun nicht länger auf den kritischen Pfad aus. Die Deaktivierung des Kontroll-Bypasses ermöglicht eine Reduzierung des kritischen Pfades um 18,6 % auf 2,27 ns und damit eine Erhöhung der maximalen Taktfrequenz auf 441 MHz.

Wird in einem nächsten Schritt der *MCR-* und der *Condition-Bypass* deaktiviert, so verläuft der kritische Pfad ausschließlich durch den Daten-Register-Bypass und endet wiederum in den Operandenregistern zwischen der Register-Read- und Execute-Pipelinestufe. Die Auswertung der Konditionen in der Execute-Pipelinestufe trägt nun nicht mehr zum kritischen Pfad bei. Durch die Deaktivierung des MCR- und des Condition-Register-Bypasses sinkt die Verzögerungszeit des kritischen Pfades um 7 % auf 2,11 ns (474 MHz).

Durch das Deaktivieren des *Daten-Register-Bypasses*, der auf Zwischenergebnisse der Execute-Pipelinestufe zugreift, können sämtliche Bypass-Komponenten aus dem kritischen Pfad entfernt werden. Der kritische Pfad verläuft nun geradlinig vom Instruktionsregister, über die Verarbeitungseinheiten und die Komponenten zur Auswertung der Konditionen bis zum Pipelineregister der Konditionen zwischen Execute-und Memory-Access-Pipelinestufe. Hierbei erreicht das System eine maximale Betriebsfrequenz von 488 MHz bei einer Verzögerungszeit von 2,05 ns ($-3\,\%$). Da der kritische Pfad nun maßgeblich durch die Verarbeitungseinheiten geprägt ist, hat das Deaktivieren der *Bypass-Pfade für die Memory-Access- oder Register-Write-Pipelinestufen* keinen direkten Einfluss auf die Verzögerungszeit. Durch das Wegfallen dieser Pfade und die daraus resultierende Reduzierung der kombinatorischen Logik reduziert sich jedoch die Gesamtfläche des Systems, was zu geringeren Leitungsverzögerungen führt. Daher lässt sich die maximale Taktfrequenz durch Entfernung

sämtlicher Bypass-Komponenten auf 498 MHz (+2 %) erhöhen. Im Vergleich zu einer Prozessorkonfiguration, bei der alle Bypass-Pfade aktiviert sind, verkürzt sich der kritische Pfad also um 28 %. Tabelle 6.1 fasst die Ergebnisse für die maximale Taktfrequenz der speziellen Bypass-Konfigurationen zusammen. Die Konfigurationen werden im Folgenden auch *Spezialfall (1–4)* genannt.

Tabelle 6.1.: *Maximale Taktfrequenz für spezielle Bypass-Konfigurationen*

Bypass-Konfiguration	T_{krit}	f_{max}	Änderung
(0) Alle Bypässe an	2,79 ns	358 MHz	±0 %
(1) Kontroll-Bypass (EX) aus	2,27 ns	441 MHz	+23,2 %
(2) MCR/Cond.-Reg.-Bypass (EX) aus	2,11 ns	474 MHz	+32,4 %
(3) Daten-Register-Bypass (EX) aus	2,05 ns	488 MHz	+36,3 %
(4) Alle Bypässe aus	2,01 ns	498 MHz	+39,1 %

6.3.1. Bestimmung einer optimierten Bypass-Konfiguration

Die Analyse des kritischen Pfades für verschiedene Bypass-Konfigurationen zeigt, dass die maximale Taktfrequenz durch das Abschalten bestimmter Bypass-Pfade deutlich erhöht werden kann. Da die Zeit und die Energie, die zur Verarbeitung eines Algorithmus benötigt werden, direkt von der Länge des kritischen Pfades abhängen, werden sie hierdurch ebenfalls reduziert. Eine vollständige Deaktivierung aller Bypässe ist jedoch nicht sinnvoll. Zur Bewertung der Performanz muss auch die Anzahl hinzukommender Strafzyklen berücksichtigt werden, die die absolute Verarbeitungszeit beeinflussen. Abhängig von der Art und Menge der deaktivierten Bypass-Pfade steigt jedoch die Anzahl der hinzukommenden Strafzyklen, wodurch die Verarbeitungszeit wiederum ansteigt (vgl. Gleichung 6.1). Die Deaktivierung von Bypass-Pfaden stellt also einen Kompromiss zwischen der Verkürzung des kritischen Pfades und der Erhöhung der Anzahl an Strafzyklen dar.

$$Verarbeitungszeit = \frac{Taktzyklen}{Taktfrequenz} = Taktzyklen \cdot Kritischer\ Pfad \tag{6.1}$$

Ziel dieses Abschnittes ist es, durch systematisches Abschalten einzelner Bypass-Pfade ein Design (vgl. Abschnitt 3.2.2) zu identifizieren, dessen Bypass-Kombination eine möglichst *kurze Verarbeitungszeit* beziehungsweise eine möglichst *hohe Energieeffizienz* ermöglicht. Folgt man dem *Greedy*-Ansatz bei der Auswahl der Bypass-Konfiguration bedeutet dieses, dass die bisherigen Deaktivierungen den kritischen Pfad verkürzen konnten, ohne die Verarbeitungszeit beziehungsweise die Energieeffizienz zu verschlechtern. Alle Bypass-Komponenten, die nach diesem Zustand

deaktiviert werden, können zwar weiterhin den kritischen Pfad verkürzen, durch die Vielzahl an hinzukommenden Strafzyklen steigt jedoch die Verarbeitungszeit oder die benötigte Energie.

Zur Bestimmung der optimierten Bypass-Konfiguration wurde ein Algorithmus entwickelt, der die Konfiguration durch iterative Simulationsdurchläufe und der damit verbundenen Bestimmung der Bypass-Auslastung ermittelt. Der Algorithmus beginnt mit der Analyse der Bypass-Auslastung bei vollständig aktivierten Bypass-Systemen. Der Algorithmus deaktiviert in einzelnen Optimierungsschritten jeweils den Bypass-Pfad, der die geringste Auslastung aufweist (rechter Abschnitt des Flussdiagramms in Abbildung 6.12).

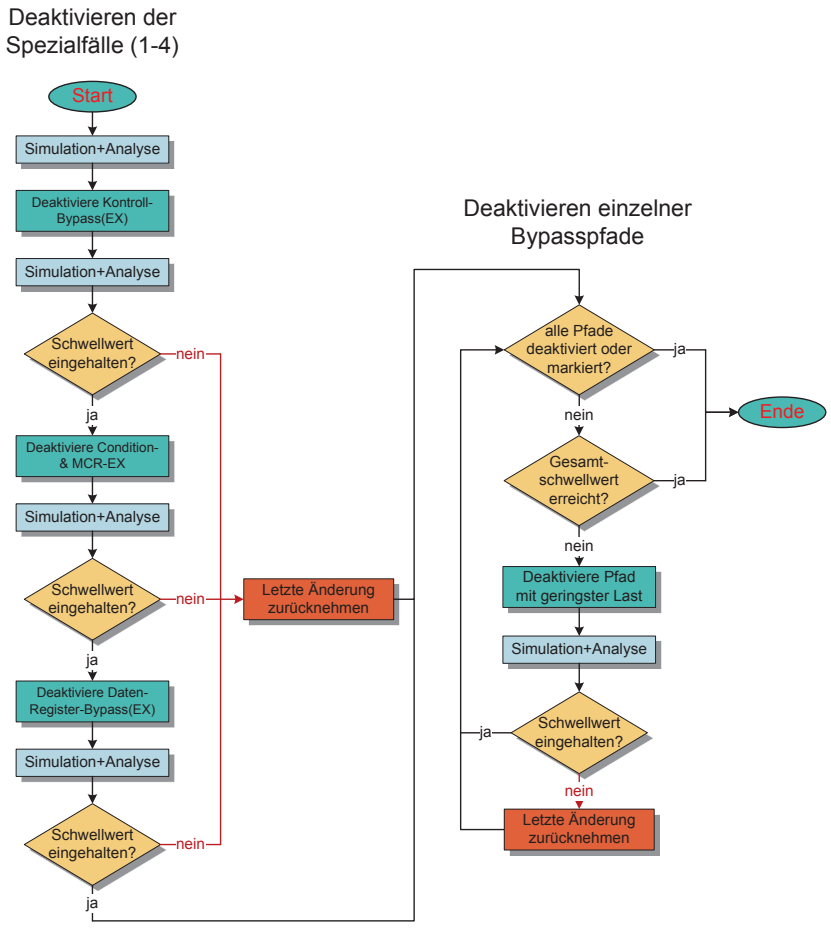


Abbildung 6.12.: Flussdiagramm des Algorithmus zur Bestimmung der optimierten Bypass-Konfiguration

Nach jedem Optimierungsschritt erfolgt eine *erneute Analyse* des Bypasses, durch die die *hinzukommenden Strafzyklen* bestimmt werden. Abhängig von der Anzahl der Strafzyklen übernimmt der Algorithmus entweder die neue Bypass-Konfiguration oder verwirft den letzten Optimierungsschritt. Hierzu werden prozentuale Schwellwerte genutzt, die detailliert in Kapitel 6.3.2 erläutert werden. Wie im vorigen Abschnitt gezeigt wurde, hat das Abschalten spezieller Bypass-Konfigurationen (Spezialfall 1–3) deutlich größere Auswirkungen auf den kritischen Pfad als das Deaktivieren einzelner Bypass-Pfade. Als Erstes analysiert der Algorithmus daher die Spezialfälle (1–3). Anschließend fährt er mit dem zuvor beschriebenen systematischen Deaktivieren der einzelnen Bypass-Komponenten fort.

6.3.2. Wahl geeigneter Schwellwerte zur Deaktivierung von Bypass-Komponenten

Bei der Wahl der geeigneten Schwellwerte können unterschiedliche Ziele verfolgt werden. Diese Ziele werden im Folgenden in den zwei Hauptgruppen *Ressourcenoptimierung* und *Zeit- beziehungsweise Energieoptimierung* zusammengefasst.

6.3.2.1. Ressourcenoptimierung

Bei der Ressourcenoptimierung sollen *möglichst viele Bypass-Komponenten* abgeschaltet werden, ohne die *Gesamtverarbeitungszeit* zu erhöhen. Dadurch soll insbesondere der Flächenbedarf des Systems und gleichzeitig durch die reduzierte Logik auch der Energiebedarf optimiert werden.

Die hierzu benötigten Schwellwerte können direkt der Definition des Optimierungslimits (vgl. Formel 6.2) entnommen werden, bei dem durch den jeweiligen Optimierungsschritt die Verarbeitungszeit nicht schlechter werden darf als zuvor. Der Schwellwert, in diesem Fall der maximale prozentuale Zuwachs der Prozessortakte, kann anschließend mit Hilfe von Formel 6.3 bestimmt werden. Die Schwellwerte aus Tabelle 6.2 orientieren sich daher direkt an der maximalen Taktfrequenz der Spezialfälle (1–4) (vgl. Tabelle 6.1).

Aus diesem Grund kann das anschließende systematische Deaktivieren *einzelner* Bypass-Pfade so lange durchgeführt werden, bis die durch die vollständige Deaktivierung der Spezialfälle erzielte Steigerung der maximalen Taktfrequenz durch hinzukommende Strafzyklen wieder „aufgebraucht“ ist.

$$Optimierungslimit : Zeit_{vorher} = Zeit_{nachher} \tag{6.2}$$

$$Schwellwert[\%] = \frac{Takte_{nachher} \cdot 100}{Takte_{vorher}} = \frac{krit.Pfad_{vorher} \cdot 100}{krit.Pfad_{nachher}} \tag{6.3}$$

Da durch diese Optimierungen möglichst viele Bypass-Komponenten deaktiviert werden sollen, werden gegebenenfalls auftretende „Zeitreserven“ in die Iterationsschritte des Algorithmus übernommen (beispielsweise von Spezialfall (1) nach Spezialfall (2)). Falls beispielsweise durch das Abschalten des Kontroll-Bypasses nicht die maximale Anzahl an zusätzlichen Takten (bis zum Erreichen des Schwellwertes) benötigt wird, kann diese Zeitreserve genutzt werden, um die Grenze beim Deaktivieren von Komponenten des MCR-Bypasses anzuheben.

Tabelle 6.2.: *Schwellwerte der Ressourcenoptimierung*

Spezialfall (1)	**Spezialfall (2)**	**Spezialfall (3)**	**Abbruchschwellwert (4)**
122,91%	132,23%	136,10%	138,81%

6.3.2.2. Zeit- und Energieoptimierung

Die Zeit- beziehungsweise Energieoptimierung zielt darauf ab, durch das Abschalten der Bypass-Komponenten die *Ausführungszeit* und damit implizit den *Energieverbrauch* zu minimieren. Im Unterschied zur Ressourcenoptimierung werden eventuelle Zeitreserven nicht in die folgenden Optimierungsstufen übernommen. Die Schwellwerte beziehen sich somit nicht auf die *ursprüngliche Version* mit vollständig aktivierten Bypass-Pfaden, sondern auf den *vorausgegangenen Durchlauf* jedes Optimierungsschrittes. Durch diese Festlegung wird das Optimierungslimit nochmals verschärft. Das System muss nach jedem Optimierungsschritt nicht nur mindestens so schnell bzw. mindestens so energieeffizient wie das *ursprüngliche System* sein, es darf zusätzlich auch nicht schlechter als der *vorangegangene Optimierungsschritt* werden.

Die Schwellwerte der Zeitoptimierung (vgl. Tabelle 6.2) können erneut direkt aus den Ergebnissen für den kritischen Pfad aus Tabelle 6.1 bestimmt werden.

Tabelle 6.3.: *Schwellwerte der Zeitoptimierung*

Spezialfall (1)	**Spezialfall (2)**	**Spezialfall (3)**	**Abbruchschwellwert (4)**
122,91%	107,58%	102,93%	101,99%

Da das Optimierungslimit der Energieoptimierung (vgl. Formel 6.4) jedoch durch die aufgewendete Energie beschrieben wird, werden ihre Schwellwerte mit Hilfe von Formel 6.5 ermittelt. Hierbei wird zusätzlich zur Reduktion des kritischen Pfades auch die Leistungsaufnahme berücksichtigt.

$$Optimierungslimit : Energie_{vorher} = Energie_{nachher} \tag{6.4}$$

$$Schwellwert[\%] = \frac{Takte_{nachher} \cdot 100}{Takte_{vorher}} = \frac{Leistung_{vorher} \cdot krit.Pfad_{vorher} \cdot 100}{Leistung_{nachher} \cdot krit.Pfad_{nachher}} \tag{6.5}$$

Da die Leistungsaufnahme bei der Verarbeitung der verschiedenen Anwendungen aufgrund unterschiedlicher Schaltaktivitäten variiert, ergeben sich für die Energieoptimierung jeweils unterschiedliche Schwellwerte (siehe Tabelle 6.4).

Tabelle 6.4.: *Schwellwerte der Energieoptimierung*

Anwendungen	**Spezialfall (1)**	**Spezialfall (2)**	**Spezialfall (3)**	**Abbruchschwellwert (4)**
Dhrystone	122,36%	105,71%	110,11%	125,30%
Coremark	122,91%	106,17%	107,16%	129,86%
Viterbi	121,89%	105,40%	110,09%	128,89%
Faltung (C)	121,01%	105,97%	109,59%	125,33%
Faltung (ASM)	121,01%	106,38%	112,55%	130,00%
FFT	116,89%	104,47%	125,80%	157,15%
IEEE 802.11b	120,05%	104,80%	115,65%	142,37%
CRC	124,01%	105,22%	111,22%	125,85%
ECC	118,50%	105,07%	120,49%	145,06%
AES	122,42%	105,49%	114,51%	132,36%
SNOW	123,93%	105,40%	107,74%	130,26%
SATD	120,73%	104,62%	108,94%	135,00%
Fibonacci	130,11%	104,52%	113,54%	124,28%

6.3.3. Analyse der ermittelten Bypass-Kombinationen

Zur Analyse der Ergebnisse des Optimierungsalgorithmus wurden die verschieden Bypass-Konfigurationen mit Hilfe des in Abschnitt 6.1.2 vorgestellten Verfahrens bezüglich der Auslastung der einzelnen Bypass-Pfade untersucht. Abbildung 6.13 zeigt die Auslastung des Daten-Register-Bypasses bei der Verarbeitung des IEEE-802.11b-Algorithmus nach Anwendung der Zeitoptimierung. Vergleicht man das Diagramm mit der ursprünglichen Auslastung bei vollständig implementiertem Bypass (vgl. Abbildung 6.1) so erkennt man, dass viele Balken, die eine sehr geringe Auslastung darstellen, in der optimierten Version verschwunden sind. Diese Balken repräsentieren die deaktivierten Bypass-Pfade.

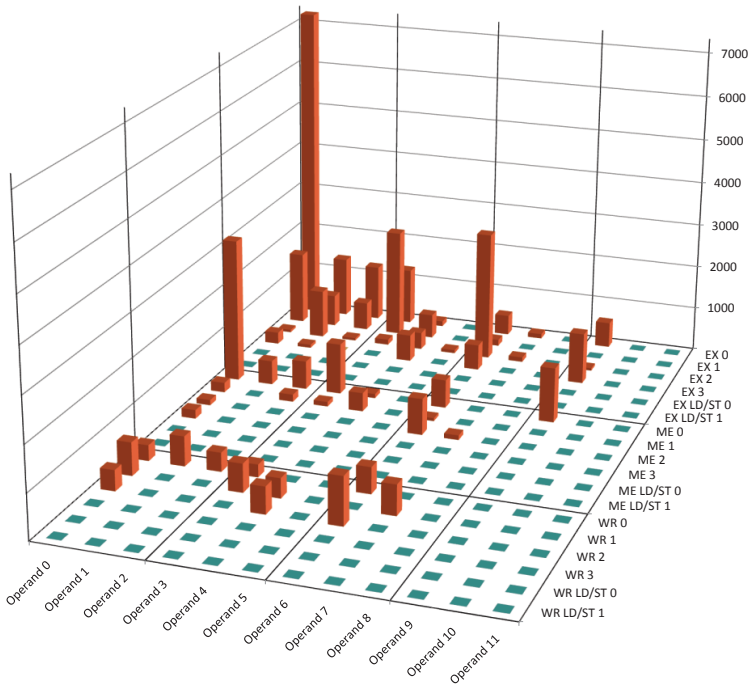


Abbildung 6.13.: Daten-Register-Bypass-Zugriffe des IEEE-802.11b-Algorithmus nach Zeitoptimierung

Insgesamt konnten für den IEEE-802.11b-Algorithmus 386 und damit 85 % aller Bypass-Pfade deaktiviert werden. Von den verbleibenden 70 Bypass-Pfaden sind 56 Pfade (80 %) Bestandteil des Daten-Register-Bypasses. Diese Pfade zeigen zu 50 % auf die Execute-Pipelinestufe, zu 29 % auf die Memory-Access-Pipelinestufe und zu 21 % auf die Register-Write-Pipelinestufe. Kontroll-Bypass (Spezialfall (1)), Condition-Register-Bypass/MCR-Bypass (Spezialfall (2)) und MLA-Bypass konnten vollständig deaktiviert werden. Da für die Energie- bzw. Ressourcenoptimierung höhere Schwellwerte angesetzt wurden, steigt hier die Anzahl der deaktivierten Bypass-Pfade auf 410 (90 %). Bei den betrachteten Anwendungen zeigte sich für den Daten-Register-Bypass ein ähnliches Ergebnis. Der Kontroll-Bypass konnte für fast alle Anwendungen vollständig deaktiviert werden. Obwohl durch das Abschalten dieser Zugriffe vergleichsweise viele Strafzyklen eingefügt werden, kann durch die enorme Verkürzung des kritischen Pfades (vgl. Tabelle 6.1) eine geringere Verarbeitungszeit erreicht werden. Lediglich für den Fibonacci-Algorithmus ist der Kontroll-Bypass dagegen zwingend erforderlich, damit die Gesamtperformanz nicht reduziert wird. Der MLA-Bypass findet insbesondere beim Faltungsalgorithmus (ASM) Verwendung,

kann jedoch in allen anderen Anwendungen vollständig deaktiviert werden.

Abbildung 6.14 zeigt den Anteil der deaktivierten Bypässe für die verschiedenen Optimierungsstrategien. Die Ergebnisse der Energie- und Ressourcenoptimierung zeigen große Ähnlichkeiten bezüglich des Anteils der deaktivierten Bypass-Komponenten, da bei beiden Optimierungsstrategien vergleichsweise hohe Schwellwerte definiert sind. Der Dhrystone-Benchmark stellt hier eine Ausnahme dar, da durch den Optimierungsalgorithmus bei der Energieoptimierung mehr Bypass-Komponenten deaktiviert werden, als bei der Ressourcenoptimierung. Insgesamt liegt die Anzahl deaktivierter Bypass-Komponenten zwischen 80 % (SATD, Zeitoptimierung) und über 99 % (Fibonacci, Zeit-, Energie- und Ressourcenoptimierung). Wie in Abschnitt 6.1.2 gezeigt wurde, ist die Bypass-Auslastung abhängig vom ILP der Anwendung. Es zeigt sich, dass ebenso die Anzahl der durch den Algorithmus deaktivierten Bypass-Komponenten in Relation zum ILP steht. Für Anwendungen wie beispielsweise SATD oder FFT (höchste Parallelität) lassen sich nur wenige Bypass-Komponenten deaktivieren. Bei Anwendungen wie Coremark oder Dhrystone (geringer ILP) sind es dagegen vergleichsweise viele Bypass-Komponenten. Beim Faltungsalgorithmus werden identische Berechnungen mehrfach wiederholt, weswegen hier sehr viele Bypass-Pfade ungenutzt bleiben und daher deaktiviert werden können.

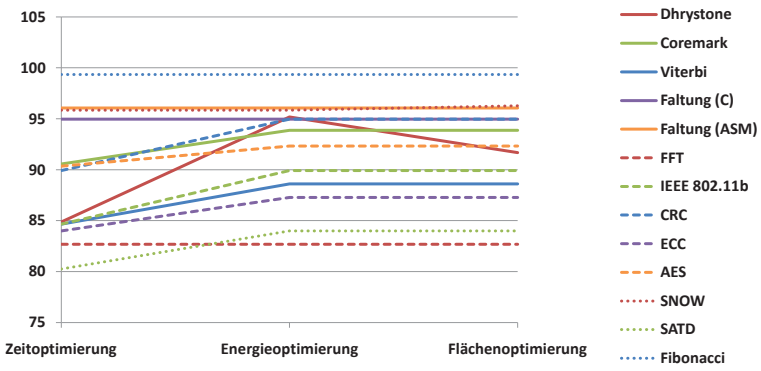


Abbildung 6.14.: Anteil der deaktivierten Bypass-Komponenten (in %)

6.3.4. Auswertung der Ressourceneffizienz der optimierten Bypass-Konfigurationen

In diesem Abschnitt wird der *Ressourcenbedarf* eines Systems mit optimierter Bypass-Konfigurationen in Bezug zur *Gesamtlaufzeit* der Anwendungen gesetzt. Hierbei wird zum einen die mögliche Erhöhung der Taktfrequenz, zum anderen aber auch eine Erhöhung der Anzahl an Taktzyklen berücksichtigt. Die Ressourceneffizienz wird beispielhaft für einen Ressourceneffizienzindex $RE = 1/(P \cdot T)$ (Energieeffizienz)

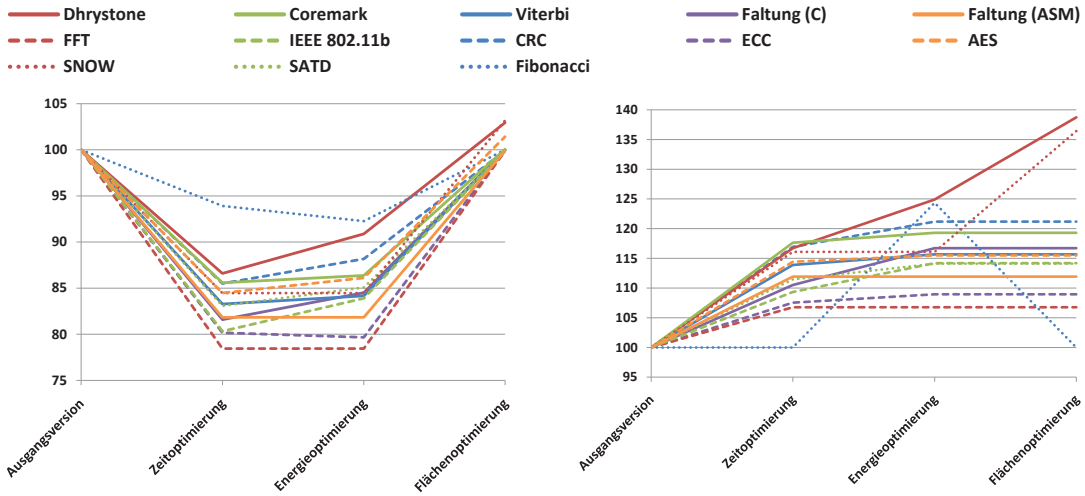


Abbildung 6.15.: Verarbeitungszeit [%]

Abbildung 6.16.: Prozessortakte [%]

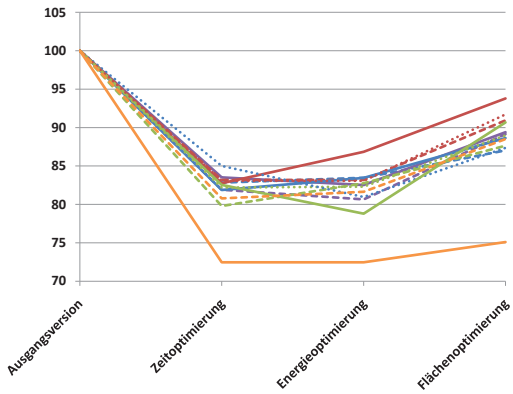


Abbildung 6.17.: Energie [%]

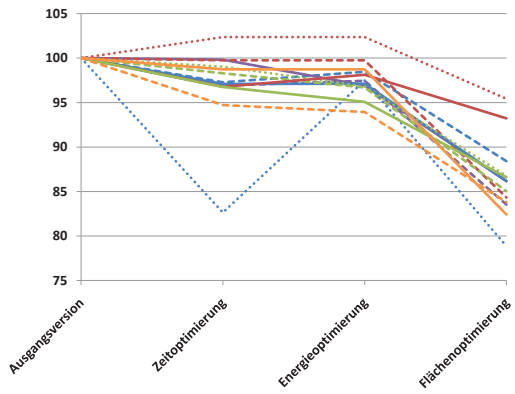


Abbildung 6.18.: Fläche [%]

bestimmt (vgl. Kapitel 5). Die Abbildungen 6.15 bis 6.18 zeigen die Resultate der Auswertung der optimierten Bypass-Konfigurationen für die Verarbeitungszeit, die Anzahl der Prozessortakte, den Energiebedarf und die Fläche.

Der in Abbildung 6.15 dargestellte Vergleich der Verarbeitungszeiten zeigt, dass die höchste Zeitersparnis jeweils durch den Einsatz der zeitoptimierten Bypass-Konfigurationen erreicht werden kann. Die Reduzierungen der Verarbeitungszeit liegen hierbei zwischen 7 % und 22 % (FFT). Ein Vergleich dieser Zeiten mit der Instruktionsverteilung in Abbildung 6.2 zeigt, dass die Zeitersparnis mit dem Parallelitätsgrad der Anwendung ansteigt. Da die Bypass-Instanzen der „hinteren“ VLIW-Slots bei höheren Parallelitätsgraden häufiger genutzt werden, ist auch die durchschnittliche Anzahl paralleler Bypass-Zugriffe höher. Durch Inkaufnahme eines Strafzyklus lassen sich somit vergleichsweise viele Bypass-Komponenten deaktivieren, weshalb die

Gesamtanzahl der hinzuzufügenden Prozessortakte nicht so stark ansteigt, wie bei Anwendungen mit geringem Parallelitätsgrad. Die Verarbeitungszeit der Anwendungen bei der energieoptimierten Bypass-Konfigurationen liegt leicht über derer der zeitoptimierten Konfigurationen. Sie liegt aber bei allen Anwendungen unterhalb der Ausführungszeit der Ausgangsversion. Dieses zeigt, dass die *Energieeinsparungen* hauptsächlich durch die *Reduktion der Ausführungszeit* und nur im geringen Maße durch die Verwendung weniger Ressourcen (durch Abschalten zusätzlicher Bypass-Komponenten) erreicht werden.

Die Verarbeitungszeit der Anwendungen bei der Ressourcenoptimierung liegt erwartungsgemäß bei 100 %. Die ressourcenoptimierende Version des Algorithmus balanciert hinzukommende Strafzyklen (vgl. Abbildung 6.16) und die Erhöhung der maximalen Taktfrequenz gegeneinander aus. Das Resultat ist insbesondere eine Reduktion der benötigten Fläche (vgl. Abbildung 6.18). Für eine Architektur mit vollständig deaktivieren Bypass-Systemen konnte ein kritischer Pfad von 2,01 ns erreicht werden (vgl. Tabelle 6.1). Für die Ressourcenoptimierung ist der Abbruchschwellwert daher auf 138,81 % festgelegt worden (vgl. Tabelle 6.2). Bei der Bestimmung der optimalen Bypass-Konfiguration für den Dhrystone-Benchmark konnte eine Konfiguration gefunden werden, die diese Grenze mit einem Anstieg der Prozessortakte auf 138,73 ns exakt erreicht. Da bei der Synthese genau dieser Konfiguration jedoch nur ein vergleichsweise schlechter kritischer Pfad mit 2,07 ns (+3 %) erreicht werden konnte, kommt es zu einer Überschreitung der geforderten Verzögerungszeit. Diese Tatsache ist auch auf die Verwendung von pseudozufälligen Sequenzen in modernen Werkzeugen zur Standardzellensynthese zurückzuführen. Ähnliche Effekte zeigen sich auch beim SNOW- und beim AES-Algorithmus.

Die Analyse des Energiebedarfs in Abbildung 6.17 zeigt, dass durch Einsatz einer energieoptimierten Bypass-Konfiguration Energieersparnisse von 13 % (Dhrystone) und bis zu 28 % (Faltung (ASM)) möglich sind. In wenigen Fällen (insbesondere beim Dhrystone-Benchmark) zeigt sich jedoch bei der *Zeitoptimierung* eine um bis zu 4 % *höhere Energieeffizienz*. Betrachtet man die Anzahl deaktivierter Bypass-Pfade für die beiden Optimierungsversionen (vgl. Abbildung 6.14) so zeigt sich, dass der Algorithmus bei der Energieoptimierung weitaus mehr Bypass-Komponenten deaktiviert, als bei der Zeitoptimierung. Der Einfluss der dadurch hinzukommenden Strafzyklen ist allerdings höher als die Reduktion der Leistungsaufnahme durch die Deaktivierung der zusätzlichen Pfade. In Einzelfällen ermöglicht eine mäßige Deaktivierung von Bypass-Komponenten und somit eine geringere Anzahl hinzukommender Strafzyklen sogar eine höhere Energieeffizienz als eine aggressivere Reduktion der mit dem Bypass verbundenen Logik.

Abbildung 6.18 zeigt den Flächenbedarf des CoreVA-Prozessors in Abhängigkeit der verschiedenen Optimierungsvarianten. Es ist wichtig zu bedenken, dass eine *Reduktion von Bypass-Komponenten* nicht automatisch mit einer *Reduktion der Chipfläche* einhergeht. Die Effizienz des Gesamtsystems kann bei einer Deaktivierung von

Bypass-Komponenten nur gesteigert werden, indem die Taktfrequenz angehoben wird. Die Erhöhung der Taktfrequenz führt aber dazu, dass das Synthesewerkzeug Gatter mit höherer Treiberstärke auswählen muss, um die höhere Taktfrequenz zu erreichen. So kann eine *Reduktion der Logik* auch eine *Erhöhung der Chipfläche* ergeben. Die Implementierung der ressourcenoptimierten Bypass-Konfigurationen ermöglicht im Vergleich zur Ausgangsversion eine Flächenreduzierung zwischen 4 % (SNOW) und 21 % (Fibonacci). Beim Faltungsalgorithmus (ASM) ergibt sich für die Zeit- und Energieoptimierung ein 2 % höherer Flächenbedarf. Bei allen anderen Anwendungen kann auch durch die Zeit- und Energieoptimierung Chipfläche eingespart werden.

6.4. Dynamische Rekonfiguration des Pipeline-Bypasses

Durch die zuvor beschriebene statische Konfiguration des Pipeline-Bypasses durch Bestimmung der jeweiligen optimierten Bypass-Konfigurationen werden ausschließlich anwendungsspezifisch optimierte Prozessoren erzeugt. Falls diese Prozessoren jedoch zur Verarbeitung anderer Anwendungen eingesetzt werden, kann die gewählte Bypass-Konfiguration die Effizienz des Gesamtsystems deutlich verschlechtern. Zur Vermeidung dieses Problems wurden die Bypass-Systeme des CoreVA-Prozessors durch dynamisch rekonfigurierbare Komponenten erweitert. Diese Komponenten ermöglichen das nachträgliche Deaktivieren oder Umkonfigurieren der implementierten Bypass-Pfade. Das Übertragen der neuen Konfigurationen erfolgt hierbei durch eine Instruktionssatzerweiterung, die in den Programmcode der Anwendungen eingefügt wird. Das dynamische Deaktivieren bestimmter Pfadgruppen (insbesondere der Spezialfälle (1–4)) während des Betriebs reduziert den kritischen Pfad des CoreVA-Prozessors, so dass die Taktfrequenz *zur Laufzeit* angehoben werden kann. Hierzu werden bei der Standardzellensynthese mehrere Zielfrequenzen für die möglichen Pfadgruppen definiert. Das Synthesewerkzeug kann dann alle Pfadgruppen getrennt optimieren (*Multi-Mode Multi-Corner*-Entwurfsablauf [130]). Des Weiteren erlaubt die dynamische Rekonfiguration einzelner Bypass-Pfade zudem die Implementierung *fehlertoleranter* Bypass-Systeme. Wie in Abschnitt 4.4 gezeigt wurde, verarbeiten die Multiplexer eines vollständig aktivierten Bypass-Systems Eingangswerte mit einer Gesamtbreite von 11008 Bit. Diese Eingangswerte werden zu einem Großteil von Bypass-Pfaden bezogen, die sich durch die VLIW-Architektur über die komplette Breite des Prozessorkerns erstrecken können. Aus diesem Grund ist die Möglichkeit, dass während des Fertigungsprozesses intrinsische Produktionsfehler bei einem dieser Bypass-Pfade auftreten, vergleichsweise hoch [175]. Um die Funktionalität des Prozessors trotz defekter Bypass-Pfade zu gewährleisten, können diese Pfade zu Beginn der Anwendungsverarbeitung deaktiviert werden [230]. Falls diese Pfade anschließend zur Auflösung von Datenkonflikten benötigt werden, wird die Verzögerungsfunktionalität angesprochen, die diese Datenkonflikte durch das Einfügen von

Strafzyklen auflöst. Die Untersuchung der Fehlertoleranz des CoreVA-Systems soll jedoch nicht Gegenstand dieser Arbeit sein. Im Folgenden wird nur der Einfluss der dynamischen Rekonfiguration der Architektur auf die Steigerung der Performanz des Gesamtsystems untersucht.

Die Erweiterung der Architektur erfolgt im Wesentlichen durch die Implementierung zusätzlicher Register zur Speicherung der aktuellen Bypass-Konfiguration (insgesamt 456 Bit für alle 456 Bypass-Komponenten). Durch das Hinzufügen des vergleichsweise großen Konfigurationsregister und der kombinatorischen Logik zum Beschreiben der Konfigurationsregister in der Decode-Pipelinestufe steigt der Flächenbedarf und die Leistungsaufnahme um etwa 5 %.

6.5. Zusammenfassung

Die Analyse der Bypass-Auslastung bei Ausführung der verschiedenen Anwendungen hat gezeigt, dass ein Großteil der implementierten Bypass-Komponenten nur selten genutzt wird. Standardzellensynthesen haben ergeben, dass der kritische Pfad des CoreVA-Prozessors durch das Deaktivieren bestimmter Bypass-Gruppen um bis zu 28 % verkürzt werden kann. Die Deaktivierung von Bypass-Pfaden stellt einen Kompromiss zwischen der Erhöhung der maximalen Taktfrequenz und dem Hinzufügen von Strafzyklen dar, weswegen eine vollständige Deaktivierung des Bypass-Systems nicht zielführend ist. Daher wurde ein *Greedy*-basierter Algorithmus entwickelt, der durch das systematische Abschalten einzelner Bypass-Komponenten optimierte Bypass-Konfigurationen ermittelt. Die abschließenden Auswertungen zeigten, dass durch das Anwenden dieses Algorithmus 80 % bis 95 % aller Bypass-Pfade deaktiviert werden können. Durch die damit verbundene Reduktion des kritischen Pfades um bis zu 26 %, sinkt die Verarbeitungszeit und die aufzuwendende Energie um bis zu 21,5 %. Der Prozessorkern wurde anschließend um eine zusätzliche Instruktion erweitert, welche die dynamische Rekonfiguration des Pipeline-Bypasses ermöglicht. Die dynamische Rekonfiguration des Pipeline-Bypasses ermöglicht eine Anpassung der Betriebsfrequenz zur Laufzeit. Anwendungen mit geringer Bypass-Auslastung können so mit einer deutlich höheren Taktfrequenz ausgeführt werden.

In den Kapiteln 5 und 6 wurde eine Entwurfsraumexploration des in Kapitel 4 vorgestellten VLIW-Prozessors CoreVA durchgeführt. Im folgenden Kapitel wird dieser Prozessorkern nun in eine Systemumgebung integriert. Die Entwurfsraumexploration wird daher auf die Komponenten der Systemebene, wie Caches, Scratchpad-Speicher und Hardware-Beschleuniger, erweitert.

7. Entwurfsraumexploration auf Systemebene

In Kapitel 4 wurde eine modulare, konfigurierbare VLIW-Architektur entwickelt. Basierend auf dieser Architektur wurde in Kapitel 5 eine Entwurfsraumexploration der Hardware-Architektur und Algorithmen verschiedener Anwendungsklassen durchgeführt. Ausgehend von den Ergebnissen der Entwurfsraumexploration wurde die vierfach-parallele VLIW-Architektur als Referenz für die weiteren Untersuchungen definiert. In Kapitel 6 wurde die statische Konfiguration der funktionalen Parallelität um die statische und dynamische (Re-)Konfiguration des Pipeline-Bypasses erweitert.

Für die Ressourceneffizienz eines gesamten *Prozessorsystems* ist neben dem *Prozessorkern* allerdings auch eine umfassende Betrachtung der weiteren *Systemkomponenten* erforderlich. Insbesondere die Art der *Speicheranbindung* hat einen großen Einfluss auf die Ausführungszeiten und den Ressourcenbedarf. Abschnitt 7.1 stellt das Speicher-Subsystem des CoreVA-Prozessors vor und führt eine Entwurfsraumexploration verschiedener Cache-Parameter durch. Abschnitt 7.2 erweitert das Speicherkonzept um eng angebundene schnelle *Scratchpad-Speicher* als *On-Chip-Speicher*. Abschnitt 7.3 zeigt die Möglichkeiten zur Steigerung der Ressourceneffizienz durch eng gekoppelte Hardware-Erweiterungen, wie Instruktionssatzerweiterungen sowie lose gekoppelte dedizierte Hardware-Beschleuniger. In Abschnitt 7.7 werden die erzielten Ergebnisse zusammengefasst und die Möglichkeiten der Steigerung der Ressourceneffizienz diskutiert.

7.1. Entwurfsraumexploration des Speicher-Subsystems

Für eine Entwurfsraumexploration des *Speicher-Subsystems* ist das Verständnis einiger Grundlagen von *Caches* vonnöten. Daher werden im Folgenden die wichtigsten Begriffe kurz eingeführt.

7.1.1. Speicherarchitekturen

Mikroprozessoren verfügen nur über eine begrenzte Anzahl an lokalen Registern. Um auf Programmcode und große Datenmengen zugreifen zu können, wird (externer) Speicher verwendet. Es stehen verschiedene Halbleiterspeicher zu diesem Zweck zur Verfügung. SRAM erreicht Taktfrequenzen ähnlich derer moderner Prozessoren und

ermöglicht daher eine enge Kopplung an das System. Andererseits nimmt SRAM jedoch viel Chipfläche ein, benötigt viel Energie und kann aus diesem Grund nur im begrenzten Maße verwendet werden. Dynamischer Speicher (Dynamic Random Access Memory (DRAM)) ist wesentlich langsamer als SRAM, benötigt jedoch pro Speicherbit wesentlich weniger Siliziumfläche. Wie bereits in Abschnitt 4.1 motiviert, liegt dem CoreVA-Prozessor eine Harvard-Architektur mit getrenntem Instruktion- und Datenspeicher zugrunde.

7.1.2. Grundlagen von Cache

Zum leichteren Verständnis des in dieser Arbeit implementierten Speicher-Subsystems und zur Definition der verwendeten Begriffe werden in diesem Abschnitt einige Grundlagen zu Caches vorgestellt. Nach [147] steigt die Leistungsfähigkeit von Prozessoren dem *Mooreschen Gesetz* folgend pro Jahr um den Faktor 1,8. Diese Wachstumsraten sind bei DRAM nicht möglich. Die Zugriffsgeschwindigkeit steigt bei DRAM-Chips pro Jahr nur um ca. 7% [139]. Durch die steigende Komplexität moderner Basisband- und Multimedia-Algorithmen benötigen eingebettete Systeme immer größere Mengen an Speicher. Auch der Einsatz von Betriebssystemen und komplexen Anwendungen erhöht den Speicherbedarf, der nicht mehr ausschließlich aus SRAM realisiert werden kann. Verhältnismäßig langsamer DRAM-basierter Speicher dient daher als Hauptspeicher. Dedizierte schnelle (SRAM-basierte) Instruktions- und Datenspeicher werden eng und meist synchron direkt auf dem Prozessor-DIE an den Prozessorkern gekoppelt. Die Größe dieser On-Chip-Speicher liegt meist im Bereich weniger Kilobytes. Die enge Kopplung und kurze Zugriffszeit von SRAM garantiert eine geringe und zugleich deterministische Latenz beim Zugriff auf die Speicher. Ist der Instruktions- oder Datenspeicher aber nicht ausreichend, so müssen Instruktionen und Daten zur Laufzeit zwischen Hauptspeicher und lokalem Speicher ausgetauscht werden. Durch die hohe, nicht deterministische Latenz beim Zugriff auf den externen Speicher muss der Prozessor-Kern während des Transfers verhältnismäßig lange angehalten werden. Um dieses Diskrepanz zwischen schnellem, lokalen und kleinem Speicher und langsamen, externen und großem Speicher zu umgehen, wird der lokale Speicher in modernen Prozessorarchitekturen als sogenannter Cache (Schattenspeicher) realisiert, der sich folgende Eigenschaften von Programmen zu nutze machen:

Zeitliche Lokalität. Nach einem Zugriff auf ein Element im Speicher ist es sehr wahrscheinlich, dass in kurzer Zeit wieder auf *dieses Element* zugegriffen wird. Aus diesem Grund wird das Element im Cache nach einem Zugriff zwischengespeichert. Somit können folgende Zugriffe schneller beantwortet werden, als wenn das Element erst aus dem Hauptspeicher geladen werden müsste.

Räumliche Lokalität. Greift ein Programm auf ein Element im Speicher zu, so ist die Wahrscheinlichkeit hoch, dass bald auf *ein Element in der Nähe dieses Elements* zugegriffen wird. Daher wird nicht nur das nachgefragte Element im Cache zwischengespeichert, sondern auch umliegende Daten. Ein Block aus Elementen, die zusammen in den Cache geladen werden, wird Cache-Zeile genannt. Anfragen nach benachbarten Elementen, die in einer Cache-Zeile liegen, können somit schnell aus dem Cache beantwortet werden. Üblicherweise kommen in eingebetteten Systemen ganze Cache-Hierarchien (vgl. Abbildung 7.1) zum Einsatz [76]. Höhere Cache-Hierarchiestufen fassen Instruktions- und Datenspeicher zusammen und stellen auch die Verbindung zu einem gemeinsamen DRAM-basierten Hauptspeicher zur Verfügung. Hier kann auch der Nachteil von Harvard-basierten Computerarchitekturen vermieden werden, dass es bei strikter Trennung von Instruktions- und Datenspeicher ansonsten nicht möglich ist, Programmcode direkt zur Laufzeit zu verändern (vgl. Abschnitt 4.1). Als höchste Hierarchiestufe verwenden moderne Betriebssysteme auf Personalcomputern Festplatten, auf denen nicht benötigte Bereiche des Hauptspeichers ausgelagert werden. Festplatten besitzen jedoch um Größenordnungen kleinere Datentransferraten und größere Latenzen im Vergleich zu Halbleiterspeichern.

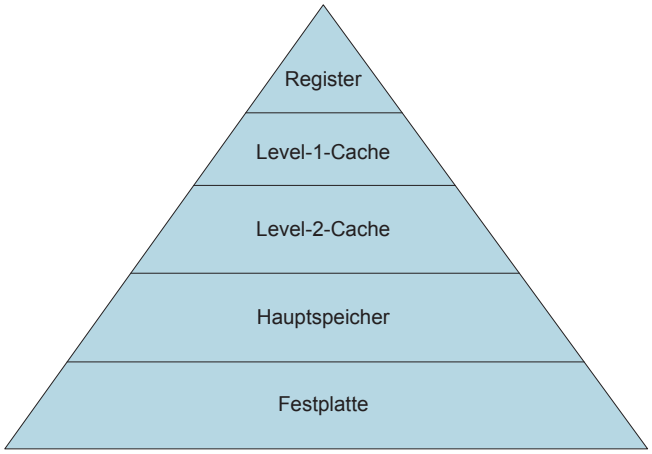


Abbildung 7.1.: Beispiel einer Speicherhierarchie eines eingebetteten Prozessorsystems

Besitzt ein Prozessorsystem mehrere Caches, muss sichergestellt werden, dass die Daten zu einer Adresse in den einzelnen Caches konsistent zueinander sind. Es muss gewährleistet werden, dass bei einer Schreiboperation die anderen Caches spätestens bei der nächsten Leseoperation auf diese Adresse die korrekten Daten enthalten. Hierzu werden Kohärenzprotokolle eingesetzt. Der CoreVA-Prozessor wird in einem Ein-Prozessor-System eingesetzt, aus diesem Grund wird im Rahmen dieser Arbeit

nicht auf Cache-Kohärenzprotokolle eingegangen. Liegt ein angefragtes Element im Cache, spricht man von einem *Treffer* (Cache-Hit). Ist das benötigte Element nicht im Cache vorhanden, nennt man dieses *Fehlzugriff* (Cache-Miss). Die *Trefferrate* (Hit-Rate) beschreibt den Anteil von Speicherzugriffen, die aus dem Cache beantwortet werden können. Die *Fehlzugriffsrate* (Miss-Rate $= 1 -$ Trefferrate) beschreibt den Anteil der Anfragen, die der Cache nicht beantworten kann. Es gibt Caches, die eine Anfrage beantworten können, während Fehlzugriffe behandelt werden (*Hit-Under-Miss*).

7.1.2.1. Cache Organisation

Direkt abgebildete Caches. Bei dieser Cache-Struktur kann ein Datenelement abhängig von seiner Adresse nur in einer Cache-Zeile gespeichert werden (vgl. Abbildung 7.2). Besteht ein Cache beispielsweise aus 16 Zeilen, werden 4 Bit der Adresse als Auswahlschlüssel (oft auch *Index* genannt) verwendet. Meistens werden die unteren Bit einer Adresse als Schlüssel benutzt, es ist jedoch auch die Verwendung jedes anderen Teils der Adresse möglich. Werden mehrere Wörter in einer Cache-Zeile gespeichert, wird der untere Teil der Adresse zur Unterscheidung dieser Wörter verwendet. Dieses wird als *Block-Offset* bezeichnet. Um feststellen zu können, ob ein Element im Cache gespeichert ist, wird zusätzlich zu den eigentlichen Daten der so genannte Tag gespeichert. Der Tag ist die Adresse des Datenworts ohne Auswahlschlüssel und Block-Offset. Um zu überprüfen, ob ein angefordertes Datenwort im Cache gespeichert ist, muss lediglich der Tag der in Frage kommenden Cache-Zeile mit dem Tag des angefragten Datenworts verglichen werden. Stimmen beide überein, können die Daten aus dem Cache gelesen werden, andernfalls müssen die Daten aus dem Hauptspeicher geladen werden. Die gelesenen Daten werden an die CPU weitergereicht und bei einem Cache-Miss zusätzlich in den Cache geschrieben. Hierbei muss auch der Tagspeicher aktualisiert werden. Zudem wird für jede Cache-Zeile ein Gültigkeitsbit (*Valid-Bit*) gespeichert. Hiermit kann der Cache zum Systemstart in einen definierten (leeren) Zustand gebracht werden. Aufgrund ihres geringeren Ressourcenbedarfs werden direkt abgebildete Caches häufig bei Prozessorarchitekturen für energiebeschränkte Anwendungsgebiete eingesetzt. Auch das Speicher-Subsystem des CoreVA-Prozessors verwendet direkt abgebildete Caches.

Ein Problem tritt auf, wenn zwei abwechselnde Zugriffe sich immer gegenseitig verdrängen. In diesem Beispiel kann die Verwendung eines direkt abgebildeten Caches sogar langsamer sein als eine Implementierung ohne Cache, da neben den benötigten Datenwörtern auch immer alle weiteren Daten einer Cache-Zeile aus dem Hauptspeicher in den Cache kopiert werden müssen [76]. In [198] und [21] wird jedoch gezeigt, dass ein auf die Cache-Architektur optimierter Compiler die Wahrscheinlichkeit für das Auftreten einer solchen Problematik stark verringern kann.

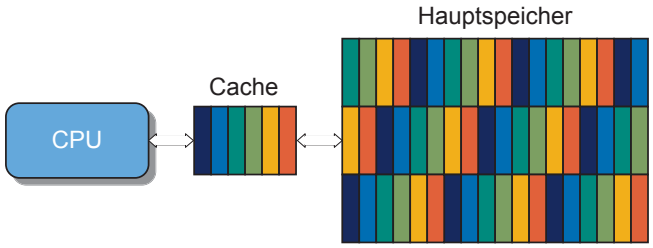


Abbildung 7.2.: Abbildung von Speicherbereichen in den Cache

Vollassoziative Caches. Anders als bei einem direkt abgebildetem Cache kann bei vollassoziativen Caches ein Datenwort in jeder beliebigen Cache-Zeile gespeichert sein. Um zu überprüfen, ob ein Datenwort im Cache gespeichert ist, müssen alle Einträge des Caches durchsucht werden. Hierzu werden spezielle Datenstrukturen, so genannter *Assoziativspeicher (Content Addressable Memory (CAM))*, verwendet. Ein CAM kann Suchoperationen in einem Taktzyklus durchführen. Vollassoziative Caches besitzen eine höhere Trefferrate als ein direkt abgebildeter Cache gleicher Größe, benötigen aber aufgrund des CAM-Speichers wesentlich mehr Energie und Chipfläche. Aus diesem Grund werden vollassoziative Caches nur in Spezialanwendungen, wie Netzwerkroutern, eingesetzt.

Teilassoziative Caches. Teilassoziative Caches ermöglichen eine ähnliche Trefferrate wie vollassoziative Caches bei nur unwesentlich höherem Hardware-Aufwand im Vergleich zu direkt abgebildeten Caches. Ein Speicherblock kann in einer festen Anzahl an Cache-Zeilen gespeichert werden. Ein Cache mit N_{Zeilen} möglichen Zeilen pro Block wird als N_{Zeilen}-fach (satz-)assoziativ bezeichnet. Üblich sind $N_{\text{Zeilen}} = 2$ bis $N_{\text{Zeilen}} = 16$. Um festzustellen, ob ein Datenwort im Cache gespeichert ist, werden alle in Frage kommenden Einträge verglichen. Hierzu werden Komparatoren in Hardware implementiert. In diesem Fall ist der Hardware-Aufwand jedoch geringer als bei einem vollassoziativem Cache. Nach [198] besitzen vollassoziative Caches bei typischen Anwendungen keinen Geschwindigkeitsvorteil gegenüber teilassoziativen Caches. Bei teilassoziativen Caches können die im Vergleich zu direkt abgebildeten Caches vorhandenen zusätzlichen Multiplexer die Latenz im kritischen Pfad verlängern [158].

Verdrängungsstrategien. Wird ein Datenwort aus dem Hauptspeicher geladen, ist es notwendig, dass es in einer Cache-Zeile gespeichert wird. Sind alle in Frage kommenden Cache-Zeilen belegt, muss eine dieser Cache-Zeilen in den Hauptspeicher verdrängt werden. Hierzu ist eine Verdrängungsstrategie notwendig, wenn

mehr als eine Cache-Zeile in Frage kommt. Dieses ist bei voll- und teilassoziative Caches der Fall. Eine optimale Strategie müsste alle Speicherzugriffe eines Programms kennen, auch die in der Zukunft liegenden. Ein Algorithmus muss sich immer an dieser optimalen Strategie messen. Bei einer Implementierung für einen eingebetteten Prozessor ist besonders darauf zu achten, dass der Verdrängungsalgorithmus wenig Energie und Chipfläche benötigt.

Verarbeitung von Schreiboperationen. Um Schreiboperationen zu ermöglichen, ist eine Erweiterung der beschriebenen Konzepte notwendig. Besonders ist darauf zu achten, dass die Daten in Cache und Hauptspeicher immer konsistent zueinander sind. Bei der Durchschreibtechnik (*Write-Through*) werden bei einem *Write Hit* alle Daten sowohl in den Cache, als auch in den Hauptspeicher geschrieben. Hierbei kann der Vorteil des Caches nicht ausgenutzt werden. Die Dauer einer Schreiboperation wird durch die Verzögerungszeit des Hauptspeichers bestimmt. Um dieses Problem abzuschwächen, kann ein Schreibpuffer (*Write-Buffer*) verwendet werden. Hier können ein oder mehrere Schreiboperationen zwischengespeichert werden, bis der Hauptspeicher sie abgearbeitet hat. Der Prozessor kann unterdessen weiterarbeiten. Ist der Schreibpuffer jedoch voll, kann der Prozessor keine weiteren Schreiboperationen durchführen und blockiert, bis wieder ein Pufferplatz frei geworden ist. Bei der Rückschreibtechnik (*Write-Back*) werden Schreiboperationen nur im Cache ausgeführt. Um die Konsistenz zum Hauptspeicher sicherzustellen, müssen bei der Verdrängung einer Cache-Zeile zunächst veränderte Daten in den Hauptspeicher geschrieben werden. Hierzu werden ein oder mehrere *Dirty-Bits* pro Cache-Zeile verwendet, die anzeigen, ob eine Cache-Zeile durch den Prozessor verändert worden ist. Eine Beschleunigung bei Schreiboperationen auf gecachte Adressen wird durch höheren Zeitbedarf beim Verdrängen erkauft. Welche der beiden Varianten bessere Ergebnisse liefert, hängt stark von der Anwendung ab. Einige Prozessoren können im Betrieb mittels eines Software-Befehls zwischen beiden Varianten wechseln.

Schreibzuordnung, Fehlzugriff beim Schreiben. Schreibt der Prozessor ein Datenwort, dessen Adresse sich nicht im Cache befindet, stehen drei Verfahren zur Auswahl:

Bei der Schreibzuordnung (*Write-Allocation*, *Allocate-on-write-miss (A-O-WM)*) werden die Daten der betroffenen Adresse aus dem Hauptspeicher in den Cache kopiert. Hierbei blockiert der Prozessor so lange, bis eine gesamte Cache-Zeile aus dem Hauptspeicher in den Cache geladen und die eigentliche Schreiboperation durchgeführt worden ist.

Findet keine Schreibzuordnung statt (*No-Write-Allocation*, *Fetch-on-write-miss (F-O-W-M)*), wird das Datenwort bei einem Schreib-Fehlzugriff direkt in den Hauptspeicher geschrieben, der Cache wird umgangen.

Als dritte Möglichkeit kann das zu schreibende Datenwort in den Cache geschrieben werden, ohne dass die Cache-Zeile zuvor aus dem Hauptspeicher in den Cache kopiert wird (*Read-Allocation*, *Read-Only-Allocation*). Hierbei ist es notwendig, neben dem *Dirty-Bit* ein *Modified-Bit* für jedes geschriebene Wort der Cache-Zeile zu setzen. Kommt es zu einer weiteren Schreiboperation auf die entsprechende Cache-Zeile, wird das Datenwort in die Cache-Zeile geschrieben sowie das entsprechende *Modified-Bit* gesetzt. Die Anfrage eines Datenwortes, das zuvor durch eine Schreibanfrage im Cache gespeichert worden ist, kann sofort vom Cache beantwortet werden. Bei einer Anfrage auf ein zuvor nicht vom Prozessor geschriebenes Datenwort wird die gesamte Cache-Zeile aus dem Hauptspeicher geladen. Es werden allerdings nur die Datenwörter in die Cache-Zeile geschrieben, die nicht mit dem *Modified-Bit* gekennzeichnet sind. Wird eine Cache-Zeile aus dem Cache verdrängt, auf die nur Schreiboperationen durchgeführt worden sind, dürfen nur die Datenwörter in den Hauptspeicher geschrieben werden, bei denen das *Modified-Bit* gesetzt ist. Durch die Strategie *Read-Allocation* wird vermieden, dass eine Cache-Zeile aus dem Hauptspeicher geladen wird, obwohl nur Schreiboperationen auf ihr ausgeführt werden. Beispielsweise behandelt der TriMedia TM5250 VLIW-Prozessor Fehlzugriffe nach diesem Muster [209].

7.1.3. Stand der Technik für Caches von VLIW-Prozessoren

Viele Ansätze für Cache-Architekturen, die bei superskalaren Prozessoren angewendet werden, sind nicht oder nur schlecht mit der VLIW-Architektur vereinbar. So ist zum Beispiel das Umsortieren von Zugriffen in einer Warteschlange bei VLIW-Prozessoren nicht möglich. Im folgenden Kapitel wird die Speicherarchitektur von ausgewählten eingebetteten Prozessoren vorgestellt, die zumeist eine VLIW-Architektur besitzen.

Der Level-1-Instruktions-Cache des TMS320C64x (vgl. Abschnitt 2.2.4) von Texas Instruments hat einen Port mit 256 Bit Breite, der Daten-Cache zwei 64 Bit (C64x) bzw. 256 Bit (C64x+) Schreib-/Leseports [207]. Da der Prozessor allerdings nur 32-Bit-Register besitzt, werden durch eine Leseoperation mehrere Register gefüllt. Hierdurch erhöht sich die effektive Speicherbandbreite des Prozessors. Der Instruktions-Cache ist direkt abgebildet, der Daten-Cache 2-fach satzassoziativ [208]. Beide Caches können auf einen gemeinsamen Level-2-Cache zugreifen.

Der Daten-Cache des Trimedia TM3270 VLIW-Prozessors [214] verfügt über einen Schreib-/Lese-Port und einen Schreib-Port. Die CPU unterstützt 32-Bit-Schreiboperationen und kann 32-Bit- und 64-Bit-Datenworte laden. Der Daten-Cache ist vierfach satzassoziativ und unterstützt heuristische Laden (*Pre-Fetching*). Ein zusätzlicher Cache-Schreibpuffer vermeidet Latenzen bei Schreibzugriffen.

Der Trimedia TM5250 besitzt einen 16 kB großen Daten-Cache, ist 4-fach satzassoziativ und eine Cache-Zeile ist 64 Byte groß. Der Prozessor besitzt lediglich einen 64-

Bit-Schreib-/Leseport, eine Leseanfrage weist eine Latenz von 7 Takten auf [209]. Über einen Software-Befehl kann eine Cache-Zeile vom Verdrängungsalgorithmus ausgenommen werden (sogenanntes *Cache Locking*). Hierdurch können wichtige Daten vom Compiler fest im Cache verankert werden. Ähnlich wie der CoreVA-Prozessor besitzt der TM5250 besitzt einen komprimierten Level-1-Instruktions-Cache (vgl. Abschnitt 4.3.1). Instruktionen werden vom Prozessor vor dem Dekodieren entpackt. Die Komprimierung spart Speicherplatz und Bandbreite des Speicherbusses, da der TM5250 leere Anweisungen (NOPs) in den Programmcode einfügen muss, wenn Ausführungseinheiten nicht ausgelastet sind. Der Instruktions-Cache ist 64 kByte groß und hat eine Blockgröße von 128 Byte. Er besitzt eine 8-fache Assoziativität.

In [40] führt eine Entwurfsraumexploration von VLIW-basierten eingebetteten Systemen durch. Bei der Exploration des Caches wird jedoch nur 1-Port-Speicher untersucht.

[146] untersucht, wie viele Speicher-Ports für eine VLIW-Architektur effizient ausgelastet werden können. Es wird berichtet, dass etwa 50 % Speicher-Ports pro VLIW-Slots einen guten Kompromiss zwischen Ressourcenbedarf und Performanz darstellen.

In [148] wird ein automatisches Werkzeug zur Entwurfsraumexploration verschiedener Cache-Architekturen anwendungsspezifischer VLIW-Prozessoren vorgestellt. Die Autoren untersuchen direkt abgebildete Caches mit einem oder zwei Schreib-/Lese-Ports sowie einem Schreib-/Lese- und einem Lese-Port. Alle Implementierungen unterstützen heuristisches Laden. Das Werkzeug analysiert verschiedene Cache-Größen, berücksichtigt allerdings nicht den Energiebedarf der Cache-Architektur.

7.1.4. Architektur des Speicher-Subsystems des CoreVA-Prozessors

Als Speicher-Subsystem wurde unter Berücksichtigung auf die Harvard-Architektur des CoreVA-Prozessors eine Cache-Architektur mit je einem dedizierten Instruktions-Cache und einem Daten-Cache für die beiden LD/ST-Einheiten implementiert [245]. In Abbildung 7.3 wird die Systemarchitektur mit den Komponenten Prozessorkern, Instruktions-Cache und Daten-Cache dargestellt [253]. Die Architektur ist bereits im Hinblick auf den später in Kapitel 8 beschriebenen ASIC-Prototyp in zwei Teile partitioniert. Der ASIC-Teil beinhaltet den Prozessorkern und die Caches. Über einen Systembus kann mit externen Komponenten (FPGA-Teil), wie z. B. einem Speicher-Controller, kommuniziert werden. Ein *Arbiter* arbitriert den Zugriff der Caches auf den Systembus. Da für das spätere Prototypensystem der SDRAM des RAPTOR-Erweiterungsmoduls DB-V2 genutzt werden soll, ist als Speicher-Controller ein SDRAM-Controller implementiert, der auf die dort vorhandene Schnittstelle abgestimmt ist. Über diesen SDRAM-Controller ist die Zusammenführung von Instruktions- und Datenspeicher auf einen gemeinsamen externen Speicher möglich. Die Cache-Controller wurden generisch implementiert. So ist es beispielsweise über generische Parameter

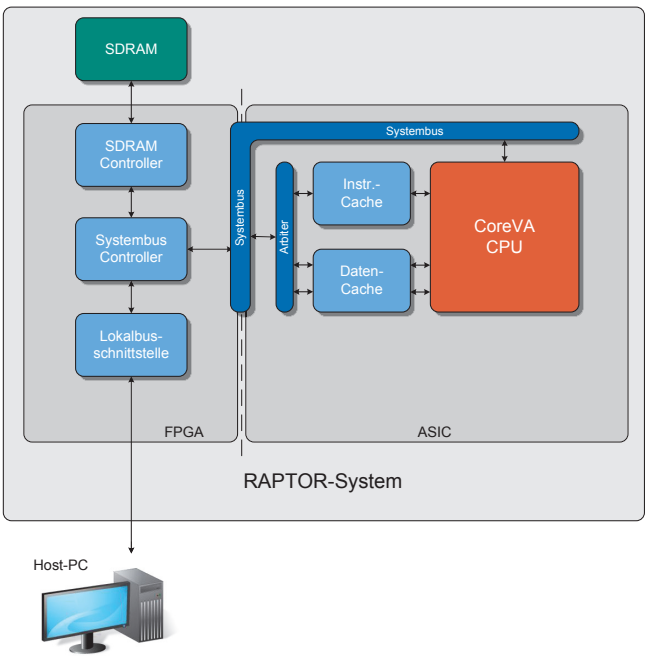


Abbildung 7.3.: Systemarchitektur des CoreVA-Systems

möglich, die Größe oder Anzahl der Cache-Zeilen zu konfigurieren.

7.1.4.1. Instruktions-Cache

Der CoreVA-Prozessor kann in der 4-Slot-Konfiguration des CoreVA-Prozessors pro Taktzyklus 128 Bit breite Instruktionsblöcke aus dem Instruktions-Cache lesen. Da der Cache keine Schreiboperationen des Prozessors verarbeiten muss, vereinfacht sich der Entwurf des Instruktions-Caches erheblich gegenüber dem des Daten-Caches. Jedes Wort des Hauptspeichers ist genau einer Cache-Zeile zugeordnet, der Instruktions-Cache ist also *direkt abgebildet* (vgl. Kapitel 7.1.2.1). Nach [139] passen viele typische Programme für eingebettete Systeme in einen 16 kByte bis 32 kByte großen Cache. Aus diesem Grund ergeben sich aus der Verwendung von satzassoziativen Caches keine Vorteile. Hierzu muss jedoch der Compiler auf den direkt abgebildeten Cache angepasst sein. In [198] wird darauf hingewiesen, dass der eigentliche Speicher bei einem Cache wesentlich für den Flächenbedarf verantwortlich ist und Logik weniger ins Gewicht fällt. Aus Kostengründen wurde im Hinblick auf die spätere ASIC-Implementierung (vgl. Kapitel 8.2) die Größe des Instruktions-Cache auf 16 kByte festgelegt.

7.1.4.2. Daten-Cache

Der Daten-Cache kann, anders als der Instruktions-Cache, sowohl Lese- als auch Schreiboperationen des CoreVA-Prozessors verarbeiten. Zudem kann der Daten-Cache zwei Anfragen in einem Taktzyklus verarbeiten. Hierzu stehen zwei Schreib-/Leseports bereit, die jeweils eine Datenbreite von 32 Bit besitzen. Der Daten-Cache ist, wie der Instruktions-Cache, *direkt abgebildet* und verwendet die *Rückschreibtechnik* (*Write-Back*) bei Schreiboperationen. Die Allokationsart kann dynamisch bestimmt werden (*Fetch-on-write-miss-Allokation* und *Allocate-on-write-miss-Allokation*). Die Möglichkeit der dynamischen Rekonfiguration der Wahl des Allokationsmodus hat keinen messbaren Einfluss auf den Ressourcenbedarf der Cache-Architektur. Der Einfluss auf die Performanz und auf die Ressourceneffizienz des Gesamtsystems wird in Abschnitt 7.1.5 detailliert beschrieben. In der Standardkonfiguration verwendet der Daten-Cache den Allocate-on-write-miss-Allokationsmodus, d.h. er lädt bei einem Schreib-Fehlzugriff die betreffende Cache-Zeile aus dem Hauptspeicher nach (vgl. Kapitel 7.1.2). Zentrale Komponente ist ein 2-Port-Speicher für Tag-Informationen und Daten. Für jede LD/ST-Einheit ist eine dedizierte Komponente zur Erzeugung der Treffersignale und je ein Zustandsautomat zur Steuerung implementiert.

Eine große Herausforderung besteht in der *Synchronisation* der Zustandsautomaten beider Ports des Daten-Caches untereinander sowie zum Instruktions-Cache. Es ist ein Ansatz denkbar, bei dem entweder nur der Instruktions-Cache oder einer der beiden Ports des Daten-Caches arbeiten. Diese einfache Implementierung würde zwar einen fehlerfreien Betrieb garantieren, der Prozessor würde jedoch durch lange Wartezyklen stark ausgebremst. Zudem könnte der Vorteil von zwei Schreib-/Leseports für den Datenspeicher nicht voll ausgenutzt werden. Aus diesem Grund arbeiten Daten- und Instruktions-Cache parallel, und die Zustandsmaschinen des Daten-Caches werden nur angehalten, wenn dieses für einen fehlerfreien Betrieb notwendig ist. Schreib- und Leseanfragen für beide Ports des Daten-Caches werden parallel verarbeitet, solange die Anfragen verschiedene Cache-Zeilen betreffen. Zwei parallele Schreibanfragen auf eine Cache-Zeile müssen nacheinander bearbeitet werden, da immer nur von einer Zustandsmaschine Daten- und Taginformationen einer Cache-Zeile aktualisiert werden können.

Der Daten-Cache kann als 1-Port- oder 2-Port-Cache konfiguriert werden. Dieses ist insbesondere für eine 1-Slot-RISC-Konfiguration der CoreVA-Architektur notwendig (vgl. Kapitel 4). Des Weiteren können Anwendungen mit geringen Speicheranforderungen von einem geringerem Ressourcenbedarf eines 1-Port-Caches profitieren. Der Allokationsmodus des Daten-Caches ist *dynamisch konfigurierbar*. Es kann zwischen *Allocate-on-write-miss-* und *Fetch-on-write-miss*-Allokation gewählt werden. Der zusätzliche Hardware-Aufwand für die dynamische Konfiguration des Allokationsmodus ist vernachlässigbar und beschränkt sich im Wesentlichen auf ein zusätzliches 1-Bit-Register und wenige Gatter zur Steuerung der Zustandsautomaten des Daten-

Caches. Der Einfluss auf die Ressourceneffizienz der verschiedenen Konfigurationsmöglichkeiten wird in Abschnitt 7.1.5 detailliert betrachtet.

Sowohl Instruktions- als auch Daten-Cache können zur Laufzeit in einen sogenannten *On-Chip-Modus* versetzt werden. Der Datenspeicher der Caches wird dann als lokaler (Scratchpad-)Speicher genutzt. Die Auswertung der Hit-/Miss-Logik entfällt. Dieses sorgt dafür, dass jeder Zugriff als Cache-Hit betrachtet wird. Durch Cache-Misses verursachte Wartezyklen entfallen komplett. Im Gegenzug ist der nutzbare Instruktions- und Datenspeicher allerdings auf die Größe des Datenspeichers des jeweiligen Cache begrenzt. Daher ist der On-Chip-Modus nur für Programme mit *geringen Speicheranforderungen* praktikabel.

7.1.4.3. SDRAM-Controller

Der *SDRAM-Controller* aus [90] wird über zwei FIFO[1] -Puffer mit der Taktdomäne des Prozessors verbunden. Der verwendete SDRAM erlaubt eine maximale Taktfrequenz von 133 MHz.

Im *Burst*-Modus benötigt eine Leseanfrage auf eine 512 Bit (dieses entspricht einer kompletten Cache-Zeile des Instruktions-Caches von 16 32-Bit-Wörtern) im besten Fall 20 Takte. Im (sehr unwahrscheinlichen) schlechtesten Fall mit einem zeitgleichen *Refresh* einer anderen Zeile des SDRAM sind 37 Takte notwendig. Eine Leseanfrage des Daten-Caches mit 256 Bit (8 32-Bit-Wörter) dauert mindestens 12 und höchstens 29 Taktzyklen[2]. Diese Betrachtung berücksichtigt ausschließlich Verzögerungen durch den SDRAM-Controller. Hinzu kommen Latenzen durch den Systembus und die SDRAM-FIFOs zur Entkopplung der verschiedenen Taktdomänen.

7.1.4.4. Systembus-Controller

Zur Kommunikation des CoreVA-Prozessorkerns wurde ein *bidirektionaler, multi-Master-fähiger Systembus* implementiert (vgl. Abbildung 7.3, [245]). Der Systembus-Controller ermöglicht den Austausch von Daten zwischen Lokalbus, SDRAM-Controller und Systembus. Ein Arbiter (Vermittler) regelt den Zugriff auf den Systembus. Es können sowohl Komponenten innerhalb (Caches etc.) aber auch außerhalb (Trace-Schnittstelle etc.) der Prozessordomäne Masterzugriff auf den Systembus anfordern. Der Arbiter stellt sicher, dass bereits laufende Zugriffe vollständig abgearbeitet werden und verhindert damit ein Blockieren der Kommunikation. Bei einer Taktfrequenz von 400 MHz beträgt die externe Kommunikationsbandbreite bis zu 12,8 GBit/s.

[1] First-In-First-Out

[2] bezogen auf die Taktdomäne des SDRAM-Controllers

7.1.5. Entwurfsraumexploration des Daten-Caches

Bei der Entwurfsraumexploration des Daten-Cache wird die Ausführungszeit von Anwendungen aus Abschnitt 5.2 in Abhängigkeit der verschiedenen Konfigurationsmöglichkeiten ermittelt.

Folgende Variationen der Parameter wurden untersucht:

- 1-/2-Port Daten-Cache
- Allocate-on-write-miss/Fetch-on-write-miss

Tabelle 7.1 zeigt das Ergebnis der Analyse des Daten-Caches bezüglich der 1-Port- und der 2-Port-Konfigurationen. Spalte drei und vier zeigen die Anzahl aller Instruktionen und deren Anteil an Load-/Store-Instruktionen. Während die absolute Anzahl der Load-/Store-Instruktionen für die 1-Port- und die 2-Port-Konfiguration identisch ist, gibt es bei der Gesamtanzahl an Instruktionen geringe Unterschiede. Diese sind auf *Datenabhängigkeiten* aufeinanderfolgender Instruktionen zurückzuführen, bei denen der Compiler Leerinstruktionen (NOPs) einfügt. Der Anteil der Speicherinstruktionen an der Gesamtzahl an Instruktionen variiert bei den verschiedenen Anwendungen stark. Während für die Ausführung des CRC-Algorithmus fast keine Speicherinstruktionen benötigt werden (hauptsächlich verursacht durch das einmalige Laden und Zurückschreiben der Nutzdaten), haben Anwendungen wie ECC und Faltung sehr hohe Speicheranforderungen. Die Auslastung der Speicher-Ports variiert stark in Anlehnung an den Anteil der Speicherinstruktionen an der Gesamtzahl an Instruktionen. Die Auslastung des zweiten Speicher-Ports variiert von 0 % (CRC) bis 36 % (Faltung). Entscheidend für die Analyse ist allerdings der Anteil paralleler Speicherzugriffe auf beiden Ports. Hier ist es wichtig zu berücksichtigen, dass die Anzahl der parallelen Speicherzugriffe in keiner Relation zu dem Performanzgewinn durch einen zweiten Speicher-Port steht. Abhängig vom Scheduling des Compilers können Speicherinstruktionen entweder parallel oder in direkt aufeinanderfolgenden Taktzyklen platziert werden, ohne dass dieses einen signifikanten Einfluss auf die Ausführungszeit haben muss. Ein Beispiel hierfür ist der *Delay-Slot* in der CoreVA-Architektur. Obwohl eine höhere Speicherbandbreite natürlich positiven Einfluß auf die Ausführungszeit hat, kann kann beim SATD-Algorithmus trotz 80 % paralleler Speicherzugriffe kein Performanzgewinn in dieser Größenordnung verzeichnet werden.

Tabelle 7.2 zeigt die Trefferrate des Daten-Caches für die zwei verschiedenen Allokationsmodi. Die Anzahl der Ports des Daten-Caches hat keinen Einfluss auf die Trefferrate. Hier ist nur die Reihenfolge der Zugriffe und die Architektur des Caches entscheidend. Die Reihenfolge der Zugriffe ist abhängig von der Implementierung des C-Compilers. Der verwendete Algorithmus erzeugt für die 1-Port- und die 2-Port-Konfiguration eine identische Zugriffsreihenfolge. Die Architektur des Caches ist in

Tabelle 7.1.: *Ergebnis der Auswertung der verschiedenen Konfigurationen des Datenspeichers*

Anwendung		Instruktionen	Load-/Store-Instruktionen		Auslastung Port 0	Auslastung Port 1	Parallele Load-/Store	
Dhrystone	1-port	119716	25444	21,25%	25,24%			
	2-port	119716	25444	21,25%	20,93%	5,16%	6638	26,09%
Coremark	1-port	849063	135528	15,96%	19,26%			
	2-port	851508	135528	15,92%	15,95%	4,10%	36776	27,14%
Viterbi	1-port	89808	12183	13,57%	28,83%			
	2-port	89808	12183	13,57%	21,61%	7,30%	935	7,67%
Faltung	1-port	3451	1893	54,85%	65,49%			
	2-port	3451	1893	54,85%	52,67%	36,00%	763	40,31%
FFT	1-port	521345	109513	21,01%	56,98%			
	2-port	518785	106953	20,62%	31,78%	29,24%	13917	13,01%
IEEE 802.11b	1-port	1544834	374779	24,26%	46,69%			
	2-port	1527233	374779	24,54%	36,99%	13,47%	54430	14,52%
CRC	1-port	129933	1029	0,79%	1,00%			
	2-port	129933	1029	0,79%	1,00%	0,00%	0	0,00%
ECC	1-port	422904	122270	28,91%	53,70%			
	2-port	421663	122270	29,00%	43,53%	16,80%	40544	33,16%
AES	1-port	135028	23273	17,24%	23,08%			
	2-port	135832	23273	17,13%	20,87%	2,82%	4162	17,88%
SNOW	1-port	794608	144926	18,24%	21,79%			
	2-port	794744	144926	18,24%	15,67%	7,81%	2004	1,38%
SATD	1 port	137000	20513	14,97%	42,22%			
	2 port	137032	20513	14,97%	25,97%	21,25%	16408	79,99%

der 1-Port- und 2-Port-Konfiguration ebenfalls gleich, weswegen auch die Trefferrate identisch ist.

Der Instruktions-Cache erzielt für alle Anwendungen sehr gute Trefferraten. Auch die Trefferraten des Daten-Caches sind bis auf wenige Ausnahmen gut. Auffällig ist die schlechte Trefferrate des CRC-Algorithmus, was jedoch auf die Beschaffenheit des Algorithmus zurückzuführen ist: Der CRC-Algorithmus liest Nutzdaten aus dem Speicher, berechnet die CRC-Prüfsumme und schreibt diese zurück. Wie in Tabelle 7.1 zu erkennen, ist der Anteil der Speicheroperationen an der Gesamtanzahl an Taktzyklen sehr gering. Für die Berechnung der Prüfsumme können die Zwischenergebnisse also fast ausschließlich in den Registern des Prozessorkerns abgelegt werden. Speicherzugriffe sind nicht notwendig. Das einmalige Lesen und das einmalige Zurückschreiben der Daten erzeugt Fehlzugriffe, die sich in der Trefferrate negativ bemerkbar machen. Da ein CRC-Algorithmus normalerweise in einer größeren Umgebung eingesetzt werden würde, bei dem sich die Nutzdaten mög-

licherweise bereits im Daten-Cache befinden, ist davon auszugehen, das in realen Einsatzszenarien die Trefferrate steigt. Wie erwartet steigt bei Nutzung des *Allocate-on-write-miss*-Allokationsmodus die Schreibtrefferrate für den Daten-Cache an. Besonders deutlich ist dieses beim Faltungs-Algorithmus der Fall ($85,38\,\% \rightarrow 99,53\,\%$). Die Lesetrefferrate sinkt dagegen leicht. Die Ursache hierfür liegt in der Kombination eines Lesezugriffes auf vorherigen Schreibzugriff auf die selbe Cache-Zeile. Im *Fetch-on-write-miss*-Allokationsmodus wäre diese Cache-Zeile beim vorherigen Schreib(-fehl)zugriffs bereits in den Cache geladen worden. Stattdessen führt der *Allocate-on-write-miss*-Allokationsmodus in dieser (seltenen) Kombination zu einem weiteren Cache-Miss.

Entscheidend für die Ausführungszeit einer Anwendung ist allerdings nicht nur die Trefferrate sondern der zeitliche Verlauf und die Abfolge der Speicherzugriffe. Aufgrund der starken (nichtdeterministischen) Variabilität der Latenzen beim Zugriff auf den (externen) SDRAM kann ein einzelner Fehlzugriff (beispielsweise während eines Refresh) eine sehr viel höhere Anzahl an Strafzyklen verursachen, als mehrere Zugriffe auf aufeinanderfolgende Speicherstellen. Auch die Reihenfolge, Verteilung und Frequenz der Zugriffe hat einen großen Einfluss auf die Latenz von SDRAM. Die

Tabelle 7.2.: *Trefferrate des Instruktions- und Daten-Caches*

Anwendung	Allokations-modus	Trefferrate I-Cache	Trefferrate D-Cache	
			Lesen	Schreiben
Dhrystone	fetch	99,87%	98,69%	99,79%
	allocate	99,87%	98,64%	99,98%
Coremark	fetch	99,95%	99,48%	99,75%
	allocate	99,95%	99,42%	99,99%
Viterbi	fetch	99,84%	99,94%	99,68%
	allocate	99,84%	99,92%	99,95%
Faltung	fetch	99,09%	98,75%	85,38%
	allocate	99,09%	98,80%	99,53%
FFT	fetch	99,98%	98,97%	98,95%
	allocate	99,98%	99,09%	99,82%
IEEE 802.11b	fetch	99,99%	99,71%	96,98%
	allocate	99,99%	99,51%	97,78%
CRC	fetch	99,99%	96,79%	25,00%
	allocate	99,99%	96,88%	75,00%
ECC	fetch	99,93%	99,88%	99,76%
	allocate	99,93%	99,84%	99,99%
AES	fetch	99,90%	99,89%	99,81%
	allocate	99,90%	99,78%	99,94%
SNOW	fetch	99,99%	99,84%	99,72%
	allocate	99,99%	99,57%	99,86%
SATD	fetch	99,96%	98,74%	99,05%
	allocate	99,96%	98,59%	99,98%

Trefferrate kann also lediglich als Indiz für die Qualität des Caches angesehen werden. Abbildung 7.4 zeigt den Anteil Verarbeitungszeit durch den CoreVA-Prozessor und den Anteil der Strafzyklen an der Gesamtlaufzeit, die durch Fehlzugriffe der Caches verursacht wurden. Die Ergebnisse sind für die vier Möglichkeiten der Variation der Konfigurationsparameter aufgeführt.

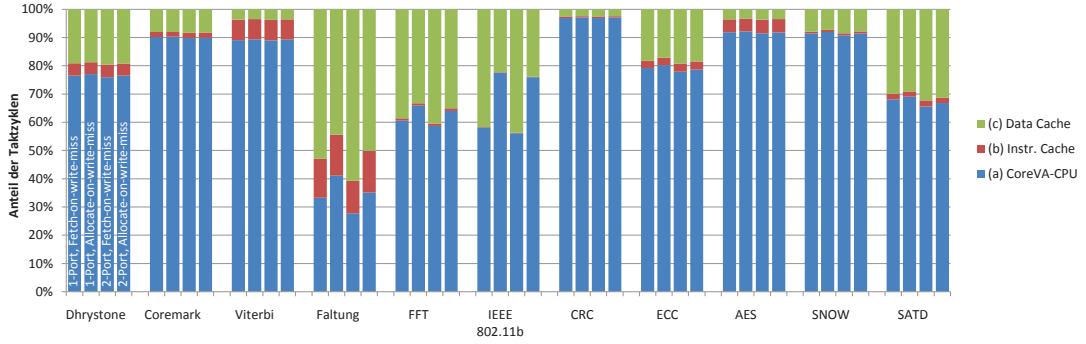


Abbildung 7.4.: Anteil der Strafzyklen an der Gesamtlaufzeit der Algorithmen

Bei Ausführung der Anwendungen Coremark, Viterbi, CRC, ECC, AES und SNOW wird der Prozessor nur in wenigen Taktzyklen von den Caches angehalten. Trotz guter Trefferraten ist der Anteil der Wartezyklen bei Anwendungen wie Dhrystone, Faltung, FFT und SATD verhältnismäßig hoch. Insbesondere bei dem Faltungsalgorithmus scheint eine Umstrukturierung der Speicherzugriffe bereits im C-Code sinnvoll.

Abbildung 7.5 zeigt den Performanzgewinn der verschiedenen Konfigurationen relativ zur 1-Port-Konfiguration bei Nutzung des *Allocate-on-write-miss*-Allokationsmodus.

Bei der Performanz profitieren alle Anwendungen vom erhöhten Datendurchsatz durch den zweiten Speicher-Port. Die größten Performanzgewinne lassen sich bei der Faltung (31 %), der Fouriertransformation (14 %) und dem IEEE-802.11b-Algorithmus (29 %) erzielen. Insbesondere der Faltungs- und der IEEE-802.11b-Algorithmus profitieren vom *Allocate-on-write-miss*-Allokationsmodus ($12\,\% \rightarrow 31\,\%$ und $4\,\% \rightarrow 29\,\%$).

Zur Bewertung der Ressourceneffizienz soll nun die Ausführungszeit in Bezug zum Ressourcenbedarf des Gesamtsystems, bestehend aus CoreVA-Prozessor und Speicher-Subsystem, gesetzt werden. Tabelle 7.3 vergleicht den Flächenbedarf und die Verlustleistung des CoreVA-Systems für die 1-Port- und die 2-Port-Konfiguration des Datenspeichers. Die Wahl des Allokationsmodus erfolgt dynamisch und hat daher keinen Einfluss auf den Ressourcenbedarf. Die Unterschiede einer statischen Konfiguration der Allokationsmodi liegen unterhalb der Toleranzen, die aufgrund pseudozufälliger Sequenzen moderner Synthesewerkzeuge entstehen, und sind daher vernachlässigbar. Das Design wurde auf eine 65 nm Standardzellentechnologie

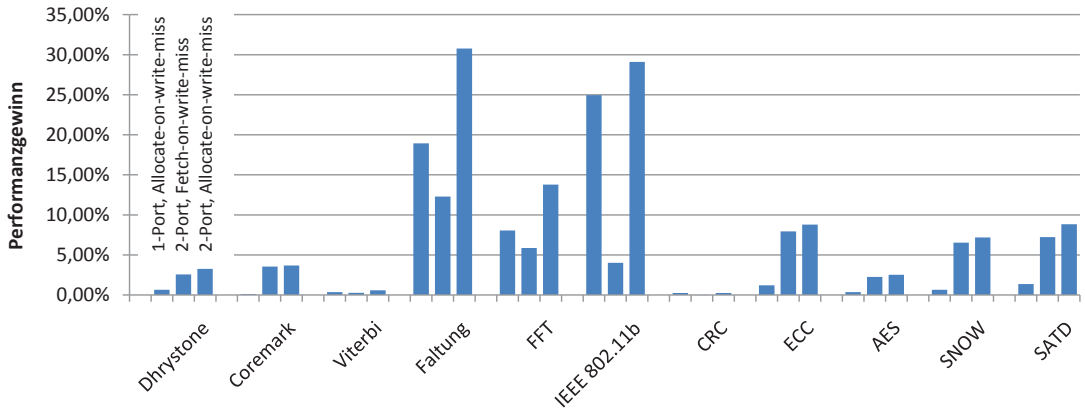


Abbildung 7.5.: Performanzgewinn der verschiedenen Konfigurationen relativ zur 1-Port-Konfiguration (Allocate-on-write-miss)

von STMicroelectronics (Worst Case) abgebildet. Der Overhead des Flächenbedarfs und der durchschnittlichen Leistungsaufnahme liegt für die 2-Port-Konfiguration unter 10 % und ist weitestgehend der höheren Komplexität der 2-Port-Datenspeicher-Makros zuzuordnen.

Tabelle 7.3.: *Vergleich des Ressourcenbedarfs der 1-Port- und der 2-Port-Konfiguration des Daten-Caches in einer 65 nm Standardzellentechnologie von STMicroelectronics*

	1-Port	2-Port	Relativ
Leistungsaufnahme (∅) [mW]	108,27	115,08	6,29%
Fläche [mm^2]	1,57	1,70	8,28%

Abbildung 7.7 zeigt die Anwendung des in Kapitel 3.4 vorgestellten Maßes zur Bewertung der Ressourceneffizienz für einen gleiche Gewichtung aller Ressourcen ($RE = 1/(A \cdot P \cdot T)$). Die Ergebnisse stellen die Unterschiede der Ressourceneffizienz normiert auf die 1-Port-Konfiguration mit *Fetch-on-write-miss*-Allokationsmodus dar. Beispielsweise ist die 2-Port-Konfiguration (*Allocate-on-write-miss*) für den Faltungsalgorithmus um etwa 26 % effizienter als die 1-Port-Konfiguration (*Fetch-on-write-miss*). Unter Berücksichtigung des Flächenoverheads ergibt sich die höchste Ressourceneffizienz für die 1-Port-Konfigurationen. Lediglich bei dem FFT-Algorithmus kann man für den zweiten Speicher-Port eine Erhöhung der Effizienz beobachten. Dieses ist auf die signifikante Erhöhung der Performanz zurückzuführen, die den Overhead des Ressourcenbedarfs ausgleichen kann.

Wird die Fläche bei der Ressourceneffizienz nicht (oder weniger) gewichtet, so

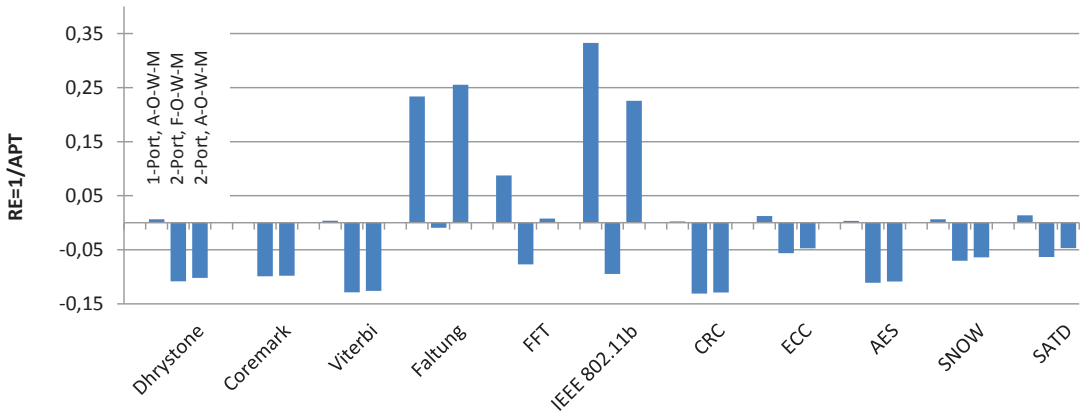


Abbildung 7.6.: Ressourceneffizienz der verschiedenen Konfigurationen des Daten-Cache für $RE = 1/(A \cdot P \cdot T)$

ist die Architektur mit zwei Speicher-Ports effizienter. Insbesondere bei energiebeschränkten Systemen erscheint eine höhere Gewichtung der Leistungsaufnahme zur Bewertung der Ressourceneffizienz sinnvoll. Abbildung 7.7 zeigt die Auswertung der Ressourceneffizienz für $RE = 1/(P \cdot T)$ (Energie) für ein energiebeschränktes Anwendungsszenario. Insbesondere der Faltungs-, FFT- und IEEE-802.11b-Algorithmus profitieren von *Allocate-on-write-miss*-Allokationsmodus (bis zu 35 %). Hier zeigt sich auch das Potential der dynamischen Rekonfiguration des Daten-Caches.

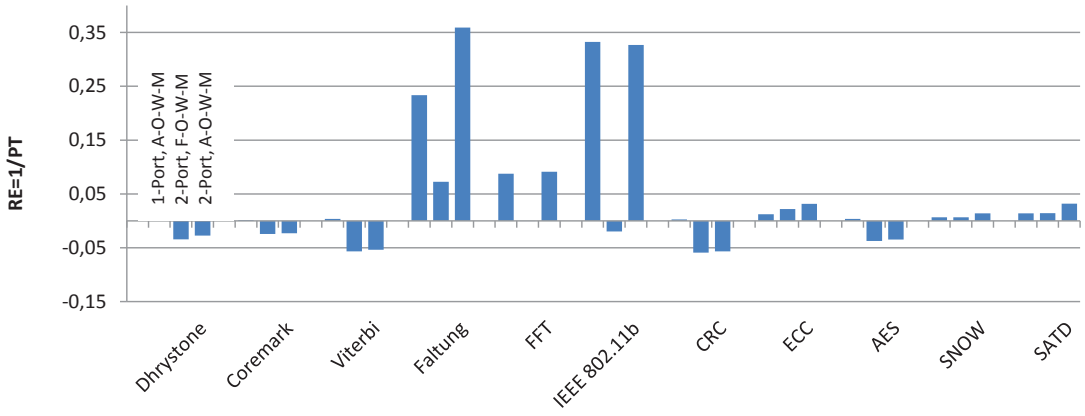


Abbildung 7.7.: Ressourceneffizienz der verschiedenen Konfigurationen des Daten-Cache für $RE = 1/(P \cdot T)$ (Energie)

7.1.6. Zusammenfassung

Es zeigt sich, dass je nach Anwendung entweder die 1-Port- oder die 2-Port-Konfiguration die effizientere Architektur darstellen kann [245]. Aufgrund des für den CoreVA-Prozessor geplanten Anwendungsszenarios wurde die 2-Port-Konfiguration für den ASIC-Prototyp ausgewählt. Die Wahl des Allokationsmodus des Daten-Caches zur Laufzeit erlaubt die Adaption der Architektur an die jeweilige Anwendung. Alle folgenden Ergebnisse für die Performanz und den Ressourcenbedarf des CoreVA-Systems beziehen sich auf die *2-Port-Konfiguration* mit *Allocate-on-write-miss*-Allokationsmodus.

7.2. Scratchpad-Speicher als schneller On-Chip-Speicher

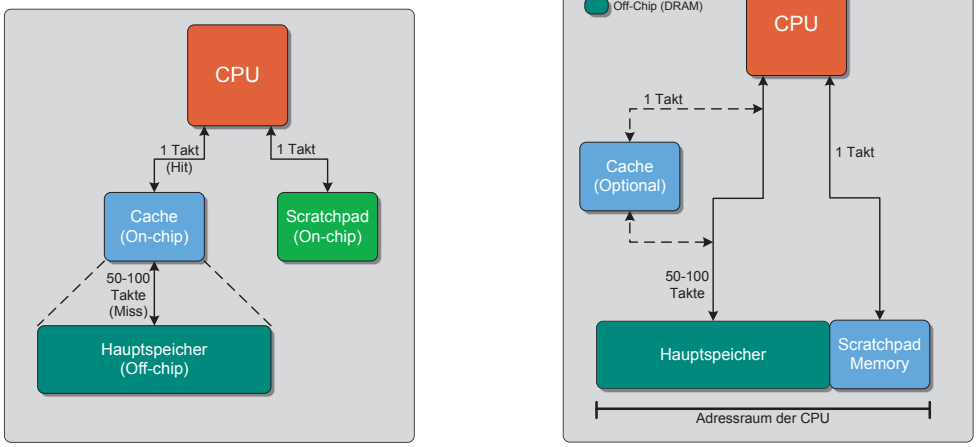
Die Auswertung der Ausführungszeiten der Anwendungen in Abschnitt 7.1 zeigt, dass in vielen Situationen, insbesondere der Daten-Cache und die damit verbundene Kommunikation mit externem Speicher, einen Flaschenhals bezüglich der Performanz darstellt. Als Alternative zu einem transparenten Cache ist die Verwendung eines schnellen SRAM-Speichers (*Scratchpad-Memory*) möglich, der vom Prozessor direkt adressiert wird. Als *Scratchpad-Speicher* oder auch *Scratchpad RAM*[3] wird ein Zwischenspeicher bezeichnet, der in der Speicherhierarchie mit Level-1-Cache vergleichbar ist. Er liegt damit nach den internen Registern am nächsten an den Funktionseinheiten einer CPU. Ebenso wie Level-1-Cache wird er physikalisch als SRAM realisiert.

Ein Scratchpad-Speicher kann energie- und flächeneffizienter sein als ein Cache gleicher Größe [16], stellt aber hohe Anforderungen an den Compiler. Sowohl Level-1-Cache als auch Scratchpad-Speicher sind üblicherweise als schneller SRAM Speicher direkt auf dem Prozessor-DIE realisiert, sodass die Zugriffszeit auf beide Speicher einen Taktzyklus beträgt. Der entscheidende Unterschied ist jedoch, dass ein Scratchpad-Speicher eine konstante Zugriffszeit von einem Takt *garantiert*. Cache hingegen kann eine Zugriffszeit von einem Takt nur bei einem Treffer erreichen. Sollte sich das angeforderte Datenwort nicht im Cache befinden, muss dieses aus dem Hauptspeicher nachgeladen werden. Dadurch ergeben sich höhere Latenzen von wenigen (bei mehreren Cache-Hierarchiestufen) bis hin zu mehreren hundert Takten. Abbildung 7.8a zeigt den Unterschied der Zugriffszeiten zwischen Cache-Architekturen und Scratchpad-Speichern.

Als weiterer Vorteil entfällt bei Scratchpad-Speicher die Kontroll-Logik zum Prüfen auf Treffer und zum Nachladen von Daten aus dem Hauptspeicher. Auch wenn dieser Overhead, wie in Abschnitt 7.1 gezeigt, im Vergleich zum Ressourcenbedarf des Speichers gering ist, so ist ein Scratchpad-Speicher offensichtlich effizienter. In Abschnitt 7.2.3 wird gezeigt, wie aufgrund des geringeren Hardware-Aufwands und der geringeren Zugriffszeiten ein System unter der Verwendung von Scratchpad-Speicher eine deutlich höhere Ressourceneffizienz erreichen kann. Der Nachteil ist jedoch die recht geringe Datenmenge die im Scratchpad-Speicher abgelegt werden kann, da er aus Kostengründen in seiner Größe eingeschränkt ist und an keinen großen Hauptspeicher gebunden ist [16, 229]. Abschnitt 7.2.3.2 zeigt die Erweiterung um Komponenten zur schnellen Kommunikation von Scratchpad-Speicher und Hauptspeicher.

Das CoreVA-System unterstützt das Prinzip von Scratchpad-Speicher bereits durch die Möglichkeit der dynamischen Rekonfiguration der Caches in den sogenannten *On-Chip-Speicher-Modus*. Im On-Chip-Speicher-Modus ist die Größe des Speichers

[3] Random Access Memory



(a) Vergleich der Zugriffszeiten **(b)** Aufteilung des Adressraumes

Abbildung 7.8.: Vergleich Cache/Hauptspeicher mit Scratchpad-Speicher

stark beschränkt und er ist daher nur für Anwendungen mit geringen Speicher- aber hohen Performanzanforderungen sinnvoll. Es sind auch Architekturen denkbar, die ausschließlich Scratchpad-Speicher ohne einen zwischengeschalteten Cache verwenden. Hierbei verursacht jeder Zugriff auf den Hauptspeicher eine sehr große Latenz, was den Einsatz in einer VLIW-Architektur mit einer hohen Datenparallelität nicht sinnvoll erscheinen lässt. Eine Lösung der parallele Einsatz von Caches und dedizierten Scratchpad-Speichern. Insbesondere bei VLIW-Architekturen mit einer höheren Datenparallelität lässt sich die Speicherbandbreite so stark erhöhen.

7.2.1. Stand der Technik bei Scratchpad-Speichern

Bei der TMS320C54x-Familie [200] von Texas Instruments wird ein Ansatz verfolgt, der die direkte Anbindung des Prozessors an einen (langsamen) Hauptspeicher und die Verwendung von (schnellem) Scratchpad-Speicher kombiniert. Auf Caches wird gänzlich verzichtet. Aufgrund der Hardwareersparnis gegenüber einem Cache-Controller kann diese Speicherarchitektur eine höhere Energie- und Flächeneffizienz erzielen. Dazu wird der Adressraum der CPU aufgeteilt, sodass ein bestimmter Teil auf den Scratchpad-Speicher abgebildet wird (vgl. Abbildung 7.8b). Zur Leistungssteigerung müssen demnach besonders häufig verwendete und zeitkritische Daten in den Scratchpad-Speicher abgelegt werden, da der Zugriff auf den Hauptspeicher sehr langsam und energieaufwendig ist. Hierzu besteht die Möglichkeit den Compiler so zu optimieren, dass dieser besonders häufig verwendete Daten in den Adressraum des Scratchpad-Speichers abbildet. So werden z. B. skalare Variablen

oder kleine Arrays in dem Scratchpad-Speicher abgelegt [156]. Eine andere Möglichkeit ist es, die Aufteilung des Adressraums dem Programmierer selbst zu überlassen. Dieses stellt jedoch hohe Anforderungen an den Programmierer, da dieser die genaue Speicherorganisation kennen muss [22].

In aktuellen Prozessoren und DSPs wird Scratchpad-Speicher zumeist neben einer Cache-Architektur verwendet, da er eine höhere Performanz verspricht als der Ansatz ohne Cache. Gleichzeitig ist er energieeffizienter als eine Cache-Architektur ohne Scratchpad-Speicher, da die Anzahl von energieaufwendigen Zugriffen auf den externen Hauptspeicher minimiert wird [11, 216]. Auch bei diesem Ansatz wird der Adressbereich der CPU in Hauptspeicher und Scratchpad-Speicher aufgeteilt, sodass sich hier hohe Anforderungen an den Compiler bzw. den Programmierer ergeben.

So verfügt beispielsweise der Tensilica Diamond 330HiFi Prozessor sowohl über Scratchpad-Speicher als auch über Caches [199, 92]. Der 2-fach satzassoziative Instruktionscache ist 4 kByte groß. Der Scratchpad-Instruktionsspeicher verfügt über bis zu 128 kByte. Der Datencache besitzt einen 64 Bit breiten Schreib-/Leseport, ist ebenfalls 2-fach satzassoziativ und 8 kByte groß. Die Größe des Scratchpad-Speicher für Daten beträgt 128 kByte und er besitzt zwei 64-Bit-Schreib-/Leseports, ist also als 2-Port-Speicher ausgeführt.

[16] vergleicht traditionelle Caches mit Scratchpad-Speichern. Mit Hilfe eines optimierenden Compilers ließ sich das Flächen-Zeit-Produkt um durchschnittlich 46 % verringern.

Der nachfolgende Abschnitt zeigt die Anbindung von zusätzlichem Scratchpad-Speicher an den CoreVA-Prozessor. Der Vorteil von Scratchpad-Speicher zeigt sich insbesondere bei solchen Anwendungen, die auf relativ kleinen Datenmengen viele Speicheroperationen durchführen. Abschnitt 7.2.3 zeigt daher die Adaption der Anwendungen IEEE 802.11b und FFT auf ein System mit Scratchpad-Speicher und die Analyse des Gesamtsystems bezüglich der Ressourceneffizienz.

7.2.2. Implementierung des Scratchpad-Speichers

In der in den vorigen Kapiteln vorgestellten, vierfach parallelen CoreVA-VLIW-Architektur sind VLIW-Slot 0 und 1 an den 2-Port-Daten-Cache angebunden. In Slot 2 und Slot 3 können keine Speicheroperationen durchgeführt werden. Daher bietet es sich an, Slot 2 und/oder Slot 3 an einen oder mehrere Scratchpad-Speicher anzubinden. Durch die generische Implementierung ist es einfach möglich, den CoreVA-Prozessor um die zwei zusätzlich benötigten LD/ST-Einheiten zu erweitern. Abbildung 7.9 zeigt die Architektur des CoreVA-Prozessors mit 2-Port-Daten-Cache und zwei Scratchpad-Speichern [247]. Die Größe und Wortbreite jedes Scratchpad-Speichers ist generisch konfigurierbar und ist für die gewählten Anwendungen auf eine Wortbreite von 32 Bit eine Größe von 2kByte eingestellt. Mit Hilfe einer Schreibmaske lassen sich neben der Datenbreite von 32Bit auch 16Bit und 8Bit Datenwörter schreiben.

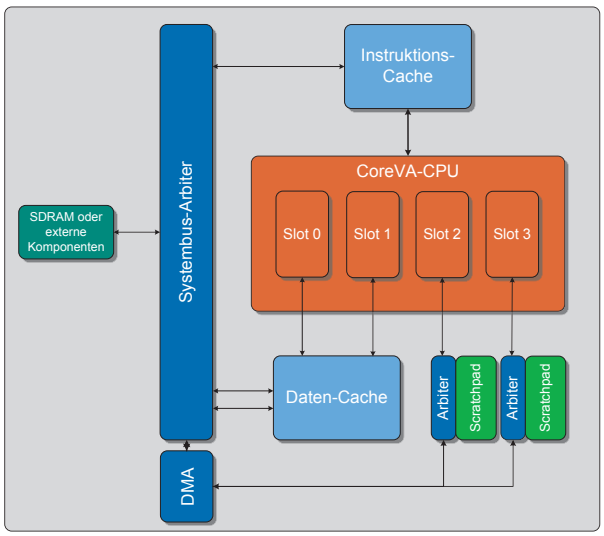


Abbildung 7.9.: Architektur und Anbindung des Scratchpad-Speichers an den CoreVA-Prozessor

Der CoreVA-VLIW-Compiler wurde dahingehend erweitert, dass die Platzierung von Speicheroperationen im C-Code über sogenannte *Intrinsics* angegeben werden kann. Listing 7.1 zeigt ein Beispiel für solche Speicheroperationen im C-Code. Speicheroperationen ohne diese Platzierungsinformationen werden standardmäßig in VLIW-Slot 0 und 1 entsprechend des Schedulings des Compilers platziert.

Listing 7.1: Beispiel für intrinsische Anweisungen im C-Code zur Vorgabe de Platzierung von Speicheroperationen

```
void _main()
{
  int a[4],b[4],c[4];
  b[0] = _ldw_slot0((unsigned int)a,0);
  b[1] = _ldw_slot1((unsigned int)a,1);
  b[2] = _ldw_slot2((unsigned int)a,2);
  b[3] = _ldw_slot3((unsigned int)a,3);
  _stw_slot0(b[0],(unsigned int)&c,0);
  _stw_slot1(b[1],(unsigned int)&c,1);
  _stw_slot2(b[2],(unsigned int)&c,2);
  _stw_slot3(b[3],(unsigned int)&c,3);
}
```

7.2.3. Entwurfsraumexploration

Die *Effizienz* der Scratchpad-Speicher-Erweiterung des CoreVA-Systems wird in diesem Abschnitt exemplarisch an den Anwendungen IEEE 802.11b und FFT aufgezeigt. Die Performanz beider Anwendungen wird vom Daten-Cache in besonderem Maße durch Fehlzugriffe beeinträchtigt (vgl. Abschnitt 7.1), so dass hier der Einsatz von Scratchpad-Speicher besonders große Vorteile verspricht.

7.2.3.1. IEEE 802.11b

Wie in Abschnitt 5.2 erläutert, basiert der IEEE-802.11b-Algorithmus auf der Ausführung von vier Grundfunktionen. Der Datenpfad der Anwendung ist also in vier Teile partitioniert. Jede Funktion arbeitet auf den Eingangsdaten, die sie von der vorherigen Funktion (als Zeiger auf einen Speicherbereich) übergeben bekommt. Über das Einfügen der intrinsischen Instruktionen wurden die Speicherzugriffe auf die Zwischenergebnisse der Funktionen, die auf den Hauptspeicher erfolgen, auf den Scratchpad-Speicher abgebildet. Dieses setzt aber voraus, dass sich die vom IEEE-802.11b-Algorithmus zu verarbeitenden Nutzdaten bereits im Scratchpad-Speicher befinden. Die Transferzeit der Nutzdaten vom Hautpspeicher in den Scratchpad-Speicher muss in den Vergleich der Ausführungszeiten mit einfließen. Um den Overhead des Kopierens von Nutzdaten aus dem Hauptspeicher in den Scratchpad-Speicher zu minimieren, liest die erste Funktion des IEEE-802.11b-Algorithmus (Scrambler) die Nutzdaten direkt aus dem Hauptspeicher, schreibt die Zwischenergebnisse jedoch in den Scratchpad-Speicher zurück. Nachfolgende Funktionen arbeiten dann direkt auf dem Scratchpad-Speicher. Abbildung 7.10 zeigt die Aufteilung der Speicherzugriffe der Funktionen des IEEE-802.11b-Algorithmus auf Hauptspeicher und Scratchpad-Speicher. Abbildung 7.11 zeigt die Verringerung der benötigten Anzahl von Taktzyklen nach der Optimierung des Algorithmus durch Nutzung des Scratchpad-Speichers CoreVA-Systems. Durch die Optimierung des Algorithmus reduziert sich die Ausführungszeit des Algorithmus um ca. 22 %. Wartezyklen durch Fehlzugriffe auf den Daten-Cache entfallen fast vollständig.

7.2.3.2. Schnelle Fouriertransformation

Der FFT-Algorithmus unterscheidet sich von dem IEEE-802.11b-Algorithmus insofern, dass er nicht aus vielen Teilfunktionen besteht. Aus diesem Grund ist es nicht praktikabel, bei der erstmaligen Verarbeitung der Nutzdaten jedes Datenwort aus dem Hauptspeicher zu laden und anschließend für die weiteren Berechnungsschritte im Scratchpad-Speicher abzulegen. Die dafür notwendige zusätzliche Programmlogik würde Performanzgewinne des Scratchpad-Speichers zunichtemachen. Stattdessen könnten die Daten vor der Verarbeitung vom Hauptspeicher in den Scratchpad-

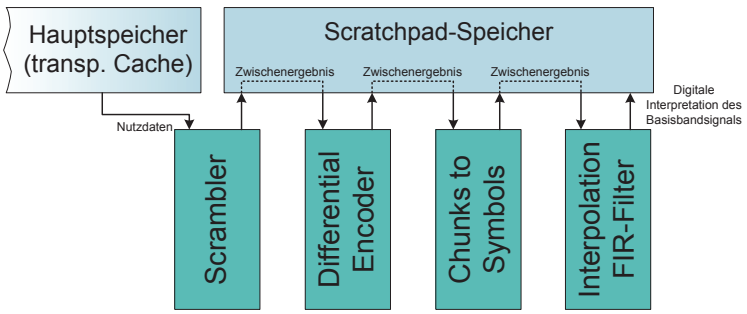


Abbildung 7.10.: Aufteilung der Speicherzugriffe der Funktionen des IEEE-802.11b-Algorithmus auf Hauptspeicher und Scratchpad-Speicher

Speicher kopiert werden. Auch dieser Ansatz ist nicht effizient. Die (erstmaligen) Zugriffe auf den Datenspeicher provozieren viele Fehlzugriffe des Daten-Caches. Das dadurch bedingte Nachladen der Nutzdaten aus dem vergleichsweise langsamen Hauptspeicher (als SDRAM implementiert) ist nicht effizient, da sie für den SDRAM (langsame) Einzelzugriffe darstellen. Ein Transfer eines zusammenhängenden Blocks (*Burst-Transfer*) ist dagegen um ein vielfaches schneller [224] [33]. Aus diesem Grund wurde die Implementierung des CoreVA-Systems mit Scratchpad-Speicher um einen sogenannten *DMA*[4] *-Controller* erweitert, der schnelle und effiziente Speichertransfers vom Hauptspeicher in einen Scratchpad-Speicher und umgekehrt ermöglicht. Ein Transfer wird von der CPU eingeleitet, findet jedoch nebenläufig zur Programmausführung statt. Die CPU kann also während eines Transfers andere Aufgaben übernehmen. Auch ermöglicht es der implementierte DMA-Controller, Daten von Peripheriegeräten direkt in den Hauptspeicher des CoreVA-Prozessors oder einen Scratchpad-Speicher zu transferieren. Gesteuert wird der DMA-Controller über sogenannte *Memory-Mapped-I/O (MMIO)*-Zugriffe. Speicherzugriffe auf zuvor definierte Speicherbereiche werden nicht auf den Datenspeicher sondern auf den DMA-Controller abgebildet. Die Abbildung dieser Speicherzugriffe erfolgt durch den in Abschnitt 7.3 detailliert beschriebenen Adressdekoder zur Anbindung von Hardware-Erweiterungen an das CoreVA-System. Über diese MMIO-Zugriffe können dem DMA-Controller die in Tabelle 7.4 dargestellten Kommandos übergeben werden.

Der DMA-Controller ist als *direkter DMA-Controller* realisiert, da er die Daten ohne Zwischenspeicherung direkt vom SDRAM bzw. anderen Peripheriegeräten in den Scratchpad-Speicher schreibt. Dieses erspart einen Zwischenspeicher für die Daten. Der direkte Transfer erfordert jedoch einen Handschlag (engl. *Handshake*) mit beiden Kommunikationspartnern, sodass diese die jeweilige Kommunikationsstruktur für

[4] Direct Memory Access

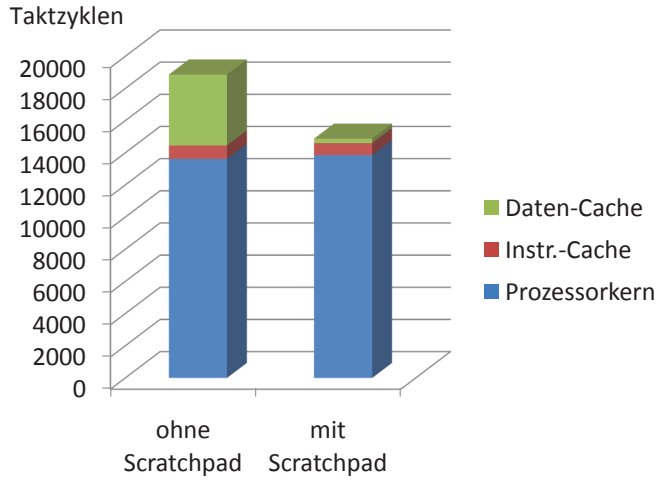


Abbildung 7.11.: Vergleich der Laufzeit des 802.11b-Algorithmus nach Optimierung durch Nutzung des Scratchpad-Speichers

den DMA-Zugriff freigeben.

Auch der DMA-Transfer der komplexen Eingangsdaten-Paare, die vom FFT-Algorithmus verarbeitet werden, muss bei der Analyse der Performanz berücksichtigt werden. Abbildung 7.12 zeigt den Vergleich der Laufzeiten des FFT-Algorithmus für unterschiedliche Anzahl an komplexen Eingangsdaten mit und ohne Scratchpad-Speicher. Durch den zusätzlichen DMA-Transfer zu Beginn der Berechnung steigt die Anzahl der Verarbeitungszyklen durch den CPU-Kern im Vergleich zur Original-Version ohne Scratchpad-Speicher. Bis zu 16 komplexen Eingangsdaten wirkt sich dieses negativ auf die Laufzeit aus. Ab 32 komplexen Eingangsdaten profitiert der

Tabelle 7.4.: *Funktionen des DMA-Controllers*

Zugriffsart	Adresse	Funktion
Schreiben	0x88000000	Startquelladresse des Datenblocks im Hauptspeicher.
Schreiben	0x88000004	Startzieladresse innerhalb des Scratchpad-Speichers.
Schreiben	0x88000008	Größe des zu transferierenden Datenblocks. Anschließend startet der Transfer.
Lesen	0x880000C	Statusregister gibt erfolgreiches Beenden des Transfers an.

Algorithmus jedoch vom Scratchpad-Speicher (Reduzierung der Laufzeit um etwa 10 %). Bei 512 komplexen Eingangsdaten reduziert sich die Laufzeit des Algorithmus bereits um ca. 31 %.

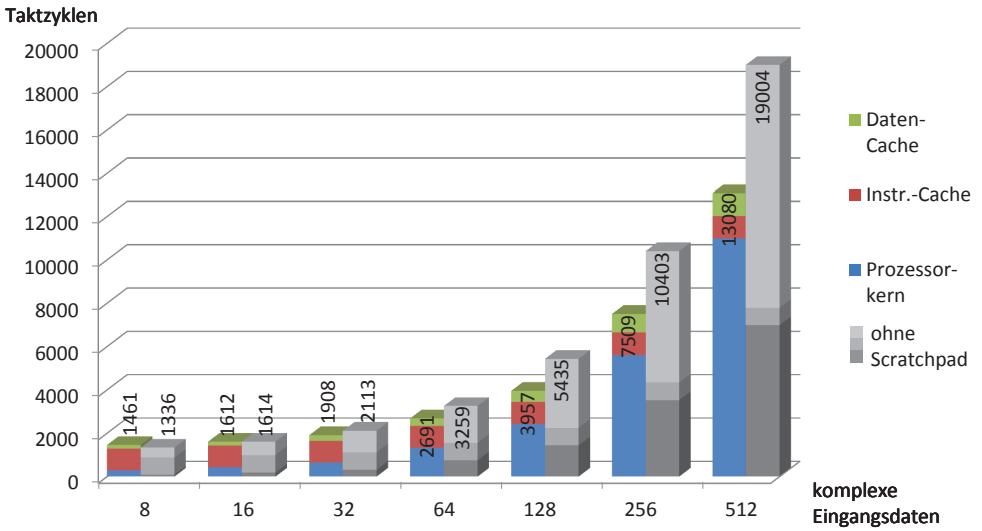


Abbildung 7.12.: Vergleich der Laufzeiten des FFT-Algorithmus für unterschiedliche Anzahl an komplexen Eingangsdaten mit und ohne Scratchpad-Speicher

7.2.4. Bewertung der Ressourceneffizienz

In diesem Abschnitt wird nun der im vorigen Kapitel ermittelte Performanzgewinn dem steigenden Ressourcenbedarf durch den zusätzlichen Scratchpad-Speicher in Relation gesetzt. Während bei dem Vergleich der Ressourceneffizienz bisher immer eine identische Code-Basis vorausgesetzt wurde, muss nun berücksichtigt werden, dass die Performanzergebnisse für die jeweilige Implementierung mit und ohne Scratchpad-Speicher auf *unterschiedlichem* Quellcode beruht. Während die eigentliche Algorithmus der Anwendung für beide Fälle identisch ist, so ergeben sich doch starke Unterschiede beim Speicherzugriff. Durch den Einsatz von Scratchpad-Speicher werden viele Speicherzugriffe vom externen Hauptspeicher zum internen Scratchpad-Speicher verschoben. Zugriffe auf externe Komponenten, wie SDRAM, verursachen allerdings eine weitaus höhere Leistungsaufnahme als auf internen statischen Speicher (SRAM). Daher wurden auch die SDRAM-Zugriffe bei der Bewertung der Ressourceneffizienz berücksichtigt. Als Beispiel wurde die Leistungsaufnahme für den auf dem FPGA-Prototyp (vgl. Kapitel 8 eingesetzten 128MB SDRAM von

Tabelle 7.5.: *Vergleich des Flächenbedarfs des CoreVA-Systems mit und ohne Scratchpad-Speicher*

Flächenbedarf [μm^2]	ohne Scratchpad	mit Scratchpad	Zuwachs [%]
Gesamt	1731765	1829198	+5,63
Prozessorkern	586321	631428	+7,69
Instr. Cache	240661	239816	-0,35
Daten Cache	420729	419251	-0,35
Systembus-Arbiter	16181	15469	-4,40
Scratchpad-Speicher	0	43417	-
DMA-Controller	0	10104	-
Sonstiges	467873	469713	+0,39

Micron Technology[5]) mit Hilfe von [136, 137] anwendungsspezifisch abgeschätzt. Tabelle 7.5 und 7.6 zeigen den Ressourcenbedarf für die beiden Anwendungen mit und ohne Scratchpad-Speicher. Die Ergebnisse beziehen sich auf eine Taktperiode von 4 ns.

Tabelle 7.6.: *Vergleich der Leistungsaufnahme [mW] des CoreVA-Systems bei Ausführung verschiedener Anwendungen mit und ohne Scratchpad-Speicher*

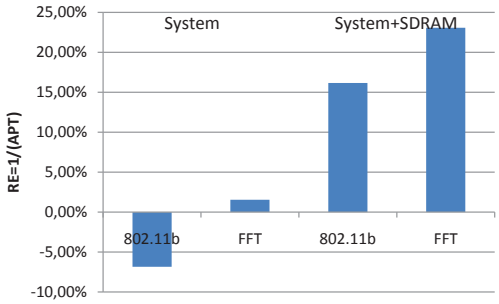
	FFT			IEEE 802.11b		
	ohne Scratchp.	mit Scratchp.	Zuwachs [%]	ohne Scratchp.	mit Scratchp.	Zuwachs [%]
Gesamt	349,4	390,5	+12,0	338,10	358,40	+6,0
Prozssorkern	13,3	22,2	+66,9	14,7	20,3	+38,1
Instruktions-Cache	16,0	30,7	+91,9	21,9	34,7	+58,5
Daten-Cache	42,4	44,4	+4,7	40,5	47,9	+18,3
Systembus-Arbiter	0,181	0,245	+35,4	0,016	0,0198	+23,8
Scratchpad-Speicher	0	0,1	-	0	0,1	-
DMA-Controller	0	0,2	-	0	0,02	-
Sonstiges	4,5	4,9	+7,5	4,6	5,0	+8,3
SDRAM	273,0	287,0	+5,1	256,0	250,0	-2,3

Der Flächenbedarf des CoreVA-Systems steigt durch den zusätzlichen Scratchpad-Speicher, den DMA-Controller und die zwei zusätzlichen LD/ST-Einheiten im Prozessorkern um knapp 6 %. Bei der Leistungsaufnahme ergibt sich in Abhängigkeit von der Anwendung ein differenziertes Ergebnis. Für beide Anwendungen steigt

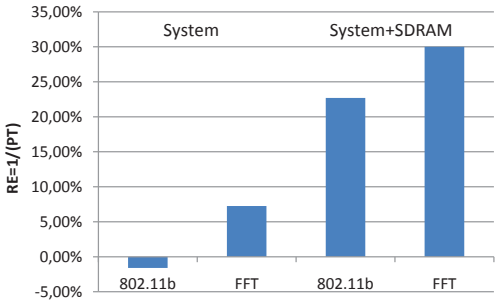
[5] 8 DQs, -7E Speed Grade, 100MHz Taktfrequenz und 3,3V Versorgungsspannung

die Leistungsaufnahme des Prozessorkerns, der Caches und des Systembus-Arbiters. Dieses ist auf die geringere Anzahl an Strafzyklen zurückzuführen, in denen der Prozessorkern vom Daten-Cache angehalten wird. Insbesondere der steigende Anteil der Speicherzugriffe auf den Instruktionsspeicher an der Gesamtlaufzeit bewirkt eine höhere durchschnittliche Leistungsaufnahme. Die Leistungsaufnahme für den SDRAM steigt beim FFT-Algorithmus um 5,1 %, beim IEEE-802.11b-Algorithmus sinkt sie sogar leicht (-2,3 %). Dieses ist auf die unterschiedlichen Zugriffsmuster auf den SDRAM vor und nach der Optimierung des Systems durch Scratchpad-Speicher zurückzuführen. Insgesamt verursachen die SDRAM-Zugriffe den Großteil der Verlustleistung (FFT: 73 %, IEEE 802.11b: 80 %). Die Leistungsaufnahme des Gesamtsystems bestehend aus CoreVA-System und SDRAM steigt um 12 % (FFT) bzw. 6 % (IEEE 802.11b).

Auf die ermittelte Ausführungszeit und den Ressourcenbedarf wurde nun das in Kapitel 3.3 vorgestellte Maß zur Bewertung der Ressourceneffizienz angewendet. Abbildung 7.13a zeigt die Steigerung der Ressourceneffizienz der Scratchpad-Erweiterung des CoreVA-Systems im Vergleich zum Original-System für einen Ressourceneffizienz-Index von $RE = 1/(A \cdot P \cdot T)$, d.h. für eine gleiche Gewichtung aller Ressourcen. Bei ausschließlicher Betrachtung des Prozessorkerns ohne SDRAM ergibt sich differenziertes Bild. Für den IEEE-802.11b-Algorithmus sinkt die Ressourceneffizienz. Beim FFT-Algorithmus steigt sie dagenen leicht. Unter Berücksichtigung der SDRAM-Zugriffe steigt die Ressourceneffizienz für die Systemarchitektur mit Scratchpad-Erweiterung für beide Anwendungen um 16 % bis 23 %.



(a) $RE = 1/(A \cdot P \cdot T)$



(b) $RE = 1/(P \cdot T)$ (Energie)

Abbildung 7.13.: Steigerung der Ressourceneffizienz des Gesamtsystems durch zusätzlichen Scratchpad-Speicher

Auch unter Annahme des energiebeschränkten Anwendungsszenarios ($RE = 1/(P \cdot T)$) dominiert die Architekturvariante mit der Scratchpad-Speichererweiterung um bis zu 30 % (vgl. Abbildung 7.13b).

7.3. Optimierung durch Hardware-Erweiterungen

Dieser Abschnitt beschreibt einen *hierarchischen Optimierungsansatz* zur Erweiterung des CoreVA-Systems um spezialisierte Hardware zur Erhöhung der Ressourceneffizienz. Der Ansatz beginnt bei *direkt in die ALUs* des Prozessorkerns integrierten *Instruktionssatzerweiterungen* (Instruction Set Extension (ISE)) und geht über *eng gekoppelte aber dedizierte Hardware-Beschleuniger* hin zu *extern und lose angebundenen Hardware-Beschleunigern* auf dedizierten, rekonfigurierbaren Bausteinen, wie z. B. FPGAs.

7.3.1. Instruktionssatzerweiterungen

Instruktionssatzerweiterungen sind das direkte Resultat *wiederholter Iterationszyklen* der Entwurfsraumexploration mithilfe des in Kapitel 3 vorgestellten Entwurfsablauf [244]. Durch die zyklenakkurate Analyse einer Anwendung mit den Profiling-Werkzeugen des Entwurfsablaufs lassen sich häufig auftretende Kombinationen von Instruktionen mit direkter Datenabhängigkeit bestimmen, die sich als mögliche Kandidaten für Instruktionssatzerweiterungen erweisen. Solche Instruktionssatzerweiterungen kombinieren die Funktionalität zweier Instruktionen. Die Reduktion der Ausführungszeit durch Instruktionssatzerweiterungen ist proportional zur Anzahl solcher „Superinstruktionen“ an der Gesamtzahl der Instruktionen und der Anzahl der kombinierten Instruktionen (Instruktionspaare, -tripel, -quadrupel etc.). Des Weiteren führt die Kombination von Instruktionen zu einer Reduktion der Anzahl der Instruktionen im Maschinencode und so ggf. zu einem geänderten Scheduling. Abbildung 7.14 zeigt den theoretischen Performanzgewinn bezüglich der Anzahl an Taktzyklen in Abhängigkeit des Anteils der kombinierten Instruktionen an der Gesamtausführungszeit für Instruktionssequenzen der Länge zwei bis fünf. An dieser Abbildung zeigt sich bereits, dass der erwartete Performanzgewinn für übliche Anwendungen eher gering ausfallen wird. Verdeutlicht werden soll dieses an einem Beispiel: Man nehme eine Instruktionssequenz bestehend aus zwei datenabhängigen Instruktionen. Tritt diese Kombination in 20 % der Taktzyklen auf, d.h. ist eine von fünf Instruktionen eine von diesen Beiden, so kann die Gesamtzahl der Taktzyklen lediglich um 10 % reduziert werden. Bei 10 % der Taktzyklen beträgt der Performanzgewinn bereits nur noch 5 %. Datenabhängige Instruktionstripel oder Instruktionsquadrupel treten sehr viel seltener auf. Auf den ersten Blick scheint diese Optimierungsmethodik ineffizient. Betrachtet man die Integration einer Instruktionssatzerweiterung jedoch als einzelnen Iterationsschritt einer größeren Entwurfsraumexploration so lässt sich das Potential dieses Optimierungsansatzes durchaus erschließen. Beginnend bei einem Basis-Instruktionssatz lassen sich mithilfe des vorgestellten Entwurfsablaufs meist viele Instruktionssatzerweiterungen finden, deren Kombination in einem Gesamtsystem ein hohes Optimierungspotential bietet. Wie in den folgenden Abschnitten gezeigt werden wird, so lässt sich

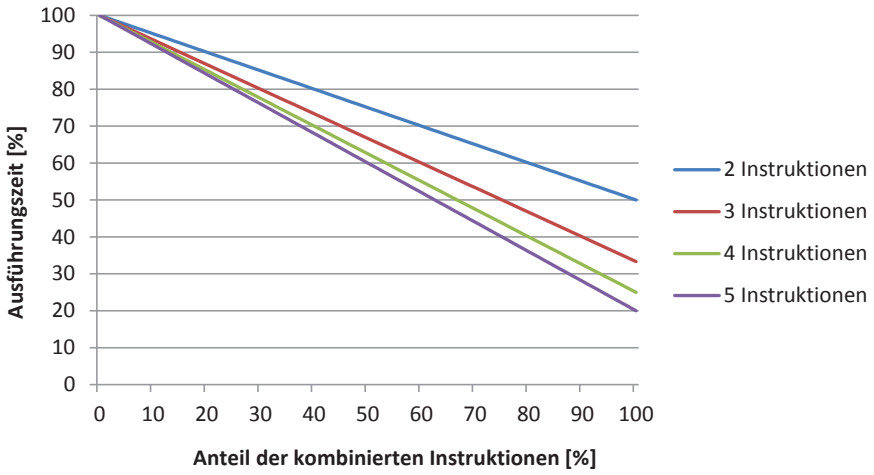


Abbildung 7.14.: Theoretischer Performanzgewinn bezüglich der Anzahl an Taktzyklen in Abhängigkeit des Anteils der kombinierten Instruktionen an der Gesamtausführungszeit für Instruktionssequenzen der Länge zwei bis fünf

die Kombinationen der Logik bestehender Instruktionen häufig mit *vernachlässigbar geringem Hardware-Aufwand* realisieren. Der kritische Pfad und damit die maximal mögliche Taktfrequenz des Systems bleibt meist gleich. Ein ähnliches Verfahren wird in [153] vorgestellt. Hier wurden Performanzgewinne von etwa 10 % erzielt. Allgemein betrachtet sind Instruktionssatzerweiterungen *anwendungsunabhängig*. Beliebige Anwendungen können von ihnen profitieren. Ihr größtes Optimierungspotential entfalten sie aber meist bei der Anwendung, aus deren Analyse sie hervorgegangen sind. Abbildung 7.15 zeigt die Architektur des CoreVA-System nach Optimierung durch Instruktionssatzerweiterungen. Die folgenden Abschnitte zeigen exemplarisch die Möglichkeiten der Steigerung der Ressourceneffizienz einer Hardware-Software-Kombination durch die Implementierung von Instruktionssatzerweiterungen.

7.3.2. Anbindung von Hardware-Erweiterungen

Dieser Abschnitt beschreibt die Erweiterung des CoreVA-Systems um dedizierte Hardware-Beschleuniger. Hardware-Beschleuniger sind in erster Linie anwendungsspezifisch, bieten dagegen aber Performanzgewinne von bis zu mehreren Größenordnungen. Der erhöhte Ressourcenbedarf und eine mögliche Beschränkung der maximalen Taktfrequenz erfordert eine genaue Analyse der Kombination aus Hardware- und Software bei der Bewertung der Ressourceneffizienz. Des Weiteren muss die Auswahl der Hardware-Beschleuniger und damit das genaue Anwendungssze-

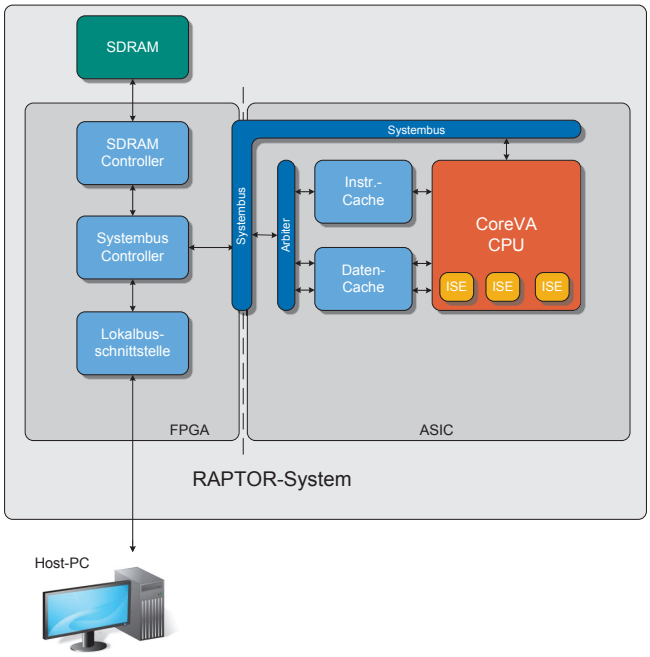


Abbildung 7.15.: Architektur des CoreVA-Systems nach Erweiterung um Instruktionssatzerweiterungen

nario bereits während der Entwicklung feststehen. Nachträgliche Änderungen an der Spezifikation der Anwendungen sind nicht mehr möglich. Auch kann durch eine Kostenbegrenzung die Anzahl von Hardware-Beschleunigern beschränkt sein. In Abschnitt 7.5 wird daher ein System vorgestellt, welches die flexible, modulbasierte Erweiterung des Systems durch Hardware-Beschleuniger auch nach Fertigung eines ASIC-Prototyps erlaubt.

Da die implementierte Schnittstelle sowohl zur Anbindung von Hardware-Beschleunigern als auch zur Erweiterung des Systems um physikalische Schnittstellen, wie *Ethernet* oder *Universal Asynchronous Receiver Transmitter (UART)*, genutzt werden kann, werden die dedizierten Hardwarekomponenten im Folgenden auch als *Hardware-Erweiterungen* bezeichnet. In Abbildung 7.16 ist die Architektur des CoreVA-Systems nach dem Hinzufügen der dedizierten Hardware-Erweiterungen dargestellt. Die Anbindung von Hardware-Erweiterungen erfolgt über das MMIO–Prinzip. Der adressierbare Adressraum wird in zwei Teile ($2 \cdot 2^{31}$ Bit) aufgeteilt. In einem Adressbereich wird der physikalische Speicher abgebildet. In dem anderen Adressbereich werden die Hardware-Erweiterungen eingeblendet. Ein Adressdekoder wird hierfür zwischen Speicherschnittstelle des Prozessors und den Daten-Cache geschaltet.

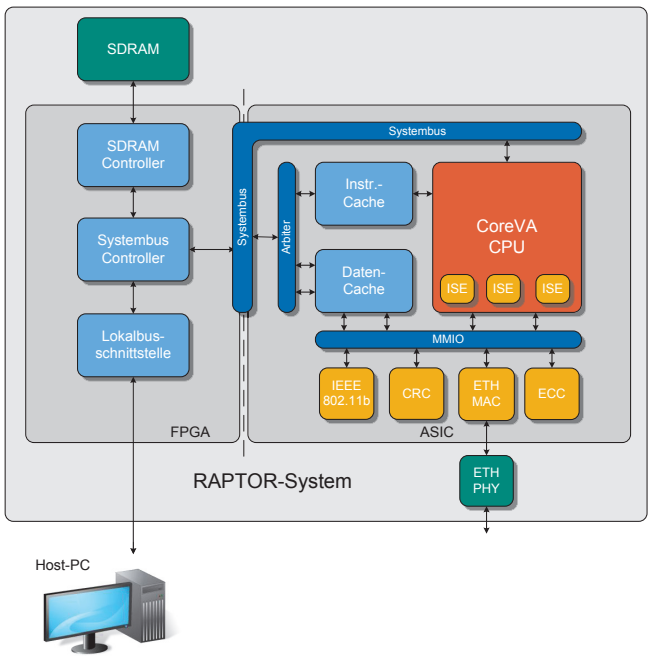


Abbildung 7.16.: Architektur des CoreVA-Systems nach Erweiterung um dedizierte Hardware-Beschleuniger

Abbildung 7.17 zeigt die Architektur des Adressdekoders. Anhand der Adresse wird entschieden, ob der Speicherzugriff an den Daten-Cache oder die Hardware-Erweiterungen weitergeleitet wird. Der Adressbereich für die Hardware-Erweiterungen wiederum wird in 32 gleiche Teile unterteilt. Jeder Hardware-Beschleuniger verfügt somit über einen Adressraum von $2^{(32-1-5)} = 2^{26}$ Bit. Abbildung 7.18 zeigt die Aufteilung des Adressraumes des Datenspeichers.

Die Latenz eines Zugriffs auf eine Hardware-Erweiterung ist im besten Fall gleich der Latenz eines Zugriffs auf den Cache bei einem Treffer, d.h. ein Taktzyklus bei einem Schreib- und zwei Taktzyklen bei einem Lesezugriff. Erfordert die Hardware-Erweiterung zur Bearbeitung des Zugriffs mehrere Taktzyklen, so kann eine Hardware-Erweiterung den Prozessorkern anhalten. Die Latenz des Zugriffs steigt dementsprechend an. Die zusätzliche Addressdekoderlogik verursacht einen Anstieg des Flächenbedarfs und der durchschnittlichen dynamischen Leistungsaufnahme von jeweils etwa 1 %.

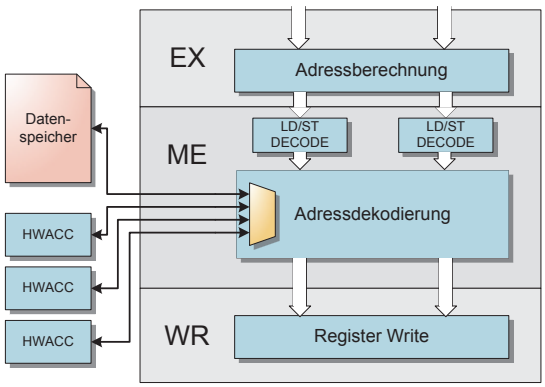


Abbildung 7.17.: Architektur des Adressdekoders

7.3.3. Optimierung eines IEEE-801.11b-Algorithmus

In diesem Abschnitt wird beispielhaft die Optimierung des in Abschnitt 5.2 vorgestellten IEEE-802.11b-Algorithmus mithilfe eng gekoppelter Instruktionssatzerweiterungen und dedizierter Hardware-Beschleuniger aufgezeigt [238, 241, 234]. Die Analyse der Anwendung zeigte eine hohe Häufigkeit von aufeinanderfolgenden und voneinander abhängigen MLA– und ASR[6] –Instruktionen. Die Funktionalität beider Instruktionen wurde daher in der Instruktionssatzerweiterung `MLA_ASR10` kombiniert. Die Änderungen der Architektur (vgl. Abbildung 7.19) beschränken sich auf wenige Multiplexer. Durch diese vernachlässigbaren Änderungen konnte die Ausführungszeit des IEEE-801.11b-Algorithmus allerdings um mehr als 4 % reduziert werden. Der kritische Pfad des Systems wurde die die zusätzliche Logik nicht verändert. Eine Erhöhung des Ressourcenbedarfs war nicht messbar. Dieses Beispiel soll das Vorgehen bei der Integration von Instruktionssatzerweiterungen verdeutlichen und kann auf weitere Instruktionspaare ausgedehnt werden.

[6] Arithmetic Shift Right

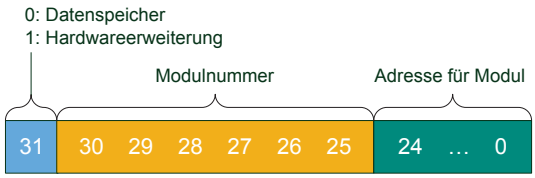


Abbildung 7.18.: Aufteilung des Adressraumes des Datenspeichers

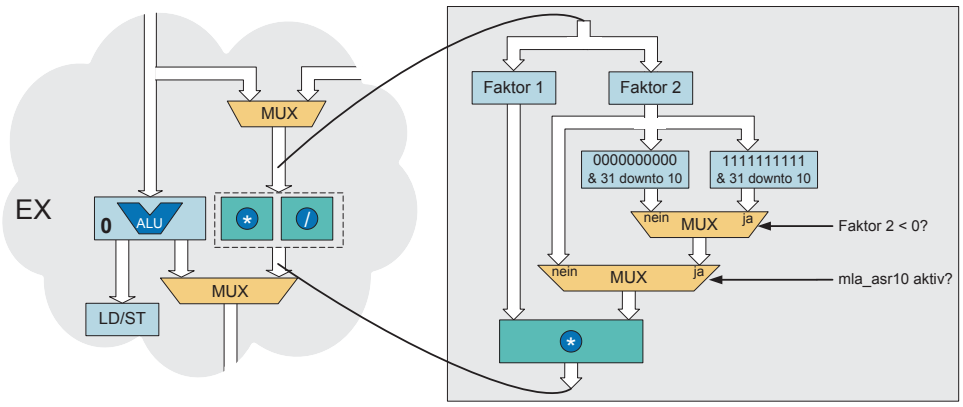


Abbildung 7.19.: Integration der MLA_ASR10–Instruktionssatzerweiterung in die Pipelinestruktur des CoreVA-Prozessors

Im weiteren Verlauf der Optimierungen wurde ein Hardware-Beschleuniger implementiert, der die Verarbeitung des IEEE-802.11b-Algorithmus durchführt. Abbildung 7.20 zeigt die Architektur dieses Hardware-Beschleunigers [250]. Um den kritischen Pfad des Gesamtsystems nicht zu verlängern wurden die Ein- und Ausgänge durch Register vom Prozessorkern entkoppelt. Die Nutzdaten werden über MMIO in die internen Register des Hardware-Beschleunigers geschrieben. Wie in Abschnitt 5.2 beschrieben, ist die Funktionalität des Algorithmus in vier Funktionen unterteilt: *Scrambler*, *Differential Encoder*, *Chunks-to-Symbols* und *Interpolation-FIR-Filter*. Die Funktionalität des Hardware-Beschleunigers orientiert sich an diesem Aufbau. Ein Zustandsautomat verteilt die Daten an die vier Komponenten. Die Latenz zur Bearbeitung eines Bits der Nutzdaten beträgt fünf Taktzyklen. In dieser Zeit kann je nach Konfiguration der Prozessorkern entweder angehalten werden oder in der Zwischenzeit andere Aufgaben übernehmen. Nach dieser Zeit können die diskreten Werte des späteren analogen Basisbandsignals aus den Ausgangsregistern des Hardware-Beschleunigers ausgelesen werden. Die Funktionalität der Software-Anwendung beschränkt sich somit fast ausschließlich auf das Schreiben der Nutzdaten und das Lesen des Ergebnisses in und aus dem Hardware-Beschleuniger. Die Ausführungszeit reduzierte sich im Vergleich zur C-optimierten Software-Implementierung um ca. 88 %. Der Flächenbedarf des IEEE-802.11b-Hardware-Beschleunigers in einer 65 nm Standardzellentechnologie von STMicroelectronics liegt bei etwa 0,12 mm^2, was 38 % der Fläche des Prozessorkerns entspricht. Die durchschnittliche Leistungsaufnahme bei einer Taktfrequenz von 358 MHz beträgt 2,8 mW (19 % der Leistungsaufnahme des Prozessorkerns).

Wendet man das in Abschnitt 3.4 vorgestellte Maß zur Bewertung der Ressour-

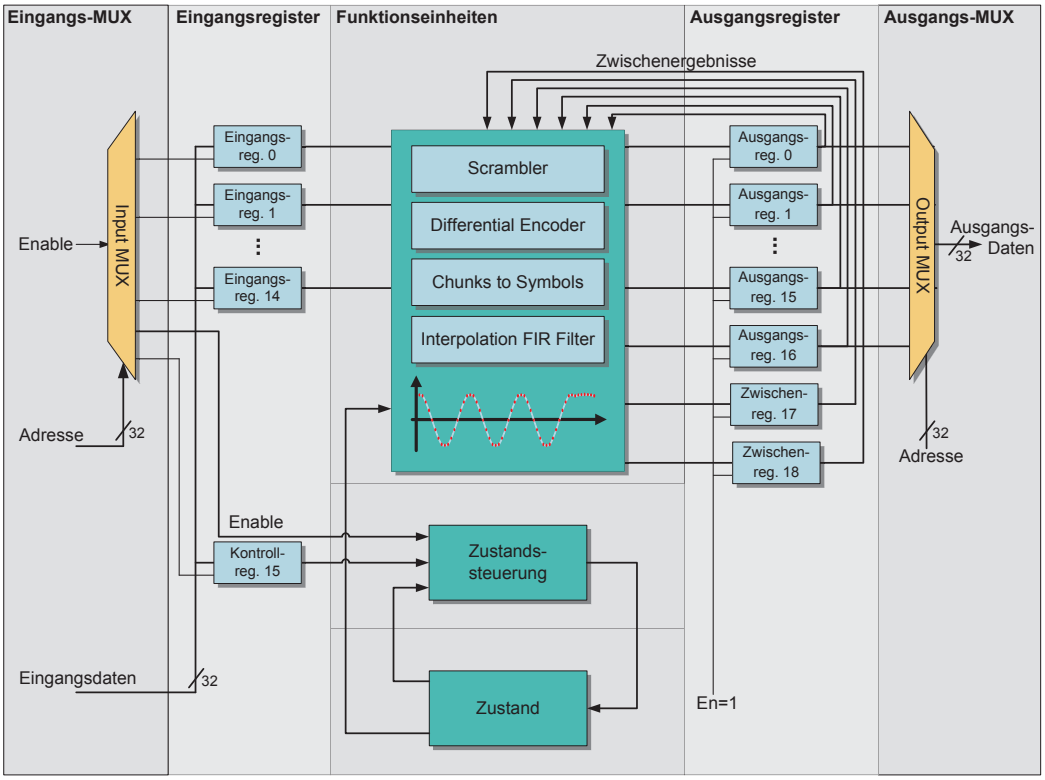


Abbildung 7.20.: Blockschaltbild des IEEE-802.11b-Hardware-Beschleunigers

ceneffizienz an, so steigert sich die Effizienz bei $RE = 1/(A \cdot P \cdot T)$ bei Einsatz der *Instruktionssatzerweiterung* proportional zur Verringerung der Ausführungszeit um ca. 4 %. Bei Einsatz des dedizierten *Hardware-Beschleunigers* steigert sich die Effizienz um den Faktor 5,12 ($\widehat{=}$+412 %). Für $RE = 1/(P \cdot T)$ ergibt sich eine Steigerung der Effizienz um +4 % (ISE) respektive +604 % (Hardware-Beschleuniger).

7.3.4. Optimierung eines Algorithmus zur zyklischen Redundanzprüfung (CRC)

Für die Beschleunigung des in Abschnitt 5.2.7 vorgestellten Algorithmus zur zyklischen Redundanzprüfung wurde in [233] ein dedizierter Hardware-Beschleuniger entwickelt, der auch bereits in [153] zum Einsatz kam. In jedem Taktzyklus kann ein 32-Bit-Datum verarbeitet werden. Die Prüfsumme wird mit jedem Schreibzugriff aktualisiert und kann anschließend ausgelesen werden. Für 1000 Byte Nutzdaten reduziert sich die Ausführungszeit von 8029 auf 1018 Taktzyklen (-87 %). Der zusätz-

liche Flächenbedarf des CRC-Hardware-Beschleunigers liegt bei 2737 μm^2 (+0,8 %). Die Leistungsaufnahme erhöht sich um 82 μW (+0,6 %). Die Ressourceneffizienz des Gesamtsystems erhöht sich durch Einsatz des Hardware-Beschleunigers um 677 % ($RE = 1/(A \cdot P \cdot T)$) respektive 684 % ($RE = 1/(P \cdot T)$).

7.3.5. Optimierung eines symmetrischen Verschlüsslungsverfahrens (AES)

Für den in Abschnitt 5.2.9 vorgestellten AES-Algorithmus wurde ein dedizierter Hardware-Beschleuniger implementiert, der sowohl die *Schlüsselberechnung*, als auch die *Ver- und Entschlüsselung von Daten* ermöglicht. Es werden Schlüssellängen von 128–192 Bits unterstützt. Über 32-Bit-MMIO-Schreibzugriffe werden der Schlüssel und die Daten an den Hardware-Beschleuniger geschickt. Nach der Verschlüsselung können die verschlüsselten Daten über entsprechende MMIO-Lesezugriffe ausgelesen werden. Durch die Hardware-Beschleunigung lässt sich die Ausführungszeit des AES-Algorithmus um mehr als 97 % reduzieren. Dagegen steigt der Flächenbedarf des Gesamtsystems um 0,108 mm^2 (+49 %) sowie die Leistungsaufnahme um 13,23 mW (+116 %, 358 MHz). Die Leistungsaufnahme des Gesamtsystem verdoppelt sich demnach während der Nutzung des AES-Hardware-Beschleunigers. Dennoch kann durch die starke Reduzierung der Ausführungszeit die Ressourceneffizienz des Systems signifikant gesteigert werden: Für $RE = 1/(A \cdot P \cdot T)$ ergibt sich eine Steigerung der Effizienz um 958 % (Faktor 11) bzw. um 1478 % (Faktor 16) für die Energieeffizienz ($RE = 1/(P \cdot T)$).

7.3.6. Optimierung eines Algorithmus zur Kryptographie mit elliptischen Kurven (ECC)

Dieser Abschnitt zeigt die Optimierung eines Algorithmus zur Kryptographie mit elliptischen Kurven (ECC, vgl. Abschnitt 5.2.8). Die Kryptographie mit elliptischen Kurven basiert auf der Arithmetik endlicher Körper der Charakteristik 2. Operationen auf endlichen Körpern werden in diesem Abschnitt aber der Einfachheit halber mit *Multiplikation*, *Quadrierung* etc. bezeichnet.

Durch das Profiling des C-Codes konnten drei *Instruktionssatzerweiterungen* identifiziert werden die sich sehr gut zur Optimierung der Performanz des Algorithmus eignen [244]: Die „Superinstruktionen“ `lslxor` (Logischen Linksschieben und XOR-Verknüpfen), `lsrxor` (Logischen Rechtsschieben und XOR-Verknüpfen) und `mvbits` (Extrahieren von Teilwörtern aus einem 32-Bit-Datenwort). Die ersten beiden Instruktionssatzerweiterungen spiegeln die Funktionsweise der Multiplikation wieder, die letzte die der Quadrierung. Die CoreVA-Implementierung mit der Erweiterung um die Instruktionssatzerweiterungen wird im Folgenden mit *CoreVA(ISE)* bezeichnet.

Als zweiter Optimierungsschritt wurden drei *Hardware-Beschleuniger* entwickelt (vgl. Abbildung 7.21):

- *CoreVA(SQU233)* repräsentieren die Funktionalität der *Quadrierung* basierend auf endlichen Körpern der Charakteristik 2.
- *CoreVA(MUL233)* repräsentieren die Funktionalität der *Multiplikation* basierend auf endlichen Körpern der Charakteristik 2.
- *CoreVA(FF233)* kombiniert die Funktionalität *beider* Hardware-Beschleuniger.

Die Latenz aller Hardware-Beschleuniger beträgt einen Taktzyklus [244].

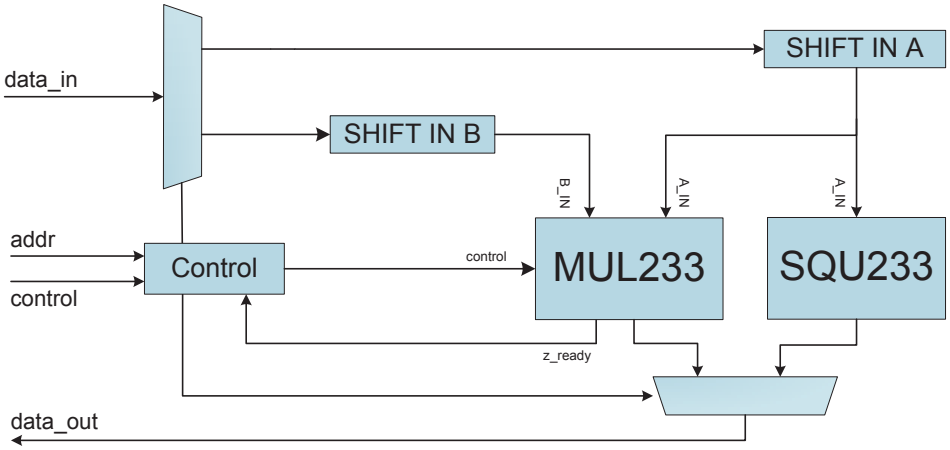


Abbildung 7.21.: Architektur des ECC-Hardware-Beschleunigers

Tabelle 7.7 zeigt die Anzahl der Taktzyklen der ursprünglichen Implementierung (CoreVA), der Implementierung mit Instruktionssatzerweiterungen und der durch Hardware-Beschleuniger optimierten Version. Die (32-Bit)-Wortmultiplikation (basierend auf endlichen Körpern – nicht zu verwechseln mit einer „arithmetischen" 32-Bit-Multiplikation) lässt sich durch die Instruktionssatzerweiterungen um 10 % gegenüber der ursprünglichen Implementierung beschleunigen. Die Ausführungszeit der eigentlichen Multiplikation auf endlichen Körpern reduziert sich um ca. 4 %. Die Quadrierung lässt sich durch die Instruktionssatzerweiterungen um knapp 3 % beschleunigen. Der Performanzgewinn des gesamten ECC-Algorithmus beträgt 4 %. Die Hardware-beschleunigten Architekturen ermöglichen einen weitaus höheren Performanzgewinn. Die Ausführungszeit der Quadrierung reduziert sich auf der Architektur *CoreVA(SQU233)* auf 13 %, d.h. die Performanz erhöht sich um den Faktor sieben. *CoreVA(MUL233)* reduziert die Ausführungszeit für die Multiplikation signifikant von 2111 Taktzyklen auf 73 Taktzyklen (-97 %, Faktor 29). Verwendet man die

CoreVA(FF233)-Architektur, welche die gesamte Skalarmultiplikation auf endlichen Körpern der Charakteristik 2 durch den Hardware-Beschleuniger ausführt, so lässt sich ein Performanzgewinn um den Faktor 14 (-92,66 %) erzielen.

Tabelle 7.7.: *Anzahl der Taktzyklen unter Verwendung verschiedener Arten der Hardware-Beschleunigung für die Arithmetik basierend auf endlichen Körpern (Charakteristik 2)*

	Skalarmultiplikation		Multiplikation		Quadrierung	
	Taktzyklen	Relativ [%]	Taktzyklen	Relativ [%]	Taktzyklen	Relativ [%]
CoreVA	3552571	100,00	2111	100,00	353	100,00
CoreVA(ISE)	3408875	95,96	2018	95,59	344	97,45
CoreVA(SQU233)	3129187	88,08	2111	100,00	49	13,88
CoreVA(MUL233)	684999	19,28	73	3,46	353	100,00
CoreVA(FF233)	260615	7,34	73	3,46	49	13,88

Die Menge *lokalen On-Chip-Speichers* ist eine kritische Größe in eingebetteten Systemen. Neben der Reduktion der Ausführungszeiten lässt sich durch Einsatz von Hardware-Beschleunigern auch die Code-Größe reduzieren. Tabelle 7.8 zeigt die Reduktion für die verschiedenen hardware-beschleunigten Implementierungen [244]. Die Instruktionssatzerweiterungen ermöglichen eine Reduktion der Codegröße um 16,25 %. Die hardware-beschleunigten Architekturen erlauben eine Ersparnis von bis zu 34,87 % (*CoreVA(FF233)*), respektive -17,8 % (*CoreVA(SQU233)*) und -32,39 % (*CoreVA(MUL233)*).

Tabelle 7.8.: *Codegröße unter Verwendung verschiedener Arten der Hardware-Beschleunigung für die Arithmetik basierend auf endlichen Körpern (Charakteristik 2)*

	Skalarmultiplikation		Multiplikation		Quadrierung	
	Codegröße [Bytes]	Relativ [%]	Codegröße [Bytes]	Relativ [%]	Codegröße [Bytes]	Relativ [%]
CoreVA	10336	100,00	2052	100,00	528	100,00
CoreVA(ISE)	8656	83,75	1940	94,54	496	93,94
CoreVA(SQU233)	8496	82,20	2052	100,00	208	39,39
CoreVA(MUL233)	6988	67,61	304	14,81	528	100,00
CoreVA(FF233)	6732	65,13	2052	100,00	208	39,39

Zur Bewertung der Ressourceneffizienz wird im Folgenden nun der zusätzliche Hardwarebedarf für die fünf verschiedenen Architekturvarianten ermittelt. Tabelle 7.9 zeigt den Ressourcenbedarf bei einer Taktfrequenz von 358 MHz (Worst Case). Die Implementierung der zusätzlichen Instruktionen bedarf nur geringfügiger Änderungen am Instruktionsdekoder und den ALUs. Der Ressourcenbedarf der

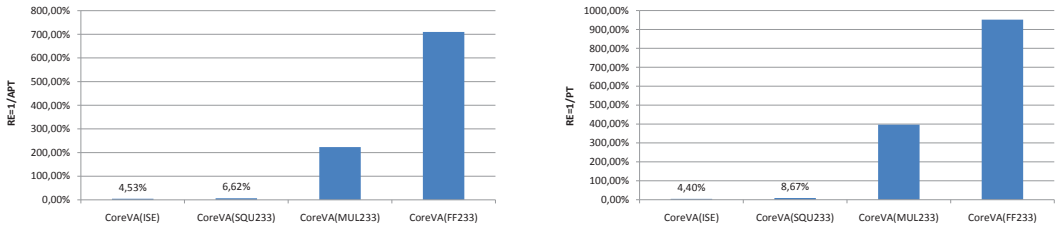
Architektur wurde hierdurch in nicht messbarer Weise beeinflusst. Durch die Verwendung pseudozufälliger Sequenzen in modernen Synthesewerkzeugen reduziert sich der Ressourcenbedarf trotz zusätzlicher Instruktionen sogar leicht. Die Logik der zusätzlichen Instruktionen liegt zudem nicht im kritischen Pfad und beeinflusst die maximale Taktfrequenz nicht. Die Fläche des *CoreVA(SQU233)*-Hardware-Beschleunigers beträgt ca. 2 % der Fläche des CoreVA-Prozessors ohne Hardware-Beschleuniger (*CoreVA(MUL233)*: +28 %). Die Erhöhung der durchschnittlichen Leistungsaufnahme beträgt ca. 4 % (*CoreVA(MUL233)*: +26 %). Der Zuwachs der Fläche und der Leistungsaufnahme für die *CoreVA(FF233)*-Architektur beträgt knapp 30 % [244].

Tabelle 7.9.: *Ressourcenbedarf der hardware-beschleunigten Architekturvarianten zur Optimierung des ECC-Algorithmus für eine Taktfrequenz von 358 MHz*

	Fläche [μm^2]	Relativ [%]	Leistungsaufnahme [mW]	Relativ [%]
CoreVA	317570	100,00%	14,80	100,00%
CoreVA(ISE)	317148	99,87%	14,77	99,83%
CoreVA(SQU233)	323677	101,92%	15,47	104,48%
CoreVA(MUL233)	405591	127,72%	18,60	125,65%
CoreVA(FF233)	412439	129,87%	19,19	129,60%

Zur Bewertung der Ressourceneffizienz wird nun die Ausführungszeit in Bezug zum Ressourcenbedarf gesetzt. Abbildung 7.22a zeigt die Anwendung des in Abschnitt 3.4 vorgestellte Maßes zur Bewertung der Ressourceneffizienz für einen gleiche Gewichtung aller Ressourcen ($RE = 1/(A \cdot P \cdot T)$). Die Ergebnisse stellen die Unterschiede der Ressourceneffizienz normiert auf das CoreVA-System ohne Hardware-Beschleuniger dar. Durch Implementierung der Instruktionssatzerweiterungen (*CoreVA(ISE)*) lässt sich die Ressourceneffizienz ohne messbaren Hardwareoverhead um knapp 5 % verbessern. Die Reduzierung der Ausführungszeit um 12 % durch Einsatz des ersten Hardware-Beschleunigers (*CoreVA(SQU233)*) wird durch die Erhöhung des Flächenbedarfs und der Leistungsaufnahme wieder ausgeglichen. Die Effizienz der Architektur erhöht sich hier nur leicht um knapp 7 %. Das größte Optimierungspotential bieten der zweite Hardware-Beschleuniger bzw. die Kombination aus beiden Hardware-Beschleunigern. Hier sind Steigerungen der Effizienz um den Faktor drei (+223 %) bzw. Faktor acht (+710 %) möglich. Unter Anwendung des Bewertungsmaßes ($RE = 1/(P \cdot T)$, vgl. Abbildung 7.22b) lässt sich die Energieeffizienz um 4,4 % (*CoreVA(ISE)*), 8,7 % (*CoreVA(SQU233)*), bzw. um den Faktor fünf (*CoreVA(MUL233)*, +396 %) und den Faktor elf (*CoreVA(FF233)*, +952 %) steigern.

Die Instruktionssatzerweiterungen bieten großes Optimierungspotential bei beschränkten Ressourcen. Sie sind flexibel und auch für andere Anwendungen nützlich. Hardware-Beschleuniger erlauben eine weitaus höhere Steigerung der Ressourceneffizienz. Allerdings sind sie anwendungsspezifisch und nur für den untersuchten



(a) $RE = 1/(A \cdot P \cdot T)$ **(b)** $RE = 1/(P \cdot T)$

Abbildung 7.22.: Vergleich der Ressourceneffizienz unter Einsatz der ECC-Hardware-Erweiterungen

ECC-Algorithmus implementiert. Werden andere Anwendungen auf einer solchen Architektur genutzt, so sinkt die Ressourceneffizienz proportional zum Hardwareoverhead der Hardware-Beschleuniger. Diese Tatsache zeigt auch die Notwendigkeit einer detaillierten Entwurfsraumexploration nicht nur auf Anwendungsebene, sondern auch unter Betrachtung des Ressourcenbedarfs.

Tabelle 7.10 fasst die Ergebnisse der für die dedizierten Hardware-Beschleuniger aus den vorherigen Abschnitten zusammen. Es zeigt sich, dass Hardware-Beschleuniger die Ressourceneffizienz eines Prozessorsystems erhöhen können, aber auch einen Anstieg der Ressourcen Fläche und Leistungsaufnahme verursachen. Sind Chip-Fläche oder Leistungsaufnahme durch eine obere Schranke begrenzt (beispielsweise durch die Kosten – vgl. Gleichung 3.6), können nicht alle Hardware-Erweiterungen gleichzeitig integriert werden. Abschnitt 7.5 stellt daher die Erweiterung des Prozessorsystems um eine *generische Schnittstelle* vor, mit der die flexible Integration von *externen Hardware-Erweiterungen* auf Systemebene, auch nach Fertigung eines ASIC-Prototyps, möglich ist.

Tabelle 7.10.: *Vergleich der Ressourceneffizienz bei Nutzung der verschiedenen Hardware-Beschleuniger*

Hardware-beschleuniger	Ausführungs-zeit (Faktor)	Flächen-bedarf	Leistungs-aufnahme	Ressourceneffizienz (Faktor) $RE = 1/(A \cdot P \cdot T)$	$RE = 1/(P \cdot T)$
802.11b	-88 % (×8)	+38 %	+19 %	**+412 % (×5)**	**+604 % (×7)**
CRC	-87 % (×8)	+0,8 %	+0,6 %	**+677 % (×8)**	**+684 % (×8)**
AES	-99 % (×66)	+49 %	+116 %	**+958 % (×11)**	**+1478 % (×16)**
ECC	-93 % (×14)	+30 %	+30 %	**+710 % (×8)**	**+952 % (×11)**

7.3.7. Hardware-Erweiterung zur externen Kommunikation

Zusätzlich zu den in den Abschnitten 7.3.3 bis 7.3.4 beschriebenen Hardware-Beschleunigern wurden zwei Hardware-Erweiterungen implementiert, die zu Debug- und Verifikationszwecken eingesetzt werden können. Die Hardware-Erweiterung `HWACC_STDOUT`[7] implementiert eine UART-Schnittstelle zur Standardein- und Standardausgabe. Die Hardware-Erweiterung `HWACC_FIFO` erlaubt den Datenaustausch zwischen Prozessor und Host– PC[8] der in Kapitel 8 vorgestellten FPGA- und ASIC-Prototypen. Die Entkopplung der verschiedenen Taktdomänen ist über asynchrone FIFO–Puffer realisiert.

7.4. Erweiterung der UPSLA-basierten Werkzeugkette um Hardware-Simulationsmodelle

Der Einsatz von Hardware-Erweiterungen verspricht in vielen Fällen eine Erhöhung der Ressourceneffizienz des Prozessorsystems. Zugriffe auf Hardware-Erweiterungen unterscheiden sich von normalen Zugriffen auf den Datenspeicher nur durch den angesprochenen Adressbereich. Im Instruktionssatzsimulator werden Zugriffe auf Hardware-Erweiterungen jedoch nicht simuliert und Zugriffe auf diese standardmäßig auf den Datenspeicher abgebildet. Dieses führt zu einer Inkonsistenz zwischen Instruktionssatzsimulation und RTL-Simulation. Das in Abschnitt 3.6.5.2 vorgestellte Verfahren zur simulationsbasierten Multidomänenvalidierung schlägt fehl. Daher wurde für den Instruktionssatzsimulator, in Zusammenarbeit mit der Fachgruppe „Programmiersprachen und Übersetzer“ von Professor Kastens, eine Schnittstelle zur Anbindung von Simulationsmodellen für Hardware-Erweiterungen implementiert. Diese fängt Lese- und Schreibzugriffe auf definierte Speicherbereiche des Adressraums des Datenspeichers (inklusive der Hardware-Erweiterungen) ab und simuliert die Verhaltensweise der entsprechenden Hardware-Erweiterung. Dieses erlaubt zum einen die Anwendung der simulationsbasierten Multidomänenvalidierung auch in Verbindung mit Hardware-Erweiterungen. Zum anderen können in der Instruktionssatzsimulation auch Ressourcen des Host-PCs, wie z. B. Ethernet-Schnittstellen, genutzt werden. Hierdurch kann der Instruktionssatzsimulator beispielsweise auch in realen Netzwerkumgebungen eingesetzt werden.

[7] Hardware Accelerator – Hardware-Beschleuniger (HWACC), Standard Out – Standardausgabe (STDOUT)

[8] Personalcomputer

7.5. Systemumgebung zur flexiblen Integration von Hardware-Erweiterungen

Der bisherige Ansatz zur Integration von dedizierten Hardware-Erweiterungen erfolgte eng gekoppelt direkt auf dem Prozessor-DIE. Die Flexibilität dieser Architektur ist stark eingeschränkt. Zur Entwurfszeit eines ASIC-Prototyps muss die Auswahl der Hardware-Erweiterungen feststehen. Nach der Fertigung können keine Veränderungen mehr an diesen vorgenommen werden. Zudem haben die Hardware-Erweiterungen großen Einfluss auf den Ressourcenbedarf. Der zusätzliche Flächenbedarf für den IEEE-802.11b-Hardware-Beschleuniger liegt bei 37 %, der des kombinierten ECC-Hardware-Beschleunigers bei knapp 30 % und der der AES-Hardware-Erweiterung bei 49 %. Dieses treibt die Fertigungskosten eines ASIC in die Höhe. Sinnvoll erscheint daher eine Lösung, die den Austausch von Hardware-Erweiterungen zur Laufzeit erlaubt. Dynamisch rekonfigurierbare integrierte Schaltkreise, wie FPGAs, stellen hierfür eine Grundlage dar. Das CoreVA-System kombiniert in Hinblick auf den späteren Prototyp bereits die Standardzellenimplementierung des CoreVA-Prozessors mit einem FPGA zur Anbindung des Prozessors an das Host-System und einen SDRAM-Controller.

In diesem Abschnitt wird die Architektur des CoreVA-Systems dahingehend erweitert, dass dedizierte Hardware-Erweiterungen sowohl eng gekoppelt auf dem Prozessor-DIE, als auch lose gekoppelt auf dem angebundenen FPGA integriert werden können [247]. Ein einheitliche Schnittstelle erlaubt die nahtlose Integration der Erweiterungen auf beiden Seiten ohne zusätzliche Logik (sogenannte *Glue-Logic*, dt. *Klebstofflogik*). Die unterschiedliche Latenz beim Zugriff auf die Hardware-Beschleuniger wird anhand von Beispielanwendungen untersucht.

In Abschnitt 7.4 wurde eine Software-Schnittstelle vorgestellt, die die Simulation von Hardware-Beschleunigern mithilfe des Instruktionssatzsimulators erlaubt. Diese implementierten Simulationsmodelle der Hardware-Beschleuniger erlauben ein konsistentes Hardware-Software-Co-Design [83] von Anwendungen und Hardwarekomponenten und deren Analyse bezüglich der Ressourceneffizienz bereits in frühen Entwicklungsstadien. Im Folgenden wird eine Hardware-Schnittstelle des CoreVA-Systems vorgestellt, die die einfache Integration von ebendiesen Hardware-Simulationsmodellen in die Hardware-Umgebung (sowohl des FPGA- als auch des ASIC-Prototyps) erlaubt. Bereits vor der Implementierung von Hardware-Erweiterungen können diese durch die Integration der Hardware-Simulationsmodelle in der Hardware-Umgebung evaluiert werden. Außerdem ist die Nutzung von Komponenten des Host-PCs (beispielsweise physikalische Schnittstellen, wie ETH[9] /WLAN-Schnittstellen oder Ein-/Ausgabebegeräte) vergleichsweise einfach möglich.

In der bisherigen Realisierung war der Adressraum zur Anbindung von Hard-

[9] Ethernet

Tabelle 7.11.: *Aufteilung des Adressraums des CoreVA Prozessors zur internen und externen Anbindung von Hardware-Erweiterungen*

Adressbereich	Größe	Verwendung
`0x00000000` – `0x7FFFFFFF`	2 GByte	Datenspeicher
`0x80000000` – `0x9FFFFFFF`	32 · 32 MByte	interne Hardware-Erweiterungen
`0xA0000000` – `0xBFFFFFFF`	16 · 32 MByte	Hardware-Simulationsmodelle auf dem Host-PC
`0xC0000000` – `0xFFFFFFFF`	16 · 32 MByte	externe Hardware-Erweiterungen auf dem FPGA

ware-Erweiterungen über MMIO, wie in Abbildung 7.18 dargestellt, aufgeteilt. 32 Hardware-Erweiterungen können auf diese Weise (intern) angebunden werden. Die Größe des adressierbaren Daten-Speichers beträgt hier 2 GByte, die eines Hardware-Beschleunigers 2^{26} Bit ($\widehat{=}$ 8 MByte). Zur Anbindung von externen Hardware-Erweiterungen wird der Adressraum der Hardware-Erweiterungen in zwei Bereiche aufgeteilt. Abbildung 7.23 und Tabelle 7.11 zeigen die neue Aufteilung. Über Bit 31 wird entschieden, ob auf den Datencache oder Hardware-Erweiterungen zugegriffen wird. Bit 30 entscheidet, ob eine externe oder interne Hardware-Erweiterung angesprochen wird. Weiterhin stehen 2 GByte adressierbarer Datenspeicher zur Verfügung. 32 Hardware-Erweiterungen können eng gekoppelt direkt an den Prozessorkern angebunden werden. Jeweils 16 Hardware-Beschleuniger und 16 Simulationsmodelle können über den dedizierten FPGA extern an das System angebunden werden. Der Adressraum für die Hardware-Erweiterungen reduziert sich somit jeweils auf 2^{25} Bit $\widehat{=}$ 4 MByte.

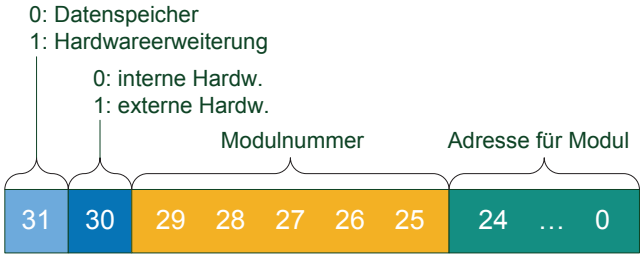


Abbildung 7.23.: Aufteilung des Adressraums zur Anbindung von internen und externen Hardware-Erweiterungen

Durch die modulare Architektur lassen sich auch andere Konfigurationen realisieren. Beispielsweise könnten in einer vierfach parallelen VLIW-Architektur Slot 0

und Slot 1 exklusiven Zugriff auf einen bis zu 4 GByte großen Adressbereich erhalten. Hardware-Erweiterungen könnten dann z. B. an Slot 2 und der Scratchpad-Speicher nur an Slot 3 angebunden werden.

7.5.1. Anbindung der externen Hardware-Erweiterungen

Um Zugriffe an externe Hardware-Erweiterungen auf dem FPGA weiterzuleiten wurde das sogenannte `HWACC_EXT`-Modul (vgl. Abbildung 7.24) entwickelt. Es ist als interne Hardware-Erweiterung an den CoreVA-Prozessor angebunden. Bei einem Zugriff werden diese allerdings vom `HWACC_EXT`-Modul an den Systembus-Arbiter weitergeleitet. Befindet sich der CoreVA-Prozessor im Mastermodus, so können neben dem `HWACC_EXT`-Modul auch der Instruktions- und Daten-Cache auf den Systembus zugreifen. Aufgabe des Arbiters ist es nun, zeitgleiche Zugriffe zu arbitrieren, indem den drei Komponenten nacheinander Zugriff auf den Systembus gewährt wird. Dieses verhindert, dass Kollisionen auf dem Systembus entstehen. Die Caches sind mit höchster Priorität an den Systembus angebunden. Die Zugriffe auf externe Hardware-Erweiterungen werden innerhalb des `HWACC_EXT`-Moduls in einem FIFO–Puffer[10] zwischengespeichert. Dieser Puffer bietet den Vorteil, dass bei *Schreibzugriffen* an die externen Hardware-Erweiterungen die CPU nicht angehalten werden muss. So können die Zugriffe nach und nach vom Systembus-Arbiter an den FPGA gesendet werden, während die CPU weiterarbeiten kann. Um den zusätzlichen Flächenaufwand durch das `HWACC_EXT`-Moduls zu minimieren, wurde mit Tiefe des FIFO-Puffers auf eine Tiefe von 16 Wörtern beschränkt. Läuft das FIFO voll, d.h. schafft es der Arbiter nicht, rechtzeitig alle Schreibzugriffe weiterzuleiten, muss auch bei Schreibzugriffen die CPU angehalten werden.

Der Zustandsautomat `fifo_out` des `HWACC_EXT`-Moduls kontrolliert das Lesen aus dem FIFO und das Weiterleiten der Daten an den Arbiter. Aufgrund von Datenkonflikten (Data Hazard, vgl. Abschnitt 4.4) kann es nötig sein, den Prozessorkern bei Zugriffen auf externe Hardware-Erweiterungen anzuhalten. Der Zustandsautomat *mmio_cpu_stall* ist für das Anhalten (Stallen) des Prozessors bei Schreibzugriffen zuständig. Der Prozessor muss angehalten werden, wenn der Schreib-Puffer voll ist (z. B. aufgrund vieler nebenläufiger Zugriffe auf die Hardware-Erweiterungen und auf die Caches), damit es nicht zu einem Überlauf kommt. Lesezugriffe auf externe Hardware-Erweiterungen haben eine Latenz von mindestens sieben Taktzyklen (vgl. Abbildung 7.25). Das FIFO innerhalb des `HWACC_EXT`-Moduls benötigt einen Taktzyklus, bis die Anfrage an den Systembus-Arbiter weitergegeben werden kann. Sowohl die ASIC- als auch FPGA-Partition sind über Ein- und Ausgangs-Register voneinander entkoppelt. Dieses dient zur Behandlung eventueller hoher Leitungsverzögerungen zwischen diesen Bausteinen. Beide Register-Stufen verursachen eine

[10] *DW_fifoctl_s1_sf* [197] der Synopsys DesignWare-Komponenten

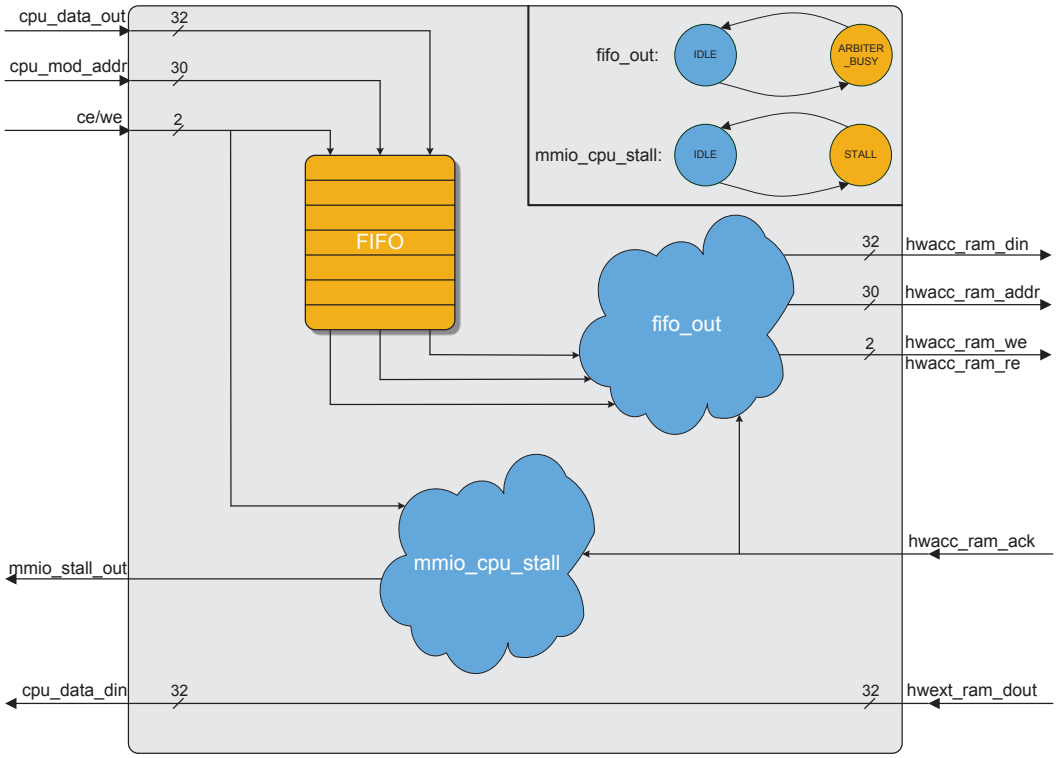


Abbildung 7.24.: Architektur des *HWACC_EXT*–Moduls zur Anbindung von externen Hardware-Erweiterungen

Latenz von insgesamt zwei Taktzyklen. Die Bearbeitung innerhalb des *HWACC–Controllers*, der die angeforderten Daten aus den Hardware-Erweiterungen ausließt, benötigt einen weiteren Taktzyklus. Für den Rückweg über die I/O-Register addieren sich zwei weitere Taktzyklen. Mit einem weiteren Taktzyklus zur Auswertung des Stall-Signals zum Anhalten des Prozessorkerns addiert sich die Latenz auf insgesamt sieben Taktzyklen. Ist der Systembus bereits durch den Zugriff einer anderen Komponente belegt, erhöht sich die Latenz für den Lesezugriff. Daher ist das Anhalten des CoreVA-Prozessors bei einem Lesezugriff, anders als bei einem Schreibzugriff, zwingend notwendig. Bei Zugriffen auf den Daten-Speicher oder interne Hardware-Erweiterungen wäre dieses nur im Falle einer direkten Datenabhängig notwendig.

Der *Systembus-Controller* wurde um den sogenannten *HWACC-Controller* erweitert. Dieser leitet Anfragen (Zugriffe auf Adressen oberhalb von 0xC0000000) vom ASIC an die (externen) Hardwaremodule weiter. Bisher wurden alle Anfragen vom Prozessor-

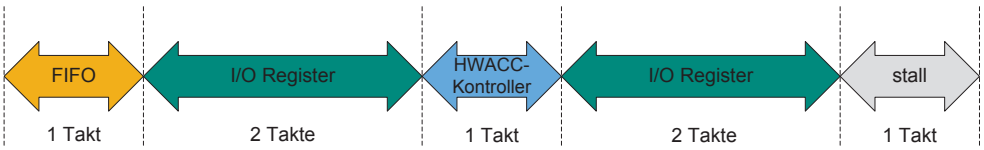


Abbildung 7.25.: Latenz bei Lesezugriffen auf externe Hardware-Erweiterungen

kern direkt an den *SDRAM-Controller* gesendet. Ein Adressdekoder innerhalb dieser Instanz entscheidet anschließend, auf welchen (externen) Hardware-Beschleuniger sich der Zugriff bezieht.

Aufgrund der Rekonfigurierbarkeit des FPGAs, können jederzeit (auch zur Laufzeit) neue Hardware-Erweiterungen an den *HWACC-Controller* angebunden werden. Abbildung 7.26 zeigt die Integration des `HWACC_EXT`-Moduls und des *HWACC-Controllers* in das CoreVA-System. Als Beispiel sind hier ein CRC-Hardware-Beschleuniger und eine Ethernet-MAC (vgl. Kapitel 8 und Abschnitt 7.3.4) über den *HWACC-Controller* als externe Hardware-Erweiterungen angebunden.

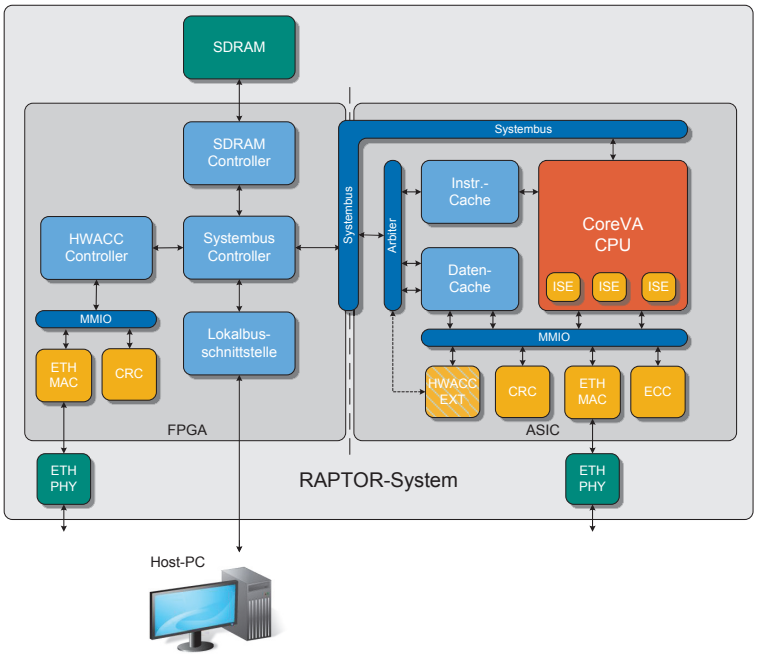


Abbildung 7.26.: Architektur des CoreVA-Systems mit Erweiterung zur Anbindung von externen Hardware-Beschleunigern

7.5.2. Anbindung von Hardware-Simulationsmodellen

Die Anbindung von *Hardware-Simulationsmodellen* der Hardware-Erweiterungen an den CoreVA-Prozessor wird über dessen Trace-Schnittstelle (vgl. Abschnitt 3.6.5.2) ermöglicht. Über diese Schnittstelle kann der Host-PCs Prozessorzustände zur Laufzeit auslesen und damit auch Speicherzugriffe detektieren.

Zur Anbindung von Hardware-Simulationsmodellen wurde diese Trace-Schnittstelle um Register erweitert, die Lese- und Schreibzugriffe zwischenspeichern. Diese Register können vom Host-PC ausgelesen werden, sodass dieser eine mögliche Anfrage an ein Hardware-Simulationsmodell bearbeiten kann. Sie beinhalten neben Informationen, ob eine Schreib- oder Leseanfrage besteht, auch die Adresse und die zu schreibenden Daten. Das Füllen dieser Register und die Kontrolle über die Zugriffe übernimmt das zusätzliches Hardware-Modul `HWACC_HOST`, welches als externe Hardware-Erweiterung an den *HWACC-Controller* angebunden ist. Die externe Anbindung bietet den Vorteil, dass Zugriffe auf die Register des `HWACC_HOST`-Moduls nicht den Systembus zur Kommunikation von Prozessorkern und Systemumgebung blockieren. Abbildung 7.27 zeigt die Architektur des `HWACC_HOST`-Moduls.

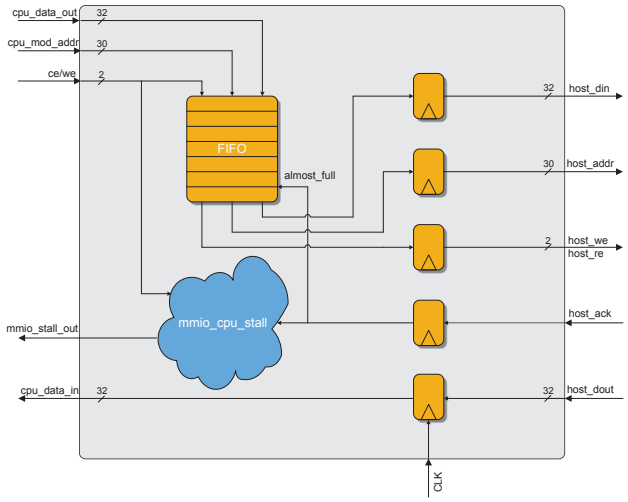


Abbildung 7.27.: Architektur des *HWACC_HOST*–Moduls zur Anbindung von Hardware-Simulationsmodellen

Abbildung 7.28 zeigt die Latenz eines Zugriffs auf ein Hardware-Simulationsmodell. Die Latenz setzt sich aus ähnlichen Komponenten wie bei einem Zugriff auf externe Hardware-Erweiterungen zusammen. Ein zusätzlicher FIFO-Puffer kann Wartezeiten abfangen, die sich durch eine eventuell zu langsame Verarbeitung der Daten durch die Hardware-Simulationsmodelle ergeben. Der größte Anteil der Latenz liegt

in der Kommunikation mit dem Host-PC über den Lokalbus des RAPTOR-Systems (vgl. Abschnitt 8). Diese Latenz ist zudem nichtdeterministisch und abhängig von der Performanz des Host-PCs. Die gesamte Latenz beträgt somit mindestens 8 Taktzyklen.

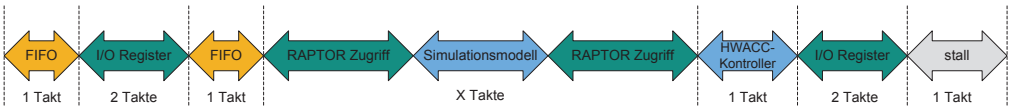


Abbildung 7.28.: Latenz eines Lesezugriffs auf ein Hardware-Simulationsmodell

Die Bearbeitung der Anfragen auf die Hardware-Simulationsmodelle erfolgt durch das Auslesen der internen Statusregister des `HWACC_HOST`-Moduls durch die in Abschnitt 8.1.2 beschriebene Anwendung zur Steuerung und Beobachtung des CoreVA-Systems (CoreVAGUI)[11] auf dem Host-PC (vgl. Abschnitt 8.1.2). Die in Abschnitt 7.4 beschriebenen Hardware-Simulationsmodelle können nahtlos an die CoreVAGUI angebunden werden. Abbildung 7.29 zeigt die Erweiterung des CoreVA-Systems um das `HWACC_HOST`-Modul zur Anbindung von Hardware-Simulationsmodellen.

7.5.3. Vergleich der Möglichkeiten der Anbindung von Hardware-Erweiterungen an den CoreVA-Prozessor

In diesem Abschnitt soll die Leistungsfähigkeit der unterschiedlichen Möglichkeiten der Anbindung von Hardware-Erweiterungen an den CoreVA-Prozessor verglichen werden. Dazu wurde der später in Abschnitt 8 vorgestellte FPGA-Prototyp verwendet.

Analyse der Performanz in Abhängigkeit des Anteils der Lesezugriffe. In diesem Abschnitt wird die Performanz in Abhängigkeit des Anteils der Lesezugriffe für die verschiedenen Anbindungsmöglichkeiten untersucht. Zu diesem Zweck wurde eine Hardware-Erweiterung implementiert, die lediglich aus 16 Registern besteht. Bei dem Hardware-Simulationsmodell werden diese 16 Register auf Integer-Variablen abgebildet. Nun wird auf dem CoreVA-Prozessor ein Programm ausgeführt, das bei jedem Schleifendurchlauf eine konstante Anzahl an Zugriffe auf die Hardware-Erweiterung durchführt. Der Anteil von Lese- und Schreibzugriffen wird hierbei variiert. Abbildung 7.30 stellt die Anzahl benötigter Taktzyklen in Abhängigkeit des Anteils an Lese- und Schreibzugriffen dar.

Die Ausführungszeit bei externer Anbindung der Hardware-Erweiterung steigt mit Anzahl der Lesezugriffe wie erwartet linear an. Grund ist hier die höhere La-

[11] Graphical User Interface (GUI)

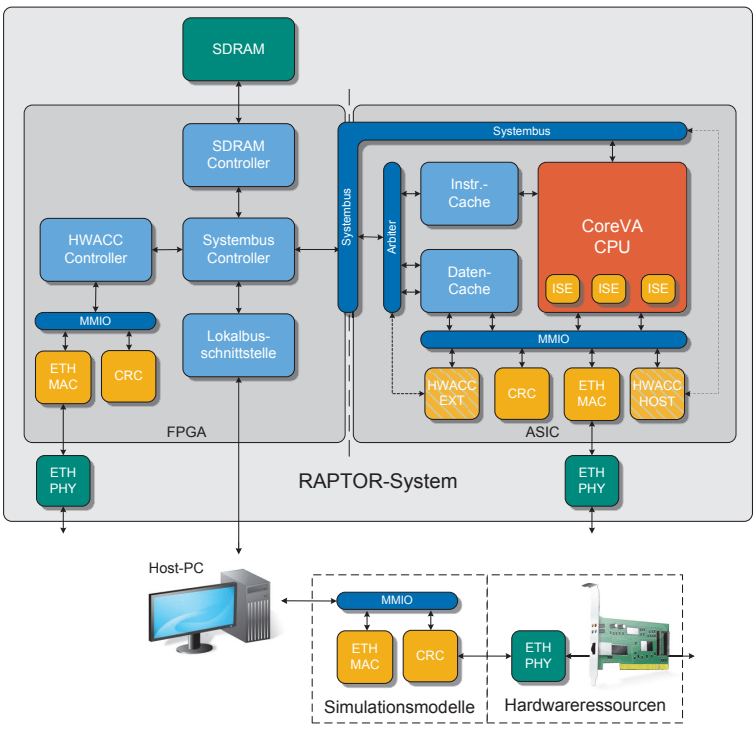


Abbildung 7.29.: Architektur des CoreVA-Systems zur Integration von Hardware-Simulationsmodellen

tenz der Kommunikation über den Systembus und das `HWACC_EXT`-Modul. Bei der internen Anbindung sinkt die Anzahl von benötigten Taktzyklen leicht. Die leichte Verringerung ist auf eine geringere Anzahl direkter Datenabhängigkeiten aufeinanderfolgender Instruktionen (Datenkonflikte) zurückzuführen. Bei der Anbindung der Hardware-Erweiterung als Hardware-Simulationsmodell ist der Einfluss der Kommunikation mit dem Host–PC zu beobachten, der eine sehr hohe Latenz verursacht. Finden keine Lesezugriffe statt, so ist die Performanz mit der internen und externen Anbindung identisch. Danach steigt die Ausführungszeit linear an. Allgemein gilt für dedizierte Hardware-Beschleuniger, dass die Performanz der Hardware-Beschleuniger mit externer Anbindung vom Unterschied der Taktfrequenzen von FPGA und ASIC abhängt. Je schneller ein externer Hardware-Beschleuniger angebunden ist, desto geringer ist der Verlust durch die zusätzliche Kommunikation über den Systembus. Die folgenden Beispiele gehen von einem synchronen System aus.

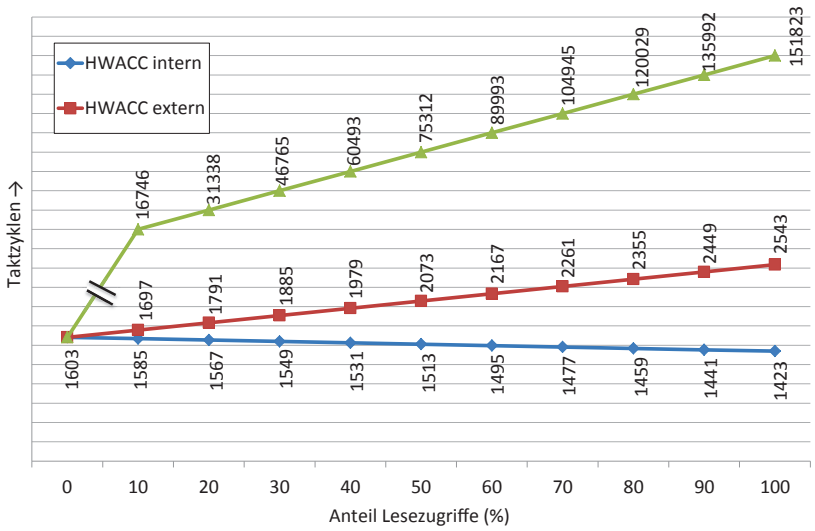


Abbildung 7.30.: Ausführungszeit beim Zugriff auf die *HWACC_REGISTER*-Hardware-Erweiterung in Abhängigkeit vom Anteil der Lese- und Schreibzugriffe

Zyklische Redundanzprüfung (CRC). Abbildung 7.31 zeigt einen Vergleich der Ausführungszeiten für den in Abschnitt 7.3.4 vorgestellten CRC-Hardware-Beschleuniger bei unterschiedlichen Arten der Anbindung an das CoreVA-System. Da der Anteil an Zugriffen auf einen Hardware-Beschleuniger Einfluss auf den durch die Anbindungsalternativen verursachten Overhead der Latenz hat, wurde die Ausführungszeit für verschiedene Paketgrößen von 4 Byte bis 64 Byte bestimmt. Der geringe Unterschied zwischen den intern und extern angebundenen CRC Hardware-Beschleunigern ergibt sich daraus, dass nicht in jedem Takt ein Zugriff auf Hardware-Beschleuniger stattfindet. Der prozentuale Unterschied nimmt mit steigender Paketgröße ab, da bei dem CRC-Algorithmus der Anteil von Lesezugriffen mit der Größe der Pakete abnimmt. Die Ausführungszeit bei Nutzung der Hardware-Simulationsmodelle ist hingegen um etwa den Faktor Hundert größer. Dieses ist auf die hohe Latenz der Kommunikation mit dem Host-PCs zurückzuführen. Zu bedenken ist jedoch, dass das Einsatzgebiet der Hardware-Simulationsmodelle in Verbindung mit dem FPGA- oder ASIC-Prototyp in erster Linie im Rahmen der Evaluation und des Prototypings zu sehen ist.

Kryptographie mit elliptischen Kurven (ECC). Abbildung 7.32 zeigt einen Vergleich der Ausführungszeiten für den in Abschnitt 7.3.6 vorgestellten ECC-Hardware-Beschleuniger bei verschiedenen Arten der Anbindung an das CoreVA-System. Die

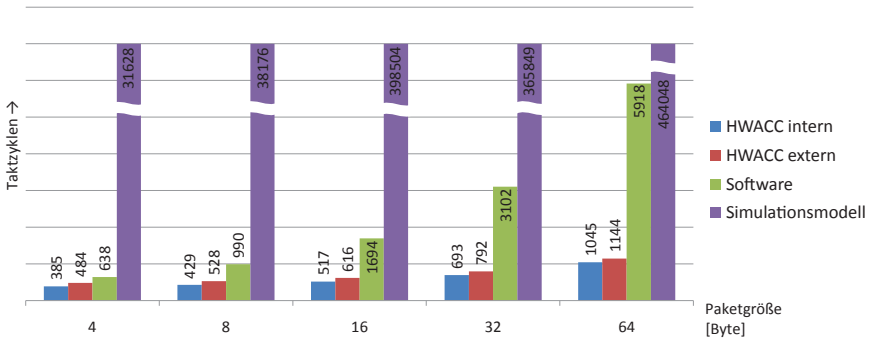


Abbildung 7.31.: Vergleich der Ausführungszeiten für den CRC-Hardware-Beschleuniger und den CRC-Algorithmus für die unterschiedlichen Möglichkeiten der Anbindung an das CoreVA-System

Ergebnisse sind ähnlich zu denen des CRC-Hardware-Beschleunigers. Die Latenz der internen und extern angebundenen Hardware-Erweiterungen steigt mit steigender Anzahl der Zugriffe auf den Hardware-Beschleuniger an. Die Ausführungszeit des Systems bei Anbindung des Hardwaresimulationsmodells ist um Größenordnungen höher als bei interner oder externer Anbindung. Vergleicht man die Ergebnisse mit denen des CRC-Hardware-Beschleunigers, fällt auf, dass die Anbindung des ECC-Hardware-Simulationsmodells die Ausführungszeit sehr viel stärker beeinträchtigt. Dieses ist auf einen höheren Anteil an Lesezugriffen zurückzuführen.

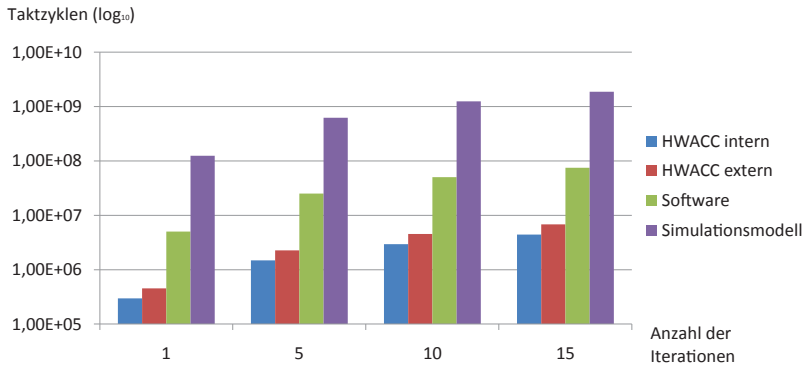


Abbildung 7.32.: Vergleich der Ausführungszeiten für den ECC-Hardware-Beschleuniger und den Algorithmus zur Kryptographie mit elliptischen Kurven für die unterschiedlichen Möglichkeiten der Anbindung an das CoreVA-System

7.6. Optimierung der Kommunikation der Systemkomponenten

Die vorigen Abschnitte haben gezeigt, dass sich durch Erweiterung des CoreVA-Systems um lokalen Scratchpad-Speicher und Hardware-Beschleuniger Performanzsteigerungen mehrerer Größenordnungen erzielen lassen. Die Analyse der Hardware-Beschleuniger zeigt jedoch weiterhin eine *Beschränkung der Performanz durch die Kommunikation mit dem externen Speicher*. Nutzdaten müssen aus dem externen Speicher in lokalen Registern abgelegt werden und durch einen weiteren Speicherzugriff in den Hardware-Beschleuniger transferiert werden. Nach der Bearbeitung der Daten werden diese aus dem Hardware-Beschleuniger ausgelesen, in Registern abgelegt und anschließend in den Hauptspeicher zurückgeschrieben (vgl. Abbildung 7.33). Der Overhead dieser Speicheroperationen übersteigt in praktischen Anwendungen,

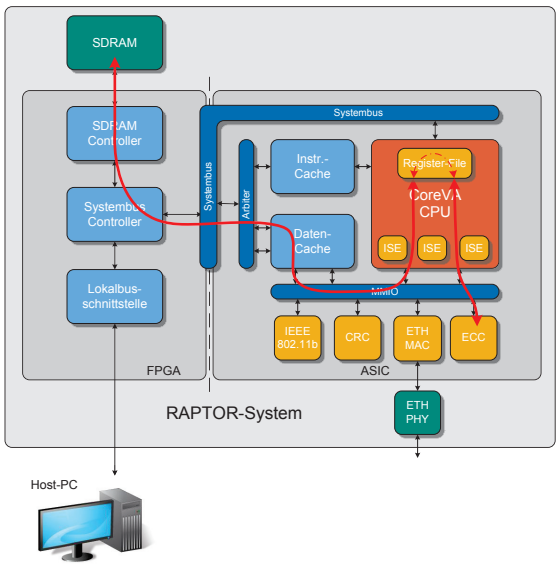


Abbildung 7.33.: Overhead durch Speichertransfers bei Nutzung von Hardware-Beschleunigern

wie beispielsweise beim IEEE-802.11b-Algorithmus, die reine Verarbeitungszeit im Hardware-Beschleuniger. Die Verlagerung der Nutzdaten in den Scratchpad-Speicher verringert zwar die Anzahl an Strafzyklen durch die Caches, löst das Problem aber nicht gänzlich. Sinnvoll erscheint hier eine Kombination des Ansatzes der Optimierung durch Hardware-Beschleuniger und Scratchpad-Speicher. Hierzu müssen Hardware-Beschleuniger jedoch um eine Schnittstelle zur Kommunikation mit einem

Scratchpad-Speicher erweitert werden (vgl. Abbildung 7.34). Die Optimierung der Kommunikation der Hardware-Beschleuniger mit dem Speicher-Subsystem wurde exemplarisch an dem in Abschnitt 7.3.4 vorgestellten CRC-Hardware-Beschleuniger durchgeführt. Über einen *DMA-Transfer* werden die Nutzdaten zuerst in den Scratchpad-Speicher geladen. Anschließend kann der Hardware-Beschleuniger selbstständig die Daten aus dem Scratchpad-Speicher laden. Um Konflikte beim zeitgleichen Zu-

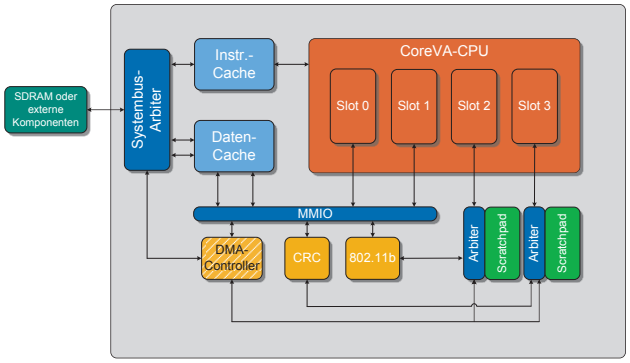


Abbildung 7.34.: Erweiterung der Hardware-Beschleuniger um Schnittstellen zur Kommunikation mit dem Scratchpad-Speicher

griff des CoreVA-Prozessors und einer Hardware-Erweiterung zu lösen, wurde die Scratchpad-Speicher-Instanz um einen Arbiter erweitert. Zugriffe des Prozessorkerns werden priorisiert. Abbildung 7.35 zeigt den Vergleich der direkten Anbindung des CRC-Hardware-Beschleunigers an den Scratchpad-Speicher zur ursprünglichen Anbindung über einfache Speichertransfers vom und zum Hauptspeicher. Es zeigt sich, dass sich erst ab einer Paketgröße von 128 Bytes ein Performanzgewinn einstellt. Ursächlich hierfür ist der zusätzlich notwendige DMA-Transfer der Nutzdaten vom Hauptspeicher in den Scratchpad-Speicher. Dieser Overhead ist aber nur einmalig bei Nutzung von Scratchpad-Speichern notwendig. Sobald die Daten im Scratchpad-Speicher vorliegen, können auch andere Anwendungen (auch in Kombination mit weiteren Hardware-Erweiterungen) vom Scratchpad-Speicher profitieren. Insbesondere eine komplexere Kombination der in diesem Kapitel vorgestellten verschiedenen Techniken, verspricht Optimierungspotential, soll aber in dieser Arbeit nicht weiter behandelt werden.

7.7. Zusammenfassung

In diesem Kapitel wurde die in Kapitel 5 beschriebene Entwurfsraumexploration der modularen VLIW-Prozessorarchitektur auf Systemebene ausgeweitet. Es wurde

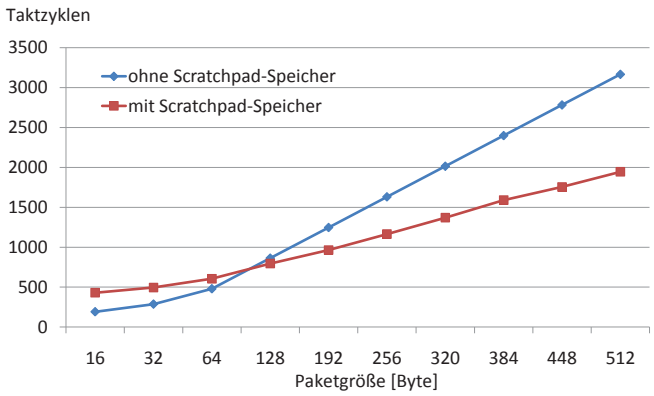


Abbildung 7.35.: Vergleich der direkten Anbindung des CRC-Hardware-Beschleunigers an den Scratchpad-Speicher zur ursprünglichen Anbindung mit zusätzlichen Speichertransfers vom und zum Hauptspeicher

der Einfluss des Instruktions- und des Daten-Caches auf die Ressourceneffizienz untersucht. Die Entwurfsraumexploration hat ergeben, dass sich für verschiedene Anwendungen die Maxima der Ressourceneffizienz bei unterschiedlichen Konfigurationen (1-Port/2-Port, F-O-W-M/A-O-W-M) ergeben. Die Variationen bei der Energieeffizienz betragen bis zu 35 %. Hier zeigt sich wieder das Potential einer umfassenden Entwurfsraumexploration des Gesamtsystems [234]. Weiterhin wurden die Möglichkeiten der Steigerung der Effizienz durch eng gekoppelten lokalen Scratchpad-Speicher aufgezeigt. Die Architekturvariante mit Scratchpad-Speichererweiterung führt zu einer Steigerung der Energieeffizienz von bis zu 34 %. Hohes Optimierungspotential bieten Hardware-Erweiterungen, wie Instruktionssatzerweiterungen oder dedizierte Hardware-Beschleuniger. Für ausgewählte Anwendungen ließ sich die Energieeffizienz um den Faktor 16 steigern. Eine generische und modulare Systemumgebung erlaubt weiterhin die flexible Anbindung von Hardware-Erweiterungen sowohl eng gekoppelt an den Prozessorkern als auch lose gekoppelt an die externe Systemumgebung. Die Anbindung von Simulationsmodellen an den Instruktionssatzsimulator ermöglicht die Validierung von Hardware-Implementierung und Compiler-Werkzeugkette auch bei Nutzung von Hardware-Erweiterungen. Zusätzlich können die Simulationsmodelle von Hardware-Beschleunigern in die Prototypenumgebung der Hardware-Architektur integriert werden. Des Weiteren wurden Schnittstellen geschaffen, über die Hardware-Erweiterungen direkt an den Scratchpad-Speicher angebunden werden können. Ein DMA-Controller realisiert hierbei den effizienten Transfer von Daten in und aus dem Scratchpad-Speicher.

8. Prototypische Implementierung des VLIW-Prozessors

Aufgrund der ständig sinkenden Strukturgrößen moderner Halbleiterprozesse und dem damit verbundenen wachsenden technologischen Aufwand steigen auch die Kosten für die Fertigung von CMOS-Schaltungen stetig an. Auch für relativ kostengünstige Multi-Wafer-Projekte (MWPs), die für den akademischen Bereich angeboten werden, liegen die Kosten in einer 65 nm Technologie bei knapp 10.000 € pro Quadratmillimeter. NRE[1] -Kosten für die Maskensätze eines kompletten Wafers liegen bereits bei mehreren Millionen Euro. Dieses erfordert eine gründliche Verifikation aller Teilkomponenten des Prozessorsystems.

Wichtig für die Qualität der *funktionalen Verifikation* ist die Testabdeckung. Diese wiederum ist abhängig von der Anzahl und Wahl der Test-Vektoren. Instruktionssatzsimulatoren erreichen zwar mit mehreren MHz eine hohe Simulationsgeschwindigkeit und würden somit auch eine hohe Testabdeckung ermöglichen, ihr Abstraktionsgrad liegt aber weit über dem des RTL-Designs. Komponenten, wie z. B. die Pipelinestruktur der Prozessorarchitektur, werden bei der Instruktionssatzsimulation nicht berücksichtigt. Feingranular rekonfigurierbare integrierte Schaltkreise, wie FPGAs, bieten die Möglichkeit, bereits in frühen Entwicklungsstadien das Gesamtsystem, aber auch Teilkomponenten, funktional zu verifizieren. Die Emulationsgeschwindigkeit von FPGA-basierten Prozessor-Designs liegt üblicherweise im Bereich von 10–100 MHz und damit um Größenordnungen höher als bei der RTL-Simulation. Auch der Detaillierungsgrad der Emulation liegt höher als beim ISS und der RTL-Simulation. In Verbindung mit der Simulation spezieller Komponenten (z. B. *clock gating*) auf Gatterebene lässt sich so eine ausreichende Verifikation des Gesamtsystems durchführen.

Der letzte Prozessschritt vor einer möglichen Serienfertigung ist die Entwicklung eines ASIC-Prototyps. Erst dieser Prototyp erlaubt genaue funktionale Tests und die Analyse der Leistungsaufnahme unter realen Bedingungen bei der erwarteten Zielfrequenz.

Dieses Kapitel beschreibt die *vollständige prototypische Umsetzung* der CoreVA-Architektur. Zur funktionalen Verifikation wurde ein *FPGA-basierter Prototyp* entwickelt (vgl. Abschnitt 8.1). Als Plattform dient hierfür das im Fachgebiet Schaltungstech-

[1] Non-recurring Engineering

nik entwickelte, modulare *Rapid-Prototyping-System RAPTOR*. Für die Einbettung von Speicherblöcken wurden Schnittstellen definiert, die das Verhalten der Speicher vereinheitlichen (z. B. Read-After-Write etc.). ASIC-Speichermakros und BlockRAM wurden hierfür in Wrapper eingebunden, die je nach Implementierung zusätzliche Eingangs- oder Ausgangsregister enthalten. Dieses ermöglicht eine weitgehend einheitliche RTL-Implementierung von FPGA- und ASIC-Prototypen. Da die Zieltechnologie eine CMOS-Standardzellentechnologie ist, wurden die BlockRAM-Blöcke funktional an die der ASIC-Speichermakros angepasst, und nicht umgekehrt. Dieses vermeidet zusätzliche Logik in der finalen Implementierung. Der modulare Aufbau des RAPTOR-Systems ermöglich es zudem, einen FPGA-Prototyp leicht um weitere Hardwarekomponenten, wie physikalische Schnittstellen, zu erweitern. Durch physikalische Schnittstellen lassen sich systematische Testdaten-Profile um zufällige Datenströme existierender Netzwerke erweitern. Des Weiteren erlauben sie die Demonstration und den Test in realen Systemumgebungen. Auch dieses kann die Fehlerabdeckung erhöhen. Eine frühe Bereitstellung einer Prototypenumgebung ermöglicht so eine effiziente parallele Entwicklung von Software und Hardware.

Der Systementwurf des FPGA-Prototyp dient als Grundlage für den anschließend entwickelten ASIC-Prototyp in einer CMOS-Standardzellentechnologie (vgl. Abschnitt 8.2). Abschnitt 8.3 beschreibt schließlich einen Demonstrator mit dem die Leistungsfähigkeit sowohl des FPGA-Prototyps als auch der ASIC-Realisierung gezeigt werden kann.

8.1. FPGA-Prototyp – Das CoreVA-Prototyping-System

Für die funktionale Verifikation der CoreVA-Architektur wurde das System auf das im Fachgebiet Schaltungstechnik entwickelte Rapid-Prototyping-System RAPTOR abgebildet. Diese Entwicklungsumgebung verfolgt einen modularen Ansatz bestehend aus einem Basisboard und bis zu vier (RAPTOR XPress) oder sechs (RAPTOR-2000 bzw. RAPTOR-X64) Erweiterungsmodulen. Das Basisboard ist über den PCI[2] (-Express[3])-Bus an einen Host-Computer angebunden. Zur Erweiterung stehen 20 Tochtermodule zur Verfügung. FPGA-Module (Virtex-II bis Virtex-5) können zur Emulation von RTL-Designs verwendet werden. Physikalische Schnittstellen ((Gigabit-)Ethernet, USB, Bussysteme etc.) dienen zur Kommunikation mit der Systemumgebung. ASIC-Verifikationsmodule können zur Verifikation von ASIC-Prototypen genutzt werden. Eine umfangreiche Entwicklungsumgebung bestehend aus fertigen Software-Werkzeugen und Programmierschnittstellen ermöglicht die Implementierung von grafischer Steuer- und Kontroll-Software.

[2] Peripheral Component Interconnect

[3] nur RAPTOR XPress

Zur Emulation des CoreVA-Systems wurde das DB[4] -V2-Modul verwendet. Dieses beinhaltet einen Xilinx Virtex-II FPGA. Für speicherintensive Anwendungen sind auf dem DB-V2-Modul 64MBit SRAM Speicher und ein SO-DIMM[5] SDRAM für bis zu 4 GB SDRAM vorhanden. Zur externen Kommunikation wurde das DB-Ethernet-Modul verwendet. Vier Fast-Ethernet-Schnittstellen bilden die physikalische Schicht des ISO/OSI-Modells von 10/100Mbit/s IEEE 802.3 Ethernet ab. Über vier Media Independent Interfaces (MIIs) können MAC[6] -Komponenten über das DB-Ethernet-Modul Daten verschicken und empfangen. Das auf dem FPGA implementierte System umfasst das CoreVA-Prozessor-System (vgl. Abbildung 8.1) mit einem VLIW-Kern mit vier ALUs, zwei Multiplizierern und zwei Dividierern. Die Slots 0 und 1 können über den in Abschnitt 7.1 vorgestellten Dual-Port-Cache auf den externen SDRAM-Speicher zugreifen. Die Slots 2 und 3 sind an jeweils 2 kByte Scratchpad-Speicher (vgl. Abschnitt 7.2) angebunden. Über die in Abschnitt 7.3 vorgestellte MMIO-Schnittstelle können Slot 0 und 1 auf vier Hardware-Erweiterungen (ECC, CRC, UART, FIFO) zugreifen. Der in Abschnitt 7.3 beschriebene DMA-Controller ermöglicht die schnelle Kommunikation von Hardware-Erweiterungen und Scratchpad-Speicher.

8.1.1. Systempartitionierung und Syntheseergebnisse

Zur Abbildung des RTL-Designs auf das FPGA wurden die Synthesewerkzeuge der Firmen Synopsys (vormals Synplicity) und Xilinx genutzt. Synopsys Synplify ermöglicht die Abbildung der in der RTL-Beschreibung genutzten Synopsys DesignWare IP Komponenten (z. B. Addierer, Multiplizierer etc.) auf optimierte FPGA-Komponenten. Zur Verifikation des Systems wurden *zwei verschiedene Partitionierungen* des Designs untersucht. Abbildung 8.1 zeigt das FPGA-Design unter Nutzung *eines FPGA-Moduls*. Sowohl Prozessor-Kern, als auch die Systemumgebung sind synchron auf denselben FPGA abgebildet. Für eine detailliertere Verifikation ist es aber notwendig auch die externe Verdrahtung zwischen ASIC Prototyp und externen Hardware-Komponenten der Systemumgebung (auf einem weiteren dedizierten FPGA) zu überprüfen. Diese Verdrahtungsleitungen, welche über die sogenannten Rechts-Links-Verbinder der FPGA-Module und Teile der RAPTOR-Hauptplatine verlaufen, verursachen unter Umständen eine im Verhältnis zur Taktfrequenz des Prozessor-Kerns hohe Leitungsverzögerung. Diese läßt sich nur schwer über interne Komponenten eines FPGA modellieren. Aus diesem Grund wurde das RTL-Design auf *zwei FPGA-Bausteine* partitioniert (vgl. Abbildung 8.2). Der Prozessorkern und die Systemumgebung wurden, wie bei der späteren ASIC-Prototypenumgebung, auf je ein separates FPGA-Modul

[4] Daughter Board

[5] Small Outline Dual Inline Memory Module

[6] Medium Access Control

abgebildet. Über *Digital Clock Manager (DCMs)* kann die Abtastflanke des Taktsignals der Eingangs- und Ausgangsregister an die Leitungsverzögerungen der externen Verdrahtung angepasst werden.

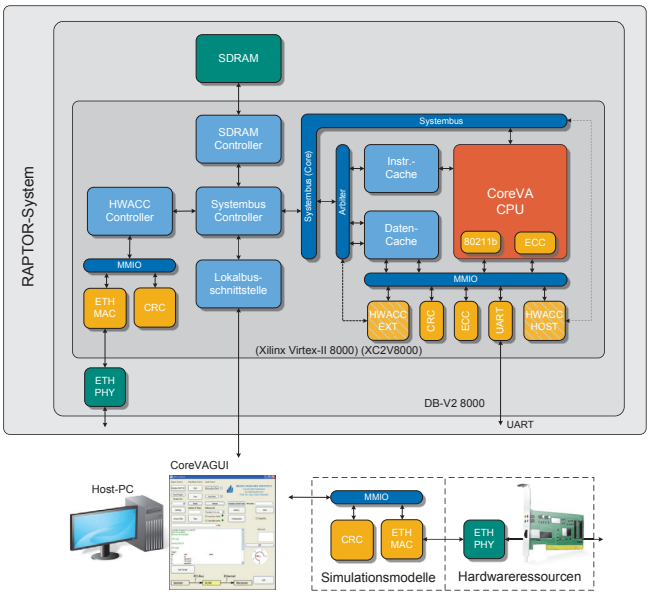


Abbildung 8.1.: Partitionierung des 1-FPGA-Designs

Das 1-FPGA-Design verwendet 79 % der 46.592 verfügbaren Slices eines Xilinx-Virtex-II 8000 (XC2V8000) FPGAs. Von den genutzten Slices sind insgesamt 21 % der Register, 71 % der *Lookup Tables (LUTs)* und 35 % der Blockrams (BRAMs) belegt. Tabelle 8.1 zeigt die Verteilung der belegten Ressourcen auf die Hauptkomponenten des CoreVA-Systems. Bei Partitionierung des Designs auf zwei FPGAs werden ein Xilinx-Virtex-II 6000 FPGAs (XC2V6000) und ein Xilinx-Virtex-II 8000 FPGAs (XC2V8000) benötigt (vgl. Abbildung 8.2). Die maximal erreichbare Taktfrequenz der FPGA-Implementierung beträgt 16,6 MHz. Verwendet man die in Kapitel 6 vorgestellten Mechanismen zur *dynamischen Rekonfiguration der Forwarding-Funktionalität* des Prozessorkerns erhöht sich die maximal mögliche Betriebsfrequenz des FPGA-Prototyps auf bis zu 25 MHz.

Neben den beiden FPGA-Modulen kommt ein DB-Ethernet-Modul zur externen Kommunikation zum Einsatz. Zwei als externe Hardware-Erweiterungen genutzte Ethernet-MACs [67, 71] binden je eine Ethernet- PHY[7] des DB-Ethernet-Moduls

[7] Physikalische Schnittstelle

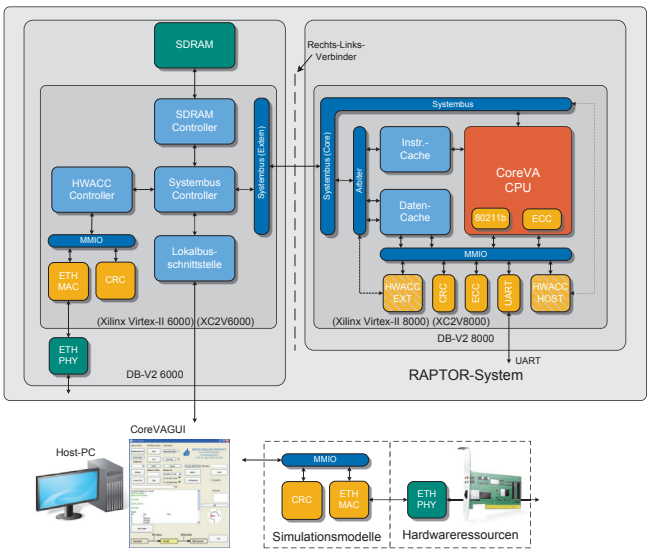


Abbildung 8.2.: Partitionierung des 2-FPGA-Designs

an. Zur Verifikation der in Abschnitt 5.2 vorgestellten Algorithmen wurde die in Abbildung 8.3 gezeigte *drahtgebundene Übertragungsstrecke* aufgebaut. Ein PC generiert Testdaten und schickt diese über eine Ethernet-Verbindung an das DB-Ethernet. Im *ersten Testszenario* empfängt der CoreVA-Prozessor diese Daten und schickt sie unverändert über die zweite Ethernet-Verbindung an einen weiteren PC. Hier können die empfangenen Daten verifiziert werden. In einem *erweiterten Testszenario* wurden

[8] bezogen auf das CoreVA-System

[9] bezogen auf die gesamten Ressourcen des Virtex-2 8000 (XC2V8000) FPGAs

Tabelle 8.1.: *Verteilung der belegten Logikressourcen eines Xilinx-Virtex-II 8000 (XC2V8000) FPGAs für das 1-FPGA-Design*

Komponente	Register	%	LUTs	%	BRAMs	%
CPU	6529	72,0 [8]	57570	87,0	0	86,4 [8]
Caches	504	5,6 [8]	5991	9,1	51	0,0 [8]
Scratchpad	66	7,3 [8]	120	0,4	8	13,6 [8]
Systemumgebung	1918	21,2 [8]	2221	3,4	0	0,0 [8]
CoreVA System	9065	9,7 [9]	66180	71,0	59	35,0 [9]
FPGA Gesamt	93184	100,0	93184	100,0	168	100,0

die Algorithmen aus Abschnitt 5.2 auf die gesendeten Daten angewendet. Somit konnten die Algorithmen auch mit sehr großen Datenmengen verifiziert werden.

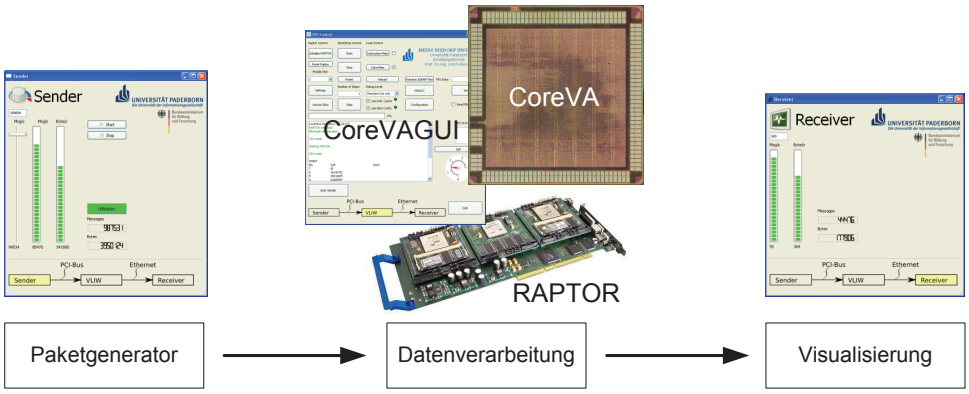


Abbildung 8.3.: Ethernet-Übertragungsstrecke von einem Paketgenerator über den CoreVA-Prozessor zu einem Paketempfänger

Um den FPGA-Prototyp über die reinen Funktionstest hinaus auch in drahtlosen Umgebungen einsetzen zu können, wurde im Rahmen dieser Arbeit das *DB-SDR-Erweiterungsmodul* für die Realisierung von *SDR-Anwendungen* entwickelt (vgl. Abschnitt 8.3.1).

8.1.2. Anwendung zur Steuerung und Beobachtung des CoreVA-Systems (CoreVAGUI)

Um die *effiziente Steuerung und Beobachtung* des CoreVA-Prototyps schon während der Hardware-Entwicklung zu ermöglichen, wurde, basierend auf der QT[10] -Bibliothek [26] und der RAPTOR-Entwicklungsumgebung, die grafische Anwendung (GUI – Graphical User Interface) CoreVAGUI entwickelt (vgl. Abbildung 8.4).

Über Lese- und Schreibzugriffe auf die Lokalbusschnittstelle des RAPTOR-Systems kann auf den Systembus des Prozessorsystems zugegriffen werden. So wird zum einen die Steuerung des CoreVA-Prozessors (Initialisieren der verschiedenen Speicher, Starten/Stoppen/Zurücksetzen des Prozessors, Konfiguration der Caches etc.), zum anderen das Beobachten der verschiedenen Komponenten (Auslesen der Speicher, Auslesen des Prozessorzustandes etc.) ermöglicht.

Interaktionen mit den ausgeführten Programmen können über zwei verschiedene Mechanismen erfolgen: Zum einen über die *FIFO-Hardware-Erweiterung*, zum anderen über die *UART-Schnittstelle* (vgl. Abschnitt 7.3.7).

[10] Quasar Technologies

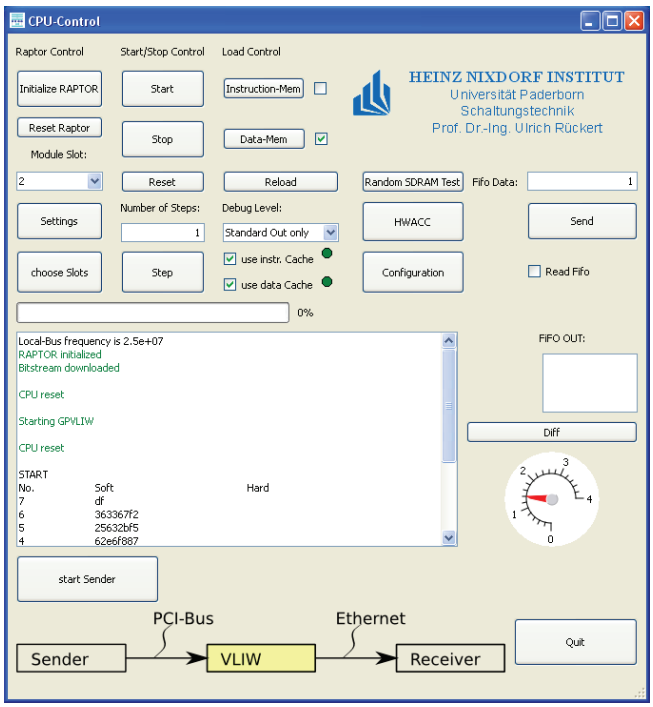
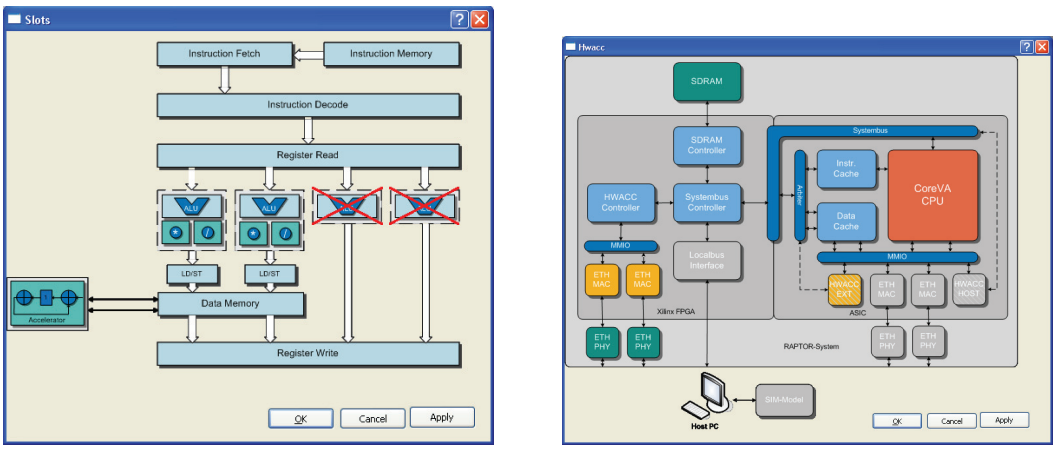


Abbildung 8.4.: CoreVAGUI – Grafische Anwendung zur Steuerung des FPGA-Prototyps

Zur Verifikation des Prototypensystems ist in der CoreVAGUI die in Abschnitt 3.6.5.2 beschriebene Funktionalität zur *Multi-Domänen-Validierung* implementiert. In jedem Taktzyklus kann der Prozessorzustand des Systems ausgelesen werden. Die daraus resultierenden Prozessor-Traces können automatisch mit denen des Instruktionssatzsimulators oder der RTL-Simulation verglichen werden.

Zwei Erweiterungen der CoreVAGUI erlauben die *Visualisierung* verschiedener Daten. Um den Einfluss der VLIW-Parallelität zu visualisieren, kann über die erste Erweiterung (vgl. Abbildung 8.5a) der CoreVAGUI die maximale mögliche Anzahl *parallel genutzter VLIW-Slots* beschränkt werden. Die zweite Erweiterung (vgl. Abbildung 8.5b) erlaubt die Konfiguration der verschiedenen Möglichkeiten der *Hardware-Erweiterungen*. Simulationsmodelle für Hardware-Erweiterungen (vgl. Abschnitt 7.4) können dynamisch zur Laufzeit an die CoreVAGUI angebunden werden. Zur Verifikation der Ethernet-Schnittstellen wurden zusätzlich zwei Anwendungen zur Paketgenerierung und zur Verifikation der empfangenen Daten entwickelt (vgl. Abbildung 8.6). Außerdem kann der Durchsatz der empfangenen Daten qualitativ und quantitativ abgelesen werden.



(a) Konfiguration der funktionalen Parallelität

(b) Konfiguration der Anbindung von Hardware-Erweiterungen

Abbildung 8.5.: Erweiterung der CoreVAGUI zur Konfiguration des CoreVA-Systems

8.2. Implementierung eines ASICs in einer 65 nm Low-Power-Standardzellentechnologie

Die Genauigkeit moderner Layout-Werkzeuge bei der Abschätzung von physikalischen Eigenschaften einer Schaltung, wie beispielsweise die Extraktion parasitärer Kapazitäten, die zur Bestimmung der maximalen Taktfrequenz oder der Leistungsaufnahme herangezogen werden, ist hoch. Um jedoch wirkliche verlässliche Angaben bezüglich des Ressourcenbedarfs von anwendungsspezifischen Schaltungen zu erhalten, sind Messungen an realer Hardware notwendig. Zu jedem Entwurf von digitalen Systemen gehört daher die Fertigung von einem oder mehreren *(Test-)Chips*, deren Analyse zur Abschätzung der Qualität eines späteren Produktes genutzt wird. Im Rahmen dieser Arbeit wurde ein ASIC-Prototyp des CoreVA-Systems in einer 65 nm Low-Power-Standardzellentechnologie von STMicroelectronics gefertigt. In modernen 65 nm Technologien beginnt die Verlustleistung durch Leckströme die Gesamtverlustleistung zu dominieren. Die *Low-Power-Technologie* von STMicroelectronics zeichnet sich im Gegensatz zu deren sogenannter *General-Purpose-Technologie* durch eine vergleichsweise *dicke Gate-Oxid-Schicht* und somit eine *geringe statische Verlustleistung* aus. Da die CoreVA-Architektur insbesondere für den mobilen Einsatz konzipiert ist, wurde diese Technologie gewählt. Der Sollwert der Kernspannung der Standardzellen beträgt 1,2 V, die I/O-Spannung der verwendeten I/O-Zellen beträgt 1,8 V. Für die Realisierung des Layouts wurde konsequent der in Abschnitt 3.6 vorgestellte automatisierte Hardware-Entwurfsablauf genutzt. Für die Platzierung und Verdrahtung wurde an erster Stelle die maximale Taktfrequenz als Optimierungsziel gewählt. An

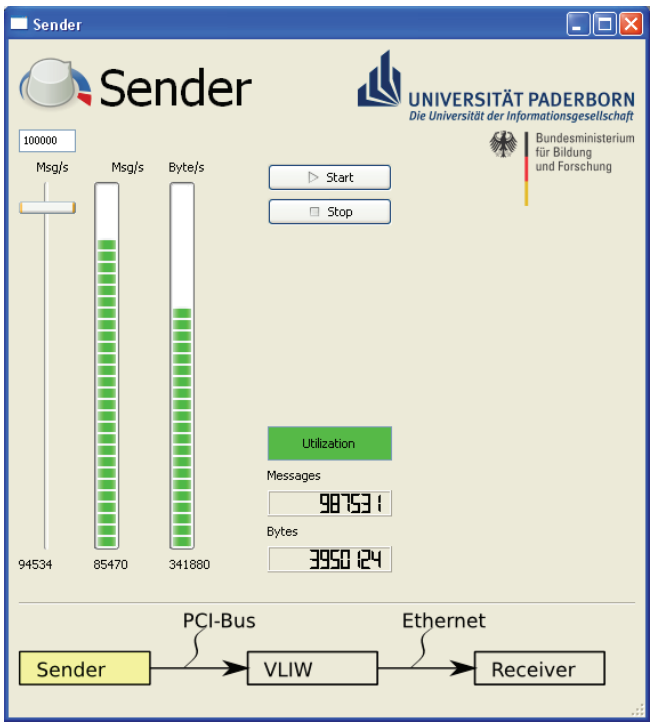


Abbildung 8.6.: Paketgenerator und Anwendung zur Verifikation der Übertragungsstrecke

zweiter Stelle wurde die Verlustleistung optimiert. Zur Reduzierung der Verlustleistung wurde Stromspartechniken, wie *Clock-Gating* und *Operand-Isolation*, genutzt. Aus Kostengründen sollte die DIE-Fläche minimiert werden. Daher wurden I/O-Zellen mit minimaler Breite gewählt und die Zellen direkt nebeneinander platziert. Somit ist der Flächenbedarf des ASICs durch die Anzahl der I/O-Ports nach unten beschränkt. Um den Datendurchsatz zum externen Speicher nicht unnötig einzuschränken, wurde eine Datenbreite von 32 Bit für den Systembus gewählt. Die Anzahl der notwendigen I/O-Versorgungspads ist durch die Treiberstärke der I/O-Zellen vorgegeben. Anhand der Ergebnisse von „Probe-Layouts“ wurde die zu erwartende Leistungsaufnahme für die Kern-Logik abgeschätzt und anhand dieser Werte die Anzahl notwendiger Versorgungspads, die die Kernspannung bereitstellen, bestimmt. Streuungen bzgl. der Umgebungstemperatur oder der externen Versorgungsspannung haben einen großen Einfluss auf die Anstiegs- und Abfall-Charakteristik und somit auf die maximale Betriebsfrequenz der I/O-Zellen. Zur Kompensation solcher Schwankungen wird eine I/O-Kompensationszelle genutzt, die als Hardmakro eingesetzt wird. Über zusätzliche I/O-Zellen kann beispielsweise ein externer Widerstand angeschlossen

werden, der als Referenz dient und die Kompensation externer Einflüsse ermöglicht. Insgesamt ergibt sich eine Anzahl von 140 I/O-Pads (vgl. Tabelle 8.2). Bei einem minimalen *Pitch*[11] von 40 μm und einer Zellenhöhe von 112 μm ergibt sich somit eine minimale Kantenlänge des ASICs von

$$L_{\text{Kante}} = 140/4 \cdot 40\ \mu m + 2 \cdot 112\ \mu m = 1624\ \mu m \tag{8.1}$$

und eine DIE-Fläche von

$$A_{\text{DIE}} = (1624\ \mu m)^2 = 2,64\ mm^2. \tag{8.2}$$

Tabelle 8.2.: *Eingangs- und Ausgangs-Pads des ASIC-Prototyps*

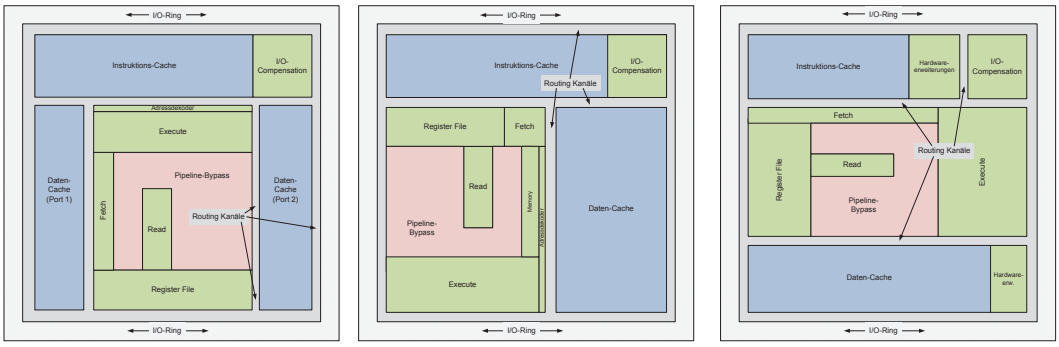
ASIC-Pad	**Richtung**	**Anzahl**
Takt	IN	1
Reset	IN	1
Systembus		
Adresse	IN/OUT	30
Daten	IN/OUT	32
Steuerung	IN/OUT	14
UART	IN/OUT	2
Spannungsversorgung		
Kernspannung	IN	24
I/O-Spannung	IN	28
I/O-Kompensation[12]	N/A	1
Reserviert	N/A	7
Summe		140

Erste Layouts unter Berücksichtigung dieser Flächenbeschränkung haben ergeben, dass sie die in Kapitel 5 ermittelte Konfiguration für das CoreVA-System (4 ALUs, 2 Multiplizierer, 2 Divisionsschritteinheiten, jeweils 16 kByte Instruktions- und Daten-Cache) gut auf dieser Fläche unterbringen lässt. Die Daten- und Tag-Speicher von Instruktions- und Daten-Cache wurden als Hardmakro-Blöcke realisiert und direkt von STMicroelectronics generiert. Als Architektur der Speicherblöcke wurden Hochgeschwindigkeits-Implementierungen gewählt. Durch den in Kapitel 3

[11] Abstand zwischen den Mittelpunkten der I/O-Zellen

[12] Die I/O-Kompensationszelle dient zur dynamischen Anpassung der I/O-Pads an Schwankungen der Umgebungstemperatur, der Versorgungsspannung etc.

vorgestellten automatisierten Entwurfsablauf war es möglich, eine sehr große Anzahl an Platzierungs- und Verdrahtungsdurchläufen durchzuführen. Im Rahmen der Entwurfsraumexploration wurden *viele verschiedene Floorplans* evaluiert. Es zeigte sich, wie erwartet, dass die Wahl des Floorplans einen großen Einfluss auf die physikalischen Eigenschaften (z. B. die maximale Taktfrequenz) der Architektur hat. Die Abbildungen 8.7a–8.7c zeigen Beispiele von verschiedenen, während der Entwurfsraumexploration evaluierten, Floorplans. Es zeigte sich, dass eine zu stringente Platzierungsvorgabe für zu viele Komponenten die Ergebnisse eher verschlechtert. Daher wurde lediglich die Platzierung von Kernkomponenten vorgegeben. Floorplan 8.7c ergab die besten Ergebnisse. Abbildung 8.8 zeigt eine Kombination von Layout und ein Chip-Foto des finalen ASIC-Prototyps. Der Prozessor besteht aus ca. 4,2 Millionen Transistoren (Prozessorkern: 1,5 Millionen Transistoren, 32 kB Speicher: 2,7 Millionen Transistoren). Dieses entspricht in etwa der Komplexität eines Intel Pentium Prozessors MMX (P55C) [62, 39, 228].



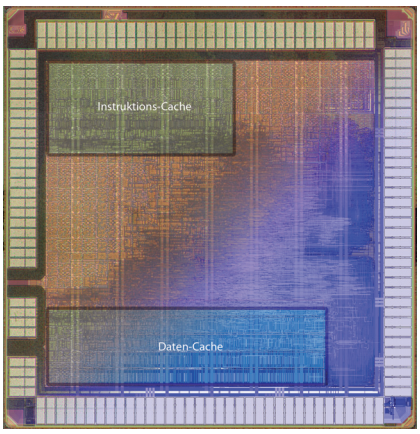
(a) Floorplan 1 **(b)** Floorplan 2 **(c)** Floorplan 3

Abbildung 8.7.: Drei verschiedene Floorplan-Beispiele

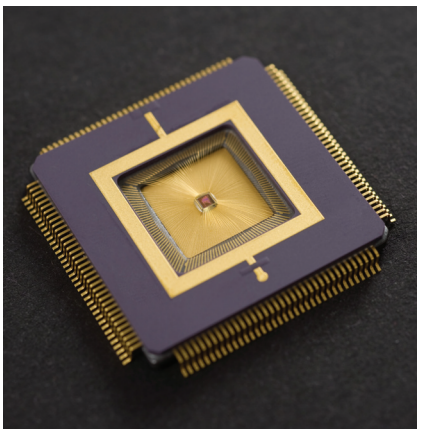
8.2.1. Leistungsdaten des ASIC-Prototyps

Zum *Test und zur Bestimmung der Leistungsdaten* wurde ein *ASIC-Evaluationsmodul* für das RAPTOR-Rapid-Prototyping-System entwickelt. Hierdurch ist es möglich, sämtliche für den FPGA-Prototyp entwickelten Steuerungs- und Analyse-Werkzeuge beizubehalten. Lediglich das FPGA-Erweiterungsmodul muss durch das neu entwickelte ASIC-Evaluationsmodul (DB-CoreVA) ersetzt werden. Abbildung 8.9 zeigt das DB-CoreVA. Der ASIC-Prototyp wird über einen CQFP[13] -144-Sockel auf die Platine aufgesetzt. Ein Spartan-6-FPGA dient zur externen Kommunikation und beinhaltet das in Abschnitt 8.1 beschriebene System-Design. Zur Bestimmung der

[13] Ceramic Quad Flatpack



(a) Layout (unten rechts) und Chip-Foto (oben links)



(b) Chip-Gehäuse mit freigelegtem Prozessor-DIE

Abbildung 8.8.: Das CoreVA-System in einer 65 nm Standardzellentechnologie von STMicroelectronics

maximalen Taktfrequenz und der Leistungsaufnahme wurden die in Abschnitt 5.2 beschriebenen Anwendungen auf dem CoreVA-Prozessor ausgeführt. Dabei wurde die in Abschnitt 3.6.5.2 beschriebene Multi-Domänen-Validierung angewandt und Prozessortraces des ASIC-Prototyps mit denen des Instruktionssatzsimulators verglichen. Tabelle 8.3 zeigt die maximal erreichbare Taktfrequenz und die dabei auftretende Leistungsaufnahme sowie die Leistungsaufnahme bei der Zielfrequenz von 400 MHz (1,2 V, 25 °C). Die maximale Leistungsaufnahme beträgt für den CRC-Algorithmus 211 mW. Die minimale Leistungsaufnahme trat bei Ausführung des Faltungsalgorithmus auf (84 mW). Der Grund liegt hier in erster Linie an dem in Abschnitt 7.1.5 diskutierten hohen Anteil an Strafzyklen durch Fehlzugriffe der Caches. Hier wird der Prozessor solange angehalten, bis die fehlende Cache-Zeile aus dem externen Speicher nachgeladen wird. Der Durchschnitt der Leistungsaufnahme liegt für die untersuchten Anwendungen bei 169 mW.

8.3. Der CoreVA Demonstrator

Der Demonstrator des CoreVA-Prototyps wurde auf der *IEEE International Conference on Communication (ICC 2009)* in Dresden auf dem Messestand der Universität Paderborn einem Fachpublikum vorgestellt (vgl. Abbildung 8.10). Von einem PC wird ein hochauflösendes Video in HD[14] -Qualität über die in Abschnitt 8.1.1 be-

[14] High Definition

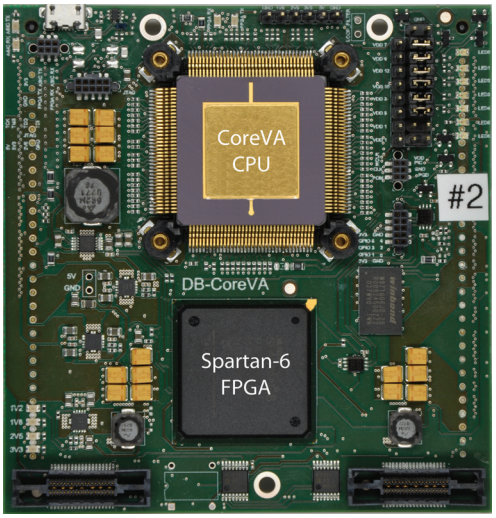


Abbildung 8.9.: Das DB-CoreVA ASIC-Evaluationsmodul

Anwendung	Leistungsaufnahme
Dhrystone	178 mW
Coremark	176 mW
Viterbi	175 mW
Faltung	84 mW
FFT	111 mW
IEEE 802.11b	194 mW
CRC	211 mW
ECC	197 mW
AES	152 mW
SNOW	196 mW
SATD	144 mW
Fibonacci	207 mW
Mittelwert	169 mW

Tabelle 8.3.: Gemessenen Leistungsdaten des CoreVA-Prozessors

schriebene Ethernet-Übertragungsstrecke geschickt. Auf dem auf einem zweiten PC emulierten VLIW-Prozessor werden die transkodierten Daten weiterverarbeitet sowie eine Paketverarbeitung inklusive Prüfsummenberechnung durchgeführt. Über die Rekonfiguration der verschiedenen Komponenten der Architektur (VLIW-Parallelität, Anbindung der Hardware-Erweiterungen etc.) kann Einfluss auf den maximalen Datendurchsatz der Übertragungsstrecke genommen werden. Dieser spiegelt sich in einer Änderung der Qualität des auf einem dritten PC empfangenen Video-Stroms wieder. Der Demonstrator zeigt, dass durch Einsatz aller Funktionseinheiten des CoreVA-Prozessors (vgl. Abschnitt 5.3.1) das HD-Video in *Echtzeit* übertragen werden kann. Wird nur ein VLIW-Slot genutzt, so treten Bildfehler auf. In einem zweiten Anwendungsbeispiel wird die Steigerung der Performanz durch Hardware-Erweiterungen demonstriert. Ohne Hardware-Beschleuniger ist keine fehlerfreie Videoübertragung möglich. Werden die intern angebundenen Hardware-Beschleuniger aktiviert, so ist die Übertragung möglich. Verlagert man die Ausführung performanzkritischer Teile des Algorithmus von den internen auf die externen Hardware-Beschleuniger so sind nur leichte Qualitätseinbußen bei der Darstellung des HD-Videos zu beobachten (vgl. Abschnitt 7.5). Der Einsatz von Hardware-Simulationsmodulen mit dem FPGA-Prototyp erlaubt zwar keinen hohen Datendurchsatz, ermöglicht aber durchaus die funktionale Verifikation des Demonstratoraufbaus mit Hilfe der Video-Übertragung in reduzierter Auflösung (vgl. Abschnitt 7.4).

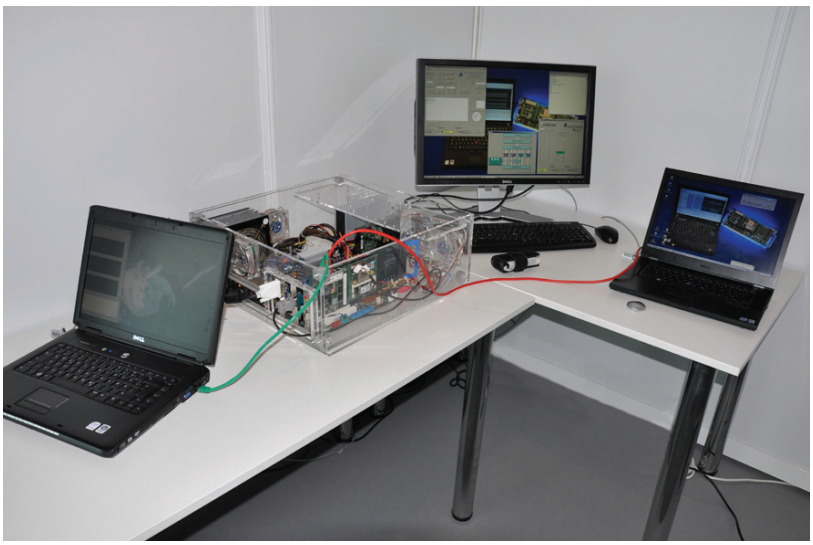
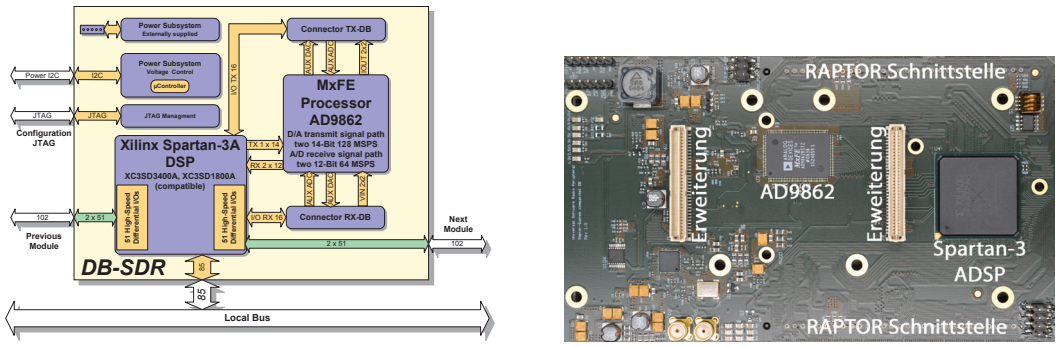


Abbildung 8.10.: Der CoreVA Demonstrator auf der ICC 2009

8.3.1. Das DB-SDR Erweiterungsmodul für drahtlose Kommunikation

Mit dem DB-SDR [242, 238, 241, 234] (vgl. Abbildung 8.11a und 8.11b) kann das RAPTOR-System auch als Prototypensystem für drahtlose Kommunikation genutzt werden. Kernkomponente des DB-SDR ist der 12-/14-Bit *Mixed Signal Front-End (MXFE)*-Prozessor AD9862 von Analog Devices. Im Empfangspfad verfügt dieser über zwei 12-Bit-A/D-Wandler mit einer Abtastrate von 64 MSamples/s, wodurch ein bis zu 32 MHz breites Frequenzband digitalisiert werden kann. Der Sendepfad verläuft über zwei 14-Bit-D/A-Wandler, die digitale Eingangsdaten mit einer Rate von 128 MSamples/s wandeln. Ein Spartan-3 ADSP FPGA von Xilinx ist für eine *Datenvorverarbeitung* und für die Anbindung des DB-SDR an das Hostsystem mit auf der Platine integriert. Über zwei Erweiterungsschnittstellen können analoge RF-Front-Ends auf das DB-SDR aufgesteckt werden. Das DB-SDR ist kompatibel zu der von Ettus Research [66, 84] entwickelten modularen SDR-Entwicklungsumgebung *Universal Software Radio Peripheral (USRP)*. Daher können die von Ettus Research vertriebenen Erweiterungsmodule für das USRP mit dem DB-SDR genutzt werden. Hier bietet Ettus eine große Auswahl an Tochterplatinen, die Frequenzbereiche von 0–5 GHz abdecken. Durch die Modularität bietet das DB-SDR eine hohe Flexibilität und ermöglicht auch die funktionale Verifikation von SDR-basierten Anwendungen. Abbildung 8.12 zeigt den Systemaufbau einer SDR-Verifikationsumgebung basierend auf dem RAPTOR-System, dem DB-Ethernet, dem DB-V2, dem DB-SDR und dem RF-Front-End XCVR2450 von Ettus Research.



(a) Blockschaltbild **(b)** Foto

Abbildung 8.11.: Das DB-SDR Erweiterungsmodul dient zur funktionalen Verifikation von SDR-basierten Anwendungen

8.4. Zusammenfassung

Dieses Kapitel hat die prototypische Umsetzung der CoreVA-Architektur beschrieben. Implementiert wurden sowohl ein FPGA-Prototyp zur funktionalen Verifikation als auch eine ASIC-Realisierung. Als Basisplattform wurde das modulare Rapid-Prototyping-System RAPTOR genutzt. Um die effiziente Steuerung und Beobachtung des CoreVA-Prototyps schon während der Hardware-Entwicklung zu ermöglichen, wurde die grafische Anwendung CoreVAGUI entwickelt. Sowohl FPGA-Prototyp als auch ASIC-Realisierung haben die Funktionsfähigkeit und Leistungsfähigkeit der CoreVA-Architektur bestätigt. Messungen des ASIC-Prototyps ergaben eine durchschnittliche Leistungsaufnahme von 167 mW. Des Weiteren wurde mit dem DB-SDR ein Modul für das RAPTOR-System entwickelt, welches die funktionale Verifikation von Software-Defined-Radio-Anwendungen ermöglicht.

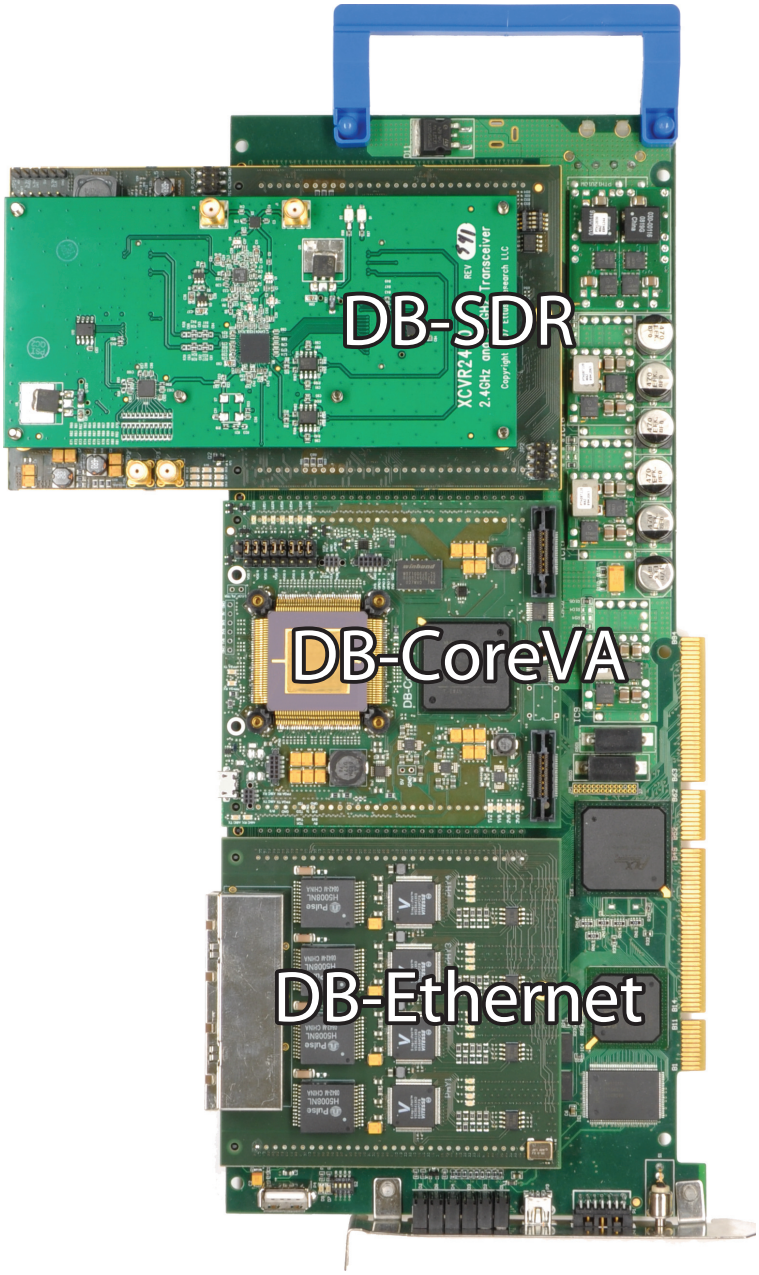


Abbildung 8.12.: Systemaufbau einer SDR-Verifikationsumgebung basierend auf der Rapid-Prototyping-Umgebung RAPTOR, dem DB-SDR mit RF-Front-End XCVR2450 von Ettus Research (Oben), dem DB-CoreVA (Mitte) und dem DB-Ethernet (Unten)

9. Zusammenfassung und Ausblick

Ressourceneffiziente und gleichzeitig *leistungsfähige* Prozessorarchitekturen spielen insbesondere in mobilen Anwendungsszenarien eine immer wichtigere Rolle. Um die benötigte Performanz bereitstellen zu können, beschäftigen sich aktuelle Forschungsarbeiten zunehmend mit Multiprozessorarchitekturen. Parallele Ausführungseinheiten ermöglichen die Erhöhung des Datendurchsatzes bei vergleichsweise moderaten Taktfrequenzen und niedriger Leistungsaufnahme. Diese Arbeit hat die Klasse der sogenannten *Very-Long-Instruction-Word (VLIW)*-Prozessoren behandelt, die sich durch eine einfache und damit effiziente Hardware-Struktur auszeichnen. Im Vergleich zu superskalaren Architekturen wird die Verteilung der Instruktionen auf die Funktionseinheiten bereits im Vorfeld vom Compiler durchgeführt. Auf zusätzliche Hardware-Komponenten zur Ablaufplanung kann verzichtet werden.

Entwurfsraumexploration. Der in dieser Arbeit durchgeführten *Entwurfsraumexploration* liegt ein dualer Entwurfsablauf zugrunde, der auf einer zentralen Prozessorspezifikation basiert. Die Dualität des Entwurfsablaufs erlaubt die parallele Entwicklung von Soft- und Hardware und eine frühe Auswertung des Ressourcenbedarfs des Gesamtsystems. Ergänzend zu der bereits möglichen Generierung einer Compiler-Werkzeugkette aus der Prozessorspezifikation wurde auf Seiten des Hardware-Entwurfs Werkzeuge implementiert, die die *automatisierte Abbildung* der Hardware-Beschreibung sowie die Abschätzung des Ressourcenbedarfs ermöglichen. Das Resultat ist ein ganzheitlicher Ansatz zur Entwurfsraumexploration von Prozessorarchitekturen. Der hohe Automatismus ermöglicht eine Verkürzung der Iterationszyklen während des Entwurfs und somit eine *Verkleinerung der Entwurfsproduktivitätslücke* bei der Betrachtung sehr großer Entwurfsräume mit vielseitig konfigurierbaren Architekturen und individuellen Anwendungsszenarien. Um die Konsistenz zwischen Hardware-Architektur und Compiler-Werkzeugkette sicherzustellen, wurden verschiedene Verfahren zur Verifikation und Validierung realisiert.

Bewertungsmaße. Die *Bewertung der Ressourceneffizienz* erfolgt durch ein eigens für diesen Zweck definiertes *Bewertungsmaß*, welches die Ressourcen einer Prozessorarchitektur miteinander verknüpft. Über geeignete Wahl der Parameter kann dieses Maß auch auf bekannte Bewertungsmaße zurückgeführt werden und ermöglicht so die Vergleichbarkeit zu anderen Arbeiten.

Modulare VLIW-Prozessorarchitektur. Als Grundlage für die in dieser Arbeit durchgeführte Entwurfsraumexploration wurde eine *modulare VLIW-Prozessorarchitektur* entwickelt. Über die Variation generischer Parameter ist die *Konfiguration* der Anzahl der Funktionseinheiten, wie ALUs, Multiplizierer und Dividierer sowie die Anzahl der Dekodierer und LD/ST-Einheiten *zur Entwurfszeit* möglich. Die Konfiguration dieser Komponenten bedingt durch die Anzahl der VLIW-Slots direkt die Datenparallelität und hat somit Einfluss auf weitere Kernkomponenten, wie die Pipelineregister, das Register-File, den Pipeline-Bypass oder das Instruktions-Alignment. Neben hochparallelen VLIW-Architekturen ist so auch die Realisierung einer skalaren RISC-Architektur mit nur einem Datenpfad möglich. Die Variabilität der Architektur ermöglichte die Exploration eines *großen Entwurfsraumes* bezüglich der funktionalen Parallelität.

Entwurfsraumexploration auf Prozessorebene. Die in dieser Arbeit entwickelte, modulare VLIW-Architektur diente als Grundlage für eine umfassende Entwurfsraumexploration, bei der der Ressourcenbedarf verschiedener Prozessorkonfigurationen in Bezug zur intrinsischen Parallelität von Algorithmen aus verschiedenen Anwendungsbereichen gesetzt wurde. Als Anwendungen wurden Algorithmen aus den Bereichen *Benchmarks*, *Codierung*, *Fehlererkennung*, *Signalverarbeitung*, *Kryptographie* und *Bildverarbeitung* betrachtet. Das Ergebnis der Analyse ist ein anwendungsspezifischer Vergleich der Ressourceneffizienz verschiedener Hardware-Konfigurationen. Die Analyse der Ausführungszeiten zeigte differenzierte Ergebnisse: Insbesondere Algorithmen aus dem Bereich der Basisbandverarbeitung und Multimediaanwendungen ermöglichen einen hohen Grad an Parallelität. Anwendungen mit hohem Kontrollanteil und die in den synthetischen Benchmarks verwendeten Programmkerne verfügen über einen geringen Anteil an intrinsischer Parallelität. Die Analyse des Ressourcenbedarfs ergab eine lineare Skalierung mit der Anzahl der Funktionseinheiten. Aufgrund von weitgehend disjunkten Datenpfaden, die VLIW-Prozessoren prinzipbedingt aufweisen, ist die maximale Taktfrequenz weitgehend unabhängig von der Parallelität. Der kritische Pfad wird im Wesentlichen durch die Ressourcenteilung gemeinsam genutzter Funktionseinheiten und den Pipeline-Bypass beeinflusst. Die Bewertung der Architekturvarianten ergab, dass eine vierfach parallele VLIW-Konfiguration mit vier ALUs, zwei Multiplizierern, zwei Dividierern und zwei Speicherkanälen zum Datenspeicher die höchste Ressourceneffizienz aufweist und somit einen geeigneten Kompromiss zwischen Performanz und Ressourcenbedarf darstellt. Im Vergleich zu einer skalaren RISC-Konfiguration mit nur einer Verarbeitungseinheit ermöglicht die feingranulare Parallelität der VLIW-Architektur einen Performanzgewinn von bis zu einem Faktor von mehr als Drei. Die Energieeffizienz verbessert sich um einen Faktor von bis zu Zwei. Diese Konfiguration wurde als Referenzarchitektur für die nachfolgenden Untersuchungen auf Systemebene definiert und als *CoreVA-*

Prozessor bezeichnet. Die Leistungsfähigkeit der CoreVA-Architektur wurde in Bezug zu kommerziellen Prozessorarchitekturen gesetzt. Bei fast allen betrachteten Anwendungen ist die in dieser Arbeit entwickelte Architektur einem kommerziellen ARM9E-Prozessor, trotz dessen hochoptimierten Compilers, überlegen. Je nach Anwendung übertrifft der *vierfach* parallele CoreVA auch die Leistungsfähigkeit des untersuchten TI TMS320C642 Prozessors mit *acht* Ausführungskanälen oder liefert zumindest eine vergleichbare Performanz.

Optimierung des Pipeline-Bypasses. Durch die Entwurfsraumexploration auf *Prozessorkern-Ebene* konnte gezeigt werden, dass, insbesondere mit steigender Anzahl von VLIW-Slots, ein vollständiger Pipeline-Bypass den Ressourcenbedarf der Architektur dominiert. Bei der vierfach parallelen CoreVA-Architektur entfallen bereits 20 % der Chip-Fläche auf den Pipeline-Bypass. Der Anteil des Pipeline-Bypasses an der Register-zu-Register-Verzögerungszeit des kritischen Pfades beträgt bis zu 28 %. Die Analyse der Bypass-Auslastung bei Ausführung verschiedener Anwendungen hat gezeigt, dass ein Großteil der implementierten Bypass-Komponenten nur selten genutzt wird. Der vollständige Verzicht auf einen Pipeline-Bypass ist jedoch aufgrund der Vielzahl hinzukommender Strafzyklen in den wenigstens Fällen zielführend. Daher wurde ein Algorithmus entwickelt, der durch das systematische Abschalten einzelner Bypass-Komponenten optimierte Bypass-Konfigurationen ermittelt. Die abschließenden Auswertungen zeigten, dass durch das Anwenden dieses Algorithmus 80 % bis 95 % aller Bypass-Pfade deaktiviert werden können. Durch die damit verbundene *Reduktion des kritischen Pfades*, sinkt die Verarbeitungszeit und die aufzuwendende Energie um bis zu 21,5 %. Der Prozessorkern wurde anschließend um eine zusätzliche Instruktion erweitert, die die *dynamische Rekonfiguration* des Pipeline-Bypasses ermöglicht. Die dynamische Rekonfiguration des Pipeline-Bypasses ermöglicht eine Anpassung der Betriebsfrequenz zur Laufzeit. Anwendungen mit geringer Bypass-Auslastung können so mit einer deutlich höheren Taktfrequenz ausgeführt werden.

Entwurfsraumexploration auf Systemebene. Um auch die Ausführung von Anwendungen mit hohen Speicheranforderungen zu ermöglichen, wurde der CoreVA-Prozessor um ein *Speicher-Subsystem* mit Level-1-Caches ergänzt. Die *Entwurfsraumexploration auf Systemebene* berücksichtigt daher eine umfassende Parametervariation der Cache-Konfiguration. Neben dem Prozessorkern wirken sich auch diese Komponenten auf den Ressourcenbedarf aus. Die Modularität der implementierten VLIW-Architektur erlaubt zur Entwurfszeit die Konfiguration der möglichen Schreib-/Lese-Ports, über die auf den Datenspeicher zugegriffen werden kann. Der Daten-Cache kann zudem bezüglich des verwendeten Allokationsmodus (Fetch-on-write-miss-/Allocate-on-write-miss) konfiguriert werden. Dieses ermöglicht die Adaption des

Systems an die jeweilige Anwendung zur Laufzeit.

Die Entwurfsraumexploration zeigte, dass die Auswahl der Cache-Konfiguration stark vom jeweiligen Anwendungsszenario abhängt. Je nach Anwendung stellt entweder die 1-Port- oder die 2-Port-Konfiguration die effizientere Architektur dar. Auch die Effizienz der Allokationsmodi divergiert mit den unterschiedlichen Algorithmen. Es zeigte sich, dass keine allgemeingültige Aussage über eine optimale Cache-Konfiguration gemacht werden kann. Aufgrund des geplanten Anwendungsszenarios wurde für den CoreVA-Prozessor die 2-Port-Konfiguration des Daten-Caches gewählt. Im Vergleich zu einer 1-Port-Konfiguration ergibt sich hierbei eine um bis zu 35 % höhere Energieeffizienz.

Weiterhin wurden die Möglichkeiten der Steigerung der Effizienz durch eng gekoppelte lokale *Scratchpad-Speicher* aufgezeigt. Die Architekturvariante mit Scratchpad-Speichererweiterung führt zu einer Steigerung der Energieeffizienz von bis zu 34 %.

Weiteres Optimierungspotential bieten *Hardware-Erweiterungen*, wie Instruktionssatzerweiterungen oder dedizierte Hardware-Beschleuniger. Für ausgewählte Anwendungen ließ sich die Energieeffizienz um den Faktor 16 steigern. Eine generische und modulare Systemumgebung erlaubt weiterhin die flexible Anbindung von Hardware-Erweiterungen sowohl eng gekoppelt an den Prozessorkern als auch lose gekoppelt an die externe Systemumgebung. Die Anbindung von Simulationsmodellen an den Instruktionssatzsimulator ermöglicht die Validierung von Hardware-Implementierung und Compiler-Werkzeugkette auch bei Nutzung von Hardware-Erweiterungen. Zusätzlich können die Simulationsmodelle von Hardware-Beschleunigern in die Prototypenumgebung der Hardware-Architektur integriert werden. Des Weiteren wurden Schnittstellen geschaffen, über die Hardware-Erweiterungen direkt an den Scratchpad-Speicher angebunden werden können. Ein DMA-Controller realisiert hierbei den effizienten Transfer von Daten in und aus dem Scratchpad-Speicher.

Prototypische Implementierung. Zur *funktionalen Verifikation* und Demonstration der Realisierbarkeit und Leistungsfähigkeit der CoreVA-Architektur wurden im Rahmen dieser Arbeit zwei Prototypen realisiert. Der *FPGA-Prototyp* basiert auf der Rapid-Prototyping-Umgebung RAPTOR. Die verwendeten rekonfigurierbaren Bausteine erlauben die prototypische Umsetzung der Architektur bereits in einem frühen Entwurfsstadium. Der CoreVA-Prozessor belegt auf einem Xilinx-Virtex-II 8000 (XC2V8000) FPGA ca. 79 % der verfügbaren Slices. Die Modularität des RAPTOR-Systems erlaubt die Erweiterung der Architektur um physikalische Schnittstellen, wie beispielsweise Ethernet oder Wireless LAN. Der *ASIC-Prototyp* des CoreVA-Prozessors wurde in einer 65 nm Standardzellentechnologie von STMicroelectronics realisiert. Bei einer maximale Taktfrequenz von 400 MHz beträgt die Leistungsaufnahme durchschnittlich etwa 169 mW. Inklusive 32 kByte Level-1-Cache belegt der CoreVA eine Chipfläche von 2,7 mm^2. Zur Verifikation und zur Messung der

physikalischen Eigenschaften des ASIC-Prototyps wurde die Erweiterungsplatine DB-CoreVA eingesetzt. Um die effiziente Steuerung und Beobachtung der CoreVA-Prototypen schon während der Hardware-Entwicklung zu ermöglichen, wurde die grafische Anwendung CoreVAGUI entwickelt. Sowohl FPGA-Prototyp als auch ASIC-Realisierung haben die Funktionsfähigkeit und Leistungsfähigkeit der CoreVA-Architektur bestätigt. Messungen des ASIC-Prototyps ergaben eine durchschnittliche Leistungsaufnahme von 169 mW bei einer Taktfrequenz von 400 MHz. Des Weiteren wurde mit dem DB-SDR ein Modul für das RAPTOR-System entwickelt, welches die funktionale Verifikation von Software-Defined-Radio-Anwendungen ermöglicht.

Verwertbarkeit der Ergebnisse. Die in dieser Arbeit entwickelte Entwurfsmethodik und das CoreVA-System ist *Grundlage sowohl für bestehende als auch für zukünftige Projekte* in der Fachgruppe Schaltungstechnik. Der CoreVA-Prozessor diente als Basisarchitektur für die Entwurfsraumexploration von Multiband-Multistandard-Algorithmen im BMBF-Projekt *MxMobile*. Im BMBF-Projekt *Easy-C* wurde der CoreVA-Prozessor zur Exploration von Basisbandalgorithmen aus dem Bereich des zukünftigen Mobilfunkstandard LTE Advanced eingesetzt. Im *Sonderforschungsbereich (SFB) 614 „Selbstoptimierende Systeme des Maschinenbaus“* wird die CoreVA-Architektur als Grundlage für fehlertolerante Systeme genutzt. In diesem Projekt wird untersucht, inwieweit redundante Funktionseinheiten die Fehlertoleranz erhöhen kann. Auf Basis der CoreVA-Architektur wurde im Rahmen der *Programme des Projektbezogenen Personenaustauschs (PPP)* des *Deutscher Akademischer Austausch Dienst (DAAD)* „Robust Ultra-Low-Power Circuits for Nano-Scale CMOS-Technologies“ ein ASIC-Prototyp in einer 65 nm Ultra-Low-Power-Standardzellenbibliothek realisiert, der unterhalb der üblichen Schwellspannung von MOS-Transistoren betrieben wird [129, 128]. Bei einer Versorgungsspannung von 0,35 V erreicht die Energieeffizienz dieser Schaltungsrealisierung ihr Maximum und übersteigt die einer konventionellen Implementierungen um den Faktor 2,58. Stromspartechniken, wie das Abschalten der Versorgungsspannung für komplette Funktionsblöcke, versprechen eine weitere Reduzierung der Leistungsaufnahme.

Eine mögliche Erweiterung des CoreVA-Systems betrifft die Erhöhung der *Robustheit* gegenüber Fehlern bei der Herstellung, durch Alterung oder durch äußere Einflüsse während des Betriebs. Eine homogene VLIW-Architektur erlaubt die Integration von Redundanz, beispielsweise durch zusätzliche Funktionseinheiten. Wird eine defekte Einheit erkannt, so kann diese durch eine redundante Einheit ersetzt werden. Eine Alternative ist es, defekte Funktionsblöcke abzuschalten. Bleibt mindestens eine Einheit übrig, so ist die grundlegende Funktionalität – wenn auch ggf. mit reduzierter Performanz – weiterhin gewährleistet.

Reicht die bereitgestellte Performanz einer VLIW-Architektur nicht aus, so bieten *On-Chip-Netzwerke (NoCs)* die Möglichkeit einer weiterführenden Skalierung der Leis-

tungsfähigkeit des Systems. Das GigaNetIC-System [153, 165][239, 240] basiert auf einem hierarchischen NoC (GigaNoC), das speziell für skalierbare Multiprozessorarchitekturen entwickelt wurde. In [234] und [248] wurde der CoreVA-Prozessor bereits in das GigaNoC integriert.

Abkürzungsverzeichnis

3DES	Triple Data Encryption Standard
3GPP	3rd Generation Partnership Project
A-O-WM	Allocate-on-write-miss
ADD	Addierer, Addition
AES	Advanced Encryption Standard
ALU	Arithmetic Logical Unit
ARQ	Automatic Repeat Request
ASIC	Application Specific Integrated Circuit
ASM	Assembler
ASR	Arithmetic Shift Right
AVC	Advanced Video Coding
BMBF	Bundesministerium für Bildung und Forschung
BPSK	Binary Phase Shift Keying
BRAM	Blockram
CAM	Content Addressable Memory
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CORBA	Common Object Resource Broker
CoreVA	Configurable Ressource Efficient VLIW Architecture
CPF	Common Power Format
CPI	Cycles per Instruction
CPU	Central Processing Unit
CQFP	Ceramic Quad Flatpack
CRC	Cyclic Redundancy Check
CSA	Carry-Save-Adder
DAAD	Deutscher Akademischer Austausch Dienst
DB	Daughter Board
DC	Instruction-Decode
DCM	Digital Clock Manager

DES	Data Encryption Standard
DFT	Diskrete Fourier-Transformation
DIV	Division
DMA	Direct Memory Access
DMIPS	Dhrystone Million Instructions Per Second
DQPSK	Differential Quaternary Phase Shift Keying
DRAM	Dynamic Random Access Memory
DSLR	Digitale Spiegelreflexkamera
DSP	Digitaler Signalprozessor
DSSS	Direct Sequence Spread Spectrum
DVI	Division Initialization
DVQ	Division Quotient
DVR	Division Remainder
DVS	Division Step
Easy-C	Enablers for Ambient Services and Systems – Part C
ECC	Elliptic Curve Cryptography
ECL	Emitter coupled logic – Emittergekoppelte Logik: Meist auf Bipolartransistoren basierende Schaltungstechnik
EDA	Electronic Design Automation
EDP	Energy-Delay-Produkt
EEMBC	Embedded Microprocessor Benchmark Consortium
ELI	Enormously Longword Instructions
EPIC	Explicitly Parallel Instruction Computing
ETH	Ethernet
EX	Execute
EXT	Extension
F-O-W-M	Fetch-on-write-miss
FE	Instruction-Fetch
FF	Flipflop
FFT	Fast Fourier Transformation
FIFO	First-In-First-Out
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
FR-V	Fujitsu RISC-VLIW
GBT	Get Best Timing
GFLOPS	Milliarden Fließkommaoperationen pro Sekunde
GOPS	Milliarden Operationen pro Sekunde

GSM	Global System for Mobile Communication
GUI	Graphical User Interface
HD	High Definition
HDL	Hardware Description Language
HDTV	High Definition Television
HP	Hewlett-Packard
HWACC	Hardware Accelerator – Hardware-Beschleuniger
I/O	Input/Output
IEC	Internationale Organisation für Normung, International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IF	Intermediate Frequency
ILP	Instruction Level Parallelism
IP	Intellectual Property
IPS	Instruktionen pro Sekunde
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ISM	Industrial Scientific Medial
ISO	International Organization for Standardization
ISS	Instruktionssatzsimulator
ITRS	International Roadmap for Semiconductors
ITU	International Telecommunication Union
LB	Load Balancing
LD	Load
LFSR	Linear Feedback Shift Register
LTE	Long Term Evolution
LUT	Lookup Table
LZI	Linear zeitinvariant
MAC	Medium Access Control
MCR	Move-Condition-to-Register
ME	Memory
MII	Media Independent Interface
MLA	Multiply-Accumulate
MMIO	Memory-Mapped-I/O
MMU	Memory Management Unit
MPEG	Moving Picture Experts Group
MPSOC	Multiprocessor-System-On-Chip

MPU	Micro Processor Unit – dt. Mikroprozessoreinheit
MRC	Move-Register-to-Condition
MUL	Multiplizierer
MULTP	Partieller Multiplizierer
MVC	Move Constant
MWP	Multi-Wafer-Projekt
MXFE	Mixed Signal Front-End
NoC	Network-on-Chip
NOP	No-Operation
NRE	Non-recurring Engineering
OPS	Operationen pro Sekunde
OSI	Open Systems Interconnection
PAR	Place and Route
PC	Personalcomputer
PCI	Peripheral Component Interconnect
PDP	Power-Delay-Produkt
PEP	Power-Energy-Produkt
PHY	Physikalische Schnittstelle
PLCP	Physical Layer Convergence Protocol
PLE	Physical Layout Estimation
PPP	Programme des Projektbezogenen Personenaustauschs
QPSK	Quadrature Phase Shift Keying
QT	Quasar Technologies
RAM	Random Access Memory
RD	Register-Read
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RSA	Rivest, Shamir, Adleman
RT	Register Transfer
RTL	Register Transfer Level
SATD	Sum of Absolute Transformed Differences
SCT	System and Circuit Technology – Schaltungstechnik
SDMA	Space division multiple access
SFB	Sonderforschungsbereich

SO-DIMM	Small Outline Dual Inline Memory Module
SoC	System-on-Chip
SRAM	Static Random Access Memory
ST	STMicroelectronics
ST	Store
st	Stop-Bit
STDOUT	Standard Out – Standardausgabe
TI	Texas Instruments
TSMC	Taiwan Semiconductor Manufacturing Company
UART	Universal Asynchronous Receiver Transmitter
UEA2	UMTS Encryption Algorithm 2
UIA2	UMTS Integrity Algorithm 2
UMTS	Universal Mobile Telecommunications System
UPSLA	Unified Processor Specification Language
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VE	Verarbeitungseinheit
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very-Long-Instruction-Word
WLAN	Wireless Local Area Network
WNS	Worst Negative Slack
WR	Register-Write
XOR	Exklusiv-Oder

Symbolverzeichnis

f_S	Abtastfrequenz
$t_{period,constr,neu}$	Angepasste Vorgabe für die Periodendauer
N_{ALU}	Anzahl der arithmetisch-logischen Einheiten
N_{cyc}	Anzahl der benötigten Taktzyklen eines Algorithmus
N_{DIV}	Anzahl der Divisionsschritteinheiten
$N_{Register\text{-}Lese\text{-}Ports}$	Anzahl der Lese-Ports des Register-Files
N_{MLA}	Anzahl der Multiplikationsakkumulatoren
$N_{Speicherschnittstelle}$	Anzahl der Ports zum Zugriff auf Speicher
$N_{Registerbypässe}$	Anzahl der Registerbypässe
$N_{Register\text{-}Schreib\text{-}Ports}$	Anzahl der Schreib-Ports des Register-Files
$N_{\text{VLIW-Slots}}$	Anzahl der VLIW-Slots
$\mathbb{A}$	Architektur
B_{IF}	Bandbreite der Zwischenfrequenz
GND	Bezeichnung für die Masse in integrierten Schaltungen
VDD	Bezeichnung für die Versorgungsspannung in integrierten Schaltungen
$\mathbb{B}$	Bildraum
$\mathbb{F}_{2^m}$	Binärer endlicher Körper
$t_{cto,reg,source}$	Clock-to-Output-Verzögerungszeit des Quellregisters
I	Coremark-Iterationen
$x[n]$	Digitalisiertes zeitdiskretes Eingangssignal
x_n	Diskretisierte Zeitfunktion
P_{dyn}	Dynamische Verlustleistung
EC	Elliptische Kurve
E	Energie
$\mathbb{E}$	Entwurfsraum

f_k	Finites Fourierspektrum
A	Fläche
A_{Zelle}	Fläche einer Standardzelle
f	Frequenz
$N_{Gesamtwortbreite}$	Gesamtwortbreite des Registerbypasses
p_{krit}	Kritischer Pfad
C_{Last}	Lastkapazität eines CMOS-Gatters
P_{last}	Lastumladeverlustleistung
P_{leck}	Leckverlustleistung
P	Leistungsaufnahme
A_{krit}	Logikfläche eines Prozessorkerns
t_{krit}	Länge des kritischen Pfades
f_{max}	Maximal erreichbare Taktfrequenz einer synchronen Schaltung
A_{max}	Obere Schranke der Fläche
P_{max}	Obere Schranke der Leistungsaufnahme
T_{max}	Obere Schranke der Zeit
p	Pfad kombinatorischer Logik durch eine synchrone Schaltung
RE	Ressourceneffizienz
O	Routing Overhead
α	Schalthäufigkeit eines CMOS-Gatters
$t_{setup,dest}$	Setup-Zeit des Zielregisters
$t_{pd.wire}$	Signalverzögerungen auf den Leitungen
s	Skalierungsfaktor
P_{stat}	Statische Verlustleistung
P_{quer}	Statische Verlustleistung durch Querströme (Subschwellströme)
$t_{pd,gate}$	Summe der Verzögerungszeiten der Gatter
$\mathbb{S}$	System
U	Utilization

P_{schalt}	Verlustleistung durch Querströme während eines Umschaltvorganges
U_{dd}	Versorgungsspannung
t_{delay}	Verzögerungszeit eines Pfades innerhalb einer kombinatorischen Logik
$t_{period,constr}$	Vorgegebene Periodendauer
t_{wns}	Worst Negative Slack – Differenz zwischen Latenz des kritischen Pfades und vorgegebener Periodendauer
T	Zeit
$h[n]$	Zeitdiskrete Impulsantwort
$y[n]$	Zeitdiskretes Ausgangssignal
$\mathbb{Z}$	Zieltechnologie

Literaturverzeichnis

[1] 3GPP: Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 1: UEA2 and UIA2 Specification. (2006), S. 1–27

[2] 3GPP: Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification. (2006), S. 1–37

[3] 3GPP: Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 5: Design and Evaluation Report. (2006), S. 1–40

[4] ADIR, A. ; ALMOG, E. ; FOURNIER, L. ; MARCUS, E. ; RIMON, M. ; VINOV, M. ; ZIV, A.: Genesys-pro: Innovations in Test Program Generation for Functional Processor Verification. In: *IEEE Design & Test of Computers* 21 (2005), Nr. 2, S. 84–93

[5] AGARWAL, A.: The Tile Processor: A 64-Core Multicore for Embedded Processing. In: *Proc. of the High-Performance Embedded Computing Workshop (HPEC)*, September 2007, S. 88–598

[6] AHARON, A. ; GOODMAN, D. ; LEVINGER, M. ; LICHTENSTEIN, Y. ; MALKA, Y. ; METZGER, C. ; MOLCHO, M. ; SHUREK, G.: Test Program Generation for Functional Verification of PowerPC Processors in IBM. (2006), S. 279–285

[7] AHUJA, P. S. ; CLARK, D. W. ; ROGERS, A.: The Performance Impact of Incomplete Bypassing in Processor Pipelines. In: *Proc. of the 28th Annual International Symposium on Microarchitecture (MICRO)*, 1995, S. 36–45. – ISBN 0-8186-7349-4

[8] ALT, M. ; ASSMANN, U. ; VAN SOMEREN, H.: Cosy Compiler Phase Embedding with the Cosy Compiler Model. In: *Proc. of the 5th International Conference on Compiler Construction*, 1994, S. 278–293

[9] AMDAHL, G. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *AFIPS '67 (Spring): Proc. of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1967, S. 483–485

[10] ANDRAUS, Z. S.: *Automatic Formal Verification of Control Logic in Hardware Designs*, Carnegie Mellon University, Dissertation, 2009

[11] ANGIOLINI, F. ; BENINI, L. ; CAPRARA, A.: Polynomial-time Algorithm for On-chip Scratchpad Memory Partitioning. In: *Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003, S. 318–326. – ISBN 1581136765

[12] ANTONELLI, D. A. ; SMITH, A. J. ; WAERDT, J.-W. van de: Power Consumption and Reduction in a Real, Commercial Multimedia Core. In: *Proc. of the 6th ACM Conference on Computing Frontiers (CF)*. New York, NY, USA : ACM, 2009, S. 171–174. – ISBN 978-1-60558-413-3

[13] AZEVEDO, R. ; RIGO, S. ; BARTHOLOMEU, M. ; ARAUJO, G. ; ARAUJO, C. ; BARROS, E.: The ArchC architecture description language and tools. In: *International Journal of Parallel Programming* 33 (2005), Nr. 5, S. 453–484. – ISSN 0885-7458

[14] BACKUS, J.: Can Programming be Liberated From the Von Neumann Style?: A Functional Style and its Algebra of Programs. In: *ACM Turing Award Lectures*, 2007, S. 1977

[15] BAGHERI, R. ; MIRZAEI, A. ; CHEHRAZI, S. ; HEIDARI, M. ; LEE, M. ; MIKHEMAR, M. ; TANG, M. ; ABIDI, A.: An 800MHz to 5GHz Software-defined Radio Receiver in 90nm CMOS. In: *Proc. of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2006, S. 1932–1941. – ISBN 1424400791

[16] BANAKAR, R. ; STEINKE, S. ; LEE, B.-S. ; BALAKRISHNAN, M. ; MARWEDEL, P.: Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. (2002), S. 73–78. ISBN 1581135424

[17] BANERJEE, K. ; LIN, S. C. ; SRIVASTAVA, N.: Electrothermal Engineering in the Nanometer Era: From Devices and Interconnects to Circuits and Systems. In: *Asia and South Pacific Conference on Design Automation*, 2006, S. 8

[18] BARON, Max: Tilera's Cores Communicate Better. In: *Microprocessor Report* 21 (2007), November, S. 11–17

[19] BELL, S. ; EDWARDS, B. ; AMANN, J. ; CONLIN, R. ; JOYCE, K. ; LEUNG, V. ; MACKAY, J. ; REIF, M. ; BAO, Liewei ; BROWN, J. ; MATTINA, M. ; MIAO, Chyi-Chang ; RAMEY, C. ; WENTZLAFF, D. ; ANDERSON, W. ; BERGER, E. ; FAIRBANKS, N. ; KHAN, D. ; MONTENEGRO, F. ; STICKNEY, J. ; ZOOK, J.: TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In: *Proc. of the IEEE International Solid-State Circuits Conference (ISSCC)*, Februar 2008, S. 88–598

[20] BENINI, L. ; BRUNI, D. ; CHINOSI, M. ; SILVANO, C. ; ZACCARIA, V. ; ZAFALON, R.: A Power Modeling and Estimation Framework for VLIW-based Embedded Systems. In: *ST Journal of System Research*, 2001, S. 110–118

[21] BEYLS, K. ; D'HOLLANDER, E.: Cache Remapping to Improve the Performance of Tiled Algorithms. In: *Euro-Par 2000 Parallel Processing*, 2000, S. 998–1007

[22] BEYLS, K. ; D'HOLLANDER, E.: Compiler-generated Dynamic Scratch Pad Memory Management. In: *Workshop on Application Specific Processors (WASP)*, 2003

[23] BHADRA, J. ; TROFIMOVA, E. ; GIORDANO, L. J. ; ABADIR, M. S.: A Trace-driven Validation Methodology for Multi-processor SoCs. In: *IEEE International SOC Conference, 2006*, 2006, S. 145–148

[24] BHATNAGAR, H.: *Advanced ASIC Chip Synthesis: Using Synopsys Design Compiler, Physical Compiler, and PrimeTime*. Springer, 2002

[25] BIN, E. ; EMEK, R. ; SHUREK, G. ; ZIV, A.: Using a constraint satisfaction formulation and solution techniques for random test program generation. In: *IBM Systems Journal* 41 (2002), Nr. 3, S. 386–402

[26] BLANCHETTE, J. ; SUMMERFIELD, M.: *C++ GUI programming with Qt 4*. Prentice Hall, 2008

[27] BLECH, I. A.: Electromigration in Thin Aluminum Films on Titanium Nitride. In: *Journal of Applied Physics* 47 (1976), April, Nr. 4, S. 1203–1208. – ISSN 0021-8979

[28] BLESKEN, M. ; RÜCKERT, U. ; STEENKEN, D. ; WITTING, K. ; DELLNITZ, M.: Multiobjective Optimization for Transistor Sizing of CMOS Logic Standard Cells Using Set-oriented Numerical Techniques. In: *NORCHIP, 2009*, 2010, S. 1–4

[29] BLÜTHGEN, H. M. ; GRASSMANN, C. ; RAAB, W. ; RAMACHER, U. ; HAUSNER, J.: A Programmable Baseband Platform for Software-defined Radio. In: *Proc. of the SDR Forum*, 2004

[30] BOBBA, S. ; THORP, T. ; AINGARAN, K. ; LIU, D.: IC power distribution challenges. In: *Proc. of the 2001 IEEE/ACM International Conference on Computer-aided Design*, 2001, S. 650

[31] BOESE, K. D. ; KAHNG, A. B. ; MANTIK, S.: On the Relevance of Wire Load Models. In: *Proc. of the International Workshop on System-level Interconnect Prediction (SLIP)*. New York, NY, USA : ACM, 2001, S. 91–98. – ISBN 1-58113-315-4

[32] BRIGHAM, E. O. ; MORROW, R. E.: The Fast Fourier Transform. In: *IEEE Spectrum* 4 (2009), Nr. 12, S. 63–70

[33] BRINKSCHULTE, U. ; UNGERER, T.: *Mikrocontroller und Mikroprozessoren*. Springer, 2002

[34] BROWN, M. D. ; PATT, Y. N.: Using Internal Redundant Representations and Limited Bypass to Support Pipelined Adders and Register Files. In: *Proc. of the 8th Annual International Symposium on High-Performance Computer Architecture*, 2001, S. 289–298

[35] BÖTTCHER, A.: *Rechneraufbau und Rechnerarchitektur*. Springer, 2006

[36] BURCH, J. R. ; DILL, D. L.: Automatic Verification of Pipelined Microprocessor Control. In: *Proc. of the 6th International Conference on Computer Aided Verification (CAV)*. London, UK : Springer, 1994, S. 68–80. – ISBN 3-540-58179-0

[37] BURNS, G. ; JACOBS, M. ; LINDWER, M. ; VANDEWIELE, B.: *Silicon Hive's Scalable and Modular Architecture Template for High-performance Multi-core Systems*. Silicon Hive, 2006

[38] CALIMERA, A. ; MACII, E. ; RAVOTTO, D. ; SANCHEZ, E. ; SONZA REORDA, M.: Generating Power-hungry Test Programs for Power-aware Validation of Pipelined Processors. In: *Proc. of the 23rd Symposium on Integrated Circuits and System Design*, 2010, S. 61–66

[39] CASE, B.: Intel Reveals Pentium Implementation Details. In: *Microprocessor Report* 5 (1993), Nr. 23, S. 9–17

[40] CATANIA, V. ; PALESI, M. ; PATTI, D.: Reducing Complexity of Multiobjective Design Space Exploration in VLIW-based Embedded Systems. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5 (2008), Nr. 2, S. 1–33

[41] CEZANNE, W.: *Allgemeine Volkswirtschaftslehre*. Oldenbourg Verlag, 2005

[42] CHANDRAKASAN, A. P. ; SHENG, S. ; BRODERSEN, R. W.: Low-power CMOS Digital Design. In: *IEEE Journal of Solid-State Circuits* 27 (2002), Nr. 4, S. 473–484

[43] CHANG, Y.-S. ; LEE, S. ; PARK, I.-C. ; KYUNG, C.-M.: Verification of a Microprocessor Using Real World Applications. In: *Proc. of the 36th ACM/IEEE Design Automation Conference (DAC)*. New York, NY, USA : ACM, 1999, S. 181–184. – ISBN 1-58133-109-7

[44] CHAOUI, J. ; CYR, K. ; DE GREGORIO, S. ; GIACALONE, J.-P. ; WEBB, J. ; MASSE, Y.: Open Multimedia Application Platform: Enabling Multimedia Applications in Third Generation Wireless Terminals Through a Combined RISC/DSP Architecture. In: *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Washington, DC, USA : IEEE Computer Society, 2001, S. 1009–1012. – ISBN 0-7803-7041-4

[45] CHATTOPADHYAY, A. ; SINHA, A. ; ZHANG, D. ; LEUPERS, R. ; ASCHEID, G. ; MEYR, H.: Integrated Verification Approach During ADL-driven Processor Design. In: *Microelectronics Journal* 40 (2009), Nr. 7, S. 1111–1123

[46] CHIEN, C.: *Digital Radio Systems on a Chip: A Systems Approach*. Springer, 2000

[47] CHOI, J. S. ; LEE, K.: Design of CMOS Tapered Buffer for Minimum Power-delay Product. In: *IEEE Journal of Solid-State Circuits* 29 (2002), Nr. 9, S. 1142–1145

[48] CLARKE, E. M.: My 27-year Quest to Overcome the State Explosion Problem. In: *24th Annual IEEE Symposium on Logic In Computer Science (LICS)*, August 2009, S. 3. – ISSN 1043-6871

[49] CLARKE, E. M. ; GRUMBERG, O.: Avoiding the State Explosion Problem in Temporal Logic Model Checking. In: *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987, S. 294–303

[50] COHEN-YASHAR, E.: From Tools to Flow: Linking the Chains in Cadence Reference Flow. In: *Proc. of the CDNLive!*, 2006

[51] COLWELL, R. P.: VLIW: The Unlikeliest Computer Architecture. In: *IEEE Solid-State Circuits Magazine* Spring (2009), Nr. 1, S. 18–22

[52] COLWELL, Robert P. ; HALL, W. E. ; JOSHI, Chandra S. ; PAPWORTH, David B. ; RODMAN, Paul K. ; TORNES, James E.: Architecture and Implementation of a VLIW Supercomputer. In: *Proc. of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1990, S. 910–919. – ISBN 0-89791-412-0

[53] COOLEY, J. W. ; TUKEY, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series. In: *Mathematics of Computation* 19 (1965), Nr. 90, S. 297–301

[54] CORMEN, T. H.: *Introduction to Algorithms*. The MIT Press, 2001. – ISBN 0-262-03293-7

[55] CORNO, F. ; CUMANI, G. ; REORDA, S. ; SQUILLERO, G.: Fully Automatic Test Program Generation for Microprocessor Cores. In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2003, S. 1006–1011

[56] DAEMEN, J. ; RIJMEN, V.: *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002

[57] DASGUPTA, S.: Common Power Format: A User-driven Ecosystem For Proven Low Power Design Flows. In: *Proc. of the 22nd International Conference on VLSI Design*, 2009, S. 5

[58] DEHNERT, J. C. ; GRANT, B. K. ; BANNING, J. P. ; JOHNSON, R. ; KISTLER, T. ; KLAIBER, A. ; MATTSON, J.: The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In: *International Symposium on Code Generation and Optimization (CGO), 2003*, 2003, S. 15–24. – ISBN 076951913X

[59] DELLNITZ, M. ; SCHÜTZE, O. ; HESTERMEYER, T.: Covering Pareto sets by multilevel subdivision techniques. In: *Journal of Optimization Theory and Applications* 124 (2005), Nr. 1, S. 113–136

[60] DESHPANDE, S. D.: Software Implementation of IEEE 802.11b Wireless LAN Standard. In: *Proceeding of the SDR 04 Technical Conference and Product Exposition*, 2004

[61] DESOLI, G. ; STRUDEL, T. ; COUSIN, J.-P. ; SAHA, K.: Current and Future Trends in Embedded VLIW Microprocessors Applied to Multimedia and Signal Processing. In: *Proc. of the 14th European Signal Processing (EUSIPCO) Conference*, September 2006

[62] EDEN, M. ; KAGAN, M.: Pentium Processor with MMX Technology. In: *The 1997 IEEE COMPCON Conference*, 1997, S. 260–262

[63] EKDAHL, P. ; JOHANSSON, T.: *SNOW – A New Stream Cipher*. 2000

[64] EKDAHL, P. ; JOHANSSON, T.: A New Version of the Stream Cipher SNOW. In: *Selected Areas in Cryptography* Springer (Veranst.), 2003, S. 47–61

[65] ELLIS, John R.: *Bulldog: A Compiler for VLIW Architectures*, Yale University, Dissertation, Januar 1985

[66] ETTUS RESEARCH LLC: – URL `http://www.ettus.com/`

[67] FACHGRUPPE SCHALTUNGSTECHNIK, UNIVERSITÄT PADERBORN: Handbuch DB-Ethernet-Modul Rev. 2. (2004)

[68] FAN, K. ; CLARK, N. ; CHU, M. ; MANJUNATH, K. V. ; RAVINDRAN, R. ; SMELYANSKIY, M. ; MAHLKE, S.: Systematic Register Bypass Customization for Application-Specific Processors. In: *Proc. of the Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, 2003, S. 64–74

[69] FARABOSCHI, P. ; BROWN, G. ; FISHER, J. A. ; DESOLI, G. ; HOMEWOOD, F.: Lx: A Technology Platform for Customizable VLIW Embedded Processing. In: *ACM SIGARCH Computer Architecture News* 28 (2000), Nr. 2, S. 203–213. – ISSN 0163-5964

[70] FARABOSCHI, P. ; HOMEWOOD, F.: ST200: A VLIW Architecture for Media-Oriented Applications. In: *Microprocessor Forum*, Oktober 2000

[71] FERBER, M.: *Ein dynamisch rekonfigurierbarer Ethernet-Switch*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2005

[72] FISHER, J. A.: Very Long Instruction Word architectures and the ELI-512. In: *Proc. of the 10th annual international symposium on Computer architecture (ISCA)*. New York, NY, USA : ACM, 1983, S. 140–150. – ISBN 0-89791-101-6

[73] FISHER, J. A.: Retrospective: Very Long Instruction Word Architectures and the ELI-512. In: *IEEE Solid-State Circuits Magazine* Spring 2009 (2009), Nr. 1, S. 34–36

[74] FISHER, J. A. ; FARABOSCHI, P. ; YOUNG, C.: *Embedded computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005

[75] FISHER, J. A. ; FARABOSCHI, P. ; YOUNG, C.: VLIW Processors: From Blue Sky To Best Buy. In: *IEEE Solid-State Circuits Magazine* Spring 2009 (2009), Nr. 1, S. 10–17

[76] FLÜGEL, S.: *Kompakte Cache-Architekturen für SPMD-Prozessoren*. Der Andere Verlag, 2008. – ISBN 978-3-89959-733-2

[77] FORNEY, G. D.: The Viterbi Algorithm. In: *Proc. of the IEEE* 61 (1973), Nr. 3, S. 268–278. – ISSN 0018-9219

[78] FUJITSU LABORATORIES LTD.: Fujitsu Develops Multi-core Processor for High-Performance Digital Consumer Products. (2005), Februar. – URL http://www.fujitsu.com/global/news/pr/archives/month/2005/20050207-01.html

[79] FURLANI, J. L.: Modules: Providing a Flexible User Environment. In: *Proc. of the 5th Large Installation Systems Administration Conference (LISA)*, 1991, S. 141–152

[80] FURLANI, J. L. ; OSEL, P. W.: Abstract yourself with modules. In: *Proc. of the 10th Large Installation Systems Administration Conference (LISA)* (1996), S. 193–204

[81] GAO, H. ; DIMITROV, M. ; KONG, J. ; ZHOU, H.: Experiencing Various Massively Parallel Architectures and Programming Models for Data-intensive Applications. In: *Workshop on Computer Architecture Education (WCAE)* Bd. 35, 2008

[82] GEILEN, M. ; BASTEN, T. ; THEELEN, B. ; OTTEN, R.: An Algebra of Pareto Points. In: *Fundamenta Informaticae* 78 (2007), Nr. 1, S. 35–74

[83] GESSLER, R. ; MAHR, T.: *Hardware-Software Co-Design*. Vieweg + Teubner, 2007

[84] GNU RADIO: – URL http://www.gnuradio.org/

[85] GOEL, N. ; KUMAR, A. ; PANDA, P. R.: Power Reduction in VLIW Processor with Compiler Driven Bypass Network. In: *Proc. of the 20th International Conference on VLSI Design (VLSID)*, 2007, S. 233–238. – ISBN 0-7695-2762-0

[86] GOLDSTEIN, H. ; VON NEUMANN, J. ; BURKS, A.: Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument / Princeton Institute of Advanced Study. 1947. – Forschungsbericht

[87] GONZALEZ, R. E.: Xtensa: A Configurable and Extensible Processor. In: *IEEE Micro* 20 (2002), Nr. 2, S. 60–70. – ISSN 0272-1732

[88] GOVINDARAJULU, S. ; PRASAD, T. J.: Considerations of performance factors in CMOS designs. In: *International Conference on Electronic Design (ICED)*, 2009, S. 1–6

[89] GRÜNEWALD, M. ; KASTENS, U. ; LE, D. K. ; NIEMANN, J.-C. ; PORRMANN, M. ; RÜCKERT, U. ; THIES, M. ; SLOWIK, A.: Network Application Driven Instruction Set Extensions for Embedded Processing Clusters. In: *Proc. of the International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, 2004, S. 209–214

[90] HAGEMEYER, Jens: *Partielle, dynamische Hardware-Rekonfiguration in der RAPTOR2000-Umgebung*, Fachgruppe Schaltungstechnik, Universität Paderborn, Studienarbeit, 2005

[91] HALFHILL, T. R.: EEMBC Releases First Benchmarks. In: *Microprocessor Report* 1 (2000)

[92] HALFHILL, T. R.: Tensilicas Preconfigured Cores - Six Embedded-Processor Cores Challenge ARM, ARC, MIPS, and DSPs. In: *Microprocessor Report* (2006)

[93] HALFHILL, Tom R.: Intel Maps Wireless Future. In: *Microprocessor Report* 23 (2003), Juni, S. 1–4

[94] HO, R. C. ; HAN YANG, C. ; HOROWITZ, M. A. ; DILL, D. L.: Architecture Validation for Processors. In: *Proc. of the 22nd Annual International Symposium on Computer Architecture*, Juni 1995, S. 404–413. – ISSN 1063-6897

[95] HONG, H. ; LEE, I. ; SOKOLSKY, O. ; URAL, H.: A Temporal Logic Based Theory of Test Coverage and Generation. In: *Tools and Algorithms for the Construction and Analysis of Systems* (2002), S. 151–161

[96] HOSSEINI, A. ; MAVROIDIS, D. ; KONAS, P.: Code Generation and Analysis for the Functional Verification of Micro Processors. In: *Proc. of the 33rd ACM/IEEE Design Automation Conference (DAC)*, 1996, S. 305–310. – ISBN 0-89791-779-0

[97] HU, C.: Future CMOS Scaling and Reliability. In: *Proc. of the IEEE* 81 (2002), Nr. 5, S. 682–689. – ISSN 0018-9219

[98] HU, X. ; EBERHART, R.: Multiobjective Optimization Using Dynamic Neighborhood Particle Swarm Optimization. In: *Proc. of the 2002 Congress on Evolutionary Computation (CES)* (2002), S. 1677–1681

[99] IANNUCCI, R. A.: Toward a Dataflow/von Neumann Hybrid Architecture. In: *ACM SIGARCH Computer Architecture News* 16 (1988), Nr. 2, S. 131–140. – ISSN 0163-5964

[100] IBRAHIM, M. E. A. ; RUPP, M. ; HABIB, S. E.-D.: Performance and Power Consumption Trade-offs for a VLIW DSP. In: *International Symposium on Signals, Circuits and Systems (ISSCS)*, Juli 2009, S. 1–4

[101] IEEE COMPUTER SOCIETY: *IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. November 2007. – URL `http://standards.ieee.org/getieee802/download/802.11-2007.pdf`

[102] INFINEON TECHNOLOGIES AG: X-Gold™ SDR 20, Product Information. (2009), August

[103] INTEL CORPORATION: Division, Square Root and Remainder Algorithms for the Intel® Itanium™ Architecture. (2003), November, S. 1–120

[104] ISHIURA, N. ; YAMAGUCHI, M.: Instruction Code Compression for Application Specific VLIW Processors Based on Automatic Field Partitioning. In: *Proc. of the Workshop on Synthesis and System Integration of Mixed Technologies*, 1998, S. 105–109

[105] ITRS: International Technology Roadmap for Semiconductors (ITRS) / European Semiconductor Industry Association (ESIA), Japan Electronics, Information Technology Industries Association (JEITA), Korean Semiconductor Industry Association (KSIA), Taiwan Semiconductor Industry Association (TSIA), and United States Semiconductor Industry Association (SIA). 2009. – Forschungsbericht

[106] ITRS: International Technology Roadmap for Semiconductors (ITRS) – Update / European Semiconductor Industry Association (ESIA), Japan Electronics and Information Technology Industries Association (JEITA), Korean Semiconductor Industry Association (KSIA), Taiwan Semiconductor Industry Association (TSIA), and United States Semiconductor Industry Association (SIA). 2010. – Forschungsbericht

[107] JANG, B. ; DO, S. ; PIEN, H. ; KAELI, D.: Architecture-aware Optimization Targeting Multithreaded Stream Computing. In: *Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*. New York, NY, USA : ACM, 2009, S. 62–70. – ISBN 978-1-60558-517-8

[108] K., Ronald L. ; V., Russell D.: *The CISSP Prep Guide, Gold Edition*. Wiley, 2002

[109] KAHN, D.: *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet, revised*. Scribner, 1996

[110] KALTE, H. ; PORRMANN, M. ; RÜCKERT, U.: A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In: *Proc. of the IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (SoC)*, 2002

[111] KANTER, D.: EEMBC Energizes Benchmarks. In: *Microprocessor Report* (2006)

[112] KASTENS, U. ; LE, D. K. ; SLOWIK, A. ; THIES, M.: Feedback Driven Instruction-set Extension. In: *ACM SIGPLAN Notices* 39 (2004), Nr. 7, S. 126–135

[113] KLAIBER, A.: The Technology behind Crusoe™ Processors. In: *Transmeta Technical Brief* (2000), Januar

[114] KLAR, H. ; HEIMSCH, W.: *Integrierte Digitale Schaltungen MOS/BICMOS*. Springer, 1993. – ISBN 3540544747

[115] KUNZLI, S. ; THIELE, L. ; ZITZLER, E.: Modular Design Space Exploration Framework for Embedded Systems. In: *Computers and Digital Techniques, IEE Proceedings* Bd. 152, 2005, S. 183–192

[116] KURUP, P. ; ABBASI, T.: *Logic Synthesis Using Synopsys*. Kluwer Academic Publishers, 1997

[117] LAMA, Carlos S. ; JÄÄSKELÄINEN, Pekka ; TAKALA, Jarmo: Programmable and Scalable Architecture for Graphics Processing Units. In: *Proc. of the 9th International Workshop on Embedded Computer Systems (SAMOS): Architectures, Modeling, and Simulation*, Springer, 2009, S. 2–11. – ISBN 978-3-642-03137-3

[118] LANGEN, D.: S-Core Reference Manual. (2001)

[119] LANGEN, D.: *Abschätzung des Ressourcenbedarfs von hochintegrierten mikroelektronischen Systemen*, Fachgruppe Schaltungstechnik, Universität Paderborn, Dissertation, Januar 2004

[120] LANGEN, D. ; NIEMANN, J.-C. ; PORRMANN, M. ; KALTE, H. ; RÜCKERT, U.: Implementation of a RISC Processor Core for SoC Designs – FPGA Prototype vs. ASIC Implementation. In: *Proc. of the IEEE-Workshop: Heterogeneous Reconfigurable Systems on Chip (SoC)*, 2002

[121] LEIBSON, S. ; KIM, J.: Configurable processors: a new era in chip design. In: *Computer* 38 (2005), Nr. 7, S. 51–59. – ISSN 0018-9162

[122] LETHIN, R. A.: How VLIW Almost Disappeared – and Then Proliferated. In: *IEEE Solid-State Circuits Magazine* Summer 2009 (2009), Nr. 2, S. 15–23

[123] LEVY, M.: EEMBC 1.0 Scores, Part 1: Observations. In: *Microprocessor Report* 14 (2000), Nr. 33, S. 132–141

[124] LIN, S. C. ; SRIVASTAVA, N. ; BANERJEE, K.: A Thermally-aware Methodology for Design-specific Optimization of Supply and Threshold Voltages in Nanometer Scale ICs. In: *Proc. of the IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, 2005, S. 411–416

[125] LINDSEY, S. ; RAGHAVENDRA, C. ; SIVALINGAM, K.: Data Gathering in Sensor Networks Using the Energy*Delay Metric. In: *Proc. of the 15th International Parallel & Distributed Processing Symposium*, 2001, S. 188

[126] LIU, D. ; NILSSON, A. ; TELL, E. ; WU, D. ; EILERT, J.: Bridging Dream and Reality: Programmable Baseband Processors for Software-defined Radio. In: *IEEE Communications Magazine* 47 (2009), Nr. 9, S. 134–140

[127] LÓPEZ, J. ; DAHAB, R.: Fast Multiplication on Elliptic Curves Over GF(2m) without Precomputation. In: *Proc. of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. London, UK : Springer, 1999, S. 316–327. – ISBN 3-540-66646-X

[128] LÜTKEMEIER, S. ; KAULMANN, T. ; RÜCKERT, U.: A Sub-200mV 32bit ALU with 0.45pJ/Instruction in 90nm CMOS. In: *Semiconductor Conference Dresden*, April 2009

[129] LÜTKEMEIER, S. ; RÜCKERT, U.: A Subthreshold to Above-Threshold Level Shifter Comprising a Wilson Current Mirror. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 57 (2010), September, Nr. 9, S. 721–724

[130] LUNG, C. L. ; HSIAO, H. C. ; ZENG, Z. Y. ; CHANG, S. C.: LP-based Multi-mode Multi-corner Clock Skew Optimization. In: *International Symposium on VLSI Design Automation and Test (VLSI-DAT)*, April 2010, S. 335–338

[131] LUNGU, A. ; SORIN, D. J.: Verification-Aware Microprocessor Design. In: *Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques*. Washington, DC, USA : IEEE Computer Society, 2007 (PACT '07), S. 83–93. – ISBN 0-7695-2944-5

[132] MAMIDI, S. ; BLEM, E. ; SCHULTE, M. J. ; GLOSSNER, J. ; IANCU, D. ; IANCU, A. ; MOUDGILL, M. ; JINTURKAR, S.: Instruction Set Extensions for Software Defined Radio. In: *Microprocessors and Microsystems* 33 (2009), Nr. 4, S. 260–272. – ISSN 0141-9331

[133] MAMIDI, S. ; BLEM, E. R. ; SCHULTE, M. J. ; GLOSSNER, J. ; IANCU, D. ; IANCU, A. ; MOUDGILL, M. ; JINTURKAR, S.: Instruction set Extensions for Software Defined Radio on a Multithreaded Processor. In: *Proc. of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded systems*, 2005, S. 266–273. – ISBN 159593149X

[134] MARTINEZ, G.: TMS320C6418 Power Consumption Summary. (2005), Februar. – URL `http://focus.ti.com/lit/an/spraa60a/spraa60a.pdf`

[135] MEI, B. ; LAMBRECHTS, A. ; MIGNOLET, J. Y. ; VERKEST, D. ; LAUWEREINS, R.: Architecture Exploration for a Reconfigurable Architecture Template. In: *IEEE Design & Test of Computers* 22 (2005), Nr. 2, S. 90–101. – ISSN 0740-7475

[136] MICRON TECHNOLOGY: SDRAM Power Calc 10. (2001). – URL `http://www.micron.com/document_download/?documentId=31`

[137] MICRON TECHNOLOGY: Synchronous DRAM MT48LC4M32B2. (2001). – URL `http://download.micron.com/pdf/datasheets/dram/sdram/128MbSDRAMx32.pdf`

[138] MILDENBERGER, O.: *System- und Signaltheorie*. Bd. 3. Vieweg + Teubner, 1988

[139] MILENKOVIC, A. ; MILENKOVIC, M. ; BARNES, N.: *A Performance Evaluation of Memory Hierarchy in Embedded Systems*. 2003

[140] MILLER, R. G. ; CARDILLO, L. A. ; MATHIESON, J. G. ; SMITH, E. R.: *Instruction Compression and Decompression System and Method for a Processor*. 6. Oktober 1998. – US Patent 5,819,058

[141] MILNE, G.: Design for verifiability. In: *Hardware Specification, Verification and Synthesis: Mathematical Aspects* (1990), S. 1–13

[142] MISHRA, P. ; DUTT, N.: Graph-based Functional Test Program Generation for Pipelined Processors. In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE)* Bd. 1, 2004, S. 182–187

[143] MITOLA, J.: The software radio architecture. In: *IEEE Communications Magazine* 33 (1995), Nr. 5, S. 26–38

[144] MITOLA III, J.: Software Radios: Survey, Critical Evaluation and Future Directions. In: *IEEE Aerospace and Electronic Systems Magazine* 8 (1993), Nr. 4, S. 25–36

[145] MONTGOMERY, Peter L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. In: *Mathematics of Computation* 48 (1987), S. 243–264. – ISSN 0025–5718

[146] MOON, S.-M. ; EBCIOGLU, K.: A Study on the Number of Memory Ports in Multiple Instruction Issue Machines. In: *Proc. of the 26th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1993, S. 49–59

[147] MOORE, G. E.: Cramming More Components onto Integrated Circuits. In: *Proc. of the IEEE* 86 (1998), Nr. 1, S. 82–85. – ISSN 0018-9219

[148] MORGAN, P. ; TAYLOR, R. ; HOSSELL, J. ; BRUCE, G. ; O'ROURKE, B.: Automated Data Cache Placement for Embedded VLIW ASIPs. In: *Proc. of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005, S. 39–44

[149] MOWBRAY, T. J. ; RUH, W.: *Inside CORBA: Distributed Object Standards and Applications*. Boston, MA, USA : Addison-Wesley, 1998. – ISBN 0-201-89540-4

[150] MULJONO, H. ; RUSU, S. ; CHERKAUER, B. ; STINSON, J.: New 130nm Itanium© 2 Processors for 2003. In: *Hot Chips*, Oktober 2003, S. 1–22

[151] MYERS, G. J.: *Advances in Computer Architecture*. Wiley, 1982

[152] NGATCHOU, P. ; ZAREI, A. ; EL-SHARKAWI, M. A.: Pareto Multi Objective Optimization. In: *Proc. of the 13th International Conference on Intelligent Systems Application to Power Systems*, 2006, S. 84–91

[153] NIEMANN, J.-C.: *Ressourceneffiziente Schaltungstechnik eingebetteter Parallelrechner – GigaNetIC*, Fachgruppe Schaltungstechnik, Universität Paderborn, Dissertation, 2009

[154] NUSSBAUMER, H. J.: Fast Fourier Transform and Convolution Algorithms. In: *Springer Series in Information Sciences* 2 (1982)

[155] OKANO, H. ; SUGA, A. ; SHIOTA, T. ; TAKEBE, Y. ; NAKAMURA, Y. ; HIGAKI, N. ; KIMURA, H. ; MIYAKE, H. ; SATOH, T. ; KAWASAKI, K. ; SASAGAWA, R. ; SHIBAMOTO, W. ; SASAKI, M. ; ANDO, N. ; YAMANA, T. ; FUKUSHI, I. ; TAGO, S. ; HAYAKAWA, F. ; KAMIGATA, T. ; IMAI, S. ; SATOH, A. ; HATTA, Y. ; NISHIMURA, N. ; ASADA, Y. ; SUKEMURA, T. ; ANDO, S. ; TAKAHASHI, H.: An 8-way VLIW Embedded Multimedia Processor Built in 7-layer Metal 0.11 um CMOS Technology. In: *Proc. of the IEEE International Solid-State Circuits Conference (ISSCC)* Bd. 1, 2002, S. 374–375

[156] PANDA, P. R. ; DUTT, N. D. ; NICOLAU, A.: Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In: *Proc. of the 1997 European Conference on Design and Test*, 1997, S. 7–11

[157] PATTERSON, D. A. ; HENNESSY, J. L.: *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA : Morgan Kaufmann, 1990. – ISBN 1-55880-069-8

[158] PATTERSON, D. A. ; HENNESSY, J. L.: *Rechnerorganisation und -entwurf - Die Hardware/Software-Schnittstelle*. Elsevier, 2005. – ISBN 3-8274-1595-0

[159] PETERSON, W. W. ; BROWN, D. T.: Cyclic Codes for Error Detection. In: *Proc. of the IRE* 49 (1961), Nr. 1, S. 228–235

[160] POOVEY, J.: Characterization of the EEMBC Benchmark Suite / North Carolina State University. 2007. – Forschungsbericht

[161] PORRMANN, M. ; HAGEMEYER, J. ; POHL, C. ; ROMOTH, J. ; STRUGHOLTZ, M.: . Bd. 19. Kap. RAPTOR – A Scalable Platform for Rapid Prototyping and FPGA-based Cluster Computing, S. 592–599. In: *Parallel Computing: From Multicores and GPU's to Petascale, Advances in Parallel Computing* Bd. 19, IOS Press, 2010. – ISBN 978-1-60750-529-7

[162] PORRMANN, M. ; HAGEMEYER, J. ; ROMOTH, J. ; STRUGHOLTZ, M.: Rapid Prototyping of Next-Generation Multiprocessor SoCs. In: *Proc. of the Semiconductor Conference Dresden (SCD)*. Dresden, Germany, 2009, S. 29–30

[163] PURNAPRAJNA, M. ; PORRMANN, M. ; RÜCKERT, U. ; HUSSMANN, M. ; THIES, M. ; KASTENS, U.: Runtime Reconfiguration of Multiprocessors Based on Compile-Time Analysis. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3 (2010), Nr. 3, S. 1–25. – ISSN 1936-7406

[164] PURNAPRAJNA, M. S. M.: *Run-time Reconfigurable Multiprocessors*, Fachgruppe Schaltungstechnik, Universität Paderborn, Dissertation, 2010

[165] PUTTMANN, C. ; NIEMANN, J.-C. ; PORRMANN, M. ; RÜCKERT, U.: GigaNoC – A Hierarchical Network-on-chip for Scalable Chip-multiprocessors. In: *Proc. of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, 2007, S. 495–502

[166] PUTTMANN, C. ; SHOKROLLAHI, J. ; PORRMANN, M.: Resource Efficiency of Instruction Set Extensions for Elliptic Curve Cryptography. In: *Proc. of the 5th International Conference on Information Technology: New Generations*, 2008, S. 131–136

[167] PUTTMANN, C. ; SHOKROLLAHI, J. ; PORRMANN, M. ; RÜCKERT, U.: Hardware Accelerators for Elliptic Curve Cryptography. In: *Advances in Radio Science* 6 (2008), S. 259–263. – ISSN 1684-9965

[168] QURESHI, S.: *Embedded Image Processing on the TMS320C6000 DSP: Examples in Code Composer Studio and MATLAB*. Springer, 2005

[169] RAMACHER, U.: Software-Defined Radio Prospects for Multistandard Mobile Phones. In: *Computer*, Oktober 2007, S. 62–69

[170] RASHID, M. ; APVRILLE, L. ; PACALET, R.: Application Specific Processors for Multimedia Applications. In: *2008 11th IEEE International Conference on Computational Science and Engineering*, 2008, S. 109–116

[171] RICHARDSON, I.: *The H.264 Advanced Video Compression Standard*. Wiley, 2010

[172] SAATY, T. L.: *Multicriteria Decision Making: The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. RWS Publications, 1990

[173] SAMI, M. ; SCIUTO, D. ; SILVANO, C. ; ZACCARIA, V. ; ZAFALON, R.: Low-Power Data Forwarding for VLIW Embedded Architectures. In: *IEEE Transactions on Very Large Scale Integration Systems* 10 (2002), Nr. 5, S. 614–622

[174] SAWADA, J. ; HUNT, W. A.: Trace Table Based Approach for Pipeline Microprocessor Verification. In: *Proc. of the 9th International Conference on Computer Aided Verification (CAV)*. London, UK : Springer, 1997, S. 364–375. – ISBN 3-540-63166-6

[175] SCHMITZ, R. ; SCHIFFMANN, W.: *Technische Informatik 1: Grundlagen der digitalen Elektronik*. Springer, 2004

[176] SCHUSTER, T. ; BRUNA, D. N. ; BOUGARD, B. ; DERUDDER, V. ; HOFFMANN, A. ; VAN DER PERRE, L.: Subword-parallel VLIW Architecture Exploration for Multimode Software Defined Radio. In: *Proc. of the IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS)*, 2007, S. 351–356. – ISBN 1424403839

[177] SENGUPTA, D. ; SALEH, R.: Power-Delay Metrics Revisited for 90nm CMOS Technology. In: *Proc. of the 6th International Symposium on Quality of Electronic Design*, 2005, S. 291–296

[178] SHEN, J. ; ABRAHAM, J. A.: An RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation. In: *Journal of Electronic Testing* 16 (2000), Nr. 1, S. 67–81

[179] SHIBA, H. ; SHONO, T. ; SHIRATO, Y. ; TOYODA, I. ; UEHARA, K. ; UMEHIRA, M.: Software Defined Radio Prototype for PHS and IEEE 802.11 Wireless LAN. In: *IEICE transactions on communications* 85 (2002), Nr. 12, S. 2694–2702

[180] SHONO, T. ; SHIRATO, Y. ; SHIBA, H. ; UEHARA, K. ; ARAKI, K. ; UMEHIRA, M.: IEEE 802.11 Wireless LAN Implemented on Software Defined Radio with Hybrid Programmable Architecture. In: *IEEE Transactions on Wireless Communications* 4 (2005), Nr. 5, S. 2299–2308

[181] SIMAR, R. ; TATGE, R.: How TI Adopted VLIW in Digital Signal Processors. In: *IEEE Solid-State Circuits Magazine* Summer 2009 (2009), Nr. 2, S. 10–14

[182] SOBOLEWSKI, J. S.: *Cyclic Redundancy Check*. S. 476–479. In: *Encyclopedia of Computer Science*, 2003

[183] STACKHOUSE, B. ; BHIMJI, S. ; BOSTAK, C. ; BRADLEY, D. ; CHERKAUER, B. ; DESAI, J. ; FRANCOM, E. ; GOWAN, M. ; GRONOWSKI, P. ; KRUEGER, D. ; MORGANTI, C. ; TROYER, S.: A 65 nm 2-Billion Transistor Quad-Core Itanium® Processor. In: *IEEE Journal of Solid-State Circuits* 44 (2009), Januar, Nr. 1, S. 18–31. – ISSN 0018-9200

[184] STALLINGS, W.: *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2010

[185] STINSON, J. ; RUSU, S.: A 1.5GHz Third Generation Itanium® 2 Processor. In: *Proc. of the 40th ACM/IEEE Design Automation Conference (DAC)*. New York, NY, USA : ACM, 2003, S. 706–709. – ISBN 1-58113-688-9

[186] STRUTZ, T.: *Bilddatenkompression. Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. Vieweg + Teubner, 2005

[187] SUGA, A. ; MATSUNAMI, K.: Introducing the FR500 Embedded Microprocessor. 20 (2000), Juli, Nr. 4, S. 21–27

[188] SUGA, Atsuhiro ; SATOSHI, Imai: FR-V Single-chip Multicore Processor : FR1000. In: *Fujitsu Scientific and Technical Journal* 42 (2006), April, Nr. 2, S. 190–199

[189] SUKEMURA, T.: FR500 VLIW-Architecture High-Performance Embedded Microprocessor. In: *Fujitsu Scientific and Technical Journal* 36 (2000), Juni, S. 31–38

[190] SYNOPSYS: DesignWare Intellectual Property. (2011), März. – URL https://www.synopsys.com/dw/doc.php/ds/o/product_overview.pdf

[191] SYNOPSYS: DW01_addsub Adder Subtractor. (2011), März. – URL {h}ttps://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw01_addsub.pdf

[192] SYNOPSYS: DW01_ash Arithmetic Shifters. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw01_ash.pdf

[193] SYNOPSYS: DW02_mac Multiplier-Accumulator. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw02_mac.pdf

[194] SYNOPSYS: DW02_mult Multiplier. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw02_mult.pdf

[195] SYNOPSYS: DW_div Combinational Divider. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw_div.pdf

[196] SYNOPSYS: DW_div_pipe Stallable Pipelined Divider. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw_div_pipe.pdf

[197] SYNOPSYS: DW_fifoctl_s1_sf Synchronous (Single-Clock) FIFO Controller with Static Flags. (2011), März. – URL https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw_fifoctl_s1_sf.pdf

[198] TANENBAUM, A. S. ; GOODMAN, J.: *Computerarchitektur*. Pearson Studium, 2001. – ISBN 308273-7016-7

[199] TENSILICA: Diamond Standard Processor Core Family Architecture. URL http://www.tensilica.com/products/literature-docs/white-papers/diamond-cores.htm, 2007. – Forschungsbericht

[200] TEXAS INSTRUMENTS: TMS320C54X, TMS320LC54X, TMS320VC54X Fixed-point Digital Signal Processors. In: *Literature Number: SPRS039C* (1999)

[201] TEXAS INSTRUMENTS: Code Composer Studio User's Guide. In: *Literature Number: SPRU328B* (2000), Februar

[202] TEXAS INSTRUMENTS: TMS320DM64x Power Consumption Summary. (2005), Februar, S. 1–4. – URL http://focus.ti.com/lit/an/spra962f/spra962f.pdf

[203] TEXAS INSTRUMENTS: TMS320DM642 Video/Imaging Fixed-Point Digital Signal Processor. (2010), S. 1–4. – URL http://focus.ti.com/lit/ds/symlink/tms320dm642.pdf

[204] THE EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM: – URL `http://www.eembc.org/`

[205] THIES, Michael: UPSLA (Unified Processor Specification Language) Language Description and Reference. (2001-2010)

[206] BRADLEY, J.: Advantages of Using the TMS320C6474 Over the TMS320C6455. (2008), Oktober. – URL `http://focus.ti.com/lit/wp/spraau9a/spraau9a.pdf`

[207] TEXAS INSTRUMENTS: TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. (2008). – URL `http://focus.ti.com/lit/ug/spru732j/spru732j.pdf`

[208] TEXAS INSTRUMENTS: TMS320C64x+ DSP Cache User's Guide. (2009). – URL `http://focus.ti.com/lit/ug/spru862b/spru862b.pdf`

[209] TRIMEDIA: TM5250 User Manual. (2005)

[210] UNGERBÖCK, G.: Trellis-coded Modulation with Redundant Signal Sets Part I: Introduction. In: *IEEE Communications Magazine* 25 (1987), Nr. 2, S. 5–11. – ISSN 0163-6804

[211] UNGERBÖCK, G.: Channel Coding with Multilevel/Phase Signals. In: *IEEE Transactions on Information Theory* 28 (2002), Nr. 1, S. 55–67. – ISSN 0018-9448

[212] UR, S. ; YADIN, Y.: Micro Architecture Coverage Directed Generation of Test Programs. In: *Proc. of the 36th ACM/IEEE Design Automation Conference (DAC)*, 1999, S. 175–180

[213] VAN DE WAERDT, J.-W. ; VASSILIADIS, S. ; DAS, S. ; MIROLO, S. ; YEN, C. ; ZHONG, B. ; BASTO, C. ; VAN ITEGEM, J.-P. ; AMIRTHARAJ, D. ; KALRA, K. ; RODRIGUEZ, P. ; VAN ANTWERPEN, H.: The TM3270 Media-Processor. In: *Proc. of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 331–342. – ISBN 0-7695-2440-0

[214] VAN DE WAERDT, J.-W. ; VASSILIADIS, S. ; VAN ITEGEM, J.-P. ; VAN ANTWERPEN, H.: The TM3270 Media-processor Data Cache. In: *Proc. of the IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, Oktober 2005, S. 334–341

[215] VAN ROMPAEY, K. ; BOLSENS, I. ; DE MAN, H. ; VERKEST, D.: CoWare – A Design Environment for Heterogenous Hardware/Software Systems. In: *Proc. of the Conference on European Design Automation* IEEE Computer Society Press (Veranst.), 1996, S. 252–257. – ISBN 081867573X

[216] VERMA, M. ; MARWEDEL, P.: *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer, Mai 2007

[217] VITERBI, A.: Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. In: *IEEE Transactions on Information Theory* 13 (2002), Nr. 2, S. 260–269

[218] WAINGOLD, E. ; TAYLOR, M. ; SRIKRISHNA, D. ; SARKAR, V. ; LEE, W. ; LEE, V. ; KIM, J. ; FRANK, M. ; FINCH, P. ; BARUA, R. ; BABB, J. ; AMARASINGHE, S. ; AGARWAL, A.: Baring it all to Software: Raw Machines. In: *Computer* 30 (1997), September, Nr. 9, S. 86–93. – ISSN 0018-9162

[219] WALL, D. W.: Limits of Instruction-level Parallelism. In: *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA : ACM, 1991, S. 176–188. – ISBN 0-89791-380-9

[220] WEICKER, R. P.: Dhrystone: A Synthetic Systems Programming Benchmark. In: *Communications of the ACM* 27 (1984), Nr. 10, S. 1013–1030

[221] WEISS, A. R.: Dhrystone Benchmark: History, Analysis, „Scores“ and Recommendations. In: *EEMBC White Paper* (2002)

[222] WESTE, N. H. E. ; ESHRAGHIAN, K.: *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 1985. – ISBN 0-201-08222-5

[223] WHITNEY, E. ; SPRAGUE, M.: Drag Your Design Environment Kicking and Screaming into the 90's with Modules! In: *Synopsys Users' Group* (2001), S. 1–18

[224] WITTGRUBER, F.: *Digitale Schnittstellen und Bussysteme*. Vieweg + Teubner, 2002

[225] WU, Q. ; PEDRAM, M. ; WU, X.: Clock-gating and its Application to Low Power Design of Sequential Circuits. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47 (2000), Nr. 3, S. 415–420

[226] XIE, Y. ; WOLF, W. ; LEKATSAS, H.: Code Compression for Embedded VLIW Processors Using Variable-to-fixed Coding. In: *IEEE Transactions on Very Large Scale Integration Systems* 14 (2006), Nr. 5, S. 525–536

[227] YIM, J.-S. ; HWANG, Y.-H. ; PARK, C.-J. ; CHOI, H. ; YANG, W.-S. ; OH, H.-S. ; PARK, I.-C. ; KYUNG, C.-M.: A C-based RTL Design Verification Methodology For Complex Microprocessor. In: *Proc. of the 34th ACM/IEEE Design Automation Conference (DAC)*, 1997, S. 83–88. – ISBN 0-7803-4093-0

[228] YU, A. Y. C.: Future of Microprocessors. In: *IEEE Micro* 16 (1996), Nr. 6, S. 46–53. – ISSN 0272-1732

[229] ZENDRA, O.: Memory and Compiler Optimizations for Low-power and -energy. In: *Proc. of the International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2006

[230] ZIPF, P.: Applying Dynamic Reconfiguration for Fault Tolerance in Fine-Grained Logic Arrays. In: *IEEE Transactions on Very Large Scale Integration Systems* 16 (2008), Nr. 2, S. 134–143. – ISSN 1063-8210

Eigene Publikationen

[231] DREESEN, R. ; JUNGEBLUT, T. ; THIES, M. ; KASTENS, U.: Dependence Analysis of VLIW Code for Non-Interlocked Pipelines. In: *Proc. of the 8th Workshop on Optimizations for DSP and Embedded Systems (ODES)*, April 2010, S. 21–28

[232] DREESEN, R. ; JUNGEBLUT, T. ; THIES, M. ; PORRMANN, M. ; RÜCKERT, U. ; KASTENS, U.: A Synchronization Method for Register Traces of Pipelined Processors. In: *Proc. of the International Embedded Systems Symposium 2009 (IESS '09)*, September 2009, S. 207–217

[233] JUNGEBLUT, T.: *Anwendungsspezifische Optimierung eingebetteter Systeme für Netzwerkanwendungen*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2005

[234] JUNGEBLUT, T. ; AX, J. ; SIEVERS, G. ; HÜBENER, B. ; PORRMANN, M. ; RÜCKERT, U.: Resource Efficiency of Scalable Processor Architectures for SDR-based Applications (Invited). In: *Proc. of the Radar, Communication and Measurement Conference (RADCOM)*, 2011

[235] JUNGEBLUT, T. ; DREESEN, R.: CoreVA Architectural Manual. (2011)

[236] JUNGEBLUT, T. ; DREESEN, R. ; PORRMANN, M. ; RÜCKERT, U. ; HACHMANN, U.: Design Space Exploration for Resource Efficient VLIW-Processors. In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE) – University Booth*, 2008

[237] JUNGEBLUT, T. ; DREESEN, R. ; PORRMANN, M. ; THIES, M. ; RÜCKERT, U. ; KASTENS, U.: „Netz der Zukunft" – MxMobile – Multi-Standard Mobile Plattform, Schlussbericht. (2009)

[238] JUNGEBLUT, T. ; DREESEN, R. ; PORRMANN, M. ; THIES, M. ; RÜCKERT, U. ; KASTENS, U.: A Framework for the Design Space Exploration of Software-Defined Radio Applications. In: *Mobile Lightweight Wireless Systems: 2nd International ICST Conference, Mobilight 2010* Bd. 45, Mai 2010, S. 148–159. – ISBN 3642166431

[239] JUNGEBLUT, T. ; GRÜNEWALD, M. ; PORRMANN, M. ; RÜCKERT, U.: Real-Time Multiprocessor SoC for Mobile Ad Hoc Networks. In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE) – University Booth*, 16. - 20. April 2007

[240] JUNGEBLUT, T. ; GRÜNEWALD, M. ; PORRMANN, M. ; RÜCKERT, U.: Realtime Multiprocessor for Mobile Ad Hoc Networks. In: *Advances in Radio Science* 6 (2008), S. 239–243

[241] JUNGEBLUT, T. ; KLASSEN, D. ; DREESEN, R. ; PORRMANN, M. ; THIES, M. ; RÜCKERT, U. ; KASTENS, U.: Design Space Exploration for Next Generation Wireless Technologies (Invited). In: *Proc. of the Electrical and Electronic Engineering for Communication Conference (EEEfCOM)*, 2009

[242] JUNGEBLUT, T. ; LISS, C. ; PORRMANN, M. ; RÜCKERT, U.: . Bd. 2. Kap. Design-space Exploration for Flexible WLAN Hardware. In: *Cross Layer Designs in WLAN Systems* Bd. 2, Troubador Publishing, 2011

[243] JUNGEBLUT, T. ; LÜTKEMEIER, S. ; SIEVERS, G. ; PORRMANN, M. ; RÜCKERT, U.: A Modular Design Flow for Very Large Design Space Explorations. In: *Proc. of the CDNLive! EMEA*, 2010

[244] JUNGEBLUT, T. ; PUTTMANN, C. ; DREESEN, R. ; PORRMANN, M. ; THIES, M. ; RÜCKERT, U. ; KASTENS, U.: Resource Efficiency of Hardware Extensions of a 4-issue VLIW Processor for Elliptic Curve Cryptography. In: *Advances in Radio Science* 8 (2010), Dezember, S. 295–305

[245] JUNGEBLUT, T. ; SIEVERS, G. ; PORRMANN, M. ; RÜCKERT, U.: Design Space Exploration for Memory Subsystems of VLIW Architectures. In: *Proc. of the 5th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, Juli 2010, S. 377–385

[246] JUNGEBLUT, T. ; THIES, M. ; KLASSEN, D. ; PORRMANN, M. ; RÜCKERT, U. ; KASTENS, U.: „Enablers for Ambient Services and Systems“ – Easy-C – Wide Area Coverage, Schlussbericht. (2010)

Betreute Arbeiten

[247] AX, J.: *Optimierung der Anbindung von Hardwareerweiterungen an einen VLIW-Prozessor*, Fachgruppe Schaltungstechnik, Universität Paderborn, Studienarbeit, 2010

[248] AX, J.: *Implementierung und Analyse eines Multiprozessorsystems auf Basis der CoreVA-VLIW-Architektur*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2011

[249] EBEL, I.: *Entwurfsraumexploration an einem ressourceneffizienten VLIW-Prozessor*, Fachgruppe Schaltungstechnik, Universität Paderborn, Studienarbeit, 2010

[250] HÜBENER, B.: *Optimierung eines 802.11b Algorithmus für einen VLIW-Prozessor*, Fachgruppe Schaltungstechnik, Universität Paderborn, Studienarbeit, 2009

[251] HÜBENER, B.: *Analyse und Optimierung der Forwarding-Architekturen eines eingebetteten VLIW-Prozessors*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2010

[252] KÄNNER, M.: *Evaluation of the Next Generation Digital Interface Technology*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2006

[253] SIEVERS, G.: *Implementierung eines Cache-Controllers für einen eingebetteten VLIW-Prozessor*, Fachgruppe Schaltungstechnik, Universität Paderborn, Diplomarbeit, 2009

Anhang A. Instruktionssatz des CoreVA-Prozessors

Dieses Kapitel beschreibt den Instruktionssatz des in Kapitel 4 vorgestellten CoreVA-Prozessors. Die Instruktionen unterteilen sich in fünfzehn Instruktionsgruppen. Die Instruktionsgruppen und der Struktur ihrer Opcodes sind in Tabelle A.1 aufgeführt. Tabelle A.2 zeigt die Instruktionsgruppen der SIMD-Erweiterungen des Instruktionssatzes.

Tabelle A.1.: *Die Instruktionsgruppen des CoreVA-Prozessors und deren Opcode-Strukturen*

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing (i)	st	Cond			0	0	0	0	0	Opcode					Rn				Rd						(u/s)	Immediate						
Data processing (i-c)	st	Cond			0	0	0	cm	1	Opcode					Rn				cm2	Cd					(u/s)	Immediate						
Data processing (r)	st	Cond			0	0	1	0	0	Opcode					Rn				Rd						0	0	0	Rm				
Data processing (r-c)	st	Cond			0	0	1	cm	1	Opcode					Rn				cm2	Cd					0	0	0	Rm				
Condition reg.logic(i)	st	Cond			0	0	0	1	1	0	1	1	1	1	0	0	Cn		0	0	Cd				0	opc		0	0	Cm		
Condition reg.logic(r)	st	Cond			0	0	1	1	1	0	1	1	1	1	0	0	Cn		0	0	Cd				0	opc		0	uimm			
Multiply-accumulate	st	Cond			0	0	1	0	0	Rs					Rn				Rd						1	S[6]	A[1]	Rm				
Store imm offset (13bit)[2] (i13)	st	Cond			0	1	0	Sz		Signed immH					Rn				Rd						Signed ImmL							
Load imm offset (13bit)[2] (i13)	st	Cond			0	1	1	1	0	Signed immH					Rn				Rd						Signed ImmL							
Load/Store, imm offset (8bit) (i8)	st	Cond			0	1	1	0	0	L[3]	Sz		AM		Rn				Rd						Signed Imm							
Load/Store, reg offset (reg)	st	Cond			0	1	1	0	1	L[3]	Sz		AM		Rn				Rd						Sc	S[4]	Rm					
Swap	st	Cond			0	1	1	0	1	1	0	0	AM		Rn				Rd						1	0	0	Rm				
Branch rel offset	st	Cond			0	1	1	1	1	L[5]	Offset 22bit																					
Branch reg.	st	Cond			0	0	1	0	0	1	1	1	1	1	0				0						0	0	L[5]	Rm				
Extension instructions	st	Cond			1	x	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

1) A=0: Multiply, A=1: Multiply and add
2) 12Bit offset Ld/St instruction without pre/post-indexed modes
3) L=0: Store, L=1: Load
4) Sign of Rm
5) L=1: Branch and link
6) Multiply signed

Tabelle A.2.: *Die SIMD-Instruktionsgruppen des CoreVA-Prozessors und deren Opcode-Strukturen*

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
□□AC□	st	Cond			1	1	0	0	0	Rs				Rn					Rd						0	Sld	Sls				Sln	
□ector multiply	st	Cond			1	0	1	opc		Rs				Rn					Rd						1	S	A	Rm				
□ector data processing (i)	st	Cond			1	0	0	0	0	Opcode				Rn					Rd						(u/s) Immediate							
□ector data processing (i-c)	st	Cond			1	0	0	cm1		Opcode				Rn					cm2		Cd				(u/s) Immediate							
□ector data processing (r)	st	Cond			1	0	1	0	0	Opcode				Rn					Rd						0	0	0	Rm				
□ector data processing (r-c)	st	Cond			1	0	1	cm1		Opcode				Rn					cm2		Cd				0	0	0	Rm				

Tabelle A.3 zeigt eine Übersicht über die Instruktionen und ihre Bedeutung. Die zusätzlichen Instruktionen der SIMD-Erweiterung sind in Tabelle A.4 aufgeführt. Detailliertere Informationen und eine vollständige syntaktische und semantische Beschreibung finden sich im Architekturhandbuch des CoreVA-Prozessors [235].

Tabelle A.3.: *Instruktionen des CoreVA-Prozessors*

Instruktion	Bedeutung
ABS	Absolutwertberechnung
ADC	Addition mit einem zusätzlichen Übertragsbit (z. B. für 64-Bit-Additionen)
ADD	Addition
AND	Bitweise logische Und-Verknüpfung
ANDN	Bitweise logische Und-Verknüpfung und Negation
ASR	Arithmetisches Rechtsschieben
BR	Absolut oder relativ Springen
BRL	Springen und Rücksprungadresse sichern
CLZ	Bestimmt führende Nullen des Operanden
CMP	Vergleich
CAN	Logische Und-Verknüpfung auf Condition-Registern
CEO	Logische Exklusiv-Oder-Verknüpfung auf Condition-Registern
COR	Logische Oder-Verknüpfung auf Condition-Registern
DEC	Dekrementieren und Vergleichen
DEC4	Dekrementieren und Vergleichen (Für vierfach abgerollte Schleifen)
DVI	Divisionsschrittberechnung initialisieren
DVS	Divisionsschrittberechnung
DVQ	Quotient der Divisionsschrittberechnung auslesen
DVR	Rest der Divisionsschrittberechnung auslesen
EOR	Bitweise Exklusiv-Oder-Verknüpfung
LDW	Wort aus Speicher laden
LSL	Logisches Linksschieben
LSR	Logisches Rechtsschieben
MLA{S}	Multiplizieren und Akkumulieren
MCR	Inhalte von Condition-Register in normale Register kopieren
MOV	Inhalte zwischen normalen Registern kopieren
MCR	Inhalte von normalen Registern in Condition-Register kopieren
MVA{L,H}	Registerinhalte kopieren und an 16-Bit-Grenzen ausrichten
MVB	Kopieren eines Bytes innerhalb der normalen Register
MVC	Laden einer Konstante
MVH	Kopieren eines Halbwortes innerhalb der normalen Register
MVSB	Kopieren eines Bytes (Zweierkomplement)
MVSH	Kopieren eines Halbwortes (Zweierkomplement)
OR	Bitweise Oder-Verknüpfung
RSB	Subtraktion mit vertauschten Operanden
RSC	Subtraktion mit vertauschten Operanden (mit zusätzlichem Übertragsbit)
SBC	Subtraktion mit zusätzlichem Übertragsbit (z. B. für 64-Bit-Subtraktionen)
STB	Byte in Speicher schreiben
STH	Halbwort in Speicher schreiben
STW	Wort in Speicher schreiben
SUB	Subtraktion

Tabelle A.4.: *SIMD-Instruktionen des CoreVA-Prozessors*

Instruktion	Bedeutung
VABS	Absolutwertberechnung
VADD	Addition
VML{A,S}{S}	Multiplizieren und Akkumulieren/Subtrahieren
VSML{A,S}{S}	Multiplizieren und Akkumulieren/Subtrahieren mit vertauschten Operanden
VSAS	Halbwörter vertauschen und Addieren/Subtrahieren
VSSA	Halbwörter vertauschen und Subtrahieren/Addieren
VASR	Arithmetisches Rechtsschieben
VCMP	Vergleich
VLSL	Logisches Linksschieben
VLSR	Logisches Rechtsschieben
VPACK	Konvertieren von einem Skalar in einen SIMD-Vektor und umgekehrt
VRSB	Subtraktion mit vertauschten Operanden
VSUB	Subtraktion

Anhang B. Generische Parameter zur Konfiguration des CoreVA-Prozessors

```
package slot_configuration is

  constant CONST_NUM_OF_SLOTS        : natural :=  4;  -- Number of slots
  constant CONST_NUM_OF_MUL_UNITS    : natural :=  2;  -- Number of multiply units
  constant CONST_NUM_OF_DIV_UNITS    : natural :=  2;  -- Number of division units
  constant CONST_NUM_OF_IMEM_SLOTS   : natural :=  4;  -- Number of load / store units
  constant CONST_NUM_OF_LDST_UNITS   : natural :=  2;  -- Number of load / store units
```

Abbildung B.1.: Konfiguration der Anzahl der einzelnen Module (Auszug aus slot_configuration.vhd)

Die Anzahl der Funktionseinheiten des CoreVA-Prozessors und deren Ressourcenverfügbarkeit innerhalb der VLIW-Slots kann über Konstanten in der Konfigurationsdatei `slot_configuration.vhd` festgelegt werden. Folgende Parameter können definiert werden:

- `CONST_NUM_OF_SLOTS`
  Anzahl der Ausführungskanäle. Jedem Ausführungskanal wird automatisch eine ALU zugewiesen.
- `CONST_NUM_OF_MUL_UNITS`
  Anzahl der Multiplikationseinheiten.
- `CONST_NUM_OF_DIV_UNITS`
  Anzahl der Divisionsschritteinheiten.
- `CONST_NUM_OF_LDST_UNITS`
  Anzahl der Kanäle zum Datenspeicher.
- `CONST_NUM_OF_IMEM_SLOTS`
  Breite der Schnittstelle zum Instruktionsspeicher (Anzahl der 32-Bit-Worte, vgl. Abschnitt 4.3.4).

Die Zuweisung der Multiplizierer, Dividierer und Datenspeicherverwaltungseinheiten zu Ausführungskanälen erfolgt mithilfe von Matrizen. Für jede Einheit wird

```
type array_config_mul   is array      (0 to CONST_NUM_OF_MUL_UNITS  -1)
                        of bit_vector (0 to CONST_NUM_OF_SLOTS      -1);

type array_config_div   is array      (0 to CONST_NUM_OF_DIV_UNITS  -1)
                        of bit_vector (0 to CONST_NUM_OF_SLOTS      -1);

type array_config_ldstr is array      (0 to CONST_NUM_OF_LDST_UNITS -1)
                        of bit_vector (0 to CONST_NUM_OF_SLOTS      -1);
```

Abbildung B.2.: Definition der Zuweisungsmatrizen

festgelegt, mit welchem Ausführungskanal sie verbunden ist. Die Verwendung solcher Matrizen hat den Vorteil, dass zum einen alle Freiheitsgrade für die Zuweisung an die Ausführungskanäle frei genutzt werden können, zum anderen eine solche Matrix relativ einfach direkt im VHDL-Programmquelltext verwendet werden kann. Abbildung B.2 zeigt die Definition dieser Matrizen. Eine Zuweisung erfolgt nach dem folgenden Schema:

$$\text{Konfigurationsmatrix} := (\text{Einheit 1}, \text{Einheit 2}, \ldots, \text{Einheit n}) \tag{B.1}$$

Eine Einheit kann hierbei vom Typ Multiplizierer, Dividierer oder Datenspeicherzugriff sein. n kennzeichnet die Anzahl der zur Verfügung stehenden Einheiten. Für jeden Einheitentyp wird jeweils eine eigene Matrix verwendet:

- `ASSIGN_MUL_TO_SLOT`
  Konfigurationsmatrix für Multiplizierer
- `ASSIGN_DIV_TO_SLOT`
  Konfigurationsmatrix für Dividierer
- `ASSIGN_LDSTR_TO_SLOT`
  Konfigurationsmatrix für Datenspeicherzugriff

Die Verbindung einer Einheit zu einem Ausführungskanal wird mithilfe einer weiteren Matrix in Form von einzelnen Bits dargestellt. Jede Zeile innerhalb eine Matrix bezieht sich auf eine Einheit $(1, \ldots, n)$. Jedes Bit innerhalb der Zeile definiert, mit welchem Ausführungskanal $(0, \ldots, m-1)$ diese Einheit verbunden ist:

$$\text{Einheit} := \text{Bit 0}, \text{Bit 1}, \text{Bit 2}, \ldots, \text{Bit m-1} \tag{B.2}$$

Ein Konfigurationsbit definiert, ob ein Ausführungskanal mit dieser Einheit verbunden ist. Der Wert eines solchen Konfigurationsbits bedeutet:

- 0 - Ausführungskanal nicht verbunden
- 1 - Ausführungskanal verbunden

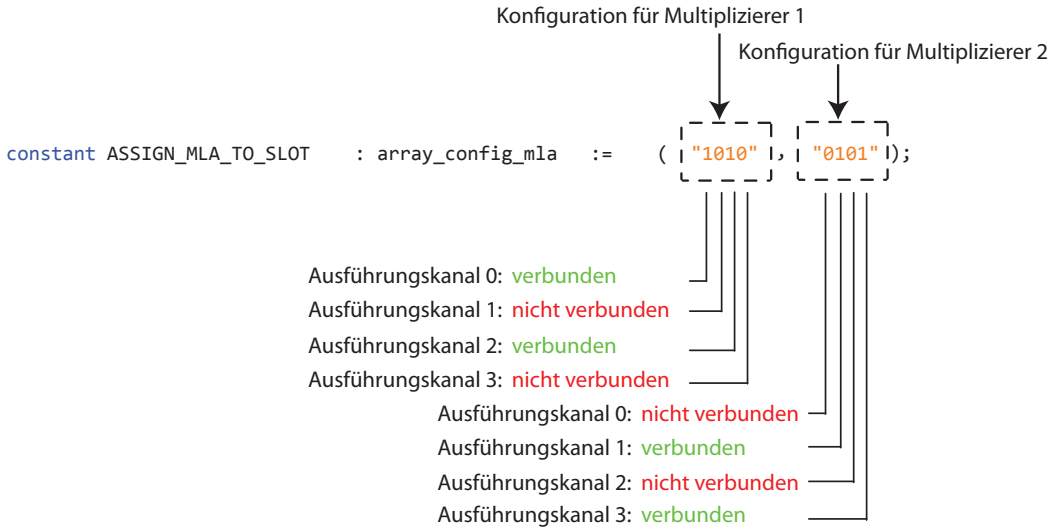


Abbildung B.3.: Beispielzuweisung für 2 Multiplizierer an 4 Ausführungskanäle

Das erste Bit steht für den ersten Ausführungskanal (Kanalnummer 0), dass $m-1$-te Bit für den Letzten. m bezeichnet die Anzahl aller Ausführungskanäle. Anhand von zwei Beispielen soll die Zuweisung an die Ausführungskanäle im Folgenden verdeutlicht werden. Im Beispiel aus Abbildung B.3 ist eine mögliche Konfiguration für zwei Multiplizierer und vier Ausführungskanäle dargestellt. Der erste Multiplizierer ist hier mit den Ausführungskanälen 0 und 2 verbunden. Der zweite Multiplizierer mit den Ausführungskanälen 1 und 3. Dieser Fall entspricht der bisherigen Konfiguration für die Multiplizierer. Das Beispiel aus Abbildung B.4 behandelt eine Zuweisung von zwei Dividierern an drei Ausführungskanäle. Hier ist der erste Dividierer mit dem Ausführungskanal 0 und 2 verbunden. Ausführungskanal 1 belegt die zweite Divisionsschritteinheit exklusiv.

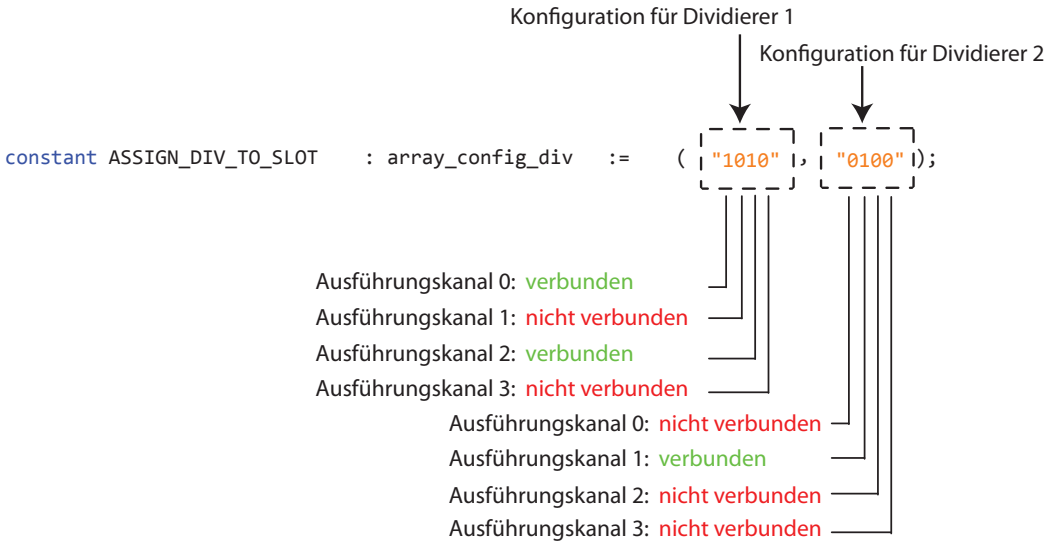


Abbildung B.4.: Beispielzuweisung für 2 Dividierer an 4 Ausführungskanäle

Anhang C. Generische Parameter zur Konfiguration des Pipeline-Bypasses

Für die Konfiguration des Pipeline-Bypasses kommen – ähnlich wie bei der Zuweisung von Funktionseinheiten zu Ausführungskanälen – Matritzen zum Einsatz. Dieses Kapitel beschreibt die statische und dynamische Konfiguration des Pipeline-Bypasses.

C.1. Statische Konfiguration des Pipeline-Bypasses

Die Zeilen der Konfigurationsmatrizen werden durch Arrays beschrieben, welche jeweils die Zugriffe auf eine der unteren Pipelinestufen steuern. Die Zuordnung zu den einzelnen Zwischenergebnissen dieser Pipelinestufen erfolgt durch die Elemente der Arrays, die somit die Spalten der Matrix bilden. Durch die einzelnen Bits der Array-Elemente wird festgelegt, welche Bypass-Instanz auf das zugehörige Zwischenergebnis zugreifen kann. Somit wird für jedes Bit, das in der Konfiguration auf 1 gesetzt wurde, beim Synthetisieren ein Bypass-Pfad implementiert. Um eine möglichst effiziente Dekodierung der Konfigurationsmatrizen zu ermöglichen, beschreibt das niederwertigste Array-Element die Konfiguration der Zugriffe auf den höchstwertigsten Slot. Das niederwertigste Bit eines Elements beschreibt jedoch die niederwertigste Bypass-Instanz.

Die Abbildung C.1 zeigt die Konfigurationsmatrix des Kontroll-Bypasses und die hierdurch implementierten Bypass-Pfade. In dieser Beispielskonfiguration kann die dritte Kontroll-Bypass-Instanz auf alle Zwischenergebnisse der Execute-Pipelinestufe zugreifen. Des Weiteren werden Pfade von sämtlichen Kontroll-Bypass-Instanzen auf den Wert des zweiten Slots der Memory-Access-Pipelinestufe (ME Slot 1) hinzugefügt.

Durch den zuvor beschriebenen Aufbau der Konfigurationsmatrizen können diese effizient mit der generisch konfigurierbaren Anzahl der Verarbeitungseinheiten skalieren. Würden die Matrizen wie in der Darstellung der implementierten Bypass-Pfade aufgebaut, müssten bei steigender Slot-Anzahl zusätzliche Konfigurationsarrays hinzugefügt werden. Bei der gewählten Matrizenanordnung müssen in diesem Fall lediglich zusätzliche Elemente in die bestehenden Arrays eingefügt werden. Hierzu werden die beiden Signaltypen `bypass_slot_enable_array` und `bypass_slot_ldst_enable_array` verwendet, die automatisch ein Variablen-

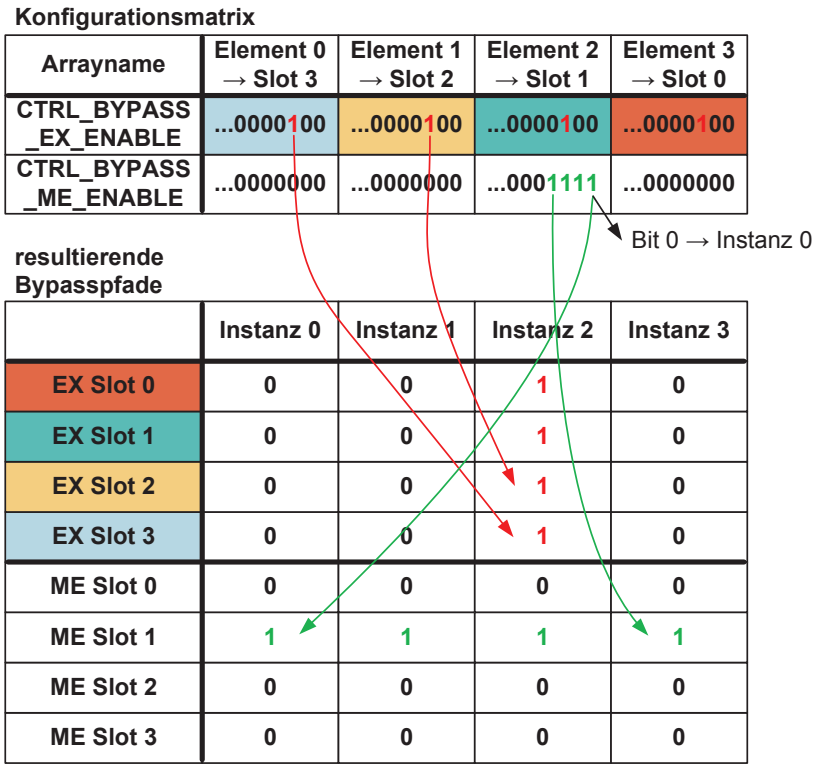


Abbildung C.1.: Konfigurationsmatrix und implementierte Bypass-Pfade

bündel mit der benötigten Elementanzahl erstellen. Da die Anzahl der Bypass-Instanzen ebenfalls von der Anzahl der Slots und der Multiplizierereinheiten abhängig ist, müssen die einzelnen Array-Elemente eine ausreichende Wortbreite besitzen. Aus diesem Grund wurde eine einheitliche Breite von 32 Bit gewählt, wodurch alle praktisch relevanten Konfigurationsmöglichkeiten abgedeckt werden können. Diese Elementbreite reicht beispielsweise aus, um die insgesamt 30 Instanzen des Daten-Register-Bypasses eines CoreVA-Prozessors mit zwölf Verarbeitungseinheiten und sechs Multiplizierereinheiten zu konfigurieren. Da generische Variablen während des Kompilierens entfernt werden, hat eine großzügig gewählte Wortbreite keine Auswirkung auf die Fläche des Prozessors.

Die Konfigurationen der verschiedenen Bypass-Systeme werden in der VHDL-Datei `bypass_configuration.vhd` beschrieben und als Bibliothek in die übergeordnete Datei `gpvliw_hwacc.vhd` eingebunden. Die obige Beispielkonfiguration des Kontroll-Bypasses wird hierbei durch folgende Zeilen realisiert.

```
CTRL_BYPASS_EX_ENABLE : bypass_slot_enable_array := (4,4,4,4);
CTRL_BYPASS_ME_ENABLE : bypass_slot_enable_array := (0,0,15,0);
```

Die Konfigurationsarrays werden von der übergeordneten Verhaltensbeschreibung über die jeweiligen Schnittstellenbeschreibungen (Entity) beziehungsweise Instanziierungen (Portmap) der einzelnen Komponenten bis in die zugehörigen Bypass-Instanzen weitergeleitet. Zur Konfiguration aller Bypass-Komponenten werden hierbei zehn Arrays mit einer Gesamtbreite von 1472 Bit genutzt (vgl. Tabelle C.1). Da die Bypass-Instanzen durch die Modularisierung in kleine eigenständige Komponenten aufgeteilt wurden, muss ihnen zusätzlich ihre Instanznummer mitgeteilt werden. Diese Nummer wird in der zentralen Datei `gpvliw.vhd` bestimmt, da in dieser Datei die verschiedenen Instanzen zusammengeführt werden. Zur Minimierung der Dekodierungslogik wird jedoch nicht der Zahlenwert der Instanznummer, sondern ihre Zweierpotenz übergeben (`inst_nr => 2**i`). Hierdurch werden die einzelnen Bypass-Instanzen, wie bei der Zuordnung der Bits der Konfigurationselemente, über die Stelle des gesetzten Bits gekennzeichnet (One-Hot-Kodierung).

C.2. Dynamische Konfiguration des Pipeline-Bypasses

Das Übertragen der Konfigurationsänderungen erfolgt durch spezielle Befehle, die in den C-Programmcode der verschiedenen Anwendungen eingefügt werden. Da der C-Compiler diese Befehle nicht durch den bestehenden CoreVA-Instruktionssatz abbilden kann, wurde durch eine Instruktionssatzerweiterung die Spezialinstruktion `BYPASS_ CONFIGURATION` in den CoreVA-Prozessor hinzugefügt. Die Spezialin-

Tabelle C.1.: *Verwendete generische Arrays*

Name des Arrays	**Anzahl der Elemente**
CTRL_BYPASS_EX_ENABLE	4
CTRL_BYPASS_ME_ENABLE	4
BYPASS_REG_EX_ENABLE	6
BYPASS_REG_ME_ENABLE	6
BYPASS_REG_WR_ENABLE	6
BYPASS_COND_EX_ENABLE	4
BYPASS_COND_ME_ENABLE	4
BYPASS_READ_ALL_COND_EX_ENABLE	4
BYPASS_READ_ALL_COND_ME_ENABLE	4
MLA_BYPASS_WR_ENABLE	4
Gesamtbreite (32 Bit pro Element)	**1472**

struktion weist große Parallelen zur Instruktion `MVC`[1] auf, welche in Kombination mit der Extensions-Instruktion `EXT` eine 32 Bit große Konstante in ein Daten-Register des CoreVA-Prozessors schreiben kann. Zur Kennzeichnung des neuen Instruktionstyps wird die noch nicht belegte Bitfolge `11100` verwendet (siehe Tabelle C.2).

Damit diese Spezialinstruktion jedoch nicht in den Instruktionssatz des C-Compilers eingefügt werden muss, wird sie innerhalb des Programmcodes durch die intrinsische Funktion `_machword64()` dargestellt. Intrinsic-Funktionen bieten die Möglichkeit, prozessorspezifische Operationen, die der C-Compiler nicht unterstützt, in einen Funktionsaufruf einzukapseln. Zur Konfiguration der Bypass-Pfade wird das Instruktionswort der jeweiligen Spezialinstruktion manuell bestimmt und in hexadezimaler Schreibweise in das Argument der Intrinsic-Funktion eingetragen. Im Vergleich zu Inline-Assembler-Realisierungen, bei denen ein Assembler-Quellcode direkt in das C-Programm eingebunden wird, kann der Compiler Intrinsic-Funktionen in vollem Umfang bei seinen Optimierungs- beziehungsweise Parallelisierungsstrategien berücksichtigen.

Um die Spezialinstruktionen effizient verarbeiten zu können und um den Logikaufwand zur Adressierung einzelner Bypass-Pfade zu minimieren, werden die Pfade zum Deaktivieren nicht direkt adressiert, sondern in Gruppen zusammengefasst. In Anlehnung an den im Kapitel C beschriebenen Aufbau der generischen Variablen, wird hierzu das zugehörige 32-Bit-Element der Konfigurationsmatrizen angepasst und vollständig neu übertragen. Neben diesem Konfigurationselement müssen jeweils vier weitere Bits zur Adressierung des Bypass-Typen und des Slots übergeben

[1] Move Constant

werden (siehe Tabelle C.3). Aus diesem Grund muss die Spezialinstruktion insgesamt 40 Bit Nutzdaten aufnehmen können. Da diese Datenbreite die Breite eines Instruktionsworts übersteigt, muss diese Instruktion ebenfalls die bestehende Extensions-Instruktion einbinden um die Konfigurations- und Adressierungsbits auf zwei Instruktionswörter zu verteilen. Somit benötigt das System zur Rekonfiguration aller 46 Konfigurationselemente mindestens 23 Prozessortakte.

Der Aufbau dieser Instruktionswörter wird im Folgenden anhand einer Konfigurationsänderung des Daten-Register-Bypasses verdeutlicht (siehe Tabelle C.2). In diesem Beispiel wird den Bypass-Instanzen der Verarbeitungseinheit in Slot 1 durch die Bypass-Konfiguration `0x00000FC7` nachträglich der Zugriff auf die Zwischenergebnisse der Execute-Pipelinestufe in Slot 2 verwehrt.

Die Instruktion `bypass_config` besitzt an der höchstwertigsten Bitstelle ein Stop-Bit. Dieses Bit kennzeichnet, ob durch eine Instruktion die umschließende Instruktionsgruppe beendet wird. Da auf diese Instruktion jedoch in jedem Fall eine Extensions-Instruktion folgt, ist dieses Stop-Bit statisch auf '0' gesetzt. Die folgenden drei Bits beinhalten die Nummer des zugeordneten Condition-Registers. Da diese Instruktion nicht bedingt aufführbar sein soll, wird an dieser Stelle immer auf das Condition-Register C7 verwiesen. Es folgen die Instruktionskennungsbits `11100` und sieben Reservebits, die beispielsweise genutzt werden können, um in späteren Implementierungen weitere Informationen zu übertragen oder die Instruktionskennung zu erweitern. Wie in Tabelle C.3 ersichtlich, wird in den anschließenden Bits der Bypass-Typ auf „Daten-Register-Bypass EX“ und der Slot auf „Slot 1“ gesetzt. Die verbleibenden Bits werden zur Übertragung der oberen acht Bits der neuen Bypass-Konfiguration verwendet. Im Unterschied zur zugrunde liegenden MVC-Instruktion, beinhaltet dieses Instruktionswort keine Informationen über die Ziel-Register-Adresse.

Die Extensions-Instruktion beginnt ebenfalls mit einem Stop-Bit. Dieses Bit wird zwar mit '0' vorbelegt, kann jedoch vom C-Compiler während der Verarbeitung verändert werden. Da die Extensions-Instruktion direkt mit der vorherigen Instruktion verbunden ist, besitzt sie keine Zuordnung zu einem Condition-Register. Aus diesem Grund folgen auf das Stop-Bit direkt die Instruktionstypenkennung und die unteren 24 Bits der Bypass-Konfiguration.

Da die Anordnung der einzelnen Instruktionswörter innerhalb einer Instruktionsgruppe von rechts nach links erfolgt und die Extensions-Instruktion immer direkt auf die Spezialinstruktion folgen muss, wird das obige Beispiel wie folgt in die Intrinsic-Funktion eingebunden.

Tabelle C.2.: *Instruktionswörter zur Beispielkonfiguration des Daten-Register-Bypasses*

Bit-Stelle	31	30	27	15	11	7
bypass_config	**0**	**111**	**11100xxxxxxx**	**0010**	**0001**	**00000000**
Instruktion	Stop	Cond	Instruktionstyp	Bypass	Slot	Konfiguration
Bit-Stelle	31	30		23		
EXT	**0**	**111xxxx**		**000000000000111111000111**		
Instruktion	Stop	Instruktionstyp		Konfiguration		

Tabelle C.3.: *Adressierung der Konfigurationselemente*

Bypass-Typ	**Adressbits**
Kontroll-Bypass EX	0000
Kontroll-Bypass ME	0001
Daten-Register-Bypass EX	0010
Daten-Register-Bypass ME	0011
Daten-Register-Bypass WR	0100
Condition-Register-Bypass EX	0101
Condition-Register-Bypass ME	0110
Read-all-Cond-Bypass EX	0111
Read-all-Cond-Bypass ME	1000
MLA-Bypass EX	1001

Slot	**Adressbits**
Slot 0	0000
Slot 1	0001
Slot 2	0010
Slot 3	0011
....	....

```
void _machword64(unsigned int high, unsigned int low);
void _main;
{
  ...
  _machword64(0x70000FC7,0x7E002100); //reg_ex_slot1 = FC7
  ...
}
```

Der Ausschnitt aus dem anschließend generierten Speicherabbild zeigt, dass die Funktionsattribute direkt übernommen werden und dass das Stop-Bit der Extensions-Instruktion nachträglich auf '1' gesetzt wurde um die Instruktionsgruppe abzuschließen:

```
... f00000fc7e002100 ...
```

Abbildungsverzeichnis

Tabellenverzeichnis