Adronis Niyonkuru

Zum Einsatz von rekonfigurierbarer Hardware in Prozessorarchitekturen

Dissertation Technische Informatik

Zum Einsatz von rekonfigurierbarer Hardware in Prozessorarchitekturen

Vom Fachbereich Elektrotechnik

der Helmut-Schmidt-Universität/Universität der Bundeswehr Hamburg

zur Erlangung des akademischen Grades einer Doktor-Ingenieurin/eines Doktor-Ingenieurs

genehmigte

Dissertation

vorgelegt von

Dipl.-Inform. Adronis Niyonkuru

aus Burambi, Burundi

Hamburg 2005

Schlüsselwörter:
Rekonfigurierbare Hardware, Prozessorarchitektur, Mikroarchitektur, partielle und dynamische Rekonfiguration, FPGA

Gutachter:	Prof. Dr.-Ing. H. Ch. Zeidler PD Dr. phil. nat. B. Klauer
Vorsitzender der Prüfungskommission:	Prof. Dr.-Ing. H. Göbel
Mündliche Prüfung:	24. Juni 2005

Gedruckt mit Unterstützung der Helmut-Schmidt-Universität/
Universität der Bundeswehr Hamburg.

Kurzfassung

Konventionelle fest verdrahtete Hardware zeichnet sich durch ein hohes Maß an Flexibilität aus, indem sie anhand von Anwendungssoftware eine Vielfalt von Problemlösungen ermöglicht. Trotz ihrer stetig steigenden Leistungsfähigkeit genügt sie für zeitkritische Anwendungen jedoch nicht immer den gestellten Anforderungen. In solchen Fällen werden meistens sog. *Application Specific Integrated Circuits* (ASICs) verwendet. Dieser Art von Hardware fehlt jedoch die Flexibilität, für unterschiedliche Anwendungen eingesetzt werden zu können. *Field-Programmable-Gate Arrays* (FPGAs) vereinigen die Vorteile beider Hardware-Plattformen: Leistungsfähigkeit und Flexibilität. Dennoch bleiben aufgrund des bisherigen damit verbundenen hohen Hardware- und Software-Aufwands (z.B. Hardware/Software-Partitionierung) der Anwenderkreis und die Anwendungsgebiete sehr beschränkt.

Die vorliegende Arbeit untersucht einen evolutionären Ansatz, in wieweit die Vorzüge rekonfigurierbarer Hardware (FPGA) in Prozessorarchitekturen eingesetzt werden können. Im Gegensatz zu den bisherigen Ansätzen sollten dabei die Kompatibilität mit herkömmlichen Rechensystemen gewährleistet sein und bestehende Hardware- und Software-Werkzeuge weiterhin verwendbar bleiben. Zu diesem Zweck wurde im Rahmen dieser Arbeit das Modell einer rekonfigurierbaren Mikroarchitektur entwickelt. Anhand von Software-Simulationen wurden unterschiedliche Möglichkeiten der Hardware-Rekonfiguration auf ihr Leistungspotenzial hin überprüft. Daraufhin wurde mit Blick auf die hohe Komplexität eines modernen Prozessors einerseits und die Einschränkungen heutiger Entwurfswerkzeuge andererseits ein realitätsnahes Modell einer partiell und dynamisch rekonfigurierbaren Mikroarchitektur vorgezogen und auf einem FPGA implementiert. Dabei wurden vordefinierte Hardware-Konfigurationen während der Programmausführung und in Abhängigkeit der anwendungsspezifischen Hardware-Anforderungen ausgetauscht. Die Mikroarchitektur implementiert den ARM-*Thumb*-Befehlssatz anhand einer fünfstufigen superskalaren Pipeline. Die erzielten Ergebnisse ermutigen zur Weiterführung des entwickelten Konzeptes, das bereits durch eine auf dieser Arbeit basierende Weiterentwicklung bestätigt wurde.

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter der Helmut-Schmidt-Universität/Universität der Bundeswehr Hamburg an der Professur für Technische Informatik im Fachbereich Elektrotechnik.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. H. Ch. Zeidler für die Betreuung und Begleitung der Arbeit. Ohne seine zahlreichen Anregungen sowie den von ihm geschaffenen, gestalterischen Freiraum wäre diese Arbeit nicht realisierbar gewesen.

Herrn PD. Dr. phil. nat. B. Klauer danke ich für die Übernahme des Korreferats und seine konstruktiven Hinweise zur schriftlichen Arbeit. Herrn Prof. Dr.-Ing. H. Göbel danke ich für die Übernahme des Vorsitzes der Prüfungskommission.

Für die unzähligen fachlichen und konstruktiven Diskussionen, die den wissenschaftlichen Fortgang maßgeblich förderten, möchte ich ganz herzlich Herrn Dr. K. Köllmann danken. Ferner gilt mein Dank all denjenigen, die zum Gelingen meiner Arbeit beigetragen haben: Hervorzuheben sind vor allen Dingen alle Kollegen und Studenten, die nicht nur durch ihre Arbeiten und die vielen kleinen alltäglichen Problemlösungen halfen und ein angenehmes Arbeitsklima schafften.

Stellvertretend für alle Studenten, die mit ihrem Fleiß und Engagement meine Arbeit unterstützt haben, danke ich Herrn Dipl.-Ing. C. Bieschke, der als Student und kurzzeitig später als Arbeitskollege maßgeblich an der Hardware-Entwicklung mitgewirkt hat.

Hamburg, im Juni 2005 Adronis Niyonkuru

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 3

1.2 Zielsetzung 7

1.3 Aufbau der Arbeit 8

2 Grundlagen **10**

2.1 Programmierbare Logikbausteine 10

2.1.1 Vom Speicher zum FPGA 11

2.1.2 FPGA-Struktur 14

2.2 Rekonfigurierbare Rechensysteme 18

2.2.1 Granularität 18

2.2.2 Art der Kopplung 19

2.2.3 Rekonfigurierbarkeit 21

2.2.4 Programmierbarkeit 22

2.3 Entwurfsmethodik FPGA-basierter Rechensysteme 24

2.3.1 Entwurfseingabe 29

2.3.2 Funktionale Simulation 30

2.3.3 Synthese 30

2.3.4 Place&Route 31

2.3.5 Baustein-Programmierung 32

2.4 Mikroprozessortechnik 33

2.4.1 Prozessorarchitektur 33
2.4.2 Grundstruktur eines Mikroprozessorsystems 35
2.4.3 Pipelining 37
2.4.4 Daten- und Befehlsabhängigkeiten 41
2.4.5 Superskalartechnik 44

3 Eine rekonfigurierbare Mikroarchitektur - Modellierung und Simulation 53

3.1 Abgrenzung 54
3.2 Modell einer rekonfigurierbaren Mikroarchitektur 60
3.3 Modellsimulation 71
3.3.1 Der *SimpleScalar* Simulator 71
3.3.2 Erweiterung des *SimpleScalar* Simulators 75
3.3.3 Leistungsbewertung 77

4 Hardware-Implementierung der modellierten Mikroarchitektur 86

4.1 Vorbereitungen 86
4.2 Implementierung einer superskalaren Befehlspipeline 87
4.2.1 Befehlsholphase 88
4.2.2 Decodierphase 93
4.2.3 Zuordungsphase 95
4.2.4 Ausführungsphase 99
4.2.5 Rückschreibphase 100
4.2.6 Integration und Test der Mikroarchitektur 101
4.3 Anwendung der partiellen und dynamischen Rekonfiguration 103

5 Schlussbetrachtung **108**

5.1 Bewertung 108

5.2 Zusammenfassung 109

5.3 Ausblick 111

Literaturverzeichnis **113**

Anhang **120**

A Simulationsergebnisse **120**

Index **122**

Abbildungsverzeichnis

1.1 Rekonfigurierbare Hardware [Har01] 2

2.1 Zweistufiges UND/ODER-Schaltnetz 12
2.2 Struktur eines Virtex-II FPGA [Xil04] 15
2.3 Look-Up Table mit SRAM-Zellen 16
2.4 Entscheidungsbaum einer ODER-Verknüpfung 17
2.5 Art der Kopplung [BL00] 20
2.6 Single Context FPGA 23
2.7 Multi Context FPGA 24
2.8 Y-Diagramm nach Gajski-Walker [Wan98] 26
2.9 Entwurfsablauf von FPGA-Schaltungen [Wan98, Sik01, LWS94] 28
2.10 Struktur eines von-Neumann-Rechners 36
2.11 Pipelining 38
2.12 Pipeline-Phasen 39
2.13 Superskalare Pipeline 45

3.1 Adaptive Hardware-Struktur [Alb98] 56
3.2 Modell einer rekonfigurierbaren Mikroarchitektur 61
3.3 Befehlsholphase 62
3.4 Aufbau eines Trace Cache 63
3.5 Das SimpleScalar Tool Set [BAB97] 73
3.6 Instructions Per Cycle 82

3.7 Cycles Per Instruction 83
3.8 Instructions Per Cycle bei erweiterten Hardware-Ressourcen . 84
3.9 Anforderung der verschiedenen Ausführungseinheiten 85

4.1 Pipeline-Phasen der superskalaren Mikroarchitektur 89
4.2 Fetch Stage 90
4.3 Decode Stage 94
4.4 Issue Logic 95
4.5 Execution/Memory Stage 99
4.6 Abbild des Mikroprozessors nach Place&Route 102
4.7 Aufteilung der Mikroarchitektur in ein festes und ein rekonfigurierbares Modul 104
4.8 Aufbau eines Bus-Macro [Xil03] 105
4.9 Abbild des FPGA nach Platzierung von Bus Macros zwischen dem festen und dem rekonfigurierbaren Modul 107

Tabellenverzeichnis

2.1 Typen programmierbarer Logikbausteine [Sik01, Wan98, Aue94] 13

3.1 Ausschnitt aus dem SPEC CPU2000 Benchmark 79
3.2 Simulierte Mikroarchitekturen 81
3.3 Simulation mit erweiterten Hardware-Ressourcen 82

A.1 Instructions Per Cycle 120
A.2 Cycles Per Instruction 120
A.3 Instructions Per Cycle bei erweiterten Hardware-Ressourcen . 121
A.4 Cycles Per Instruction mit erweiterten Hardware-Ressourcen 121
A.5 Anforderung der verschiedenen Ausführungseinheiten 121

Abkürzungen

ALU	*Arithmetic and Logical Unit*
AMD	*Advanced Micro Devices*
ARM	*Advanced RISC Machines*
ASIC	*Application Specific Integrated Circuit*
BRAM	*Block SelectRAM*
CAE	*Computer-Aided Engineering*
CISC	*Complex Instruction Set Computer*
CLB	*Configurable Logic Block*
CPI	*Cycles Per Instruction*
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
DCM	*Digital Clock Manager*
DMA	*Direct Memory Access*
E/A	Eingabe/Ausgabe
EDIF	*Electronic Design Interchange Format*
EPIC	*Explicitly Parallel Instruction Computing*
EPROM	*Erasable and Programmable Read Only Memory*
FPGA	*Field-Programmable Gate Array*
FPU	*Floating-Point Unit*
GCC	*GNU C Compiler*
HDL	*Hardware Description Language*
IA	*Intel Architecture*
IC	*Integrated Circuit*
ILP	*Instruction-Level Parallelism*
I/O	*Input/Output*
IOB	*Input/Output Block*
IPC	*Instructions Per Cycle*
ISA	*Instruction Set Architecture*
JTAG	*Joint Test Action Group*

LSU	*Load/Store Unit*
LUT	*Look-Up Table*
MDU	*Multiply/Divide Unit*
MIPS	*Microprocessor Without Interlocking Pipeline Stages*
MMU	*Memory Management Unit*
MMX	*Multimedia Extension*
PAL	*Programmable Array Logic*
PC	*Program Counter*
PCI	*Peripheral Component Interconnect*
PDA	*Personal Digital Assistant*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PROM	*Programmable Read Only Memory*
RAW	*Read After Write*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RTL	*Register-Transfer Level*
SIMD	*Single Instruction Multiple Data*
SPARC	*Scalable Processor Architecture*
SPEC	*Standard Performance Evaluation Corporation*
SRAM	*Static Random Access Memory*
VHDL	*Very High Speed Integrated Circuits Hardware Description Language*
VLIW	*Very Long Instruction Word*
WAR	*Write After Read*
WAW	*Write After Write*

1 Einleitung

Der Rechner hat endgültig den Einzug in das alltägliche Leben gefunden. Die eindrucksvolle Verbreitung verdanken wir vielen Faktoren wie der regelmäßigen Einführung von immer leistungsfähigeren Prozessoren und sinkenden Preisen für ältere Modelle. Wir erleben seit vielen Jahrzehnten eine rasante Entwicklung der Prozessorleistung, die sowohl eine breite Auswahl als auch ein günstiges Preis-/Leistungsverhältnis ermöglicht. Damit kann für aufwendige Berechnungen z.B. in Forschungseinrichtungen auf Hochleistungsprozessoren zurückgegriffen werden, während sich immer mehr Anwender den „Volks-PC“ für den Hausgebrauch leisten können.

Zwei Faktoren haben maßgeblich dazu beigetragen, die Leistungsfähigkeit von Prozessoren ununterbrochen zu steigern [VMS97, XA00]. Zum einen werden ständig immer mehr Transistoren auf einem Chip nach dem immer noch geltenden Gesetz von Moore[1] integriert. Solche hoch integrierten Schaltungen ermöglichen es, immer mehr Funktionen auf der gleichen Siliziumfläche zu realisieren. Darüber hinaus tragen diese Fortschritte in der Halbleitertechnologie durch die Miniaturisierung zu einer stetigen Erhöhung der Prozessortaktfrequenz bei. Zum anderen verhelfen verbesserte Implementierungstechniken heutiger Mikroprozessoren zu erkennbarer Leistungssteigerung. Zu solchen Maßnahmen gehören beispielsweise Pipelining, Superskalarität, VLIW und *Multithreading*.

Moderne Prozessoren bieten darüber hinaus weitere Vorteile mit deutlich erkennbarem Einfluss auf die Verbreitung des PC. Die hoch gelobte Kompatibilität zwischen unterschiedlichen Prozessorgenerationen z.B. führt dazu, dass Anwendungen nicht neu programmiert werden müssen, wenn ein neuer Prozessor eingeführt wird. Außerdem lässt sich ein Prozessor durch Ausführung beliebiger Software-Anwendungen universell einsetzen.

[1] Gordon Moore hat 1965 vorausgesagt, dass sich die Anzahl der auf einem Chip integrierten Transistoren alle 24 Monate verdoppelt [Moo65]

Dennoch, je mehr Rechenleistung zur Verfügung steht, desto mehr Geschwindigkeit und kürzere Reaktionszeiten werden von der Seite der Anwender verlangt [DeH96]. Außerdem existiert eine Reihe von speziellen Anwendungen, z.B. die sog. Echtzeitanwendungen, die eine strenge Einhaltung festgelegter Ausführungszeiten fordern. Bei dieser Art von Anwendungen machen sich die Leistungsgrenzen herkömmlicher Prozessoren bemerkbar, da sie fest verdrahtet hergestellt werden und sich nicht nachträglich optimieren lassen. Um solchen harten Anforderungen gerecht zu werden, wird spezialisierte Hardware in Form von Co-Prozessoren oder *Application Specific Integrated Circuits* (ASICs) eingesetzt [VMS97, XA00, DeH96]. Derartige Hardware-Plattformen werden für die Lösung einer ganz speziellen Aufgabe (Grafik, Signalverarbeitung, Kryptografie, ...) optimiert. Ändert sich die Aufgabe, so muss eine andere Hardware-Lösung neu entwickelt werden. Mit dieser Vorgehensweise wird lediglich eine Leistungssteigerung für eine bestimmte Problemstellung erzielt. Dabei geht die Flexibilität verloren, unterschiedliche Lösungen auf der gleichen Hardware-Plattform realisieren zu können.

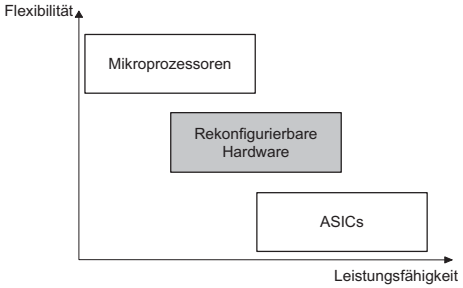

Abbildung 1.1: *Rekonfigurierbare Hardware [Har01]*

Zwischen den flexiblen und universellen Prozessoren auf der einen Seite und den schnellen und hoch spezialisierten ASICs auf der anderen Seite etabliert sich zunehmend die rekonfigurierbare Hardware, die eine flexible Implementierung einer Vielzahl von Lösungen auf dem gleichen programmierbaren Baustein ermöglicht [Har01]. Es handelt sich um die sog. *Field-Programmable Gate Arrays* (FPGAs), die sich durch den Anwender beliebig

oft programmieren (bzw. konfigurieren) lassen. Durch wiederholte Programmierung eines FPGA ist es möglich, unterschiedliche Funktionen in Hardware zu implementieren. Damit verbinden FPGAs den Vorteil einer schnellen Hardware-Lösung mit der Flexibilität eines universell einsetzbaren Prozessors [PB99] (siehe Abb. 1.1). In Analogie zum *General Purpose Computing* (auch *Mainstream Computing* genannt), das auf den fest verdrahteten Prozessoren basiert, wird das Rechnen auf FPGA-basierten Rechensystemen mit *Reconfigurable Computing* bezeichnet.

1.1 Motivation

Das Gebiet des *Reconfigurable Computing* gehört zu den heute aktivsten Forschungsgebieten, sowohl im industriellen, als auch im universitären Umfeld. Dies belegen zum einen die zahlreichen wissenschaftlichen Veröffentlichungen und weltweiten Konferenzen auf diesem Gebiet [Har01]. Zum anderen existiert bereits eine Vielzahl von kommerziellen Produkten, welche auf rekonfigurierbarer Hardware basieren [Ten05, Cel05, PAC05, Str05]. Die Forschungsaktivitäten im Zusammenhang mit rekonfigurierbarer Hardware lassen sich in drei Kategorien unterteilen:

- **Architekturen**:
  Arbeiten in dieser Gruppe beschäftigen sich hauptsächlich mit der Modellierung und Entwicklung rekonfigurierbarer Mikroarchitekturen. Einige Beispiele dafür sind: MorphoSys [SLL+00], MATRIX [DeH96], Xputer [HHKW90], PipeRench [GSM+99], Chimaera [YMH+02]. Dabei wird von den speziellen Eigenschaften von FPGAs Gebrauch gemacht, um eine deutliche Leistungssteigerung zu erreichen. Die meisten daraus resultierenden Architekturen sind jedoch anwendungsspezifisch, so dass sich die erzielte Leistung nicht auf andere Anwendungsfälle übertragen lässt. In jüngster Zeit rückt zunehmend besonders die Fähigkeit der dynamischen Rekonfiguration in den Mittelpunkt der Forschungsaktivitäten in dieser Kategorie (*dynamic reconfiguration*). Das Ziel dabei ist die Entwicklung flexibler Architekturen, die sich für einen breiteren Umfang von Problemstellungen eignen. Die

vorliegende Arbeit fügt sich in diese Kategorie ein und setzt dabei einen besonderen Akzent auf die Universalität und Kompatibilität rekonfigurierbarer Architekturen.

- **Anwendungen**:
  Für den größten Teil der Forschungsarbeiten auf dem Gebiet des *Reconfigurable Computing* geht es darum, die Ausführung einer bestimmten Anwendung anhand einer Hardware-Implementierung zu beschleunigen. Dies geschieht oftmals durch Verlagerung aufwendiger Algorithmen oder sogar vollständiger Applikationen auf die flexiblen und leistungsfähigen FPGAs. Ein universeller Prozessor übernimmt dabei die Steuerungsaufgaben, während eine rekonfigurierbare Einheit die spezielle Anwendung implementiert (siehe Abschn. 2.2.2). Das Anwendungsspektrum umfasst beispielsweise die Signal- und Bildverarbeitung [JM02, Mar04], die Kryptografie [PW04], numerische Berechnungen [LLVC04], Netzwerkprotokolle [MBV+02] etc. Die ersten Implementierungen dienten überwiegend zum *Rapid Prototyping*. Heutzutage ersetzen sie mehr und mehr traditionelle ASIC-Realisierungen.

- **Werkzeuge**:
  Die letzte Gruppe von Arbeiten widmet sich der Entwicklung von Hardware- und Software-Werkzeugen, um Rekonfiguration zu unterstützen bzw. überhaupt erst zu ermöglichen. Dazu zählen die diversen Technologie-Anbieter, die sowohl rekonfigurierbare Bausteine (FPGAs) als auch herstellerabhängige Software-Pakete zur FPGA-Programmierung zur Verfügung stellen [Xil05, Alt05]. Darüber hinaus gibt es zahlreiche Entwicklungen, welche herstellerunabhängig die Leistungsfähigkeit rekonfigurierbarer Rechensysteme steigern und deren Handhabbarkeit erleichtern sollen [BKS04, Str04]. Betriebssysteme für solche Systeme [WP03, WK01] oder spezielle Compiler, die beispielsweise den Quellcode in getrennte Hardware-und Software-Anteile partitionieren [Koc02], gehören ebenfalls zu dieser Kategorie.

Trotz der erkennbaren Fortschritte, *Reconfigurable Computing* als zukunftsweisender Technologie zum Durchbruch zu verhelfen, gibt es immer noch eine Vielzahl an offenen Fragen und Verbesserungsmöglichkeiten. Dazu gehören beispielsweise weiterführende Arbeiten, um den Umgang mit rekonfigurierbaren Rechensystemen zu erleichtern und somit eine höhere Akzeptanz durch ein breites Publikum zu erreichen. Hierfür sind Ansätze zur Bereitstellung von einfach zu bedienenden Werkzeugen sowie kompatiblen rekonfigurierbaren Rechenarchitekturen von größer Bedeutung. Die vorliegende Arbeit beschränkt sich im Hinblick auf den damit verbundenen Aufwand jedoch auf die Architektur-Ebene und untersucht dabei die Einsatzmöglichkeiten rekonfigurierbarer Hardware in Prozessorarchitekturen. Dabei soll folgenden Fragestellungen nachgegangen werden:

- In wie weit lassen sich die viel versprechenden Eigenschaften von FPGAs auf das Gebiet der Prozessorarchitekturen übertragen?
- Wie wird die Leistungsfähigkeit des Prozessors dadurch beeinflusst und mit welchem Entwicklungsaufwand ist dabei zu rechnen?

Im Gegensatz zu den zahlreichen existierenden Arbeiten, die sich bereits mit solchen Fragen beschäftigt haben, wird hier ein evolutionärer Ansatz bevorzugt. Die Untersuchung soll auf weit verbreiteten und allgemein akzeptierten Techniken basieren und universell anwendbare Ergebnisse liefern. Im Unterschied dazu geht es bei alternativen Forschungsprojekten auf diesem Gebiet häufig darum, verbesserte und/oder optimierte Lösungen im Hinblick z.B. auf Geschwindigkeit, Leistungsaufnahme, Chip-Fläche etc. zu entwickeln, welche aber nur für eine ausgewählte Klasse von Problemen geeignet sind.

Eine weitere Singularität dieser Arbeit beruht auf einer Beobachtung der jüngsten Entwicklungen in der Mikroprozessortechnik. Neben den ständig steigenden Taktfrequenzen zeichnen sich moderne Prozessoren besonders dadurch aus, dass sie über eine wachsende Anzahl an spezialisierten Funktionseinheiten verfügen. Jede dieser Funktionseinheiten ist für die Ausführung einer bestimmten Klasse von Befehlen optimiert: *Load/Store*

Units (LSUs) für die Ausführung von Speicherzugriffen, *Arithmetic and Logic Units* (ALUs) für Integer-Operationen und *Floating-Point Units* (FPUs) für Gleitkomma-Arithmetik. Der Intel Pentium 4 Prozessor und der AMD Athlon Prozessor beispielsweise enthalten bis zu 13 solcher spezialisierter Funktionseinheiten, der IBM PowerPC750 fünf Ausführungseinheiten, der MIPS R1000 sieben Ausführungseinheiten und der UltraSPARC-III sechs Ausführungseinheiten [BU02, OV03].

Allerdings kommen nicht alle diese Funktionseinheiten gleichzeitig zum Einsatz. Einer der Gründe dafür ist, dass sich Befehle nur begrenzt parallelisieren lassen. Selbst Maßnahmen zur Steigerung der Befehls-Parallelisierung (*Instruction-Level Parallelism*) schaffen es nicht, alle Einheiten gleichzeitig mit Befehlen zu versorgen. Abhängig von dem gerade ausgeführten Programm werden bestimmte Ausführungseinheiten überproportional beansprucht, während die restlichen Einheiten überhaupt nicht benötigt werden. Dieser Umstand tritt dann ein, wenn beispielsweise ein Programm nur aus Integer-Operationen besteht, wie z.B. im Fall der SPEC Int2000 *Benchmark* [SPE05]. In diesem Fall werden ausschließlich ALUs und LSUs aktiviert und die FPUs werden kaum genutzt. Letztere belegen trotzdem weiterhin Hardware-Ressourcen, da der Prozessor fest verdrahtet hergestellt wurde. Wie wäre es aber, wenn der Prozessor auf einer rekonfigurierbaren Hardware basierte? Damit hätte der Anwender die Möglichkeit, die Anzahl der zur Verfügung stehenden Ausführungseinheiten dynamisch an das Programm anzupassen. Diese Arbeit findet ihren Ursprung auch in dieser Fragestellung.

Viele Forschungsgruppen haben schon erkannt, dass der Einsatz rekonfigurierbarer Hardware in Prozessorarchitekturen viele Vorteile mit sich bringt. Damit werden z.B. Flexibilität und hohe Ausführungsgeschwindigkeit vereint [WH95, HW97, XA00, SLC02]. In den meisten Fällen wird ein Hauptprozessor für Steuerungsaufgaben verwendet, während das FPGA rechenintensive Aufgaben übernimmt. Trotz der bemerkenswerten Leistung einiger rekonfigurierbarer Prozessoren verhindert bisher die oft komplizierte Programmierung sowie die im Allgemeinen aufwändige Handhabung ihre breite Akzeptanz [MSH97, CCH+00]. Bis auf wenige Ausnahmen, die eine

benutzerfreundliche Programmierumgebung anbieten, muss der Entwickler zunächst ein neues rekonfigurierbares Rechensystem mit der darunter liegenden Hardware-Architektur erlernen, bevor er Anwendungen darauf optimieren und ausführen kann. Eine wesentliche Hürde für einen potenziellen Anwender besteht oftmals darin, dass er sein Programm in bestimmte Partitionen aufteilen muss, von denen ein Teil in Hardware übersetzt und ein anderer Teil als Software compiliert wird. Außerdem trägt die fehlende Kompatibilität unter den verschiedenen rekonfigurierbaren Prozessoren nicht gerade zu ihrer Verbreitung bei.

Daher genügt es nicht, dass rekonfigurierbare Hardware für ausgewählte Anwendungen zu einer erkennbaren Leistungssteigerung führt, um sich als echte Alternative zu fest verdrahteten Prozessoren zu etablieren. Sie müssen darüber hinaus noch ihre Standfestigkeit bezüglich Universalität, Flexibilität und Benutzerfreundlichkeit beweisen.

1.2 Zielsetzung

In dieser Arbeit geht es nicht darum, für eine vorgegebene Anwendung eine möglichst schnelle Hardware-Lösung auf einem FPGA zu implementieren. Genauso wenig wird die Idee verfolgt, ein rekonfigurierbares leistungsfähiges Rechensystem zu realisieren, das sich nur für eine bestimmte Klasse von Anwendungen eignet. Vielmehr geht es in dieser Arbeit darum zu untersuchen, ob und mit welchem Ergebnis sich ein universaler (*general purpose*) Prozessor anhand von rekonfigurierbarer Hardware realisieren lässt. Dabei soll mit Hilfe der so vielgelobten Eigenschaften der partiellen und dynamischen Rekonfiguration eine Leistungssteigerung angestrebt werden.

Der wesentliche Vorteil gegenüber bereits bestehenden Ansätzen liegt darin, dass die Hardware-Ressourcen transparent für den Benutzer an die jeweiligen Anwendungen angepasst werden. Es entsteht kein zusätzlicher Hardware-spezifischer Aufwand für den Programmierer und jeder Anwendung wird automatisch die beste mögliche Hardware-Konfiguration angeboten. Damit wird die Einstiegshürde in die Welt des *Reconfigurable Computing* deutlich reduziert. Darüber hinaus wird die Kompatibilität mit exis-

tierenden Programmen und Entwicklungswerkzeugen weitgehend sichergestellt. Dies ist darauf zurückzuführen, dass die Untersuchung auf einer bestehenden Prozessorarchitektur basiert. Ausgehend von einem festgelegten Programmiermodell wird lediglich versucht, Programme ohne zusätzlichen Software-Aufwand mit Hilfe einer rekonfigurierbaren Mikroarchitektur auszuführen.

Die Untersuchung beginnt mit der Modellierung einer neuartigen rekonfigurierbaren Mikroarchitektur. Anhand von Software-Simulationen sollen zunächst Erkenntnisse über die Realisierungsmöglichkeiten gewonnen werden. Zusätzlich ermöglicht eine solche Software-Simulation eine vorläufige Abschätzung von Leistungskenngrößen. Den Kern der Arbeit bildet der Versuch, nach Festlegung eines bestimmten Befehlssatzes die modellierte Mikroarchitektur tatsächlich auf rekonfigurierbare Hardware (FPGA) umzusetzen.

1.3 Aufbau der Arbeit

Im folgenden Kapitel wird zunächst eine grobe Einteilung der programmierbaren Logik vorgenommen, um rekonfigurierbare Hardware (FPGA) besser einordnen zu können. Dieser Art von programmierbaren Bausteinen gilt hier das besondere Augenmerk. Zusätzlich bietet das zweite Kapitel einen tieferen Einblick in die verschiedenen Ausführungsmöglichkeiten FPGA-basierter Rechensysteme. Der letzte Teil dieses Grundlagen-Kapitels stellt einige moderne Mikroarchitektur-Techniken dar, die in heutigen Prozessoren zu Leistungssteigerungen führen. Aufbauend auf einigen dieser Techniken und unter Berücksichtigung wichtiger Eigenschaften von FPGAs wird die Basis einer neuen Mikroarchitektur geschaffen.

Das dritte Kapitel widmet sich dem theoretischen Modell einer partiell und dynamisch rekonfigurierbaren Mikroarchitektur. Das Modell basiert auf modernen Implementierungstechniken wie z.B. Pipelining und Superskalarität, um den steigenden Leistungsanforderungen gerecht zu werden. Weiterhin erlaubt die partielle und dynamische Hardware-Rekonfiguration zur Laufzeit eine ständige Anpassung der Mikroarchitektur an die gerade

ausgeführten Anwendungen. Anhand von Software-Simulationen werden in diesem Kapitel verschiedene Ansätze zu einer rekonfigurierbaren Mikroarchitektur erprobt. Eine erste Leistungsbewertung der simulierten Mikroarchitekturen erfolgt mit Hilfe des SPEC-Benchmarks.

Im vierten Kapitel wird die Implementierung eines ausgewählten Modells einer rekonfigurierbaren Mikroarchitektur auf realer Hardware beschrieben. Die Mikroarchitektur basiert auf dem ARM-*Thumb*-Befehlssatz [ARM00]. Die Umsetzung der bisher noch wenig erprobten Technik der partiellen Rekonfiguration auf den entstandenen Mikroprozessor wird ebenfalls erörtert.

Das fünfte Kapitel fasst die Ergebnisse dieser Arbeit zusammen. Offen gebliebene Fragestellungen sowie Anregungen für weiterführende Arbeiten werden als Ausblick behandelt.

2 Grundlagen

2.1 Programmierbare Logikbausteine

Digitale Schaltungen bestehen im Allgemeinen aus einer Vielzahl von elementaren logischen Blöcken (Basiszellen), welche über ein fest verdrahtetes oder programmierbares Leitungsnetz miteinander verbunden werden. Darauf aufbauend lassen sich einfache sowie komplexe logische Funktionen realisieren. Bereits über 30 Jahren werden immer mehr solcher elementarer Blöcke auf standardisierten Chip-Flächen integriert. Dadurch entstehen die sog. integrierten Schaltungen (*Integrated Circuits*, ICs). Wie bereits im einleitenden Kapitel angedeutet, lassen sich solche integrierten Bausteine hinsichtlich ihrer Funktionalität heute in drei Gruppen einteilen [Sik01, Wan98]:

- **Fest verdrahtete universelle Bausteine**
  Diese Art von Bausteinen wird vom Hersteller entwickelt, um eine bestimme Klasse von Anwendungen auszuführen. Der Anwender hat keine Möglichkeit, die Hardware nachträglich in ihrer Struktur zu verändern. Ein festgelegtes Programmiermodell ermöglicht das Erstellen von Programmen, die anschließend durch derartige fest verdrahtete und universelle Hardware ausgeführt werden. Mikroprozessoren und zahlreiche Mikrocontroller sind die bekanntesten Vertreter dieser Gruppe von Bausteinen.

- **ASICs**
  ASICs stellen eine Hardware-Plattform dar, auf welcher eine maßgeschneiderte Lösung für eine bestimmte Anwendung implementiert wird. Für den Entwurf solcher Lösungen hat der ASIC-Entwickler die Möglichkeit, Bauelemente der Komplexität eines Gatters oder eines Flipflops selbst auf dem Chip zu platzieren. Zusätzlich muss er

für die notwendigen Verbindungen sorgen, um eine kundenspezifische Funktion zu realisieren (*Semi-Custom-Entwurf*). Der Einsatz solcher Komponenten reduziert den Entwicklungsaufwand, da diese schon optimiert und fehlerfrei zur Verfügung stehen. In einem *Full-Custom-Entwurf* dagegen setzt der Entwickler die gewünschte Funktionalität ausschließlich anhand von Transistoren und elektrischen Leitungen um. Der Entwurfsaufwand ist dementsprechend hoch und die Fehlerquote ebenfalls. Abhilfe schafft der Einsatz von rechnergestützten Entwurfswerkzeugen.

Das Ergebnis ist in beiden Fällen wiederum eine fest verdrahtete Hardware, auf der eine bestimmte Funktion optimal abgebildet wird. Im Gegensatz zum Prozessor, welcher durch das Laden unterschiedlicher Programme verschiedene Anwendungen ausführen kann, beschränkt sich die Anwendungsbreite eines ASIC auf eine einzige Applikation, denn ein ASIC lässt sich nur einmal „beschreiben". In ihren Hardware-Strukturen sind ASICs genau wie die im vorangegangenen Punkt besprochenen Bausteine vom Werk aus fest verdrahtet. Sie unterscheiden sich jedoch dadurch, dass Letztere erst durch die Ausführung einer zweckentsprechenden Anwendungssoftware zur Lösung eines Problems beitragen. Dagegen stellt ein ASIC von vornherein eine optimierte Hardware-Lösung für eine bestimmte Anwendung bereit.

- **Programmierbare Logikbausteine**
  Programmierbare Logikbausteine bieten die Möglichkeit, anwendungsspezifische Hardware-Lösungen beliebig oft auf der gleichen Plattform zu implementieren. Im nachstehenden Abschnitt (Absch. 2.1.1) wird auf diese Familie von integrierten Schaltungen näher eingegangen.

2.1.1 Vom Speicher zum FPGA

Die Tatsache, dass jede boolesche Funktion als disjunktive Normalform (DNF) dargestellt werden kann, hat zur Folge, dass sie auch leicht in einem ROM-Speicher abgebildet werden kann. Ein ROM entspricht logisch einem zweistufigen Schaltnetz mit einer UND-Matrix und einer ODER-

Matrix, die beide über elektrische Leitungen miteinander verbunden sind (Abb 2.1). Dabei stellt die UND-Matrix den Decoder dar, welcher aus der Konjunktion der Eingangssignale die entsprechende Speicheradresse bildet. Die ODER-Matrix dagegen erfüllt die Funktion der Speichermatrix und liefert das gewünschte Speicherwort an den Ausgängen des Schaltnetzes. Aus der Disjunktion der logischen Werte („0“ oder „1“) an den adressierten Speicherzellen wird das entsprechende Speicherwort zusammengesetzt [Sik01, Wan98, Aue94, Möl03, OV03] .

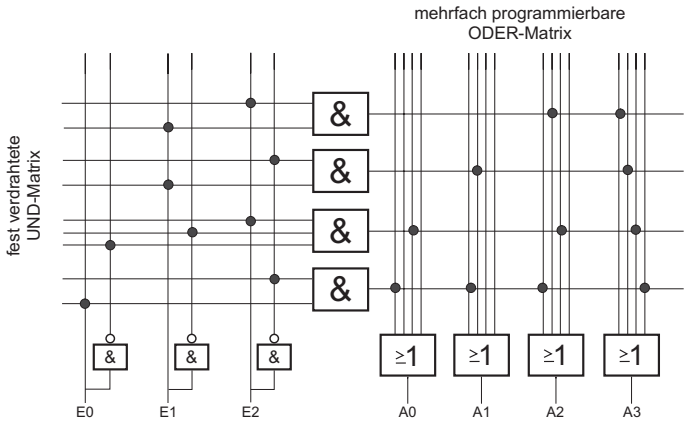

Abbildung 2.1: *Zweistufiges UND/ODER-Schaltnetz*

Solche zweistufigen UND/ODER-Schaltnetze werden im Allgemeinen als *Programmable Logic Devices* (PLDs) bezeichnet. Eine feinere Unterscheidung kann danach getroffen werden, ob die UND/ODER-Matrizen fest verdrahtet werden oder mehrfach programmierbar bleiben.

Vor dem eigentlichen Programmieren einer Matrix stehen zahlreiche elektrische Leitungen zur Verfügung, aber es besteht keine feste Verbindung zwischen unterschiedlichen Leitungen. Bei einer festen Verdrahtung werden lediglich solche Verbindungen hergestellt, die für die Realisierung einer bestimmten Funktionalität notwendig sind. Die nicht verwendeten Verbindungen werden entfernt und es findet somit eine einmalige Programmierung statt. Falls eine erneute Programmierung möglich sein soll, bleiben die

nicht verwendeten Leitungen bestehen (siehe Abb. 2.1). Diese und eventuell gelöste alte Verbindungen ermöglichen zu einem späteren Zeitpunkt die Implementierung einer neuen Funktion. Aus diesen Überlegungen können folgende PLD-Bausteine klassifiziert werden:

- Falls die UND-Matrix fest verdrahtet wird und die ODER-Matrix mehrfach programmierbar ist, dann entsteht ein *Programmable Read Only Memory* (PROM).
- Kann dagegen die UND-Matrix wiederholt programmiert werden und die ODER-Matrix wird fest verdrahtet, dann spricht man von *Programmable Array Logic* (PAL).
- Lassen sich sowohl die UND-Matrix als auch die ODER-Matrix mehrmals programmieren, so spricht man von einem *Programmable Logic Array* (PLA).

Tabelle 2.1 fasst noch einmal die Eigenschaften der verschiedenen programmierbaren Logikbausteine zusammen, wenn nur die Programmierbarkeit der Matrizen betrachtet wird.

Baustein	UND-Matrix	ODER-Matrix
PROM	fest verdrahtet	mehrfach programmierbar
PAL	mehrfach programmierbar	fest verdrahtet
PLA	mehrfach programmierbar	mehrfach programmierbar

Tabelle 2.1: *Typen programmierbarer Logikbausteine [Sik01, Wan98, Aue94]*

PLDs reichen nur für die Implementierung einfacher logischer Funktionen aus. Für komplexere, rein kombinatorische Funktionen werden mehrere PLDs hintereinander geschaltet. Dabei entstehen die sogenannten *Complex Programmable Logic Devices* (CPLDs). Ein PLD bildet somit die Basiszelle eines CPLD. Beide Arten von programmierbaren Logikbausteinen eignen sich besonders für die Implementierung von Funktionen, die sich

durch reine kombinatorische Logik darstellen lassen (Schaltnetze). Wird dagegen eine Funktionalität mit Rückkopplung bzw. mit Speichermöglichkeiten (Schaltwerk) gewünscht, so hat sich herausgestellt, dass sie nicht die optimale Plattform darstellen. Eine Lösungsmöglichkeit bieten die *Field-Programmable Gate Arrays* (FPGAs), die aus Basiszellen mit Speicherungseigenschaften (*Look-Up Tables*, Flipflops, etc.) und einem programmierbaren Verbindungsnetz bestehen.

2.1.2 FPGA-Struktur

Ein FPGA besteht generell aus einer Vielzahl von konfigurierbaren Logikblöcken (*Configurable Logic Blocks*, CLBs), die als Basiszellen in einer Matrix-Struktur angeordnet sind. Gegenüber PLDs zeichnen sich FPGA-Basiszellen durch ihre Eigenschaft aus, eine höhere Anzahl logischer Funktionen realisieren zu können. Zur Illustration dieser Eigenschaft soll ein CLB der Xilinx Virtex-II FPGAs herangezogen werden. Abbildung 2.2 stellt ein solches CLB dar, welches sich aus vier sog. *Slices* zusammensetzt, wobei jedes davon aus zwei programmierbaren Funktionsgeneratoren mit jeweils vier Eingängen besteht. Derartige Funktionsgeneratoren können als *Look-Up Tables* (LUTs) mit vier Eingängen, 16-Bit-Speicherelementen oder sogar als 16-Bit-Shift-Register programmiert (konfiguriert) werden. Darüber hinaus finden in einem *Slice* logische Bauelemente für arithmetische Operationen, ein Multiplexer sowie zusätzliche Speicherelemente Platz [Xil04, Xil02].

Weitere Strukturmerkmale eines FPGA bilden die programmierbaren Verbindungsglieder (*Switching Matrix*, S) sowie die zahlreichen elektrischen Leitungen (*Connectivity Matrix*, C), die zwischen den CLBs angelegt werden. Außerdem ermöglichen spezielle Ein- und Ausgangsblöcke (*I/O-Block*, IOB) den Datenaustausch zwischen dem FPGA und seiner Umgebung [HB02, Wan98, Sik01]. Darüber hinaus verfügen Xilinx FPGAs über zahlreiche Speicherblöcke (18-Kbit Block SelectRAM), schnelle 18-Bit x 18-Bit Multiplizierer sowie *Digital Clock Manager* (DCM), die auf der ganzen FPGA-Chipfläche für einen unverzögerten Taktgeber sorgen.

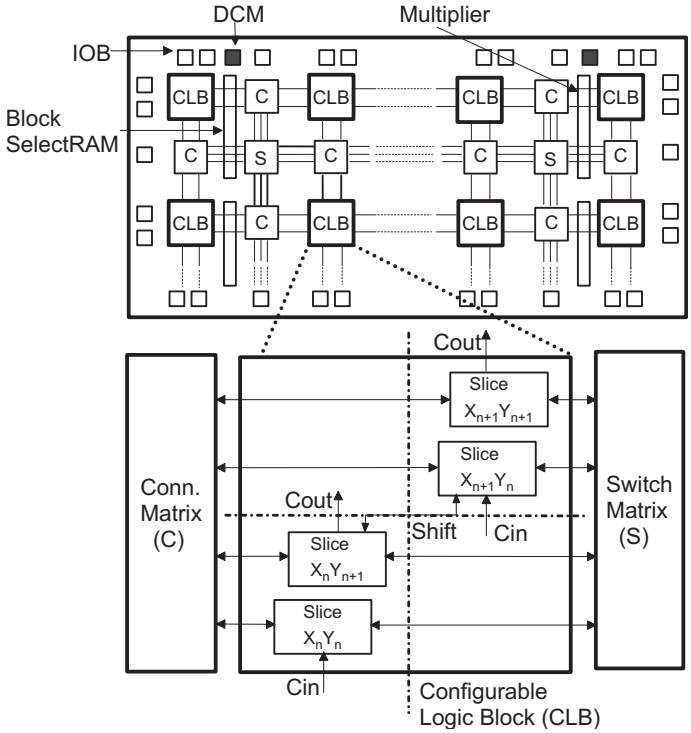

Abbildung 2.2: *Struktur eines Virtex-II FPGA [Xil04]*

Trotz dieser generellen und einheitlichen Grundstruktur jedes FPGA ist es erwähnenswert, dass zahlreiche Hersteller unterschiedliche Technologien zur Implementierung einer Basiszelle und zur Programmierung der Verbindungsmatrizen einsetzen. Daher existieren unterschiedliche Arten von FPGAs mit unterschiedlichen Eigenschaften. Bei der Wahl eines geeigneten FPGA für eine bestimmte Anwendung muss daher besonders geprüft werden, ob die gewünschten Eigenschaften unterstützt werden.

Die heutzutage verfügbaren FPGAs basieren auf unterschiedlichen CLB-Technologien. Im Wesentlichen sind sie aus *Look-Up Tables*, Multiplexern oder als sog. *Sea-of-gate* aufgebaut [HB02, Wan98, Aue94, Sik01].

- ***Look-Up Tables* (LUTs)**:
  FPGAs dieser Art bestehen aus CLBs, die hauptsächlich auf LUTs aufgebaut sind. Es handelt sich um „Nachschlagetabellen“, die für jede Eingangskombination der festgelegten Eingangssignale einen entsprechenden logischen Funktionswert zurück liefern. Eine LUT mit n Eingängen besitzt für jede der 2^n möglichen Kombinationen der Eingänge eine eigene Speicherzelle, aus der ein bestimmter Funktionswert abgelesen wird. Ein davor geschalteter Decoder sorgt dafür, dass aus den 2^n Speicherzellen diejenige ausgewählt wird, welche der zu realisierenden logischen Funktion entspricht. Abbildung 2.3 zeigt ein Beispiel einer LUT mit 2 Eingängen, welches derart konfiguriert ist, dass eine XOR-Verknüpfung der beiden Eingänge realisiert wird. Um komplexe logische Funktionen mit solchen Gattern zu implementieren, werden mehrere LUTs hintereinander geschaltet und gegebenenfalls Rückkopplungen hinzugefügt.

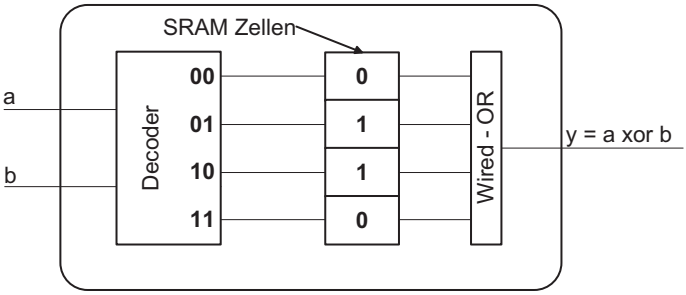

Abbildung 2.3: *Look-Up Table mit SRAM-Zellen*

Die Speicherinhalte der LUTs müssen bei jedem Systemstart neu beschrieben werden, da SRAMs flüchtige Speicher sind. Zu diesem Zweck muss bei solchen SRAM-basierten FPGAs immer ein nichtflüchtiger Speicher (z.B. EPROM) zur Verfügung stehen, aus dem die Anfangskonfiguration des FPGA geladen werden kann. Zu den Vorteilen dieser Art von FPGAs zählt dagegen die Tatsache, dass SRAM-

Zellen im Betrieb neu geschrieben werden können, was die einzigartige Eigenschaft der Rekonfiguration des FPGA zur Laufzeit ermöglicht.

- **Multiplexer**:
  Mit einer LUT wird ein logischer Funktionswert anhand der entsprechenden Funktionstabelle ermittelt. Es ist durchaus möglich, den Funktionswert mit Hilfe eines binären Entscheidungsbaums zu bestimmen (*Binary Decision Diagram*, BDD) [HB02, BDM05]. Durch ein sukzessives Abfragen der Belegung der einzelnen Eingangssignale wird der logische Wert der Funktion schrittweise ermittelt. Abbildung 2.4 zeigt ein Beispiel für die Bestimmung des Funktionswerts einer ODER-Verknüpfung dreier Eingangssignale anhand eines binären Entscheidungsbaums. Wenn jeder Knoten im Entscheidungsbaum hardwaremäßig durch einen 2:1 Multiplexer ersetzt wird, so ist es möglich, eine komplette Schaltung ausschließlich mit Multiplexern zu realisieren. Es gibt tatsächlich eine Reihe von FPGAs, die mit Hilfe solcher Multiplexer hergestellt werden.

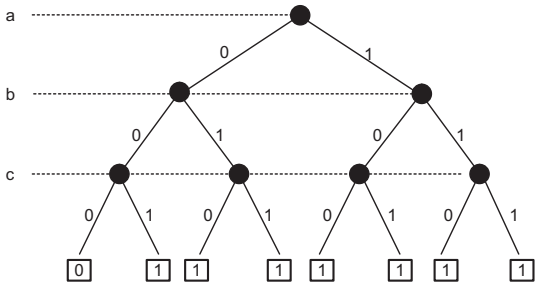

Abbildung 2.4: *Entscheidungsbaum einer ODER-Verknüpfung*

- ***Sea-of-Gates***:
  Diese Art von Basiszellen baut auf einer Vielzahl von PLDs (siehe Abschnitt 2.1.1) auf, die durch eine komplexe programmierbare Verbindungsmatrix gekoppelt sind. Darüber hinaus unterscheiden sich FPGAs, die auf dieser Technologie aufgebaut sind, von den einfachen PLDs bzw. CPLDs durch zusätzliche Ein-/Ausgabeblöcke (IOB). Sol-

che Blöcke dienen dem Datenaustausch zwischen dem FPGA und der „Außenwelt“.

2.2 Rekonfigurierbare Rechensysteme

Mit zunehmender Verbreitung der FPGA-Technologie werden immer mehr Rechensysteme sowie zahlreiche Anwendungen auf dieser rekonfigurierbaren Hardware implementiert. Besonders beliebt ist das FPGA in Kreisen von Hardware-Entwicklern, die jahrelang „*One-Way*“-ASICs entworfen und jetzt erkannt haben, dass sie den gleichen Baustein mehrmals für unterschiedliche Anwendungen einsetzen können. Besonders vorteilhaft erweist sich das FPGA während der Entwicklungsphase, da die mehrfach notwendigen Iterationen zur Fehlerbehebung auf der realen Hardware getestet werden können. Erst, wenn alle Fehler beseitigt worden sind, kann eine Serienproduktion auf Basis eines ASIC oder eines FPGA erfolgen (*Rapid Prototyping*).

FPGAs eignen sich nicht nur für die Implementierung ausgewählter Algorithmen und Anwendungen, sondern sie lassen sich zugleich gut für den Aufbau kompletter Rechensysteme verwenden. Derartige FPGA-basierte Systeme werden als rekonfigurierbare Rechensysteme und das Rechnen auf solchen Systemen wird mit *Reconfigurable Computing* bezeichnet.

Ähnlich der Vielfalt unter den fest verdrahteten Rechensystemen gibt es auch bei rekonfigurierbaren Rechensystemen unterschiedliche Ansätze hinsichtlich ihres Aufbaus sowie ihrer Operationsprinzipien. Generell lassen sie sich durch folgende Kriterien klassifizieren: Granularität (*Granularity*), Art der Kopplung (*Interface*), Programmierbarkeit (*Programmability*) und Rekonfigurierbarkeit (*Reconfigurability*) [SLL+00, BL00, vRU99, CH99].

2.2.1 Granularität

Die Granularität eines rekonfigurierbaren Rechensystems bezieht sich auf die Komplexität und die Datenbreite der rekonfigurierbaren Bildungsblöcke, auf denen das System aufgebaut ist. Bei einem *fine-grained* Rechensystem besteht das System teilweise aus elementaren logischen Gattern (AND-,

OR-Gatter etc.) sowie aus Flipflops mit einer Datenbreite von maximal vier Bits. Solche Gatter können nur boolesche Funktionen auf der Bit-Ebene implementieren. Werden dagegen komplexere Funktionseinheiten wie Addierer, Multiplizierer, Multiplexer usw. verwendet, so handelt es sich um ein *coarse-grained* Rechensystem. Mit solchen *coarse-grained* Einheiten lassen sich Funktionen mit einer höheren Datenbreite implementieren. Zu diesen beiden Arten von Granularität existiert eine Mischung von *fine-* und *coarse-grained* logischen Blöcken. Ein Rechensystem, das darauf aufgebaut wird, wird als *mixed-grained* oder hybrides System bezeichnet.

2.2.2 Art der Kopplung

Rekonfigurierbare Hardware eignet sich nicht besonders gut für Programmteile mit vielen Verzweigungen. Für derartige Programmkonstrukte ist am effektivsten, diese durch einen herkömmlichen Prozessor ausführen zu lassen. Rechenintensive Code-Portionen lassen sich dagegen sehr gut auf rekonfigurierbare Hardware ausgelagern. Der Prozessor sorgt dabei für die notwendige Synchronisation. Dies geschieht im Allgemeinen durch Erweiterung des Befehlssatzes um spezielle Befehle, um Berechnungen auf der rekonfigurierbaren Hardware auszulösen und entsprechende Ergebnisse abzuholen. Die daraus entstehenden rekonfigurierbaren Rechensysteme unterscheiden sich hinsichtlich der Art der Kopplung zwischen der rekonfigurierbaren Hardware und dem Prozessor wie folgt (siehe Abb. 2.5):

- **Funktionseinheit**: Die erste und engste Art der Kopplung besteht darin, die rekonfigurierbare Hardware als eine der Funktionseinheiten des Prozessors zu betrachten. Diese spezielle Funktionseinheit wird in die entsprechende Mikroarchitektur eingebettet und je nach Bedarf für unterschiedliche Aufgaben konfiguriert. Der Datenaustausch mit dem Rest des Prozessors erfolgt üblicherweise über prozessor-interne Register. Die T1000-Architektur ist ein typisches Beispiel für derartige rekonfigurierbare Prozessoren [ZM00]. Im Rahmen dieser Arbeit wurde ebenfalls eine Mikroarchitektur mit rekonfigurierbaren Funktionseinheiten entwickelt. Der Unterschied zu dieser Kategorie von

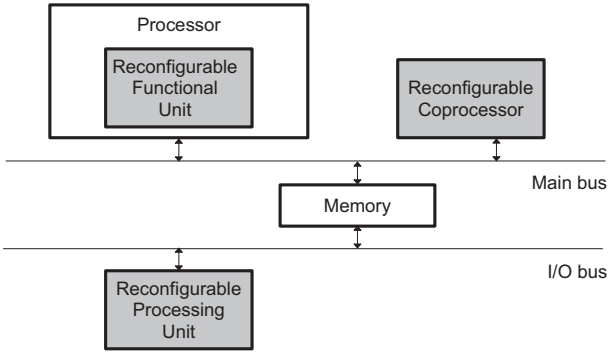

Abbildung 2.5: *Art der Kopplung [BL00]*

Mikroarchitekturen besteht darin, dass die rekonfigurierbaren Funktionseinheiten in der neu entwickelten Mikroarchitektur nicht nur für spezielle Aufgaben verwendet werden. Vielmehr bilden die zu einem bestimmten Zeitpunkt vorhandenen rekonfigurierbaren Funktionseinheiten die eigentliche Mikroarchitektur und führen alle anstehenden Aufgaben aus.

- **Co-Prozessor**: In diesem Fall wird die rekonfigurierbare Hardware zwar in den Datenfluss des Prozessors mit einbezogen, aber sie wird auf einem separaten Chip (FPGA) implementiert. Sie fungiert als üblicher Co-Prozessor und die Kommunikation mit dem Hauptprozessor findet über den Systembus statt. Beispiele hierfür bilden die Hades-Hardware [Lud97], MorphoSys [SLL+00] sowie die DISC-Architektur [WH95].

- **Ein/Ausgabe-Einheit**: Bei dieser Art der Kopplung wird die rekonfigurierbare Hardware als eigenständige Recheneinheit eingesetzt. Sie unterstützt den Prozessor für ausgewählte Rechenaufgaben und wird über eine E/A-Schnittstelle, z.B. über den externen Bus, angesprochen. Eine Realisierungsmöglichkeit dieser Art der Kopplung besteht beispielsweise darin, ein FPGA auf einer Einsteckkarte zu integrieren und diese an die PCI-Schnittstelle eines PC anzuschließen. Ein sol-

ches System wurde beispielsweise im Rahmen des RAGE-Projektes [BDH+97] implementiert.

Es soll an dieser Stelle nicht unerwähnt bleiben, dass es noch eine Reihe von rekonfigurierbaren Rechensystemen gibt, die in keine der drei genannten Kategorien fallen. Es sind vor allem FPGA-basierte *stand-alone* Mikroarchitekturen, welche sämtliche Steuerungs- und Berechnungsaufgaben ohne einen zusätzlichen fest verdrahteten Hauptprozessor eigenständig erledigen. Beispiele solcher Architekturen sind MATRIX [DeH96] und Xputer [HHW89, HHKW90]. Die im Rahmen dieser Arbeit entwickelte Mikroarchitektur enthält ebenfalls keinen Host-Prozessor, sondern bildet selbst den einzigen Hauptprozessor mit rekonfigurierbaren Eigenschaften. Daher kann sie auch dieser speziellen Kategorie zugeordnet werden.

2.2.3 Rekonfigurierbarkeit

Der Vorteil von FPGAs gegenüber anderen Hardware-Plattformen (wie fest verdrahteter Hardware, ASICs) liegt wie bereits erwähnt darin, dass sie aufgrund ihrer Flexibilität mehrfach für die Implementierung unterschiedlicher Anwendungen in Hardware benutzt werden können. Dafür werden die zu verschiedenen Anwendungen gehörigen Hardware-Lösungen in Form von Konfigurationen (auch Kontexte genannt) vorbereitet und je nach Bedarf in das FPGA geladen. Diesen Ladevorgang bezeichnet man als FPGA-Programmierung oder Rekonfiguration. Das Erstellen einer Konfiguration erfolgt in mehreren Entwicklungsschritten, die in Abschnitt 2.3 näher erläutert werden. Genauso wie ein fest verdrahteter Prozessor durch das Laden unterschiedlicher Programme eine Vielzahl von Anwendungen in Software ausführen kann, implementiert das FPGA verschiedene Anwendungen in Hardware durch das Laden unterschiedlicher Kontexte. Generell werden zwei Arten von Rekonfiguration unterschieden:

- **Statische Rekonfiguration**: Bei einer statischen Rekonfiguration wird meist bei jedem Systemstart, und zwar während der Initialisierungsphase, eine bestimmte Konfiguration geladen. Diese Konfiguration bleibt aktiv, bis das System erneut gestartet wird. Daraus

ergibt sich, dass lediglich eine Anwendung während einer Arbeitsperiode des Systems ausgeführt wird. Damit eine weitere Anwendung durch das rekonfigurierbare Rechensystem ausgeführt werden kann, muss das System neu gestartet werden. Eine erneute statische Konfiguration lädt den entsprechenden Kontext als Hardware-Lösung für die nächste Applikation in das FPGA.

- **Dynamische Rekonfiguration**: Die Rekonfiguration eines FPGA kann auch erfolgen, während eine Anwendung darauf ausgeführt wird. Dies geschieht, indem beispielsweise mehrere Konfigurationen im Speicher vorgehalten und nacheinander ohne erneuten Systemstart in das FPGA geladen werden. Der Ladevorgang kann durch den Haupt-Prozessor in Gang gesetzt werden, wenn das FPGA als Peripheriegerät oder als Co-Prozessor betrieben wird. Alternativ kann das FPGA so aufgeteilt werden, dass ein Teil davon den Ladevorgang übernimmt, während der Rest des FPGA als Ausführungseinheit dient. Im Allgemeinen spricht man von einer partiellen Rekonfiguration, wenn nur ein Teil des FPGA rekonfiguriert wird und der Rest unverändert bleibt. Dies ist allein bei dynamischer Rekonfiguration möglich, da bei statischer Rekonfiguration typischerweise das FPGA beim Systemstart vollständig neu beschrieben wird.

2.2.4 Programmierbarkeit

Während die Rekonfigurierbarkeit sich auf den Zeitpunkt des Ladens eines bestimmten Kontextes bezieht, geht es bei der Programmierbarkeit darum, wie ein oder mehrere Kontexte auf dem FPGA organisiert werden. Eine Möglichkeit besteht darin, dass immer zu einem bestimmten Zeitpunkt lediglich ein Kontext auf dem FPGA verfügbar ist (*single context*). Nur durch (statische) Rekonfiguration ist es möglich, einen Kontext durch einen anderen zu ersetzen und somit beispielsweise einen Task-Wechsel zu realisieren (siehe Abb. 2.6). Alternativ dazu können gleichzeitig mehrere Kontexte auf das FPGA geladen werden. Das FPGA wird derart aufgeteilt, dass jede einzelne Konfiguration einen eigenen Teilbereich auf dem FPGA be-

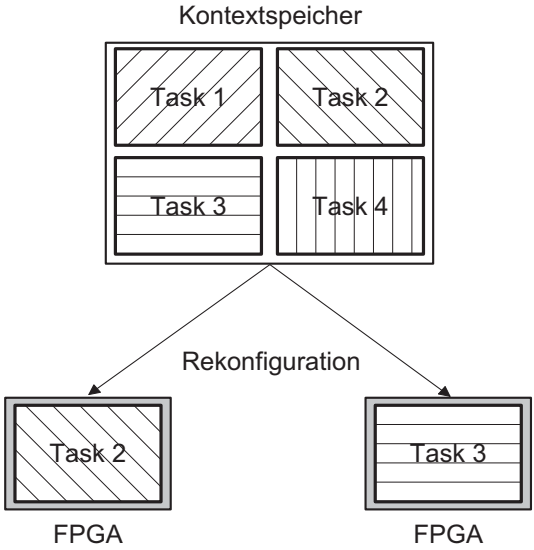

Abbildung 2.6: *Single Context FPGA*

legt. Mit der Annahme, dass mit jeder Konfiguration eine separate Task implementiert wird, kann somit ein echter Parallelbetrieb erreicht werden. Dies setzt allerdings voraus, dass das FPGA über entsprechende Hardware-Ressourcen verfügt, um mehrere Kontexte gleichzeitig aufzunehmen (siehe Abb. 2.7). Außerdem müssen die Tasks so aufgeteilt sein, dass keine unmittelbare Kommunikation zwischen verschiedenen Tasks erforderlich ist. Andernfalls müssen dafür zusätzliche Hardware-Ressourcen vorgesehen werden. Hierfür eignet sich besonders die Anwendung der partiellen Rekonfiguration mit einem entsprechenden Kommunikationsmechanismus (z.B. *Bus-Macro*). Es bietet sich sonst die Möglichkeit, die Kommunikation über einen gemeinsamen Speicher zu steuern.

Eine quasi-parallele Ausführung mehrerer Tasks wird auch erreicht, wenn dynamische Rekonfiguration mit einbezogen wird. Dabei wird jedem Kontext eine bestimmte Zeit zugeordnet, in der die entsprechende Hardware-Lösung ausgeführt wird. Wenn am Ende des festgelegten Zeitraums eine Task noch nicht vollständig abgearbeitet ist, so wird die aktuelle Konfi-

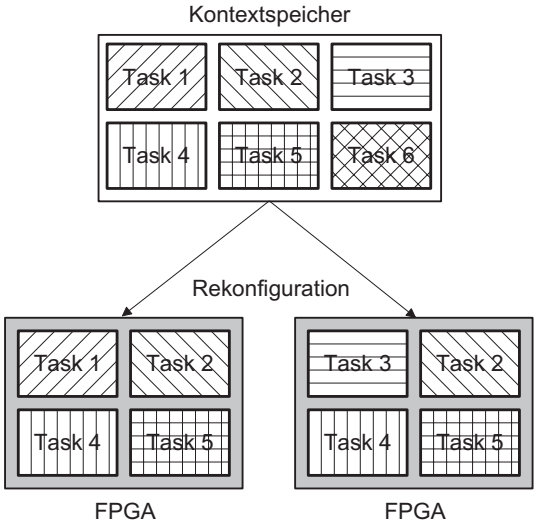

Abbildung 2.7: *Multi Context FPGA*

guration in den Speicher zurückgeschrieben und der nächste Kontext wird geladen. Diese Funktionsweise entspricht dem Prinzip des *Swapping* bei den Software-basierten Systemen.

2.3 Entwurfsmethodik FPGA-basierter Rechensysteme

Nach dem vorangegangenen Abschnitt können rekonfigurierbare Rechensysteme als Systeme verstanden werden, die teilweise oder vollständig auf rekonfigurierbarer Hardware (FPGAs) aufgebaut sind. Dabei erfüllt das FPGA als *Stand-Alone*-Baustein oder in Kombination mit anderen Hardware-Plattformen zwei Aufgaben: zum einen wird es für die Implementierung ausgewählter und meist rechenintensiver Applikationen verwendet, zum anderen ist sein Einsatz aufgrund der speziellen Eigenschaften für die Rekonfiguration des ganzen Rechensystems unumgänglich. Dieser Ab-

schnitt gewährt einen Einblick in die Entwurfsmethodik solcher FPGA-basierten Rechensysteme.

Im Allgemeinen geht es beim Hardware-Entwurf darum, eine spezifizierte Funktionalität auf eine fest verdrahtete oder rekonfigurierbare Hardware abzubilden. Dabei stehen zwei wichtige Gesichtspunkte im Vordergrund. Einerseits werden reale Hardware-Anwendungen zunehmend so komplex, dass der Entwurfsaufwand nicht mehr ohne eine gewisse Systematik und Rechnerunterstützung zu beherrschen ist. Dies ist z.B bei der Steuerungselektronik für große Systeme (Maschinen, Motoren, etc.) der Fall. Derartige komplexe Schaltungen können nicht mehr durch ein manuelles „Stöpseln" elementarer Bauelemente (Transistoren, Dioden, Widerstände, ...) auf einer Platine realisiert werden. Andererseits ist der Hardware-Entwurf mit den immer dichter integrierten Schaltungen (*Integrated Circuits*) überhaupt nicht mehr manuell zu bewerkstelligen. Außerdem müssen heutzutage sehr kurze Entwicklungszeiten eingehalten werden, um der Forderung nach einem schnellen *Time-To-Market* gerecht zu werden. Darüber hinaus wird von einem modernen Hardware-Entwurf - genau wie in allen anderen Branchen - mehr Leistungsfähigkeit in Form von Rechengeschwindigkeit, geringem Platzbedarf, reduziertem Stromverbrauch etc. verlangt.

Aus diesen Gründen lässt sich eine zuverlässige Entwicklung eines FPGA-basierten Systems nur mittels einer strukturierten Vorgehensweise durchführen. Daher werden zunehmend Design-Techniken wie Entwurfsablauf (*Design Flow*), modularer Aufbau, Abstraktionsebenen, rechnergestützter Entwurf (*Computer-Aided Engineering*) etc. eingesetzt. Abbildung 2.8 zeigt ein Modell von Entwurfssichten und Abstraktionsebenen als Y-Diagramm, das sich beim Entwurf integrierter Schaltungen etabliert hat [Wan98, Sik01, LWS94, Möl03].

Das Y-Diagramm betrachtet die Entwicklung integrierter Schaltungen aus drei verschiedenen Sichten (auch Domänen genannt). Demnach kann eine Schaltung über ihr Verhalten, ihre Struktur oder ihre Geometrie beschrieben werden. Mit der Verhaltensbeschreibung wird das System durch seine Reaktionen auf die Änderungen der Eingangssignale beschrieben. Alternativ dazu kann das zu entwickelnde System anhand seiner Bestandteile, aus de-

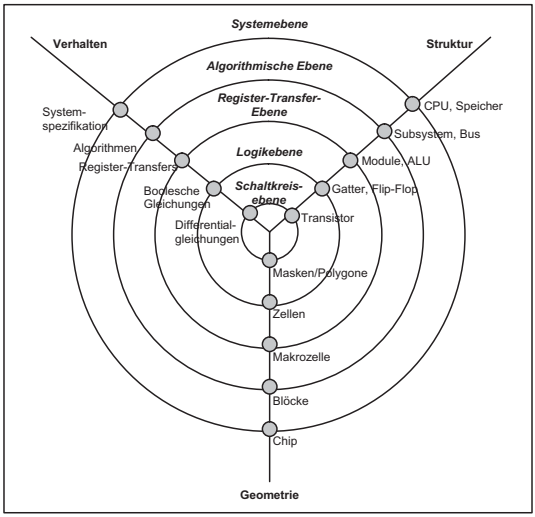

Abbildung 2.8: *Y-Diagramm nach Gajski-Walker [Wan98]*

nen es sich zusammensetzt, definiert werden. In diesem Fall handelt es sich um eine strukturelle Beschreibung. Eine dritte Möglichkeit besteht darin, mit Hilfe von realen elektronischen Bauelementen und einem Schaltplan die Implementierung durchzuführen. In der Praxis wird ein hardware-basiertes System nicht nur aus einer der drei Sichten entwickelt, sondern es wird oftmals je nach Abstraktionsebene zwischen den Sichten hin und her gesprungen. Auf diesen drei Entwurfswegen können bis zu fünf Abstraktionsebenen unterschieden werden. In Abbildung 2.8 werden sie durch die fünf Kreise dargestellt, wobei die Abstraktionsstufe vom äußeren zum inneren Kreis hin abnimmt.

- **Systemebene**: In der Systemebene wird das Systemverhalten sowie die Struktur festgelegt, und zwar ohne jeglichen Hinweis auf die Realisierungsmöglichkeiten. Das System wird auf dieser Ebene mit Hilfe von natürlicher Sprache und Skizzen beschrieben. Dabei beschränkt sich die Beschreibung auf die grundlegenden Funktionsblöcke des Systems wie z.B. Prozessorarchitektur, Speicher, Schnittstellen etc. Die

Wahl einer geeigneten Hardware-Plattform sowie eine erste Verteilung der Blöcke auf dem Chip werden auf dieser Ebene durchgeführt.

- **Algorithmenebene**: Auf dieser Ebene wird das Systemverhalten mittels Prozessen, Funktionen, Prozeduren, Ein-und Ausgangssignalen usw. programmtechnisch beschrieben. Die Funktionalitäten der aus einer Unterteilung der groben Funktionsblöcke resultierenden Subsysteme bilden die Grundlage der Implementierung.

- **Register-Transfer-Ebene**: Die RTL-Ebene (*Register-Transfer Level*) dient der Implementierung einzelner Funktionseinheiten durch kombinatorische und arithmetische Rechenoperationen. Außerdem werden in dieser Ebene Daten und Steuersignale zwischen den Funktionseinheiten und meistens synchron zu einem Taktsignal transferiert. Wie die Bezeichnung dieser Ebene erahnen lässt, werden die Daten in Registern gehalten und durch relativ kleine Funktionseinheiten wie Addierer, Multiplexer, Codierer usw. miteinander verknüpft. Anschließend werden die Ergebnisse aus diesen Verknüpfungen taktgesteuert von einem verarbeitenden Block zum nächsten weitergereicht.

- **Logikebene**: Auf der Logikebene werden kombinatorische und arithmetische Funktionen in elementare logische Gatter (*AND, OR, NAND, NOR, XOR,...*) umgesetzt. Zusätzlich wird aus dem vorangegangenen synchronen Datentransfer das zeitliche Verhalten der Schaltung abgeleitet. Dabei wird meistens auf herstellerabhängige Hardware-Bibliotheken zurück gegriffen, welche eine automatische Übersetzung einer RTL-Beschreibung in eine entsprechende logische Schaltung ermöglichen.

- **Schaltkreisebene**: Die niedrigste Abstraktionsebene bildet die Schaltkreisebene, die in der Praxis voll automatisiert abläuft. Auf dieser Ebene besteht das zu implementierende System lediglich aus elektronischen Bauelementen (Transistoren, Dioden, Widerstände, ...). Das zeitliche Verhalten wird nicht mehr diskret angegeben, sondern es wird aufgrund realer Kenndaten der Bauelemente in physikalischen Größen

ausgedrückt. Die gesamte integrierte Schaltung wird auf einem fest verdrahteten oder programmierbaren Chip realisiert.

Wie bei den verschiedenen Sichten des Y-Diagramms gilt ebenfalls für die Abstraktionsebenen, dass eine Implementierung nicht einer festen Abstraktionsebene zugeordnet werden kann. Vielmehr wechselt man während der Entwicklungsphase zweckmäßigerweise zwischen den verschiedenen Ebenen.

Neben dem Y-Diagramm mit den aufgezeigten Abstraktionsebenen kann zusätzlich ein geeigneter Ablaufplan dem Hardware-Entwurf zum bestmöglichen Ergebnis verhelfen. Einen solchen Entwurfsablaufplan (*Design Flow*), der heutzutage eine breite Anwendung findet, zeigt die Abbildung 2.9. Bei der technischen Umsetzung des in der vorliegenden Arbeit entwickelten Konzepts eines rekonfigurierbaren Prozessors wurde weitgehend nach diesem Entwurfsablaufplan vorgegangen.

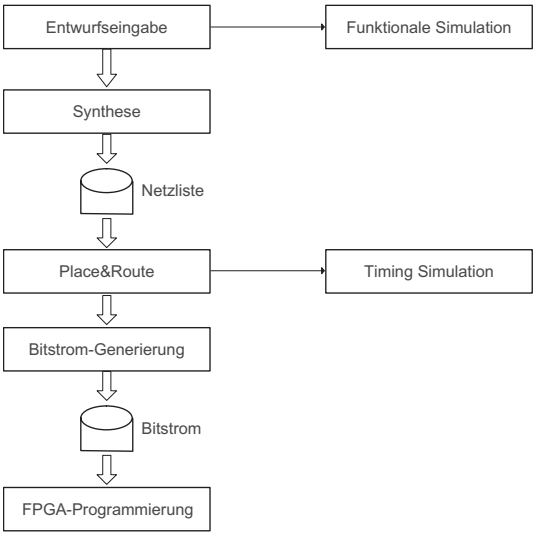

Abbildung 2.9: *Entwurfsablauf von FPGA-Schaltungen [Wan98, Sik01, LWS94]*

2.3.1 Entwurfseingabe

Der erste Schritt zur rechnergestützten Implementierung einer bereits spezifizierten Funktionalität in Hardware besteht darin, diese in geeigneter Art und Weise programmtechnisch umzusetzen. In Analogie zur Software-Entwicklung geschieht die Umsetzung mit Hilfe von Programmiersprachen, die sich besonders für die Hardware-Beschreibung eignen: die sog. *Hardware Description Languages* (HDLs). Die bekanntesten Vertreter dieser Klasse von Programmiersprachen sind die Sprachen *Very High Speed Integrated Circuits Hardware Description Language* (VHDL) und *Verilog*. In jüngster Zeit werden zunehmend neue Hardware-Programmiersprachen entwickelt, die sich stark an Software-Programmiersprachen anlehnen. Die meisten davon werden aus der Programmiersprache „C“ durch Hinzunahme von Hardware-Konstrukten hergeleitet (*SystemC, Handel C, SpecC* etc.). Das verfolgte Ziel dabei ist es, den Einstieg von Software-Entwicklern in die Hardware-Entwicklung zu erleichtern. Diese Notwendigkeit ist aus der Tatsache entstanden, dass immer mehr moderne eingebettete Systeme (Mobil-Telefone, *Pocket* PCs, Spielkonsolen, aktive Netzwerk-Komponenten, Navigationssysteme, ...) keine strenge Trennung mehr zwischen Hardware- und Software-Entwicklung erlauben. Vielmehr muss heutzutage ein Entwickler mit beiden Entwicklungsgebieten vertraut sein (*Hardware/Software Codesign*).

Neben den reinen Programmiersprachen bieten die neuesten Entwicklungssysteme zusätzlich Werkzeuge zur grafischen Entwurfseingabe an. Derartige grafische Werkzeuge umfassen u.a Schaltpläne, Zustandsdiagramme, Ablaufpläne und Wahrheitstabellen. Sie helfen ihrerseits solchen Entwicklern, die sich jahrelang z.B. nur mit Schaltplänen beschäftigt haben, den Sprung in die rechnergestützte Entwicklung hochintegrierter Systeme zu schaffen.

Unabhängig vom benutzten Eingabemittel geht es primär auf dieser Ebene darum, eine bestimmte Funktionalität so zu beschreiben, dass sie durch die nachfolgenden Entwurfsschritte in Hardware umgesetzt werden kann. Einige Entwicklungssysteme stellen bereits in dieser ersten Phase zusätzlich zu den reinen Eingabe-Tools weitere Programme (z.B. Compiler) für die

Syntaxprüfung sowie andere Plausibilitätsprüfungen zur Verfügung. Damit ist der Entwickler in der Lage, schon in der Frühphase der Implementierung einfache Beschreibungsfehler zu bereinigen.

2.3.2 Funktionale Simulation

Das alleinige Beschreiben der Funktionalität stellt noch lange nicht sicher, dass diese auch programmtechnisch richtig umgesetzt wird. Wichtiger noch muss die Beschreibung den funktionalen Anforderungen genügen. Erfahrungsgemäß sind mehrere Iterationsschritte notwendig, bis eine Beschreibung genau der gewünschten Funktionalität entspricht. Dies gilt gleichermaßen für die Software- und die Hardware-Entwicklung. Die funktionale Simulation dient daher dem Überprüfen, ob das Beschriebene (in welcher Form auch immer) auch tatsächlich die spezifizierte Funktion erfüllt. Dafür wird die syntaktisch korrekte Entwurfseingabe in einen Hardware-Simulator geladen (z.B. *ModelSim*). Durch einen Stimulus wird das Systemverhalten anhand seiner Reaktionen untersucht. Entspricht das Systemverhalten der festgelegten Funktionalität, so wird die nächste Phase eingeleitet. Andernfalls wird die Hardware-Beschreibung solange verbessert, bis die funktionale Simulation ein fehlerfreies Verhalten des Systems nachweist.

2.3.3 Synthese

In dieser Phase der Hardware-Entwicklung wird erst die sprachliche oder grafische Beschreibung in logische Hardware-Komponenten (siehe Abb. 2.8, Logikebene) umgesetzt. Alle sprachlichen Konstrukte sowie alle grafischen Objekte werden, soweit es möglich ist, in logische Gatter übersetzt. Beispielsweise wird aus der funktionalen Beschreibung $c = a + b$ ein Addierer *synthetisiert*.

Der Entwurf wird in eine einheitliche und herstellerunabhängige Schaltungsnotation (*Electronic Design Interchange Format*, EDIF) überführt und als übertragbare Netzliste (*Netlist*) gespeichert. Eine Besonderheit des Synthese-Prozesses soll an dieser Stelle erwähnt werden: Das Ergebnis aus der Synthese hängt maßgeblich vom Beschreibungsstil und vom verwendeten Synthese-

Tool ab. Dies bedeutet, dass nicht alles, was funktional richtig beschrieben und simuliert wird, auch optimal in Hardware umgesetzt wird. Tatsächlich lassen sich einige Sprachkonstrukte überhaupt nicht synthetisieren. Beispielsweise bildet der synthesefähige Sprachschatz von VHDL eine echte Teilmenge des gesamten VHDL-Sprachumfangs. Außerdem hat sich gezeigt, dass unterschiedliche Synthese-Werkzeuge aus der gleichen Hardware-Beschreibung unterschiedliche Hardware-Implementierungen erzeugen. Weiterhin werden einige Konstrukte durch ein bestimmtes Tool fehlerfrei synthetisiert, während sie durch ein anderes Synthese-Tool verworfen werden. Diese Unterschiede sind darauf zurückzuführen, dass sich hinter den Synthese-Werkzeugen herstellerabhängige Bibliotheken verbergen, mit deren Hilfe die sprachliche oder grafische Beschreibung in entsprechende Gatter-Funktionen übersetzt wird. Daher sollte die Wahl des Synthese-Tools zweckmäßig alle diese Erkenntnisse berücksichtigen. In der Tat handelt es sich bei der Synthese um eine Konkatenation mehrerer NP-vollständiger Probleme, welche von den verschiedenen Tools-Herstellern unterschiedlich heuristisch gelöst werden.

2.3.4 Place&Route

Die synthetisierte Netzliste wird weiter verarbeitet, um die zu implementierende Funktionalität auf einer bestimmten Hardware-Plattform zu realisieren. In dieser Entwurfsphase wird zunächst die allgemeine EDIF-Beschreibung in eine technologie-abhängige Notation umgesetzt. Dies bedeutet, dass die durch ein Synthese-Tool erstellten Gatter-Funktionen in spezifische Basiszellen eines ausgewählten FPGA übersetzt werden (*Technology Mapping*). Außerdem können einige Vorgaben (*Constraints*) bezüglich Platzierung (*Area Constraints*) und Zeitverhalten (*Timing constraints*) mit einbezogen werden. Unter Einhaltung solcher Vorgaben werden anschließend die nun erhaltenen technologieabhängigen Hardware-Komponenten auf die Ziel-Plattform platziert (*Floorplanning*) und miteinander verbunden (*Routing*). Dabei entsteht ein echtes Abbild des FPGA mit der implementierten Funktionalität.

Zusätzlich kann auf dieser Ebene ein nahezu reales Zeitverhalten des Systems durch die sog. *Back Annotation* gewonnen werden. Es kann nämlich aus dem Abbild des FPGA ein simulationsfähiges Modell der implementierten Funktion mit dem dazugehörigen Zeitverhalten zurückgewonnen werden. Die daraus resultierenden Daten ermöglichen eine *Timing*-Simulation, die genau das Zeitverhalten des Systems widerspiegelt. Dadurch soll überprüft werden, ob das System auch den spezifizierten zeitlichen Anforderungen genügt. Dies bildet einen der wichtigsten Vorteile von FPGAs gegenüber anderen Hardware-Platformen.

Eine letzte Verarbeitung auf dieser Ebene besteht darin, das Baustein-Abbild mit der implementierten Funktionalität in ein Dateiformat umzuwandeln, das auf den physikalischen Baustein übertragen wird. Das Ergebnis ist ein Bitstrom (*Bitstream*), der einen bestimmten Baustein mit einer darauf zu implementierenden Funktionalität vollständig beschreibt.

2.3.5 Baustein-Programmierung

Der durch die vorangegangenen Entwurfsschritte entstandene Bitstrom muss auf den Ziel-Baustein übertragen werden. Dafür wird meistens das JTAG-Protokoll (*Joint Test Action Group*) verwendet. Eine Kabelverbindung beispielsweise bildet die entsprechende Schnittstelle, durch die der erstellte Bitstrom von dem Entwicklungssystem auf das FPGA übertragen wird. Das gleiche Verfahren kann auch dazu benutzt werden, um die Daten zurück zu gewinnen, die den Zustand eines FPGA beschreiben (*Readback*). Solche Daten dienen meistens zu Verifikationszwecken.

Der Bitstrom enthält, wie bereits erwähnt, das gesamte Abbild des Ziel-Bausteins. Daher kommt es für größere Bausteine oft vor, dass die Zeit, bis der ganze Bitstrom vollständig übertragen wird, im Minuten-Bereich liegt. Es ist mittlerweile möglich, mit Hilfe spezieller Tools einen umfangreichen Bitstrom in kleinere Dateien aufzuteilen. Zusammen mit der Anwendung der partiellen Rekonfiguration lassen sich damit durch Überlappung der Konfigurationszeit und der Verarbeitungszeit die Wartezeiten stark reduzieren. Darüber hinaus ist damit die Möglichkeit gegeben, die Funktionsweise

des FPGA und somit der implementierten Hardware im laufenden Betrieb zu beeinflussen. Die Umsetzung des in der vorliegenden Arbeit dargestellten Konzeptes eines dynamisch rekonfigurierbaren Prozessors basiert auf dieser Technik der partiellen Rekonfiguration.

2.4 Mikroprozessortechnik

2.4.1 Prozessorarchitektur

Die Begriffe Prozessorarchitektur und Mikroarchitektur bzw. Prozessor und Mikroprozessor werden oftmals verwechselt. Diese Begriffe sollen an dieser Stelle präzisiert werden.
Eine **Prozessorarchitektur** umfasst primär einen festgelegten Satz von Befehlen. Das ist der Befehlssatz, mit Hilfe dessen sich ein ausführbares Programm herstellen lässt. Zusätzlich definiert sie alle zulässigen Befehlsformate, welche für die Darstellung eines gültigen Befehls verwendet werden können. Weitere Festlegungen einer Prozessorarchitektur sind die Adressierungsarten für Speicherzugriffe, die unterstützten Datenformate sowie die Art der Datenhaltung innerhalb des Prozessors (Akkumulator, Stack oder Register). Gleichwertige Begriffe für eine Prozessorarchitektur sind ***Instruction Set Architecture*** **(ISA)** und **Programmiermodell**. Letzteres unterstreicht noch mehr die Bedeutung, dass es sich um den für den Programmierer sichtbaren Teil eines Prozessors handelt [FL98, BU02, vRU99, PH96].

Eine der bekanntesten Prozessorarchitekturen ist die *Intel Architecture-32* (IA-32), auf der die marktführenden Intel- Prozessoren, aber auch die AMD-Prozessoren basieren. Sie gehört zu der Familie der *Complex Instruction Set Computer* (CISC), welche sich durch einen umfangreichen Befehlssatz auszeichnet (wird aufgrund der Pipeline-Implementierung oft als RISC verkauft). CISC-Architekturen verfügen über komplexe Befehle mit zahlreichen Befehlsformaten, deren Decodierung oft nur mit Hilfe von Microcodes zu bewältigen ist (Mikroprogrammierung). Intel- und AMD-Prozessoren

umgehen die zeitaufwendige Mikroprogrammierung durch erweiterte Implementierungstechniken, z.B. durch die Trennung der Ausführungspfade einfacher und komplexer Befehle voneinander.

Weitere verbreitete Prozessorarchitekturen sind u.a. die *Microprocessor Without Interlocking Pipeline Stages* (MIPS), die *Scalable Processor ARChitecture* (SPARC) und die *Advanced* RISC *Machine* ISA (ARM ISA). Alle diese Architekturen basieren auf dem *Reduced Instruction Set Computer* (RISC)-Prinzip, was vor allem einen reduzierten, aber vollständigen Befehlssatz beinhaltet. RISC-Prozessorarchitekturen bevorzugen einfache Befehle und Befehlsformate, die dank Pipelining schnell und in einem relativ kurzen Zeitfenster ausgeführt werden können (typischerweise innerhalb eines Taktzyklus). Darüber hinaus operieren RISC-Befehle im Gegensatz zu CISC-Befehlen, welche unmittelbare Verknüpfungen auf Speicherinhalte ermöglichen, ausschließlich auf Register-Inhalten. Somit wird die Verarbeitungszeit deutlich reduziert, da während der Befehlsabarbeitung nicht auf die langsame Verbindungsstruktur (Systembus) zwischen Prozessor und Arbeitsspeicher zugegriffen wird.

Eine **Mikroarchitektur** dagegen bezeichnet die Implementierung einer bestimmten Prozessorarchitektur in Hardware. Dabei können für die Implementierung der gleichen ISA unterschiedlichste Techniken eingesetzt werden. Es entstehen daraus verschiedene Arten und Generationen von Prozessoren, denen dieselbe Prozessorarchitektur zugrunde liegt. Beispielsweise verwendete Intel bereits in den 80er Jahren die IA-32 (auch Intel x86-Architektur genannt) in den damaligen Intel x86 Prozessoren. Die gleiche Architektur wird heute noch (2005) in aktuellen Prozessoren (Intel Pentium 4 Prozessor) benutzt, wobei lediglich im Laufe der Jahre der Befehlssatz um weitere zweckmäßige Befehle erweitert wurde. Die bekanntesten Erweiterungen umfassen:

- *Multimedia Extension* (MMX)-Befehle für Multimedia-Anwendungen
- *Streaming* SIMD (*Single Instruction Multiple Data) Extension 1-3* für aufwendige Berechnungen.

Außerdem können Prozessorhersteller eine bestimmte Prozessorarchitektur lizenzieren (wenn sie nicht wie die SPARC-Architektur frei verfügbar ist) und diese anhand einer proprietären Mikroarchitektur als eigenen Prozessor vermarkten. Somit ist es möglich, dass die Firma AMD eigene Prozessoren entwickelt, die auf der IA-32-Architektur ihres Mitbewerbers Intel basieren. In gleicher Weise ist Intel selbst genauso wie viele andere Firmen Lizenznehmer der ARM-Architektur, die sich besonders für stromsparende, eingebettete Systeme eignet und zunehmend eine breite Anwendung findet.

Dank der wachsenden Chip-Integration bestehen heutige **Mikroprozessoren** neben dem Prozessorkern (*Central Processing Unit*, CPU) zusätzlich aus einer Vielzahl von Komponenten wie der *Memory Management Unit* (MMU), einer DMA-Einheit, verschiedenen mehrschichtigen *Cache*-Speichern usw.

2.4.2 Grundstruktur eines Mikroprozessorsystems

Die Mehrzahl heutiger Mikroprozessoren basieren immer noch auf dem von-Neumann-Prinzip. Es beschreibt die grundsätzlichen Komponenten eines digitalen Rechensystems, die Funktionsweise der einzelnen Komponenten sowie das Zusammenwirken aller Rechner-Komponenten. Daher wird auch der Begriff des von-Neumann-Rechners korrekterweise verwendet, wie in der Abbildung 2.10 dargestellt [FL98, BU02, MZ97].

Der von-Neumann-Rechner besteht aus folgenden Komponenten:

- Prozessor (CPU): Die Zentraleinheit hat die Aufgabe der Steuerung sämtlicher Abläufe innerhalb des Rechners. Das Steuerwerk bildet die dafür notwendige Subkomponente und ist zusätzlich für die Decodierung der Befehle zuständig. Die decodierten Befehle erzeugen Steuersignale, die an das Rechenwerk übermittelt und dort ausgeführt werden. Dies geschieht mit Hilfe der darin enthaltenen arithmetischen und logischen Einheit (ALU). Das Ergebnis einer gerade ausgeführten Rechenoperation wird in einen Zwischenspeicher (Akkumulator oder Register) abgelegt. Durch einen weiteren Befehl (Transportbefehl)

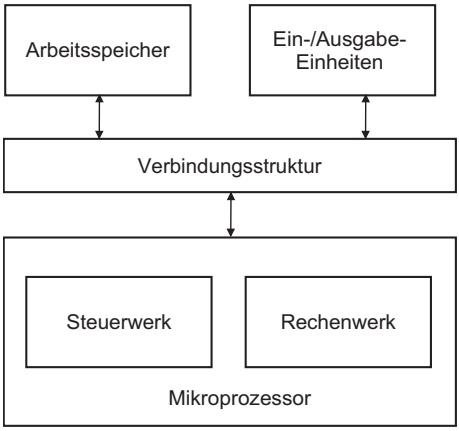

Abbildung 2.10: *Struktur eines von-Neumann-Rechners*

gelangt das Ergebnis in den Arbeitsspeicher zur endgültigen Speicherung.

- Arbeitsspeicher: Diese Komponente speichert ausführbare Programme in Form von Befehlen und Daten ab. Das Steuerwerk stellt den Inhalt des Befehlszählers (*Program Counter*, PC) als Speicheradresse zur Verfügung. Diese wird im Arbeitsspeicher decodiert und aus der dazu gehörigen Speicherzelle wird ein Befehl oder ein Datum in die CPU geladen. Darüber hinaus werden die Ergebnisse der Programmausführung in diesen Speicher zurückgeschrieben.

- Ein- und Ausgabesystem: Damit Daten zum Prozessor gelangen, dort durch ein Programm abgearbeitet und anschließend die Ergebnisse an den Benutzer ausgegeben werden können, ist ein Ein-/Ausgabesystem notwendig. Somit kann das digitale Rechensystem benutzer-relevante Aufgaben übernehmen und zweckmäßig erledigen.

- Verbindungsstruktur: Alle Rechner-Komponenten kommunizieren miteinander über eine Verbindungsstruktur, welche im Allgemeinen durch ein internes Bussystem oder einen Multiplexer realisiert wird.

Zusätzlich zu der aufgezeigten Grundstruktur definiert das von-Neumann-Prinzip das Operationsprinzip dieses Rechnertyps. Demnach werden Befehle streng sequentiell nach einem zwei-Phasen-Schema ausgeführt. Während der ersten Phase (Befehlsholphase, *fetch stage*) wird der zum aktuellen PC gehörige Befehl aus dem Arbeitsspeicher geholt. In der darauf folgenden Phase (Ausführungsphase, *execute stage*) wird die im Befehl kodierte Rechenoperation ausgeführt. Außerdem zeichnet sich der von-Neumann-Rechner durch seine einfache Hardware-Implementierung aus. Dies war vor allem zu seiner Entstehungszeit Ende der vierziger Jahre, als die Hardware-Kosten noch sehr hoch waren, ein wichtiger Punkt. Durch die stetig fallenden Kosten für Hardware und insbesondere durch die steigende Chip-Integration sind im Laufe der Jahre beträchtliche Weiterentwicklungen des ursprünglichen von-Neumann-Prinzips möglich geworden. Eine der wichtigsten Änderungen bildet die Einführung der getrennten Speicherung und Zuführung von Befehlen und Daten. Daraus resultiert die sog. **Harvard-Architektur**. Sie entschärft den im von-Neumann-Rechner vorhandenen Engpass infolge des hohen Daten- und Befehlstransfers zwischen Prozessor und Arbeitspeicher (auch **„von-Neumann-Flaschenhals“** genannt).

Eine weitere Entwicklung besteht in der Aufhebung der strengen Sequentialität der Befehlsabarbeitung. Durch die Einführung der Pipeline-Technik wird ermöglicht, dass ein Befehl gestartet wird, bevor die vorangegangenen Befehle vollständig abgearbeitet worden sind. Diese sowie einige der in modernen Prozessoren implementierten Techniken und die damit verbundenen Fragestellungen werden in den nachfolgenden Abschnitten näher erläutert. Für eine ausführliche Behandlung dieses hoch komplexen Gebiets der Mikroprozessortechnik sei auf die einschlägige Literatur verwiesen ([Bär02, BU02, PH96, FL98, OV03]).

2.4.3 Pipelining

Pipelining gehört mittlerweile zu den etablierten Techniken zur Implementierung moderner Mikroprozessoren und dient vorrangig zur Steigerung des Befehlsdurchsatzes. Durch eine Überlappung der Verarbeitung aufeinander folgender Befehle werden durchschnittlich mehr Befehle pro Zeiteinheit

ausgeführt als ohne Pipelining (*Instructions Per Cycle*, IPC > 1). Voraussetzung dafür ist, dass die Ausführung eines Befehls in mehrere Schritte (Pipeline-Phasen, Pipeline-Stufen) aufgeteilt wird. Die Ergebnisse der verschiedenen Abarbeitungsphasen werden dabei synchron nach einem einheitlichen Takt von einer Phase zu der nächsten mit Hilfe von zwischengeschalteten Registern weitergereicht [BU02, OV03].

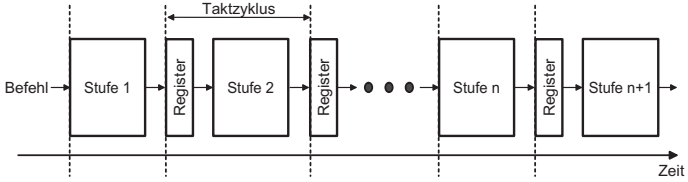

Abbildung 2.11: *Pipelining*

Die benötigte Verarbeitungszeit für die langsamste Pipeline-Phase wird als Maschinenzyklus, Taktzyklus oder einfach Takt bezeichnet [BU02, PH96, vRU99]. Im Idealfall benötigt ein Befehl genau k Takte, um durch k Pipeline-Phasen abgearbeitet zu werden. Für n Befehle wären dementsprechend $n * k$ Takte notwendig, wenn die Vorteile der Pipeline-Technik außer Acht bleiben. Tatsächlich lässt sich die Verarbeitungszeit in einer Pipeline durch eine einfache Addition der folgenden zeitlichen Komponenten bestimmen [BU02, vRU99]:

- Es werden k Taktzyklen für die Einschwingzeit benötigt, um die Pipeline vollständig mit Befehlen zu füllen.
- Die restlichen $n - 1$ Befehle benötigen idealerweise $n - 1$ Taktzyklen, falls keine Sprungbefehle oder sonstige Hindernisse die Pipeline zum Halten (*Stall)* zwingen (vgl. Absch. 2.4.4).

Daraus ergibt sich eine durch die Pipeline-Technik hervorgerufene Beschleunigung S (*Speedup*), die sich nach der Formel 2.1 berechnen lässt:

$$S = \frac{n * k}{n + k - 1} = \frac{k}{\frac{k}{n} + 1 - \frac{1}{n}} \tag{2.1}$$

Für ein sehr großes n entspricht die Beschleunigung S einer Pipeline der Anzahl k ihrer Phasen.

Die Anzahl der Pipeline-Phasen ist nicht festgelegt und hängt von den jeweiligen Mikroarchitekturen ab. Dennoch gliedert sich in Anlehnung an die eher theoretische DLX-Architektur [PH96] die Abarbeitung eines Befehls bei den meisten Mikroprozessoren in fünf Pipeline-Phasen [Bär02, BU02, OV03]:

1. Befehlsholphase (*Fetch Stage*, IF)
2. Decodierphase (*Decode Stage*, ID)
3. Ausführungsphase (*Execution Stage*, EX)
4. Speicherzugriffsphase (*Memory Stage*, MEM)
5. Rückschreibphase (*Write Back Stage*, WB)

Die Abbildung 2.12 veranschaulicht die Ausführung aufeinander folgender Befehle durch die aufgezeigten fünf Pipeline-Phasen.

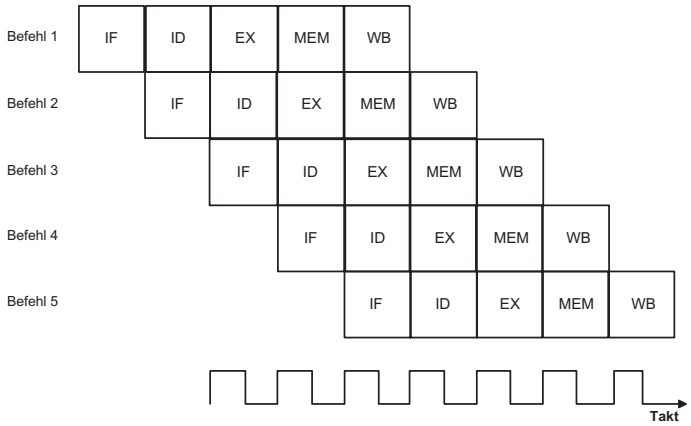

Abbildung 2.12: *Pipeline-Phasen*

- **Befehlsholphase:**
  Während der ersten Phase der Befehlsabarbeitung (IF, auch Bereitstellungsphase genannt) wird ein auszuführender Befehl aus dem Speicher geholt. Dafür überträgt die Fetch-Einheit den Inhalt des Befehlszählers zum Arbeitsspeicher und erhält ein entsprechendes Befehlswort zurück. Anschließend wird der aktuelle PC inkrementiert und

zeigt somit auf den nächsten auszuführenden Befehl. Die Ausführung eines vorangegangenen Sprungbefehls kann gegebenenfalls den Befehlszähler verändern.

- **Decodierphase:**
  In der darauf folgenden Phase (ID) findet die Befehlsdecodierung statt. Da eine Prozessorarchitektur oftmals eine Vielzahl von Befehlsformaten zulässt, muss das gerade aus dem Speicher geholte Befehlswort entschlüsselt werden. Daraus resultieren die auszuführende Operation und die Adressen der dafür benötigten Operanden. Bei einer **Load/Store-Architektur** befinden sich die Operanden in Prozessor-Registern, in die sie zuvor durch Lade-Operationen transportiert werden müssen. In dieser Phase werden die Operanden aus den Registern gelesen und zusammen mit dem Operationsteil des Befehls für die nachfolgenden Schritte bereitgestellt. Spezielle Architekturen lassen es zu, dass im Speicher befindliche Operanden auch direkt von dort aus verknüpft werden können.

- **Ausführungsphase:**
  In der dritten Phase (EX) wird die aus dem Befehlswort decodierte Operation ausgeführt. An dieser Stelle stehen eine ganz bestimmte Operation, die dafür notwendigen Operanden und sonstige Steuersignale zur Verfügung. Daraufhin wird die ausstehende Rechenoperation auf die mitgelieferten Daten angewendet. Für *Load*- und *Store*-Operationen werden die effektiven Adressen für anschließende Speicherzugriffe berechnet.

- **Speicherzugriffsphase:**
  Load- und *Store*-Operationen benötigen zusätzlich eine vierte Phase (MEM). In dieser Phase werden Daten aus dem Speicher in den Laderegistern zwischengespeichert bzw. Daten aus den prozessorinternen Registern in den externen Datenspeicher geschrieben. Außerdem wird diese Phase für bedingte Sprungbefehle benötigt, die in Abhängigkeit eines errechneten Ergebnisses zu einem Sprung führen.

- **Rückschreibphase:**
  Während der fünften und letzten Phase (WB) wird das Ergebnis der Ausführungsphase bzw. des Speicherzugriffs aus einem temporären Register in das im Programm angegebene Ziel-Register (*Destination-Register*) geschrieben.

In jüngster Zeit werden zunehmend Prozessoren mit immer mehr Pipeline-Phasen implementiert mit dem Ziel, den Befehlsdurchsatz weiter zu erhöhen. Beispielsweise erzielt Intel mit seinem jüngsten Mikroprozessor, dem Pentium 4E Prescott (2004), eine erkennbare Leistungssteigerung nicht zuletzt dank der beträchtlichen Länge seiner Pipeline mit 32 Phasen.

2.4.4 Daten- und Befehlsabhängigkeiten

In einer streng sequentiellen Befehlsausführung nach dem von-Neumann-Prinzip stellen Daten- und Befehlsabhängigkeiten keine besondere Herausforderung dar. Bei dieser Art der Ausführung wird ein Befehl vollständig ausgeführt, bevor die Ausführung des nächsten Befehls gestartet wird. Bei einer Pipeline-Ausführung jedoch können sie zu Konfliktssituationen führen, die sog. ***Pipeline-Hazards*** (Pipelinehemmnisse [Bär02]). In ungünstigen Fällen wird die Pipeline dadurch zum Halten gezwungen und damit die gesamte Ausführungsgeschwindigkeit stark beeinträchtigt.

Generell können drei Arten von Pipeline-Hazards auftreten [vRU99, BU02, PH96, OV03, FL98]:

- **Strukturelle Hazards**
  Strukturelle Hazards entstehen, wenn die zeitliche Überlappung der Verarbeitung mehrerer Befehle mehr Hardware-Ressourcen verlangt als tatsächlich vorhanden sind. Dies ist z.B. der Fall, wenn zwei Befehle gleichzeitig eine Mikroarchitektur-Komponente beanspruchen, welche aber nur eine exklusive Nutzung zulässt. In solchen Fällen muss im Allgemeinen einer der beiden Befehle warten, bis die benötigte Hardware-Ressource freigegeben wird. Eine derartige Konfliktsitua-

tion führt zwangsläufig zu einem Wartezyklus der ganzen Pipeline (*Pipeline Stall*).

- **Daten-Hazards**
  Daten-Hazards werden durch aufeinander folgende Lese- und/oder Schreibzugriffe auf die selben Prozessor-Register verursacht. Ausgehend von einem Befehl i und seinem unmittelbaren Nachfolger $i+1$ können folgende Fälle eintreten:

  1. Ein Operand für den Befehl $i+1$ wird durch einen Lesezugriff auf ein Register erhalten, welches zugleich als Ziel-Register für den Befehl i dient. Wenn beide Befehle i und $i+1$ sich gleichzeitig in der Pipeline befinden, so kann es vorkommen, dass der Befehl $i+1$ einen Registerinhalt erhält, der noch nicht durch den vorhergehenden Befehl i aktualisiert worden ist. Diese Art von Konflikt wird als *Read After Write* (RAW)-Hazard oder *True Data Dependence* bezeichnet. Das nachstehende Beispiel verdeutlicht den Sachverhalt:

     ```
     i:   ADD R1, R2, R3    (R1 = R2 + R3)
     i+1: SUB R4, R1, R5    (R4 = R1 - R5)
     ```

     In diesem kleinen Beispiel benötigt die Substraktion den aktuellen Inhalt von $R1$ aus der vorherigen Addition. Bei einer Pipeline-Ausführung beider Befehle unmittelbar hintereinander würde der Befehl $i+1$ mit falschen Daten operieren. Abhilfe schaffen zahlreiche Software- und Hardware-Maßnahmen zur Erkennung und Lösung solcher Abhängigkeiten (*Delayed Load*, *Pipeline Interlock* etc.) Eine mögliche Hardware-Lösung besteht darin, das Ergebnis einer Operation einem in der Pipeline befindlichen Befehl zu Verfügung zu stellen, bevor es in das Ziel-Register geschrieben wird. Diese Technik nennt sich *Forwarding*, *Bypassing* oder *Short-Circuiting*.

  2. Ein weiterer Konfliktfall entsteht, falls der Befehl i zuerst einen Operanden aus einem Register liest, welches durch den nachfolgenden Befehl $i+1$ als Ziel-Register beschrieben wird. In un-

günstigen Fällen kann es dann vorkommen, dass der Befehl $i+1$ in das Register schreibt, bevor der vorherige Wert gelesen wird. Damit würde der Befehl i den falschen Wert des gemeinsamen Registers erhalten. Daraus resultiert ein sog. *Write After Read* (WAR)-Hazard oder *Anti-Dependence*.

```
i:    ADD R4, R1, R3     (R4 = R1 + R3)
i+1: MULT R3, R5, R6     (R3 = R5 * R6)
```

In diesem Beispiel schreibt der Befehl $i+1$ sein Ergebnis aus der Multiplikation in das Register $R3$ zurück. Die vorherige Addition liest gerade dasselbe Register $R3$. Wenn die Reihenfolge der Lese- und Schreibzugriffe durch die Pipeline geändert wird, so entsteht ein WAR-Hazard. Daher soll möglichst gewährleistet werden, dass ein Befehl erst sein Ergebnis in ein Register schreibt, nachdem alle vorhergehenden Befehle vollständig ausgeführt worden sind (*In-Order Completion*). Diese Problematik entsteht vor allem bei superskalaren sowie VLIW-Mikroprozessoren (siehe Abschn. 2.4.5).

3. Die dritte Möglichkeit besteht darin, dass die beiden aufeinander folgenden Befehle ihre Ergebnisse in dasselbe Register schreiben wollen. In diesem Fall handelt es sich um ein *Write After Write* (WAW)-Hazard oder *Output Dependence*.

```
  i:   LOAD R2, !R5
i+1: MULT R2, R5, R6
```

Im aufgezeigten Beispiel dient das Register $R2$ als Ziel-Register für beide Befehle i und $i+1$. Nach der korrekten Ausführung beider Befehle befindet sich im Register $R2$ das Ergebnis des Befehls $i+1$. Es ist durchaus möglich, insbesondere bei superskalaren Prozessoren, dass das Ergebnis des Befehls $i+1$ in $R2$ geschrieben wird, während die Ausführung des Befehls i noch nicht abgeschlossen ist (vgl. Abschn. 2.4.5). In solchen Fällen würde die Ausführung eines nachfolgenden Befehls, welcher den Inhalt von $R2$ verwendet, mit einem falschen Operand erfolgen. Diese Art von Konflikten wird ebenfalls durch *In-Order Completion* gelöst.

- **Kontroll-Hazards**
  Sprungbefehle beeinflussen maßgeblich den Programmablauf. Für eine Ausführung ohne Pipeline ist dies nicht weiter tragisch. Dagegen wird die Leistung einer Pipeline-Implementierung stark durch Sprungbefehle beeinträchtigt. Dies ist vor allem darauf zurückzuführen, dass bei einem bedingten Sprungbefehl die Sprungadresse während der *Execute Phase* berechnet wird und erst bei der anschließenden MEM-Phase der *Program Counter* neu gesetzt wird. In einer fünfstufigen Pipeline, wie sie z.B. in Abb. 2.12 dargestellt ist, befinden sich zum Schluss der MEM-Phase bis zu vier Befehle in der Pipeline. Falls ein bedingter Sprungbefehl tatsächlich die Programmausführung an einem unvorhergesehenen Sprungziel fortsetzt, so müssen drei Befehle, die sich bereits in der Pipeline befinden, verworfen werden. Dies ist in der Tat ein Leistungsverlust, der umso größer wird, je mehr Pipeline-Phasen implementiert werden. Abhilfe schaffen zahlreiche Software- und Hardware-Techniken zur Sprungvorhersage (*Branch Prediction*), die in der einschlägigen Literatur ausführlich behandelt werden.

2.4.5 Superskalartechnik

Mit der stetig steigenden Chip-Integration eröffnen sich neben dem Pipelining noch weitere Möglichkeiten, leistungsfähige Prozessoren zu entwickeln. Die wachsende Anzahl von Transistoren, die auf immer kleinerer Chip-Fläche integriert werden, erlaubt die Implementierung zusätzlicher Funktionseinheiten auf dem Prozessor-Chip. Somit wurde seit den 90er Jahren eine Mikroprozessor-Technik entwickelt, welche auf einer parallelen Ausführung mehrerer Befehle pro Taktzyklus in einer Pipeline-Phase beruht.

Im ursprünglichen Sinne bedeutet der Begriff „superskalar“, dass aus einem konventionellen linearen Befehlsfluss taktweise mehrere Befehle gleichzeitig in die Ausführungsphase geleitet werden (mehrfache Zuordnung, *Multiple Issue*) [BU02, vRU99, Bär02, PH96, FL98]. Die Superskalartechnik basiert auf der aufgezeigten Pipeline-Technik mit dem wesentlichen Unterschied, dass hier mehrere Befehle gleichzeitig aus dem Speicher abgeholt,

decodiert und ausgeführt werden. Eine der Grundvoraussetzungen einer solchen parallelen Ausführung bildet das Vorhandensein einer Vielzahl von Ausführungseinheiten. Damit erreicht man im Allgemeinen einen höheren Befehlsdurchsatz als beim einfachen Pipelining (IPC $>$ 1 bzw. CPI $<$ 1) [Bär02]. Die Abbildung 2.13 zeigt den prinzipiellen Verarbeitungsfluss einer superskalaren Pipeline.

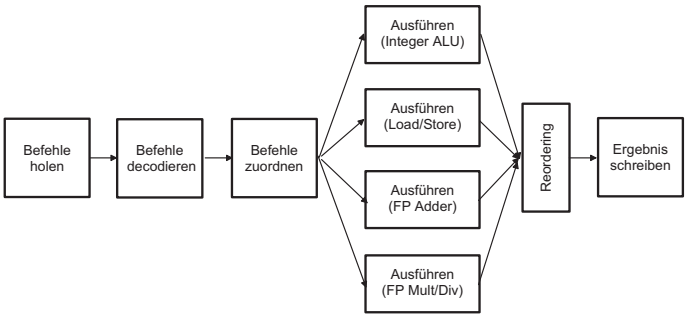

Abbildung 2.13: *Superskalare Pipeline*

Eine superskalare Pipeline besteht grundsätzlich aus den gleichen Pipeline-Phasen wie eine einfache Pipeline. Des öfteren werden zusätzliche Phasen eingeführt, um die mit der Anzahl der parallel auszuführenden Befehle wachsende Komplexität besser in den Griff zu bekommen. Weiterhin kann durch eine feinere Aufteilung der Pipeline-Phasen der Taktzyklus kurz gehalten und dementsprechend eine hohe Taktrate erreicht werden.

2.4.5.1 Die Befehlsholphase

Genau wie bei einer einfachen Pipeline beginnt die Programmausführung damit, Befehle aus dem Befehlsspeicher zu holen. In diesem Fall wird jedoch nicht nur ein einziger Befehl abgeholt, sondern mehrere Befehle werden gleichzeitig bereitgestellt. Dies bildet die Voraussetzung dafür, dass eine spätere parallele Ausführung erfolgen kann. Darüber hinaus unterstützen zusätzliche Komponenten beispielsweise zur Sprungvorhersage (*Branch Prediction Buffer*, *Branch Target Buffer*) diese Phase. Hiermit wird die Leis-

tungsfähigkeit der Pipeline erhöht, indem die bereits erörterten Kontroll-Hazards reduziert werden.

Weitere unterstützende Maßnahmen sind dahingehend, möglichst große Blöcke zusammenhängender und/oder parallelisierbarer Befehle aus dem ausführbaren Programm zu extrahieren und diese als Einheit dem Decoder zur Verfügung zu stellen. Dabei wird zwischen statischen und dynamischen Techniken unterschieden. Statische Techniken werden in Software implementiert und gehören im Allgemeinen zum Compiliervorgang. Dagegen werden dynamische Techniken erst zur Laufzeit aktiviert und sind daher feste Bestandteile der Mikroarchitektur. Letztere bieten den wesentlichen Vorteil, dass eine Leistungssteigerung erzielt werden kann, ohne die bestehenden Software-Tools ändern zu müssen. Der Preis dafür ist die Verlagerung der Komplexität in die Hardware-Implementierung.

Eine sowohl in statischer als auch in dynamischer Ausführung existierende Technik basiert auf dem sog. *Tracing*. Bei dieser Technik geht es darum, eine längere und zusammenhängende Befehlsfolge aus den möglichen Programmverzweigungen zu bilden. Ihre statische Variante, auch unter der Bezeichnung *Trace Scheduling* [PH96] bekannt, wird durch einen verbesserten Compiliervorgang realisiert. Die Grundidee dabei ist, während des Compilierens Befehlsblöcke zu bilden, die über einfache Schleifen (*Loops*) hinausgehen. Mit Hilfe z.B. des *Profiler* wird das Programmverhalten hinsichtlich der möglichen Verzweigungen und Sprünge analysiert. Manche Sprungziele werden aufgrund der geschätzter hohen Wahrscheinlichkeit, dass sie tatsächlich ausgeführt werden, spekulativ ausgewählt. Diese Vorauswahl wird als *Trace Selection* bezeichnet. Anschließend werden die in den aufeinander folgenden *Loops* und in den sonstigen Programmkonstrukten gewählten Befehle zu einem sprungfreien Befehlsblock zusammengesetzt (*Trace Compactation*). Damit resultiert ein Trace als *Bundle* von Befehlen, die gleichzeitig aus dem Speicher abgeholt werden und somit den „von-Neumann-Flaschenhals“ entschärfen.

Auf der Mikroarchitektur-Ebene wird stattdessen das dynamische *Tracing* durch eine zusätzliche Hardware-Komponente, den ***Trace Cache***, realisiert [RBS96, RJSS97, RBS99]. Bei dieser Art der Realisierung finden *Trace Se-*

lection und *Trace Compactation* zur Laufzeit statt. Zum Beginn der Programmausführung wird üblicherweise die Fetch-Phase durchgeführt. Nach dem (Pre-)Decode-Vorgang werden die Befehle ausgeführt und zugleich werden Kopien davon im Trace Cache gespeichert. Das Ziel dabei ist es, Befehle, die mehrmals ausgeführt werden – z.B. Befehle innerhalb einer oft wiederholten Schleife – nicht jedes Mal aus dem Befehlsspeicher oder aus dem Befehlscache abzuholen. Stattdessen stehen sie bereits in decodierter Form im Trace Cache zur Verfügung. Dadurch werden zum einen zeitaufwendige Speicherzugriffe vermieden. Zum anderen können mit Hilfe von Sprungvorhersage-Techniken genauso wie bei dem statischen *Tracing* lange und zusammenhängende Befehlsfolgen, und zwar während der Programmausführung, gebildet werden. Somit wird das Einstellen mehrerer Befehle pro Takt in die Pipeline realisiert. Der Intel Pentium 4 Prozessor integriert als erster kommerzieller Prozessor einen *Execution Trace Cache* als Bestandteil der *Netburst*-Mikroarchitektur und verwirklicht damit das Konzept des dynamischen *Tracing* [Int02].

2.4.5.2 Decodierphase

Wie in den vorangegangen Abschnitten schon angedeutet, nimmt die Komplexität der Pipeline-Implementierung mit der Anzahl der parallel auszuführenden Befehle zu. Dieser Grundsatz gilt auch für die Decodierphase. In dieser Phase müssen prinzipiell alle aus dem Speicher oder aus dem *Trace Cache* geholten Befehle, und zwar in einem Taktzyklus, decodiert werden. Wenn mehrere Befehle gleichzeitig vorliegen, ist es offensichtlich, dass dieser Vorgang ein längeres Zeitfenster in Anspruch nimmt als bei einer einfachen Pipeline. Eine derartige parallele Decodierung mehrerer Befehle führt möglicherweise zu einem langen Taktzyklus und somit zu einer insgesamt niedrigeren Prozessor-Taktrate, falls keine unterstützenden Maßnahmen getroffen werden. Eine davon besteht darin, eine weitere Unterteilung der Decodierphase vorzunehmen, so dass sie sich über mehrere Takte erstreckt. Mit Hilfe von Pipelining, Superskalarität und weiteren Implementierungstechniken bleibt dennoch die Leistungsfähigkeit erhalten.

Eine weitere Technik zur Unterstützung dieser Phase ist das *Predecoding*. Mit dieser Technik werden einige Teilaufgaben des Decodier-Vorgangs bereits während der Fetch-Phase erledigt. Dabei werden einige für die Ausführung notwendige Informationen aus dem Befehlswort extrahiert, z.B. die Befehlsart (*Instruction Class*), die Art der für einen Befehl benötigten Ausführungseinheit (*Execution Unit*) etc. [SFK97].

Für manche superskalaren Mikroarchitekturen gehört zum Decodier-Vorgang auch der Zugriff auf den Registersatz, um die Operanden aus den Registern zu lesen. Außerdem werden zur Lösung der aufgezeigten Hazards die gleichzeitig verwendeten Register je nach Möglichkeit umbenannt (*Register Renaming*). Andere Mikroarchitekturen verlagern diese Aufgaben des Register-Zugriffs sowie der Konfliktlösung in die darauf folgende Pipeline-Stufe.

2.4.5.3 Zuordnungsphase (*Dispatch/Issue/Scheduling*)

Die Zuordnungsphase bildet den Kern der Superskalartechnik, bei der, wie bereits erwähnt, pro Takt mehrere Befehle gleichzeitig zur Ausführung freigegeben werden (*Multiple Issue*). Voraussetzung dafür ist neben einer hohen Anzahl an decodierten Befehlen das Vorhandensein mehrerer parallel arbeitender Ausführungseinheiten.

Demnach besteht die Aufgabe dieser Phase hauptsächlich darin, die Vielzahl von decodierten Befehlen den vielen Ausführungseinheiten zuzuordnen. Zusätzlich müssen die Operanden für die anstehenden Operationen aus den jeweiligen Registern gelesen werden, sofern dies während der Decodierphase nicht erfolgt ist. Des Weiteren kann mit Hilfe der bereits angesprochenen Technik des *Forwarding* das Ergebnis einer gerade ausgeführten Operation als Operand für nachfolgende Operationen weitergereicht werden, bevor es in das Ziel-Register geschrieben wird. Eine Reihe von Techniken werden in den heutigen superskalaren Prozessoren eingesetzt, um die komplexe Aufgabe der Befehlszuordnung zu lösen. Auf die meist verwendeten Techniken wird nachstehend eingegangen.

Die parallel decodierten Befehle müssen in einer geeigneten Struktur aufgenommen werden und auf alle möglichen *Hazards* überprüft werden. Das sog. ***Scoreboard*** bietet hierfür eine Möglichkeit, die parallele Ausführung mehrerer Befehle zu verwalten. Dabei werden alle Informationen über den Fortgang der Ausführung jedes Befehls protokolliert. Befehle können sich in der Ausführung oder im Wartezustand befinden. Zwei Gründe führen dazu, dass ein Befehl in den Wartezustand versetzt wird. Zum einen können die benötigten Operanden aufgrund bestehender Datenabhängigkeiten zwischen aufeinander folgenden Befehlen nicht zur Verfügung stehen. Zum anderen kann zwar der Befehl über alle Operanden verfügen, jedoch verzögert sich seine Ausführung, wenn alle für ihn in Frage kommenden Ausführungseinheiten mit der Ausführung vorangegangener Befehle beschäftigt sind.

Eine weitere Aufgabe des *Scoreboard* ist dafür zu sorgen, dass die potentiellen *Hazards* nicht nur aufgedeckt werden, sondern auch, dass sie tatsächlich gelöst werden. Mit Hilfe des sog. ***Tamasulo Algorithmus*** wird z.B. zusätzlich die Umbenennung der Register (*Register Renaming*) für die Lösung der Datenabhängigkeiten verwendet [PH96].

Die nächste wichtige Fragestellung betrifft die Zuordnungsstrategie (*Issue Policy*). Ausgehend von der Voraussetzung, dass mehrere voneinander unabhängige Ausführungseinheiten zur Verfügung stehen, soll ein optimaler Belegungsplan für die ankommenden Befehle ausgearbeitet werden. Dabei muss der Tatsache Rechnung getragen werden, dass Befehle unterschiedliche Ausführungszeiten benötigen. Manche Befehle wie z.B. die Multiplikation zweier Gleitkommazahlen benötigen mehrere Taktzyklen. Eine durchdachte Zuordnungsstrategie erlaubt, dass z.B. wartende Integer-Operationen einer unmittelbar bevorstehenden Gleitkomma-Operation vorgezogen werden, wenn gerade eine Integer-Einheit zur Verfügung steht. Mit Hilfe dieser ***Out-of-Order***-Zuordnung wird verhindert, dass zeitaufwendige Operationen die gesamte Pipeline verlangsamen. Somit dürfen Befehle, welche alle *Issue*-Bedingungen erfüllen, nicht im Wartezustand verweilen, weil vorhergehende Befehle noch nicht vollständig ausgeführt worden sind. Dies wird durch die sog. ***Reservation Stations*** oder ***Shelving Buffers*** ermöglicht, indem vor jeder Ausführungseinheit ein kleiner Puffer eingerichtet

wird, welcher alle auf die spezielle Ausführungseinheit wartenden Befehle aufnimmt. In manchen Mikroprozessoren wird stattdessen eine globale *Reservation Station* verwendet, und anhand eines geschickten Zuordnungsverfahrens werden die Befehle den entsprechenden Ausführungseinheiten zugeteilt [BU02, vRU99, PH96, SFK97, Bär02, Sta03].

2.4.5.4 Ausführungsphase

Nach dem vorangegangenen Abschnitt lässt sich die Superskalartechnik u.a. dadurch charakterisieren, dass mehrere Befehle gleichzeitig ausgeführt werden können. Dies ist nur möglich, wenn mehrere unabhängige Ausführungseinheiten zur Verfügung stehen. Dafür wird für jede Klasse von Befehlen eine spezielle Ausführungseinheit vorgehalten. Für Integer- und logische Operationen wird eine für diese Aufgabe optimierte arithmetische und logische Einheit (ALU) implementiert. Dementsprechend steht für die Ausführung von Transportbefehlen eine sog. *Load-Store*-Einheit (LSU) zur Verfügung. Den zeitaufwendigen Gleitkomma-Operationen stehen eine spezielle *Floating-Point*-Einheit (FPU) zur Verfügung. Weitere Variationen aus diesen grundlegenden Ausführungseinheiten sind möglich. Beispielsweise enthält der Intel Pentium 4 Prozessor bis zu sieben und sein direkter Konkurrent, der AMD Athlon XP, bis zu fünf unterschiedliche Arten von Ausführungseinheiten. Jede der möglichen Ausführungseinheiten kann einfach oder mehrfach in einem Prozessor implementiert werden.

Der wesentliche Vorteil dieser Unterscheidung der Funktionseinheiten in der Ausführungsphase besteht darin, dass die einfachen Operationen mit einer höheren Taktrate möglichst in einem Takt ausgeführt werden können. Dagegen darf die Ausführung zeitaufwendiger Befehle mehrere Takte in Anspruch nehmen, ohne dabei die gesamte Pipeline zum Halten zu zwingen. Außerdem lässt sich durch diese Klassifizierung der verschiedenen Ausführungseinheiten die Komplexität in Hinblick auf die Hardware-Implementierung besser beherrschen und eine getrennte Optimierung realisieren.

2.4.5.5 Rückschreibphase

Nach der parallelen Ausführung der Befehle müssen die entsprechenden Ergebnisse in die dafür vorgesehenen Register zurückgeschrieben werden (*write back*). Dabei ist es zwingend notwendig, dass unabhängig von den unterschiedlichen Ausführungsgeschwindigkeiten der verschiedenen Ausführungseinheiten die Ergebnisse in der richtigen Programmreihenfolge gespeichert werden. Andernfalls können die auftretenden Hazards das endgültige Programmergebnis verfälschen. Damit die Programmreihenfolge eingehalten wird, werden die Ergebnisse aus der Ausführungsphase zunächst in einen ***Reorder Buffer*** gespeichert (*Completion*). Die darauf wartenden Befehle, die sich bereits in der Zuordnungsphase befinden, erhalten hieraus ihre Operanden (*Bypassing, Forwarding*). Falls in einer der vorhergehenden Pipeline-Stufen eine Register-Umbenennung stattgefunden hat, muss sie an dieser Stelle rückgängig gemacht werden. Anschließend werden die Ergebnisse in die richtigen Register zurückgeschrieben (*Commitment*). Zum Schluss werden die nun vollständig ausgeführten Befehle aus der gesamten Pipeline (*Scoreboard, Reorder Buffer*) entfernt (*Retirement*).

Neben der Superskalartechnik sollen vollständigkeitshalber noch zwei Prozessor-Techniken erwähnt werden, welche ebenfalls eine parallele Befehlsausführung (*Instruction Level Parallelism*, ILP) realisieren. Zum einen handelt es sich um die *Very Long Instruction Word* (VLIW)-Technik, die sich dadurch auszeichnet, dass mehrere, voneinander unabhängige Befehle aus einem Programm in einem langen Befehlswort zusammengelegt werden. Solche Befehlsblöcke werden während des Compiliervorgangs, und zwar durch einen speziellen VLIW-Compiler, gebildet. Sie werden als einheitliches Befehlswort aus dem Befehlsspeicher abgeholt und parallel ausgeführt. Zum anderen wurde im Jahre 1994 von den Firmen Hewlett-Packard und Intel die *Explicitly Parallel Instruction Computing* (EPIC)-Technik ins Leben gerufen. Diese unterscheidet sich von der VLIW-Technik hauptsächlich dadurch, dass sie das Auffinden der parallelisierbaren Befehle nicht dem Compiler überlässt. Bei der EPIC-Technik werden stattdessen parallel auszuführende

Befehle bereits beim Codieren durch den Programmierer explizit angegeben. Auf dieser Technik basiert die neue *Intel 64-Bit Architecture* (IA-64).

3 Eine rekonfigurierbare Mikroarchitektur - Modellierung und Simulation

Nachdem die grundsätzlichen Eigenschaften rekonfigurierbarer Hardware sowie die verschiedenen Techniken der modernen Prozessorentwicklung aufzeigt sind, beschäftigt sich nun der Hauptteil der Arbeit mit der kombinierten Einsatzmöglichkeit dieser beiden Hochtechnologien. Einer der möglichen Lösungswege zu dieser eher allgemein formulierten Problemstellung, der zugleich als Grundlage dieser Arbeit vorgeschlagen wird, basiert auf einem neuen Ansatz einer dynamisch rekonfigurierbaren Mikroarchitektur. Im Gegensatz zu den üblichen fest verdrahteten Mikroprozessoren, bei denen eine nachträgliche Verbesserung der Mikroarchitektur zwangsläufig die Entwicklung eines neuen Produkts bedeutet, wird hier eine anwendungsabhängige Anpassung der Mikroarchitektur zur Laufzeit realisiert. Grundvoraussetzung dafür bildet die Verwendung eines partiell und dynamisch rekonfigurierbaren FPGA als Ziel-Plattform, welches eine flexible Gestaltung unterschiedlicher Hardware-Konfigurationen ermöglicht.

Die Verwirklichung dieses sowohl technologisch als auch wissenschaftlich sehr ehrgeizigen Vorhabens wurde im Laufe dieser Arbeit in drei wesentlichen Schritten durchgeführt. Zunächst galt es, das Modell einer rekonfigurierbaren Mikroarchitektur zu entwerfen, welches die bewährten Eigenschaften eines zeitgemäßen RISC-Mikroprozessors, z.B. Pipelining und Superskalarität, mit den Vorzügen der partiellen und dynamischen Hardware-Rekonfiguration vereinigt. Zur Validierung des Modells wurde eine Leistungsabschätzung anhand von Software-Simulationen herangezogen. Die daraus resultierenden Simulationsergebnisse haben das Potential, aber auch die Grenzen einer solchen rekonfigurierbaren Mikroarchitektur aufgezeigt. Darauf aufbauend wurde das Modell in eine Hardware-Implementierung

umgesetzt, um die Standfestigkeit sowie die Machbarkeit dieses neuen Ansatzes nachzuweisen. Damit soll zumindest exemplarisch ein Weg aufgezeigt werden, wie künftige Mikroarchitekturen von den Vorteilen rekonfigurierbarer Hardware profitieren können. Die drei Schritte bilden den Kern dieser Arbeit und werden in den nachstehenden Kapiteln näher erörtert.

3.1 Abgrenzung

Aus den vorangegangenen Abschnitten ist deutlich geworden, dass eine Vielzahl von Arbeiten auf dem Gebiet des *Reconfigurable Computing* durchgeführt worden ist und bereits die Vorteile rekonfigurierbarer Hardware nachgewiesen wurden. Dabei sind zahlreiche rekonfigurierbare Mikroarchitekturen und Werkzeuge entwickelt worden und unterschiedlichste Anwendungen effizienter implementiert worden. Unabhängig von der Granularität, der Art der Kopplung, der Rekonfigurierbarkeit und von der Programmierbarkeit (siehe Abschn. 2.2) können die Arbeiten auf diesem Gebiet in zwei Gruppen eingeteilt werden:

- Die erste Gruppe, hauptsächlich aus dem industriellen Umfeld, umfasst Arbeiten, welche rekonfigurierbare Hardware als flexible Entwicklungsplattform verwenden. Durch Rekonfiguration des selben Bausteins können unterschiedliche Lösungen in Hardware getestet werden (*Rapid Prototyping*). Nach einer erfolgreichen Testphase wird ein entsprechendes ASIC entwickelt, welches das Endprodukt realisiert. Denn ein ASIC stellt trotz seiner geringen Flexibilität immer noch eine leistungsfähige und kostengünstige Hardware dar.

- Der größte Teil der Arbeiten jedoch beschäftigt sich mit der Verbesserung der Leistungsfähigkeit durch Parallelisierung der Ausführung auf unterschiedlichen Ebenen. Mit Hilfe rekonfigurierbarer Hardware werden Anwendungen, Threads, Funktionen oder sogar Befehle parallel oder quasi-parallel ausgeführt. Dadurch wird in der Regel die Ausführung beschleunigt und eine dementsprechende Leistungsteigerung erzielt. Für die herkömmliche Hardware stellen die immer noch

wachsenden Taktraten ein überzeugendes Verkaufsargument dar. Dagegen wird bei den rekonfigurierbaren Rechensystemen oftmals die Leistungsfähigkeit anhand der durch parallele Ausführung erhöhten Anzahl von ausgeführten Befehlen pro Zeiteinheit (*Millions of Instructions Per Second*, MIPS) angegeben. Stellvertretend für die Arbeiten in dieser Gruppe ist der rekonfigurierbare Mikroprozessor der Firma Stretch [Str05, Ele04] zu nennen, der laut Hersteller zu den leistungsfähigsten Prozessoren im *embedded* und DSP-Bereich gehört.

Dazu gibt es eine Reihe von Arbeiten, die in beiden Gruppen zu finden sind, welche sich mit der Entwicklung von unterstützenden Werkzeugen (*Design Tools*, Compiler, Betriebssysteme etc.) beschäftigen.

Ein großer Nachteil charakterisiert fast alle Arbeiten auf diesem Gebiet trotz aller positiven Leistungsdaten: die Software muss immer an die rekonfigurierbare Hardware angepasst werden. Dies geschieht beispielsweise durch Partitionierung der Anwendung in Software- und Hardware-Anteile. Rechenintensive Aufgaben werden auf der rekonfigurierbaren Hardware ausgeführt, während einfache Aufgaben sowie die Steuerung meistens weiterhin in Software implementiert werden. Die Folge davon ist, dass ein entsprechender Compiler entwickelt werden muss und bestehende Anwendungen angepasst und neu übersetzt werden müssen. Im Allgemeinen sind Arbeiten auf diesem Gebiet mit einem sehr hohen Aufwand sowohl für die Software- als auch die Hardware-Entwicklung verbunden.

In dieser Arbeit soll deshalb der umgekehrte Weg untersucht werden. Die Software soll nicht an die Hardware angepasst werden, sondern die Hardware soll sich den Software-Erfordernissen anpassen. Es hat sich sehr oft gezeigt, dass vielversprechende Technologien nicht zuletzt daran scheiterten, dass sie nicht mit bestehender Software/Hardware kompatibel sind. Die vorliegende Arbeit schließt sich dieser Ansicht an und verfolgt das Ziel, mit gängigen Werkzeugen Software-Anforderungen durch Hardware-Rekonfiguration nachzukommen (evolutionärer Ansatz). Dies bedeutet zwar einen großen Hardware-Aufwand, aber es trägt durch die gewahrte Kompatibilität mit bestehenden Systemen sowie durch den äußerst geringen Software-Aufwand

zu einem leichten Einstieg und einer breiten Akzeptanz dieser zukunftsweisenden Technologie bei.

Einige Arbeiten haben schon die Idee der Anpassung der Hardware an die Software-Anforderungen verfolgt. Beim sog. *Complexity-Adaptive Processor, CAP* [Alb98, XA00] wird beispielsweise eine mittlerweile patentierte Technik angewendet, um die Komplexität konventioneller, fest verdrahteter Hardware für eine bestimmte Anwendung zu optimieren. Das Ziel der Optimierung ist vor allem die Reduzierung der verbrauchten Hardware-Ressourcen. Die dabei angewendete Technik basiert auf den Fortschritten in der Halbleitertechnik.

Die Forscher in der CAP-Arbeitsgruppe [CAP05] haben erkannt, dass die in der CMOS-Technologie eingebauten Puffer zur Reduzierung der Latenzzeiten (*wire delay*) für andere Zwecke verwendet werden können. Anhand dieser in Busverbindungen und Speicherkomponenten zahlreich vorhandenen Schaltungselemente wird eine konventionelle statische Hardware in eine dynamische Hardware-Struktur umgewandelt. Dies geschieht in Abhängigkeit der Software-Anforderungen durch einfaches Ein- oder Ausschalten einiger Hardware-Elemente aus dem aktiven Bereich. Aus einer 8-elementigen Busverbindung wird z.B. durch Ausschalten der letzten vier Elemente eine 4-elementige Verbindung geschaffen (vgl. Abb. 3.1). Dies führt zu einer Reduzierung der Latenzzeit, zu einer geringeren Verlustleistung und nicht zuletzt zu einem reduzierten Stromverbrauch in konventionellen Mikroprozessoren.

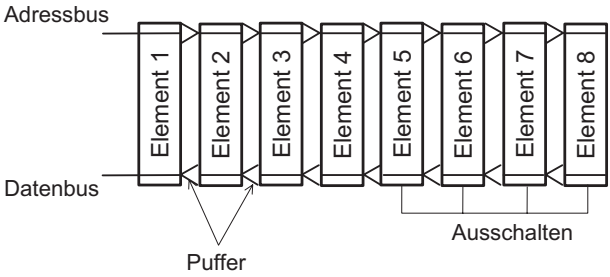

Abbildung 3.1: *Adaptive Hardware-Struktur [Alb98]*

Daraus wird deutlich, dass der CAP-Ansatz zwar auch auf der Idee der software-abhängigen Hardware-Anpassung basiert, jedoch besteht das Ziel beim CAP lediglich darin, den Hardware-Verbrauch in konventionellen Mikroprozessoren zu reduzieren. Echte rekonfigurierbare Hardware (FPGAs) kommt dort nicht zum Einsatz.

Die in den vorangegangenen Abschnitten formulierte Kritik über den hohen Software-Aufwand, der notwendig ist, um eine Anwendung auf einem rekonfigurierbaren Rechensystem ablaufen zu lassen, wurde auch in [Dal99] geäußert. Außerdem brachte die gleiche Arbeit die existierende Problematik zum Ausdruck, dass unterschiedliche Anwendungen auch unterschiedliche Hardware-Ansprüche haben. Daher auch die Idee einer Anpassung der Hardware an die unterschiedlichen Software-Anforderungen mit Hilfe rekonfigurierbarer Hardware. Die Umsetzung erfolgte jedoch im Rahmen einer der bereits erwähnten Lösungswege. Ein rekonfigurierbarer Coprozessor wurde entwickelt, welcher lediglich ausgewählte rechenintensive Berechnungen ausführen soll [Dal99, Dal01, Dal03]. Außerdem wurden die praktischen Untersuchungen nur anhand von Software-Simulationen durchgeführt.

Im vorliegenden Beitrag sollen dagegen nicht nur die Ausführung spezieller Befehle, die zuvor identifiziert werden müssen, beschleunigt werden, sondern es werden parallel zu der Ausführung — und zwar durch kontinuierliche Auswertung des Programmsablaufs (*Tracing*) — die benötigten Hardware-Ressourcen ermittelt. Anhand dieser *just-in-time* Evaluierung des Hardware-Bedarfs wird durch Rekonfiguration versucht, eine sich ändernde Mikroarchitektur für das gesamte Programm anzubieten. Gerade die Umsetzung allein auf der Mikroarchitektur-Ebene schafft — im Gegensatz zu allen bisherigen Ansätzen — die Kompatibilität mit anderen Rechensystemen und minimiert somit den Software-Aufwand. Darüber hinaus erfolgt in dieser Arbeit die Untersuchung nicht nur als Software-Simulation, sondern sie wird durch eine entsprechende Implementierung auf rekonfigurierbarer Hardware untermauert.

Einen Extremfall von Anpassung der Hardware-Ressourcen an Software-Anforderungen stellt der sog. *Flexible Instruction Processor* (FIP) [SLC00, SLC02] dar. Anhand von vorgefertigten Komponenten (*Templates*) wird zu-

nächst in Abhängigkeit des gerade ausgeführten Programms eine geeignete Mikroarchitektur aufgebaut. Dabei können unterschiedliche Prozessortypen, z.B. ein Stack- oder ein Register-basierter Prozessor, realisiert werden. Während der Programmausführung wird das dynamische Verhalten des Programms überwacht. Mit Hilfe der gesammelten Daten werden im Betrieb einige Parameter der Mikroarchitektur, z.B. die Datenbreite, angepasst. Ferner soll es möglich sein, zusätzliche benutzerdefinierte Befehle (*custom instructions*) in Form von optimierten Hardware-Konfigurationen zum bestehenden Befehlssatz dynamisch hinzuzufügen.

Leider wird aus den entsprechenden Veröffentlichungen nicht ersichtlich, wie alle diese Konzepte praktisch umgesetzt werden sollen. Denn es besteht angesichts der existierenden Einschränkungen bei den heutigen Methoden zum Hardware-Entwurf der berechtigte Zweifel, ob sich eine dynamische Implementierung aller möglichen Prozessor-Komponenten wirklich durchführen lässt. Eine uneingeschränkte Änderung der Mikroarchitektur im Betrieb erfordert neue Entwurfsmethoden, die eine dynamische Hardware-Beschreibung zulassen. Mit den heutigen Hardware-Entwicklungstools müssen vor der Umsetzung der Hardware-Beschreibung in Gattern alle Parameter mit festen und statischen Werten belegt werden. Somit erscheint dieser Ansatz im Hinblick auf die heutige Technologie unrealistisch. Er kann erst dann realisiert werden, wenn entsprechende Hardware- und Software-Tools entwickelt worden sind. Es ist z.B. bekannt, dass im Rahmen dieses Projektes eine neue Hardware-Beschreibungssprache entwickelt wurde, welche die geltenden Regeln der Synthese nur mit statischen Werten aufheben soll. Darüber hinaus ist die Entwicklung eines speziellen Compilers unumgänglich und tatsächlich wurde ein FIP-Compiler gebaut [SLC02].

Die im Rahmen dieser Arbeit entwickelte Mikroarchitektur verfolgt zwar den gleichen Grundgedanken einer dynamischen Anpassung der Mikroarchitektur an die Software-Anforderung wie in [SLC02]. Da sie mit Hilfe von *state-of-the-art* Soft- und Hardware-Mitteln implementiert wurde, musste jedoch eine spezielle und geeignete Art von Rekonfiguration konzipiert werden. Sie stützt sich auf folgende Argumente: Zum einen ist es mit dem heutigen Stand der Technik praktisch unmöglich, einen kompletten

Prozessor (inkl. Befehlssatz und Mikroarchitektur) zeitgleich zum angestrebten Prozessorbetrieb zu bilden bzw. zu optimieren. Zum anderen soll eine rekonfigurierbare Mikroarchitektur entstehen, bei der alle Komponenten ohne Spezialisierung für alle Anwendungen (*General Purpose Computing*) verwendet werden können. Dies bedeutet, dass nicht nur eine spezielle Komponente vorzusehen ist, die allein rekonfiguriert wird, um ausgewählte Berechnungen effizienter auszuführen.

Zusammenfassend zeichnet sich diese Arbeit von den bekannten Ansätzen durch folgende Elemente aus:

- Anpassung der rekonfigurierbaren Hardware an die Anwendungsanforderungen und nicht umgekehrt, wie von der Mehrzahl der Arbeiten auf diesem Gebiet praktiziert.
- Gewährleistung der Software-Kompatibilität durch alleinige Änderung der Mikroarchitektur; die Folge davon ist, dass der Anwendungsentwickler sich nicht mit der Hardware beschäftigen muss.
- Verwendung üblicher Werkzeuge und Berücksichtigung der aktuell existierenden Einschränkungen. Somit ist der Ansatz realitätsnah und erfordert keine zusätzlichen Software- und Hardware-Tools (z.B. Compiler, Programmiersprachen, Synthese-Tools etc.).

3.2 Modell einer rekonfigurierbaren Mikroarchitektur

In einem ersten Arbeitsschritt ging es um die Entwicklung des Modells einer rekonfigurierbaren Mikroarchitektur mit dem Ziel, sowohl die Vorzüge als auch die Grenzen des Einsatzes von rekonfigurierbarer Hardware in Prozessorarchitekturen untersuchen zu können. Das Modell beruht in erster Linie auf dem von-Neumann-Prinzip, welches um eine fünf-stufige Pipeline-Ausführung erweitert wurde, wie es bei den heutigen Mikroprozessoren üblich ist. Darüber hinaus handelt es sich um eine zeitgemäße superskalare Mikroarchitektur mit mehreren parallel arbeitenden Ausführungseinheiten. Des Weiteren wurde Wert auf die Kompatibilität mit einer existierenden Prozessorarchitektur gelegt, so dass man von einem evolutionären Ansatz sprechen kann. Dies bietet den Vorteil, dass weder ein neuer Compiler entwickelt noch bestehende Software umgeschrieben werden muss. Es wird lediglich die Art und Weise geändert, wie die einzelnen Befehle ausgeführt werden (s. Abschn. 2.4.1). Mit der Fähigkeit einiger FPGAs, partiell und dynamisch rekonfiguriert werden zu können, ergeben sich weitere Gestaltungsmöglichkeiten bei der Konzeption einer neuartigen Mikroarchitektur. Abbildung 3.2 zeigt das Blockdiagramm der modellierten Mikroarchitektur.

Das neu konzipierte Modell besteht aus folgenden Komponenten:

- ***Fetch Unit / Predecoder***
  Die *Fetch Unit* bildet die Komponente, welche zu jedem Takt einen aktuellen Befehlszählerstand (PC) festlegt, der als Speicheradresse für die nächsten Befehle verwendet wird. Dazu wird im Allgemeinen der Inhalt des PC zum Befehlsspeicher (*Instruction Memory*) übertragen, aus welchem dann ein oder mehrere Befehle abgeholt werden. Im darauf folgenden Taktzyklus werden diese Befehle dem Decoder zur weiteren Ausführung zugeführt. Die *Fetch Unit* realisiert somit die Befehlsholphase (s. Abschn 2.4.3). Außerdem wird der aktuelle Befehlszählerstand zu einer speziellen Komponente geleitet, dem *Trace Cache*, welcher bereits ausgeführte Befehle enthält. Dort wird

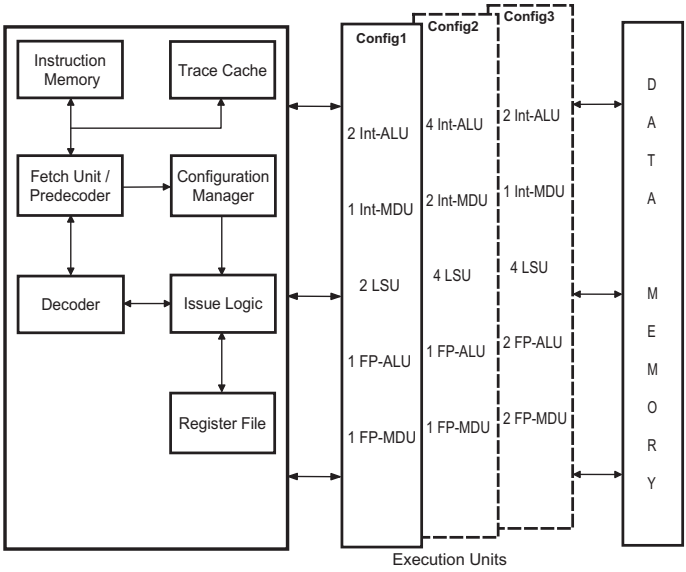

Abbildung 3.2: *Modell einer rekonfigurierbaren Mikroarchitektur*

er mit den zuvor darin gespeicherten PCs verglichen, und im Falle einer Übereinstimmung (*Trace valid*) werden die dazu gehörigen Befehle in die Pipeline gebracht (siehe 3.3). Die aus dem *Instruction Memory* kommenden Befehle werden dabei verworfen. Somit fungiert der *Trace Cache* als zweiter Befehlsspeicher für Befehle, die bereits mindestens einmal ausgeführt worden sind. Voraussetzung für diese Funktionalität des *Trace Cache* bildet der *Predecoder*. Im Gegensatz zu den herkömmlichen Mikroarchitekturen, bei denen ein Befehl ohne zusätzliche Verarbeitung von der *Fetch Unit* zum Decoder übertragen wird, findet hier eine Vor-Decodierung statt. Ziel dabei ist, Befehle für das zwischenzeitliche Speichern im *Trace Cache* geeignet vorzubereiten. Damit sollen sie bei einer späteren Ausführung schneller und effizienter abgearbeitet werden können. Der in diesem Modell vorgesehene *Trace Cache* speichert beispielsweise nicht nur Befehle mit den dazu gehörigen Befehlszählerständen, sondern enthält zusätzliche Informationen über die Art jedes Befehls. Solche Informatio-

nen werden durch den *Predecoder* bereitgestellt und dienen einer späteren Entscheidungsfindung darüber, wann und in welchem Umfang die Hardware-Rekonfiguration erfolgen soll. Werden Befehle aus dem *Trace Cache* abgeholt, informiert die *Fetch Unit* eine weitere Komponente, den *Configuration Manager*, über die Art der vorliegenden Befehle. Diese Komponente trifft ihrerseits alle Vorkehrungen, damit eine geeignete Hardware-Konfiguration im Hinblick auf eine optimale Ausführungsgeschwindigkeit dynamisch geladen werden kann (siehe Abb. 3.3).

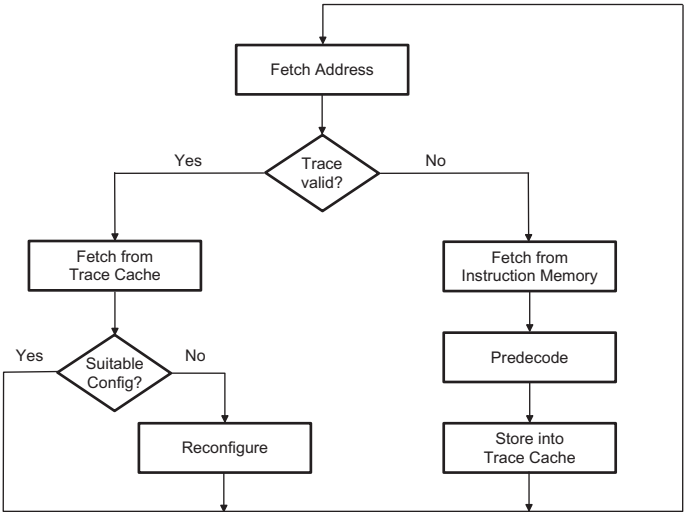

Abbildung 3.3: *Befehlsholphase*

- ***Trace Cache***

  Der *Trace Cache* besteht aus einem prozessorinternen Befehlsspeicher, welcher ursprünglich für die Unterstützung der Befehlsholphase entwickelt wurde [RBS96]. Mit einem herkömmlichen und im Arbeitsspeicher angesiedelten Befehlsspeicher ist die Anzahl der Befehle, die parallel übertragen werden können, eingeschränkt („von-Neumann-

Flaschenhals“). Wird dagegen ein *Trace Cache* eingesetzt, so ist es möglich, die Wortlänge geeignet zu erweitern und so eine Vielzahl von Befehlen gleichzeitig in die Pipeline einzuleiten. Ihre genaue Zahl hängt maßgeblich von der Organisation des *Trace Cache*, vom Parallelisierungsgrad der gesamten Pipeline (z.B. Superskalarität) und nicht zuletzt von dem programmabhängigen Grad der Befehlsparallelisierung (*Instruction-Level Parallelism*, ILP [PH96]) ab.

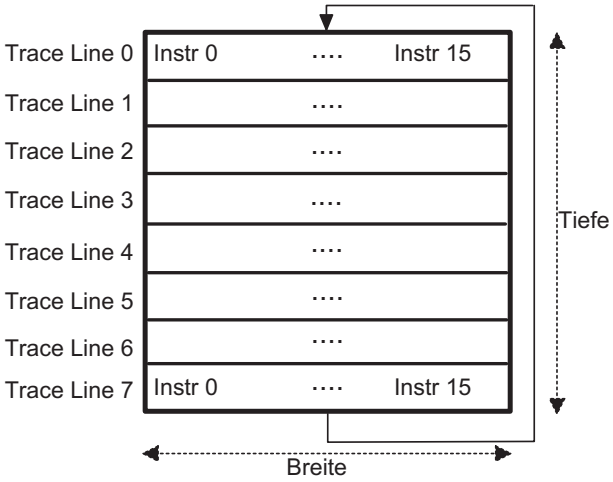

Abbildung 3.4: *Aufbau eines Trace Cache*

Ein *Trace Cache* wird üblicherweise in *Trace Lines* organisiert (siehe Abb. 3.4). Die Tiefe des *Trace Cache* gibt die maximale Anzahl der verfügbaren *Trace Lines* an. Die Breite des *Trace Cache* entspricht der maximalen Anzahl an Befehlen, die in einer *Trace Line* gespeichert werden können (Länge). Darin soll eine zusammenhängende Befehlsfolge zwischengespeichert werden und bei einer späteren Ausführung effizienter bereitgestellt werden können. Das Standardbeispiel einer solchen Befehlsfolge bildet die Befehlskette innerhalb der Schleifenkonstruktion eines Programms. Die in einer Schleife verwendeten

Befehle einschließlich des abschließenden Sprungbefehls werden bei ihrer ersten Ausführung in einer *Trace Line* hintereinander aufgenommen. Es gibt Ansätze dahingehend, Befehlsfolgen zu bilden, welche über einen Sprungbefehl hinaus gehen. Zu diesem Zweck werden mit Hilfe des sog. *Branch History Table* Sprungziele aus der vorhergehenden Ausführung mit einbezogen. Falls die Länge einer Befehlsfolge die Breite einer *Trace Line* überschreitet, wird die Speicherung in der darauf folgenden fortgesetzt.

Bei erneuter Ausführung der gleichen Befehlsfolge erhält der *Trace Cache* von der *Fetch Unit* die aktuelle *Fetch*-Adresse. Daraufhin wird aus allen belegten *Trace Lines* nach einem Befehlszählerstand gesucht, welcher mit der gelieferten Adresse übereinstimmt. Bei einer erfolgreichen Suche wird nicht allein der dazu gehörige Befehl geholt, sondern sämtliche Befehle innerhalb der identifizierten *Trace Line* werden gleichzeitig bereitgestellt. Die einzige Einschränkung besteht darin, dass nur so viele Befehle geholt werden können, wie die Pipeline auch parallel verarbeiten kann (Parallelisierungsgrad). Es ist theoretisch nicht ausgeschlossen, dass eine im *Trace Cache* gespeicherte Befehlsfolge die Anzahl an Befehlen übersteigt, die gleichzeitig durch die Pipeline aufgenommen werden können. In solchen Fälle wird das Befehlsholen aus dem *Trace Cache* wiederholt durchgeführt, bis die vorliegende Befehlsfolge vollständig ausgeführt worden ist.

- ***Decoder***

  Es handelt sich hier um einen herkömmlichen Decoder, wie er in allen Mikroprozessoren zur Decodierung von Befehlen eingesetzt wird. Er erhält von der *Fetch Unit* die auszuführenden Befehle, und diese werden auf ihre jeweiligen Bestandsteile hin überprüft. Generell werden bei RISC-Prozessoren Befehle in 16 Bits oder 32 Bits komprimiert, um u.a. den Speicherplatzbedarf von Programmen in Grenzen zu halten. Der Decoder muss aus solchen komprimierten Befehlen erkennen, ob es sich überhaupt um gültige Befehle handelt. Außerdem bestimmt er die Art der darin codierten Rechenoperationen (Opcode) sowie die dafür notwendigen Argumente (Operanden). Diese Angaben zusammen

mit allen zusätzlichen Steuerungsinformationen werden anschließend zu der *Issue Logic* weitergeleitet, die für die weiteren Verarbeitungsschritte sorgt.

- ***Issue Logic***
  Die modellierte rekonfigurierbare Mikroarchitektur basiert u.a. auf einer superskalaren Mikroarchitektur (siehe Abb. 2.13). Es wird davon ausgegangen, dass mehrere Befehle gleichzeitig ausgeführt werden können. Voraussetzung dafür ist einerseits die gleichzeitige Bereitstellung mehrerer Befehle, was durch die *Fetch Unit*, den *Trace Cache* und den Decoder gewährleistet wird. Andererseits ist eine Vielzahl von Ausführungseinheiten erforderlich, damit voneinander unabhängige Befehle auch parallel ausführt werden können. Die *Issue Logic* übernimmt im Einzelnen folgende Aufgaben:
  - Kurzzeitige Speicherung der vom Decoder ankommenden Befehle in einer zyklischen Wartenschlange (*Issue Queue*), bis sie vollständig ausgeführt worden sind.
  - Lese-Zugriff auf den Registersatz (*Register File*), um die Operanden für die verschiedenen Befehle bereitzustellen.
  - Behandlung der möglichen *Hazards*.
  - Verteilung der vorhandenen Befehle auf die zur Verfügung stehenden Ausführungseinheiten und somit die Erfüllung der Aufgaben eines *Dispatcher*. Dabei wird ein Befehl sofort ausgeführt, sobald die Argumente und eine entsprechende Ausführungseinheit dafür bereitstehen, unabhängig von der Reihenfolge seines Eintreffens in der Warteschlange (*out-of-order execution*).
  - Aufnahme der Ergebnisse aus den Ausführungseinheiten in einen Puffer, den sog. *Reorder Buffer*. Aus diesem Puffer werden die Ergebnisse in die Ziel-Register (*Destination Register*) geschrieben. Dabei ist darauf zu achten, dass die richtige Befehlsreihenfolge eingehalten wird, damit nachfolgende Operationen nicht mit falschen Werten rechnen (*in-order completion*). Außerdem können Ergebnisse aus dem *Reorder Buffer* ohne Umweg zum

Register File als Operanden für ausstehende Befehle zur Verfügung gestellt werden (*forwarding/bypass*). Somit verkürzt sich in manchen Fällen die Wartezeit eines Befehls, bis er zur Ausführung kommt.

- Schreib-Zugriff auf das *Register File*, um die Ergebnisse der ausgeführten Befehle in die jeweiligen Ziel-Register zu speichern.
- Ausführung von Sprung-Befehlen und sonstigen Steueraufgaben.

Eine der wesentlichen Neuerungen dieses Modells besteht darin, in Abhängigkeit der zur Ausführung anstehenden Befehle die Anzahl der unterschiedlichen Ausführungseinheiten durch Hardware-Rekonfiguration zu reduzieren oder zu erhöhen. Damit die *Issue Logic* in jedem Fall eine optimale Verteilung der Befehle an die zahlenmäßig wechselnden Ausführungseinheiten vornehmen kann, muss sie ständig über deren gerade aktuelle Konfiguration informiert werden. Deshalb wird sie jedes Mal durch den *Configuration Manager* benachrichtigt, wenn sich die Konfiguration ändert.

- ***Register File***

  Ein weiteres Merkmal des entwickelten Modells ist die Umsetzung der etablierten *Load/Store*-Architektur (vgl. Abschn. 2.4.3). Demnach operieren Befehle ausschließlich auf Register-Inhalten. Lediglich *Load*- und *Store*-Befehle ermöglichen den Datenaustausch über Register zwischen Prozessor und Arbeitsspeicher. Das *Register File* besteht aus einem Satz von Registern, welche zur Speicherung der Operanden und der berechneten Ergebnisse dienen. Die *Issue Logic* ist die einzige Komponente, die darauf zugreifen kann, um entweder die Operanden zu lesen oder die Ergebnisse zu schreiben.

- ***Execution Units***

  Wie bereits erwähnt, basiert eine superskalare Mikroarchitektur weitgehend auf der Existenz einer Vielzahl von Ausführungseinheiten. Dies ist eine der notwendigen Voraussetzungen für das laut Definition geforderte $\mathrm{IPC} > 1$ bzw. $\mathrm{CPI} < 1$. Die verschiedenen Ausführungseinheiten unterscheiden sich durch ihre jeweilige Spezialisierung auf die

Ausführung einer bestimmten Klasse von Befehlen. Darüber hinaus können mehrere Ausführungseinheiten zur Verfügung stehen, welche für die gleiche Art von Befehlen geeignet sind (z.B. 2 ALUs). Im vorliegenden Modell sind folgende Ausführungeinheiten vorgesehen:

- Int-ALU: Führt einfache arithmetische und logische Operationen für Integer-Werte aus.
- Int-MDU: Wird für die Ausführung von Multiplikationen und Divisionen mit Integer-Werten benötigt.
- LSU: Ist für das Laden (*Load*) der Operanden aus dem Datenspeicher sowie das Speichern (*Store*) der Ergebnisse in den Datenspeicher zuständig. Daher die Bezeichnung *Load/Store Unit*.
- FP-ALU: Führt einfache arithmetische und logische Operationen für Gleitkomma-Zahlen (*Floating-Point*) aus.
- FP-MDU: Übernimmt die Ausführung von Multiplikationen und Divisionen mit Gleitkomma-Zahlen.

Jede dieser Ausführungseinheiten kann wie bereits erwähnt einfach oder mehrfach zur Verfügung gestellt werden. Mit Hilfe der partiellen und dynamischen Hardware-Rekonfiguration wird erst zur Laufzeit entschieden und laufend aktualisiert, welche Ausführungseinheiten und wie viele davon konfiguriert werden.

Aufgrund der existierenden Einschränkung im Hinblick auf die Hardware-Implementierung bezieht sich die dynamische Rekonfiguration im Rahmen dieser Arbeit ausschließlich auf die Ausführungseinheiten. Theoretisch ist es möglich, jede Prozessor-Komponente im Betrieb zu konfigurieren. Beispielsweise könnte auch die *Fetch Unit* rekonfiguriert werden und anstelle von zwei Befehlen pro Takt würden vier Befehle gleichzeitig abgeholt werden. Wird dynamische Rekonfiguration für die gesamte Mikroarchitektur zugelassen, so ist die daraus resultierende Komplexität nicht mehr mit herkömmlichen Mitteln zu bewältigen. Das führt zu dem bereits erwähnten Ansatz des *Flexible Instruction Processor* (vgl. Abschn. 3.1). Daher muss eine Balance gefunden

werden zwischen dem, was erwünscht ist, und dem, was realisierbar ist.

Der entscheidende Vorteil des für diese Arbeit gewählten Ansatzes gegenüber herkömmlichen, fest verdrahteten Mikroarchitekturen besteht darin, dass keine unnötigen Ausführungseinheiten vorgehalten werden, während andere, dringend benötigte Einheiten knapp werden. Statt eine Menge von Komponenten im Betriebszustand zu halten, unabhängig davon, ob sie gebraucht werden oder nicht, sollen dynamisch nur solche Einheiten bereitgestellt werden, welche auch tatsächlich benötigt werden. Dies ist z.B. der Fall, wenn ein Programm aus Integer-Operationen besteht. In solchen Fällen sollen die Hardware-Ressourcen statt zur Implementierung der für diese Art von Befehlen irrelevanten Gleitkomma-Einheiten dafür verwendet werden, eine höhere Anzahl an Integer-Einheiten zu implementieren. Wenn später Gleitkomma-Operationen ausgeführt werden müssen, so soll wiederum durch Rekonfiguration eine passende Konfiguration mit Gleitkomma-Einheiten geladen werden. Darin besteht die zentrale Neuerung der im Rahmen dieser Arbeit entwickelten Mikroarchitektur gegenüber bereits bestehenden.

Allerdings muss ein zusätzlicher Zeitaufwand angerechnet werden, da während der dynamischen Hardware-Rekonfiguration die Programmausführung teilweise behindert wird, bis eine neue Konfiguration geladen worden ist. Insofern müssen die Vor- und Nachteile dieses Konzepts gegenübergestellt und diskutiert werden.

Für eine praktische Umsetzung werden die verschiedenen Ausführungseinheiten in unterschiedlichen, aber zweckmäßigen Kombinationen zusammengefasst. Daraus entstehen untereinander austauschbare Konfigurationen. In Abbildung 3.2 sind beispielhaft drei Konfigurationen (*Config1, Config2, Config3*) aufgetragen, welche im Betrieb ausgetauscht werden können. Die Bereitstellung dieser vordefinierten Konfigurationen ist damit zu rechtfertigen, dass deren Zusammenstellung zur Laufzeit einen erheblichen zusätzlichen Zeitaufwand bedeuten würde. Daher sollten sie vorher in sinnvoller Weise entwickelt werden.

Die in der Abbildung 3.2 angegebenen Mengen der verschiedenen Ausführungseinheiten basieren auf folgenden Überlegungen:

- *Config1* stellt eine superskalare *Start-Up*-Konfiguration dar
- *Config2* ist für Programme mit einem hohen Anteil an Integer-Operationen geeignet
- *Config3* soll für die Ausführung von Programmen mit vielen Gleitkomma-Operationen geladen werden.

Besonders zu beachten ist die Tatsache, dass bei beiden alternativen Konfigurationen (*Config2* und *Config3*) die Zahl der LSUs gleichermaßen heraufgesetzt wird. Sie sollen die vielen, bei intensiven Rechenoperationen anfallenden Speicherzugriffe, z.B. um Operanden zu lesen, beschleunigen und somit zu einer insgesamt besseren Leistung beitragen.

- ***Instruction Memory***

  Bekanntermaßen besteht ein Programm aus einer Folge von Befehlen und Daten, die vor der Programmausführung in den Arbeitsspeicher geladen werden. Das *Instruction Memory* bildet den Teil des Arbeitsspeichers, welcher ausschließlich für die Speicherung von Befehlen verwendet wird. Dieser Teil wird während der Programmabarbeitung nur gelesen, da selbst-modifizierender Code nicht vorgesehen ist. In der modellierten Mikroarchitektur wird während der Befehlsholphase zunächst versucht, Befehle aus dem prozessorinternen *Trace Cache* zu holen. Nur im Fall eines *Trace Cache-Miss* werden die Befehle aus dem *Instruction Memory* übernommen. Erwartungsgemäß ist zum Beginn der Programmausführung der *Trace Cache* leer, so dass zwangsläufig die ersten auszuführenden Befehle aus dem *Instruction Memory* kommen. Mit der Fortsetzung der Programmausführung füllt sich der *Trace Cache* zunehmend. Erst später trifft dann der Fall zu, dass die aktuelle *Fetch*-Adresse mit einem Eintrag im *Trace Cache* übereinstimmt und die Befehle von dort übernommen werden können.

- ***Data Memory***
  Bei modernen Mikroprozessoren wird die Trennung zwischen Befehlsspeicher und Datenspeicher bevorzugt, um den hohen Datenverkehr zwischen Prozessor und Arbeitsspeicher über einen einzigen Speicherbus zu entschärfen (*Harvard*-Architektur, vgl. Abschn. 2.4.2). Aus dem gleichen Grund wird im vorliegenden Modell ebenfalls ein getrennter Datenspeicher vorgesehen. Gleichzeitig wird damit auf eine zusätzliche Komponente zur Arbitrierung des Zugriffs auf einen gemeinsamen Speicherbus verzichtet. Das *Data Memory* speichert die in einem Programm definierten Daten (Variablen). Diese werden durch die LSUs mittels *Load*-Befehlen als Argumente für die anfallenden Rechenoperation geladen. Ergebnisse aus den Berechnungen werden wiederum durch *Store*-Befehle in diesem Datenspeicher dauerhaft gespeichert.

- ***Configuration Manager***
  Wie bereits erwähnt wird der *Configuration Manager* durch die *Fetch Unit* über die Art der gerade abgeholten Befehle unterrichtet. Daraufhin überprüft er, ob die aktuell geladene Konfiguration den erforderlichen Ausführungseinheiten am besten entspricht. Ist dies der Fall, unternimmt der *Configuration Manager* keinen weiteren Schritt. Andernfalls muss er herausfinden, welche von den vordefinierten Konfigurationen am besten den Anforderungen genügt (s. Abb. 3.3). Gegebenenfalls initiiert er eine partielle Rekonfiguration, damit die passenden Ausführungseinheiten zur Verfügung gestellt werden. Zu diesem Zweck wird ein zusätzlicher Speicher benötigt, in dem die vordefinierten Konfigurationen vorgehalten werden.

Damit ist das Modell einer rekonfigurierbaren Mikroarchitektur definiert, welche als Basis für die nachfolgenden Untersuchungen anhand von Software-Simulation und Hardware-Implementierung dienen wird.

3.3 Modellsimulation

Heutzutage werden zunehmend Systeme (Mikroprozessor, Motor, Immobilie) vor ihrer endgültigen Implementierung zuerst simuliert, wenn sie einen gewissen Komplexitätsgrad überschreiten. Dabei werden im Allgemeinen Computerprogramme entwickelt, welche das zu entwickelnde System in unterschiedlichen Abstraktionsstufen darstellen. Damit ist es möglich, das Systemverhalten in einer frühen Phase der Entwicklung auf eventuelle Schwachstellen hin zu untersuchen und gegebenenfalls zu optimieren. Der Vorteil liegt auf der Hand: Ohne Verbrauch von wertvollen Materialien können unterschiedliche Systemparameter und somit unterschiedliche Modelle erprobt werden. Schließlich wird das Modell implementiert, welches am besten den Anforderungen oder den persönlichen Wertvorstellungen entspricht.

In diesem Sinne wird das im vorangegangenen Abschnitt vorgestellte Modell zunächst mit Hilfe einer Software simuliert. Das Ziel dabei ist zu untersuchen, wie es sich mit der daraus resultierenden Prozessorleistung verhält. Darüber hinaus soll mit Hilfe der durch die Software gebotenen Flexibilität die partielle und dynamische Hardware-Rekonfiguration nachgebildet werden. Von großem Vorteil ist auch die damit gegebene Möglichkeit, alternative Rekonfigurationsmodelle in Betracht zu ziehen und sich ein insgesamt besseres Verständnis der Vorgänge in einem Mikroprozessor zu verschaffen. All dies bildet die Grundlage einer anschließenden Implementierung der Mikroarchitektur auf realer Hardware.

3.3.1 Der *SimpleScalar* Simulator

Für die Simulation der modellierten rekonfigurierbaren Mikroarchitektur wird ein geeigneter Prozessor-Simulator benötigt. Einer der möglichen Lösungswege besteht darin, einen eigenen Simulator dafür selbst zu implementieren. Dies bietet den Vorteil, dass man damit über eine freie Auswahl der Tools, z.B. Programmiersprache, Entwicklungsumgebung etc. verfügt. Außerdem lassen sich dadurch unterschiedliche Simulationsziele — funktionale und/oder zeitliche Simulation — leichter realisieren. Leider ist ein solches Vorhaben nur mit einem erheblichen Zeitaufwand zu bewältigen. Deshalb

fiel im Rahmen dieser Arbeit die Entscheidung dahingehend, einen in der Forschung sehr verbreiteten Prozessor-Simulator zu verwenden und diesen so zu erweitern, dass er die gestellten Anforderungen hinreichend erfüllt. Damit lässt sich der Zeitaufwand hierfür enorm verringern und stattdessen besser in die anschließende aufwendige Hardware-Implementierung investieren.

Auf dem Gebiet der Prozessorarchitekturen hat sich seit einigen Jahren ein frei verfügbarer Simulator als sehr sinnvoll und nützlich erwiesen. Er wird in zahlreichen Forschungsprojekten verwendet, deren Ergebnisse auf renommierten internationalen Konferenzen wie MICRO[1], ISCA[2], HPCA[3] sowie in auf diesem Gebiet hoch angesehenen wissenschaftlichen Zeitschriften (z.B. *IEEE Computer Society*) veröffentlicht werden. Es handelt sich um den *SimpleScalar* Simulator, der aus einer Dissertation an der Wisconsin-Madison Universität (USA) hervorgegangen ist und bis heute weltweit eine breite Verwendung findet [BAB97, Sim05]. Abbildung 3.5 zeigt einen Überblick der Funktionsweise des *SimpleScalar*-Simulators in der ursprünglichen Version.

Das *SimpleScalar Tool Set* besteht aus einer Reihe von Prozessor-Simulatoren, die unterschiedliche Zielsetzungen verfolgen. So ist beispielsweise der einfachste Simulator (*sim-fast*) lediglich in der Lage, einzelne Befehle eines Programms streng sequentiell abzuarbeiten. Insbesondere ist er auf die Laufzeit optimiert und dementsprechend äußerst einfach gehalten (300 Code-Zeilen). Von größerer Bedeutung ist dagegen der komplexere und umfangreichere Simulator *sim-outorder*. Dieser Simulator ist genau wie alle im *Tool Set* mitgelieferten Simulatoren in der Programmiersprache C implementiert. Das Besondere an *sim-outorder* ist, wie sein Name schon erkennen lässt, dass er einen Prozessor mit einer *out-of-order* Befehlsausführung simuliert. Er beinhaltet alle bis zu seiner Fertigstellung (1996) aktuellen

[1] *International Symposium on Microarchitecture*

[2] *International Symposium on Computer Architecture*

[3] *International Symposium on High-Performance Computer Architecture*

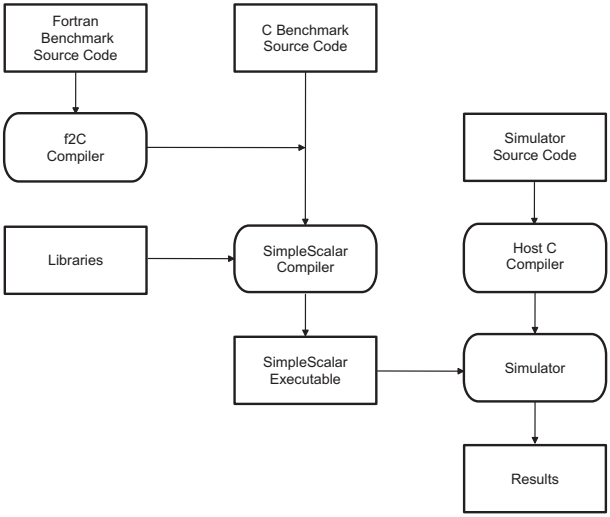

Abbildung 3.5: *Das SimpleScalar Tool Set [BAB97]*

Techniken der Mikroprozessortechnik. Dazu zählen die Superskalarität, die genannte *out-of-order* Ausführung, die Sprungvorhersage-Technik, der Einsatz von Cache-Speichern etc. Da sein Quellcode offen gelegt ist, bietet er die Möglichkeit, sehr schnell neue Ideen durch eine einfache Erweiterung des Simulators zu testen, bevor sie später in innovative Produkte umgesetzt werden. Diese Vorgehensweise hat sich durch die zahlreichen Verbesserungen auf dem Gebiet der Mikroprozessortechnik bewährt.

Mit Hilfe eines mitgelieferten *SimpleScalar* Compilers ist es möglich, beliebige C-Programme speziell für diesen Simulator zu übersetzen. Dabei wird ein modifizierter GCC-Compiler verwendet und anhand der dazu gehörigen Bibliotheken ein ausführbares Programm generiert. Fortran-Programme lassen sich ebenfalls in ein auf *SimpleScalar* ausführbares Programm übersetzen. Dafür wird zunächst das Fortran-Programm durch einen *Cross*-Compiler (*f2C*) in ein C-Programm übersetzt. Anschließend durchläuft das so erstellte Programm die gleichen Schritte wie ein übliches C-Programm. In jüngster Zeit wurden ein weiterer Simulator und ein entsprechender

Cross-Compiler zur Verfügung gestellt, mit deren Hilfe sich auch der ARM-Befehlssatz [ARM00] simulieren lässt.

Bevor ein erfolgreich übersetztes Programm auf dem *SimpleScalar*-Simulator ausgeführt werden kann, muss der zunächst als C-Quellcode vorliegende Simulator generiert werden. Zu diesem Zweck bedient man sich eines beliebigen GNU-kompatiblen C-Compilers. Das Ergebnis des Übersetzungsvorgangs ist wiederum ein Programm, dessen Ausführung die Funktionsweise eines implementierten Mikroprozessors simuliert. Dabei ist für eine sinnvolle Simulation mindestens ein Parameter unbedingt notwendig: Es muss angegeben werden, welches Programm der simulierte Prozessor nun ausführen soll. Ein derartiger Aufruf der Simulation sieht folgendermaßen aus:

```
sim-outorder test-math
```

wobei `sim-outorder` dem Aufruf des Simulators entspricht und `test-math` das Programm ist, das durch den Simulator ausgeführt werden soll.

Die Ausgabe der Simulation besteht aus zwei Teilen. Einerseits wird das Ergebnis des durch den Simulator ausgeführten Programms ausgegeben. Dies kann z.B. die Berechnung einer mathematischen Funktion sein. Andererseits — und das ist für den Entwickler das Wesentliche — erhält man ein umfangreiches Protokoll, welches eine Reihe von Leistungsmerkmalen des simulierten Prozessors enthält. Nachfolgend sind Beispiele solcher Daten aufgelistet:

- Die Anzahl der ausgeführten Befehle.
- Die gesamte Simulationszeit in Taktzyklen und in Sekunden.
- Die durchschnittliche Anzahl der ausgeführten Befehle pro Taktzyklus (*Instructions Per Cycle*, IPC).
- Die durchschnittliche Anzahl der benötigten Taktzyklen, um einen Befehl auszuführen (*Cycles Per Instruction*, CPI).
- Die Anzahl der *Cache-Hits* bzw. *Cache-Misses*.
- Der in Anspruch genommene Speicher (in Seiten ausgedruckt).
- Die Anzahl der richtig bzw. falsch vorhergesagten Sprünge etc.

Der Simulator *sim-outorder* aus dem *SimpleScalar Tool Set* bietet drei wesentliche Vorteile. Erstens können mit Hilfe der vielen einstellbaren Parameter unterschiedliche Mikroarchitekturen in einfacher Weise erprobt werden. Der Benutzer hat die Möglichkeit, bei jedem Simulationslauf die Parameter mit anderen Werten zu belegen und somit eine Änderung der Mikroarchitektur herbeizuführen. Somit können z.B. unterschiedliche Cache-Strategien durch einfache Variation der Cache-Parameter untersucht werden. Für einen anderen Benutzer kann z.B. das Interesse an der Untersuchung einer geeigneten Sprungvorhersage-Technik liegen. In diesem Fall würde er die entsprechenden Parameter bei jeder Programmausführung ändern und protokollieren, wie sich die Leistung dadurch beeinflussen lässt.

Der zweite Vorteil basiert darauf, dass das Befehlsformat der *SimpleScalar* Prozessorarchitektur bis zu 64 Bits vorsieht. Davon bleiben 16 Bits unbenutzt und sollen Benutzern eine Möglichkeit geben, zusätzliche Befehle zum ursprünglichen Befehlssatz hinzuzufügen.

Drittens wird durch die bereits erwähnte Offenlegung des C-Quellcodes (und bei Zustimmung zum beigelegten *Copyright*) ermöglicht, den Simulator für eigene Zwecke zu verändern oder zu erweitern. Hiervon wird in der vorliegenden Arbeit Gebrauch gemacht, indem der Simulator so weiterentwickelt wurde, dass das im vorangegangenen Abschnitt dargestellte Modell einer rekonfigurierbaren Mikroarchitektur damit umfassend simuliert werden kann.

3.3.2 Erweiterung des *SimpleScalar* Simulators

Der Simulator *sim-outorder* weist in seiner ursprünglichen Version viele Eigenschaften auf, die sich in der modellierten Mikroarchitektur wiederfinden. Pipelining, Superskalarität, *out-of-order* Ausführung usw.; alle diese Konzepte sind bereits darin verwirklicht. Er muss somit lediglich um die speziellen Komponenten zur Unterstützung der partiellen und dynamischen Rekonfiguration erweitert werden. Dazu zählen der *Trace Cache*, der *Predecoder* sowie der *Configuration Manager*.

Aufgrund der Komplexität des *Trace Cache* wird er als zusätzliche, eigenständige Komponente des Simulators realisiert und wird im nächsten Ab-

satz näher behandelt. Der *Predecoder* dagegen kann mit wenig Mühe als zusätzlicher Block im verfügbaren Simulator-Quellcode implementiert werden. Dies erfolgt in dem Teil des Quellcodes, welcher die *Fetch*-Phase realisiert. An dieser Stelle gilt es hauptsächlich, die zu jedem vorkommenden Befehl passende *Ausführungseinheit* zu ermitteln. Dies geschieht durch die Überprüfung der im Befehl codierten Operation. Diese Angabe wird zusammen mit dem jeweiligen Befehl im *Trace Cache* gespeichert. Des Weiteren wird im gleichen Codesegment das Holen der Befehle entweder aus dem *Trace Cache* oder aus dem bisherigen Befehlsspeicher eingeleitet. Wenn eine Befehlsfolge aus dem *Trace Cache* geholt werden soll, werden gleichzeitig die dafür benötigten Ausführungseinheiten geprüft. Daraufhin werden die Parameter aktualisiert, welche die Anzahl der tatsächlich vorhandenen Ausführungseinheiten angeben. Auf diese Weise wird zugleich die Funktion des *Configuration Manager* realisiert. Es sei noch erwähnt, dass anstatt diese Parameter während der Programmausführung laufend zu aktualisieren, diese gleich vor jedem Simulationslauf unterschiedlich belegt werden können. Damit kann zusätzlich eine statische Rekonfiguration (vgl. Abschn. 2.2.3) simuliert werden.

Zum Schluss der Vorstellung des erweiterten Simulators soll noch auf die Software-Implementierung des *Trace Cache* näher eingegangen werden. Es handelt sich lediglich um eine einfache Implementierung, da diese Komponente trotz ihrer großen Bedeutung nicht den Kern der vorliegenden Arbeit bildet. Die Implementierung basiert auf dem bereits im vorherigen Abschnitt dargestellten Modell eines *Trace Cache* (siehe Abb. 3.4). Die darin definierten *Trace Lines* werden mit Hilfe einer zyklischen Warteschlange (*queue*) verwaltet, wobei jede *Trace Line* durch eine bestimmte Position in der Warteschlange gekennzeichnet wird. Das Auffüllen der Befehle in den *Trace Cache* geschieht dadurch, dass zunächst eine gültige Warte-Position belegt wird und anschließend Befehle in die dazu gehörigen *Trace Line* aufgenommen werden. Falls kein Platz mehr in der Warteschlange zur Verfügung steht — alle *Trace Lines* sind mit Befehlen gefüllt — werden zyklisch die zuerst belegten *Trace Lines* überschrieben. Eine wandernde Marke gibt stets die aktuelle Position in der Warteschlange an, wo sich die nächste freie bzw.

zu überschreibende *Trace Line* befindet. An dieser Stelle ist festzustellen, dass andere Formen der Verwaltung eines solchen *Trace Cache* denkbar sind; einige Arbeiten beschäftigen sich speziell mit dieser Problematik [FPP97].

Die maximale Anzahl der vorhandenen *Trace Lines* (Tiefe des *Trace Cache*) sowie die Länge jeder *Trace Line* (Breite des *Trace Cache*) werden in globalen Variablen gehalten und können daher leicht verändert werden. Durch diese Parametrisierung lässt sich dementsprechend die Speicherkapazität des gesamten *Trace Cache* flexibel gestalten. In der implementierten Version wurde die Tiefe auf 8 *Trace Lines* und die Breite auf 16 Befehle voreingestellt. Die Bestimmung der Breite ergibt sich aus der Annahme, dass maximal 16 Befehle mit einem einzigen Sprungbefehl als zusammenhängende Befehlsfolge aus einem Programm extrahiert werden können. Angesichts des recht hohen Anteils von Sprungbefehlen innerhalb von Programmen [PH96] kann man bei diesen Gegebenheiten von einer näherungsweise angemessenen Breite ausgehen.

Zu jedem Befehl werden zusätzlich der entsprechende Befehlszählerstand sowie die dafür benötigte Ausführungseinheit gespeichert. Bei einer späteren Suche nach einer bestimmten Befehlsfolge im *Trace-Cache* findet ein Vergleich zwischen der aktuellen *Fetch*-Adresse und dem gespeicherten Befehlszählerstand statt. Ein *Trace-Cache-Hit* entsteht, wenn es zur Übereinstimmung kommt. Andernfalls wird ein *Trace-Cache-Miss* ausgegeben.

3.3.3 Leistungsbewertung

Der erweiterte *SimpleScalar* Simulator!erweitert wurde verwendet, um die im Abschnitt 3.2 beschriebene rekonfigurierbare Mikroarchitektur hinsichtlich der erzielbaren Leistung zu untersuchen. Zu diesem Zweck wurden unterschiedliche Testprogramme herangezogen, welche zunächst für die *SimpleScalar*-Architektur übersetzt wurden. Anschließend wurden sie durch die simulierte Mikroarchitektur ausgeführt. Unter den vielen frei wählbaren Testprogrammen fiel die Entscheidung auf den SPEC CPU2000-Benchmark, da er eine breite Akzeptanz sowohl in der Industrie als auch in der Wissenschaft genießt [SPE05]. Es handelt sich um eine Ansammlung von an-

wendungsorientierten Testprogrammen, die eine international anerkannte Vergleichbarkeit von Computersytemen ermöglicht.

3.3.3.1 Simulationsdaten

Die SPEC CPU2000-Testprogramme sind in zwei Gruppen gegliedert: Die erste Gruppe, CINT2000, enthält Programme, welche vorwiegend integerbasierte Berechnungen aufweisen. Dagegen umfasst die zweite Gruppe, CFP2000, solche Anwendungen, die hauptsächlich auf Gleitkomma-Operationen aufgebaut sind. Im Allgemeinen lässt sich der gesamte SPEC CPU2000-*Benchmark* für jedes beliebige Zielsystem mittels eines standardisierten Compilers übersetzen. Der anschließenden Programmausführung werden die jeweiligen systemabhängigen Leistungsdaten entnommen und veröffentlicht. Somit können unterschiedliche Rechensysteme untereinander quantitativ verglichen werden.

Ein Nachteil dieses Simulationsaufbaus ist, dass es leider nicht gelungen ist, alle aus dem SPEC CPU2000-*Benchmark* verfügbaren Programme mit dem speziellen *SimpleScalar* Compiler zu übersetzen. Dies ist darauf zurück zu führen, dass dieser Compiler zwar aus dem GCC entwickelt worden ist, jedoch ist er durch die Anpassung auf den *SimpleScalar*-Simulator zu speziell geworden. So lassen sich zwar einfache Testprogramme übersetzen, aber für manche komplexen Anwendungen aus den gängigen Benchmarks ist ein zusätzlicher Software-Aufwand bzw. eine Programmänderung erforderlich.

Einige der erfolgreich kompilierten Testprogramme sind in der Tabelle 3.1 aufgetragen. Diese bilden die Grundlage der aufgeführten Simulationsergebnisse.

Ein weiterer Nachteil betrifft die Laufzeiten der Simulationen. Bei der Ausführung dieser zumeist rechenintensiven Anwendungen sowohl auf einem realen als auch auf einem simulierten Rechensystem müssen unterschiedliche Testdaten herangezogen werden. Beispielsweise benötigt das Testprogramm „*gzip*“ einen Parameter, welcher die zu komprimierende Datei angibt. Für jedes Programm werden solche Testdaten mitgeliefert, die sich in ihrem Umfang allerdings stark unterscheiden. Je nach ausgeführtem Testprogramm

	Name	Beschreibung
CINT2000	164.gzip	Programm zur Datenkompression
	175.vpr	Place&Routing Algorithmen für FPGAs
	176.gcc	C Compiler
CFP2000	183.equake	Simulation von Finiten Elementen Modellierung von Erdbeben
	188.ammp	Programm mit chemischen Berechnungen

Tabelle 3.1: *Ausschnitt aus dem SPEC CPU2000 Benchmark*

und in Abhängigkeit der verwendeten Testdaten kann die Durchführung eines Testlaufs mehrere Stunden oder sogar Tage dauern. Diese langen Laufzeiten steigen noch überproportional, wenn das Programm auf einer simulierten Architektur ausgeführt wird, da in Software simulierte Systeme ja wesentlich langsamer als reale Hardware-Systeme sind. Genau dieser negative Aspekt trifft auf die im Laufe dieser Arbeit durchgeführten Simulationen zu, bei denen ein Ausschnitt des SPEC CPU2000-*Benchmark* durch die modellierte rekonfigurierbare Mikroarchitektur ausgeführt wird.

Während der Entwicklungsphase wurden zusätzlich die einfacheren, mit dem *SimpleScalar Tool Set* zur Verfügung gestellten Testprogramme herangezogen. Damit konnte doch noch hinreichend schnell simuliert werden. Darüber hinaus bestand die Möglichkeit, die hinsichtlich der Datenmenge optimierten Testdateien zusammen mit dem ursprünglichen SPEC CPU2000 *Benchmark* zu verwenden [KL02]. Alle diese Möglichkeiten kamen zum Einsatz, um den Zeitaufwand für die Software-Simulation zu reduzieren. Den unten stehenden endgültigen Ergebnissen liegen die nach [KL02] für *SimpleScalar* angepassten Testdaten zu Grunde.

3.3.3.2 Simulationsergebnisse

Das Modell einer rekonfigurierbaren Mikroarchitektur sollte nun auf eine potentielle Leistungsteigerung hin überprüft werden. Dafür wurde mit Hilfe des SPEC CPU2000 *Benchmark* die damit erzielbare Leistung der einer vergleichbaren Mikroarchitektur gegenübergestellt. Eine vergleichbare Mikroarchitektur ist sicherlich die zuvor erwähnte *sim-outorder*-Mikroarchitek-

tur in ihrer unveränderten Fassung. Denn sie ist bis auf die Rekonfiguration der Ausführungseinheiten und die damit verbundenen zusätzlichen Komponenten dem neu entwickelten Modell sehr ähnlich. Die mit *sim-outorder* simulierte Mikroarchitektur bildete daher die Referenz, nach welcher über die Güte des Modells entschieden wurde (*sim-ref*). In der Tabelle 3.2 ist zunächst aufgetragen, über welche Arten von Ausführungseinheiten und in welcher Anzahl diese Mikroarchitektur verfügt. Die dort eingetragenen Werte blieben während eines Simulationslaufs unverändert, da hierfür keine Hardware-Rekonfiguration stattfand. Weiterhin befinden sich in der gleichen Tabelle unter der Bezeichnung *sim-predef* die drei austauschbaren Konfigurationen, wie sie in der modellierten Mikroarchitektur vorgeschlagen sind (*config1, config2, config3*).

Dank der durch die Software gebotenen Flexibilität konnten darüber hinaus zwei weitere Rekonfigurationsmodelle simuliert werden. Zum einen handelt es sich um ein hochgradig dynamisch rekonfigurierbares Modell, bei dem es keine vordefinierten Konfigurationen wie bei *sim-predef* gibt. Stattdessen werden die verschiedenen Ausführungeinheiten im laufenden Betrieb zwischen einem minimalen (1) und einem maximalen (4) Wert gemäß der aktuellen Software-Anforderungen variiert (*sim-dynrec*). Dabei orientiert sich die obere Grenze an der existierenden Beschränkung hinsichtlich der verfügbaren Hardware-Ressourcen. Diese Art von Rekonfiguration ist eher als idealistisch zu betrachten, da sie sich mit den heute zu Verfügung stehenden Mitteln kaum in Hardware umsetzen ließe. Zum anderen wird die statische Rekonfiguration mit zwei unterschiedlichen Konfigurationen simuliert (*sim-statrec1, sim-statrec2*). Laut Abschnitt 2.2.3 aus dem zweiten Kapitel geht es bei dieser Art der Rekonfiguration darum, bei jedem Simulationslauf unterschiedliche Konfigurationen zu verwenden, die sich aber während der Ausführung nicht mehr ändern. Zu diesem Zweck werden beim Aufruf des Simulators die entsprechenden Parameter durch die in der Tabelle angegebenen Werte belegt.

Um die Leistung der fünf simulierten Mikroarchitekturen quantitativ untereinander vergleichen zu können, werden zwei der nach [PH96] wichtigsten Leistungsparameter herangezogen. Als erster Parameter wird die Anzahl

		Int-ALU	Int-MDU	LSU	FP-ALU	FP-MDU
sim-ref		4	1	2	4	1
sim-predef	config1	2	1	2	1	1
	config2	4	2	4	1	1
	config3	2	1	4	2	2
sim-dynrec		1 - 4	1 - 4	1 - 4	1 - 4	1 - 4
sim-statrec1		4	2	4	1	1
sim-statrec2		2	1	4	2	2

Tabelle 3.2: *Simulierte Mikroarchitekturen*

ausgeführter Befehle pro Taktzyklus (*Instructions Per Cycle*, IPC) dem Vergleich zu Grunde gelegt. Abbildung 3.6 zeigt je nach ausgeführtem Testprogramm, wie sich dieser Parameter in Abhängigkeit der simulierten Mikroarchitekturen verhält. Es sind zwar keine großen Unterschiede zu beobachten, dennoch kann man feststellen, dass die *sim-dynrec*-Mikroarchitektur bei einigen Programmen (*route, equake, ammp*) zu besseren Ergebnissen führt. Außerdem ist erkennbar, dass sich die realistischere Mikroarchitektur *sim-predef* bei allen verwendeten Testprogrammen als zweitbeste Lösung erweist.

Abbildung 3.7 dagegen vergleicht die fünf Mikroarchitekturen anhand eines weiteren Leistungsparameters: Aufgetragen wird die Anzahl an Taktzyklen, die durchschnittlich für die Ausführung eines Befehls benötigt werden (*Cycles Per Instruction*, CPI). Je weniger Taktzyklen für die Ausführung eines Befehls benötigt werden, umso besser ist die entsprechende Mikroarchitektur. Demzufolge gilt die gleiche Schlussfolgerung wie vorher, wonach *sim-dynrec* und *sim-predef* einen geringen Vorsprung im Leistungspotenzial vorweisen.

Die Flexibilität der Software ließ die Durchführung noch weiterer Untersuchungen zu. So konnte in dieser Arbeit z.B. auch folgender Fragestellung nachgegangen werden: Wie verhält sich die Leistung der verschiedenen Mikroarchitekturen, wenn die zur Verfügung stehenden Hardware-Ressourcen erweitert werden? Dieser Sachverhalt wird teilweise dadurch simuliert, dass die Anzahl einiger Ausführungseinheiten erhöht wird (s. Tabelle 3.3).

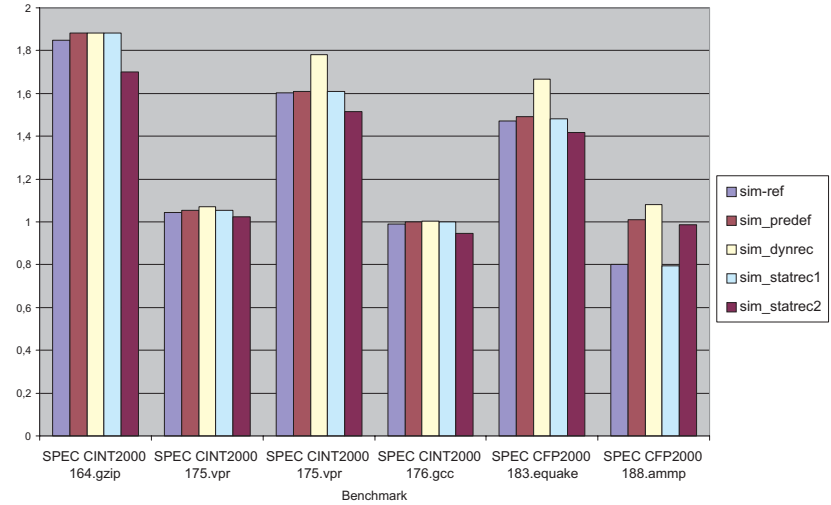

Abbildung 3.6: *Instructions Per Cycle*

Die daraus resultierenden Leistungswerte bezüglich IPC sind in Abbildung 3.8 dargestellt. Die simulierte Erweiterung von Hardware-Ressourcen bringt offensichtlich keine Leistungssteigerung gegenüber den vorherigen Ergebnissen. Der Grund dafür liegt darin, dass im Allgemeinen Programme und insbesondere der verwendete *Benchmark* nur eine eingeschränkte Befehlsparallelisierung (*Instruction-Level Parallelism*, ILP) aufweisen [PH96]. Eine

		Int-ALU	Int-MDU	LSU	FP-ALU	FP-MDU
sim-ref		4	1	2	4	1
sim-predef	config1	4	1	2	4	1
	config2	8	4	4	1	1
	config3	2	1	4	4	4
sim-dyn-rec		1 - 8	1 - 8	1 - 8	1 - 8	1 - 8
sim-stat-rec1		8	4	4	1	1
sim-stat-rec2		2	1	4	4	4

Tabelle 3.3: *Simulation mit erweiterten Hardware-Ressourcen*

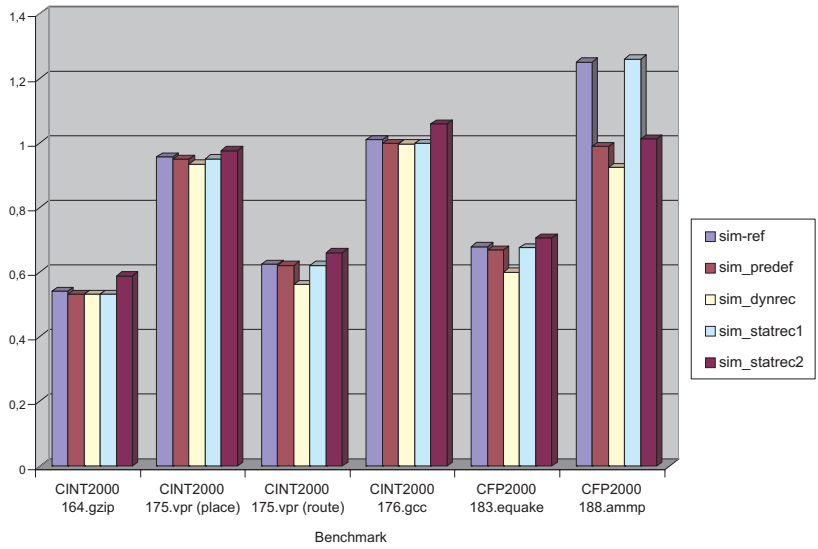

Abbildung 3.7: *Cycles Per Instruction*

Erhöhung der Hardware-Ressourcen kann sich auf die Leistung nur in dem Verhältnis auswirken, wie sich das ausgeführte Programm parallelisieren lässt. In diesem Fall scheint die erreichbare Befehlsparallelisierung bereits mit dem ersten Ansatz erschöpft zu sein, so dass zusätzliche Ausführungseinheiten zu keiner weiteren Leistungszunahme führen.

Im Zusammenhang mit der Simulation der eher theoretischen dynamisch rekonfigurierbaren Mikroarchitektur (*sim-dynrec*) wurde weiterhin untersucht, wie sich die Nachfrage nach den verschiedenen Ausführungseinheiten verhält. Für diesen Versuch wird protokolliert, wie viele von den verfügbaren Ausführungseinheiten tatsächlich in Anspruch genommen werden. Abbildung 3.9 zeigt die erzielten Ergebnisse. Es ist daraus zu beobachten, dass die Int-ALUs, die LSUs und die FP-ALUs bei der Ausführung aller Testprogramme stets bis zum gesetzten Maximum von jeweils vier Einheiten verwendet wurden. Deutlich zu erkennen ist weiterhin die starke Nachfrage

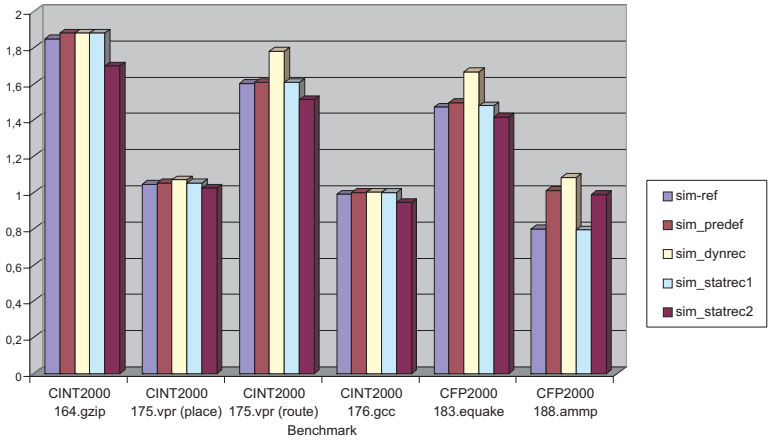

Abbildung 3.8: *Instructions Per Cycle bei erweiterten Hardware-Ressourcen*

nach FP-MDUs nur bei den CFP2000-Programmen (*equake, ammp*), die einen hohen Anteil an Gleitkomma-Operationen beinhalten.

Die erzielten Simulationsergebnisse lassen zunächst keinen beeindruckenden Leistungsgewinn durch die modellierte rekonfigurierbare Mikroarchitektur prognostizieren. Weil dieser neue Ansatz aber auch keine Leistungseinbußen zeigt, ist der Weg zu weiteren Schritten frei, um den Synergieeffekt zwischen rekonfigurierbarer Hardware und Mikroprozessortechnik untersuchen zu können. Dies kann zum einen durch eine weiter verfeinerte Simulation erfolgen. Dabei können zusätzliche Parameter, z.B. die Rekonfigurationszeit, Chipfläche, Verlustleistung etc. einbezogen werden. Zum anderen bietet sich an, die modellierte Mikroarchitektur auf einer rekonfigurierbaren Hardware zu implementieren, um somit die gleichen Parameter durch realistische und physikalisch messbare Werte bestimmen zu können. Dieser Weg wurde im Laufe der vorliegenden Arbeit eingeschlagen. Das nächste Kapitel widmet sich deshalb ausführlich dieser Thematik. Dabei steht zunächst die Frage nach der praktischen Umsetzung des Konzeptes einer partiell und

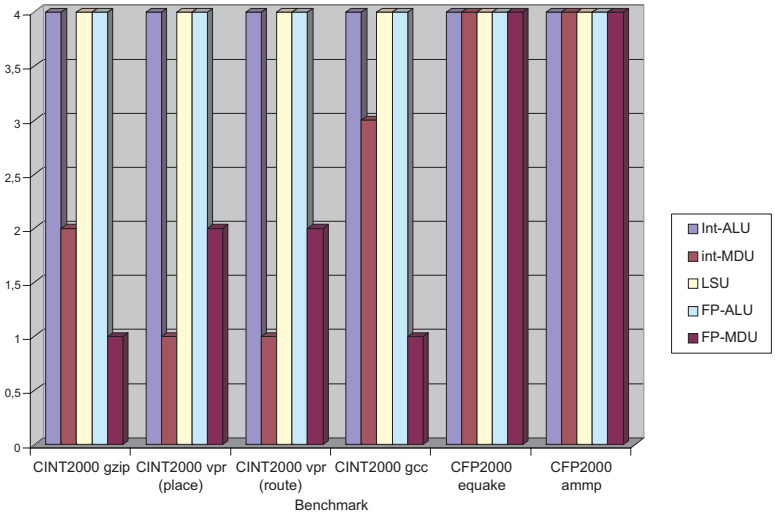

Abbildung 3.9: *Anforderung der verschiedenen Ausführungseinheiten*

dynamisch rekonfigurierbaren Mikroarchitektur im Mittelpunkt. Erst wenn die Machbarkeit des Konzeptes bewiesen ist (*proof-of-concept*), lohnt sich anschließend eine umfassende Leistungsbetrachtung.

4 Hardware-Implementierung der modellierten Mikroarchitektur

Im Mittelpunkt dieses Kapitels steht die im Rahmen der vorliegenden Arbeit durchgeführte Implementierung der rekonfigurierbaren Mikroarchitektur auf realer Hardware. Zunächst werden die wichtigsten Entscheidungen erörtert, die zur Vorbereitung eines solchen Vorhabens erforderlich sind. Im Anschluss daran wird die Implementierung der superskalaren Befehlspipeline ausführlich behandelt. Sie bildet die Grundlage für eine abschließende Durchführung der dynamischen und partiellen Rekonfiguration der Mikroarchitektur, wie sie im vorangegangenen Kapitel modelliert wurde.

4.1 Vorbereitungen

Wie in den vorangegangenen Abschnitten mehrfach verdeutlicht, handelt es sich bei einer Mikroarchitektur — rekonfigurierbar oder fest verdrahtet — um die spezielle Implementierung einer bestimmten Prozessorarchitektur. Sie umfasst insbesondere die zumeist vertraulich gehaltene Hardware-Realisierung eines ausgewählten Befehlssatzes (auch Microsoft veröffentlicht nicht den Source-Code seines Windows-Betriebssystems). Bevor eine Mikroarchitektur implementiert werden kann, gilt es, einen geeigneten Befehlssatz zu definieren oder aus den vielen frei verfügbaren *Instruction Set Architectures* den für die jeweiligen Zwecke am besten passenden zu identifizieren.

Für die Hardware-Implementierung der in dieser Arbeit modellierten Mikroarchitektur erwies sich die simulierte *SimpleScalar*-Architektur aufgrund ihres breiten Befehlsformats von 64 Bit als nicht besonders geeignet. Eine dazugehörige Mikroarchitektur würde einen zu hohen Verbrauch von Hardware-Ressourcen mit sich bringen, wenn man von einer parallelen Aus-

führung mehrerer solcher Befehle ausgeht. Aus diesem Grund wurde stattdessen der ARM-*Thumb*-Befehlssatz [ARM00] vorgezogen, welcher lediglich ein Befehlsformat von 16 Bit vorsieht. Die ARM-Architektur wird zunehmend in zahlreichen stromsparenden und prozessorgesteuerten Geräten (PDAs, mobile Telefone, digitale Kameras etc.) eingesetzt und gilt daher als eine der modernsten Prozessorarchitekturen überhaupt.

Eine weitere Entscheidung, die vor der Hardware-Implementierung getroffen werden muss, betrifft die Ziel-Plattform. Es steht zwar fest, dass aufgrund der geforderten Hardware-Rekonfiguration nur ein FPGA als Ziel-Plattform in Frage kommt. Dennoch muss aus den vielfältigen Angeboten ein FPGA heraus gesucht werden, welches tatsächlich die ganz spezielle partielle und dynamische Rekonfiguration unterstützt. Eine entsprechende Marktanalyse ergab, dass lediglich eine bestimmte Familie von FPGAs der Firma Xilinx, *Virtex-II* (mittlerweile auch deren Nachfolger, *Virtex-Pro* und *Virtex-4*), den gestellten Anforderungen genügte.

Nachdem die beiden wichtigsten Voraussetzungen — *Befehlssatz* und Ziel-Plattform — geklärt waren, folgten die verschiedenen, in Abschnitt 2.3 beschriebenen Schritte zur Hardware-Implementierung der Prozessorpipeline mit Hilfe einer frei wählbaren Hardware-Entwicklungsumgebung.

4.2 Implementierung einer superskalaren Befehlspipeline

Der Weg zu einer partiell und dynamisch rekonfigurierbaren Mikroarchitektur führt zwangsläufig zur Implementierung einer Basis-Mikroarchitektur, welche als Ausgangspunkt für die nachfolgenden Schritte dient. Dieser notwendige Zwischenschritt ist besonders wichtig, da die Implementierung eines superskalaren Mikroprozessors schon für sich allein eine recht komplexe, zeitaufwendige und anspruchsvolle Aufgabe darstellt. Erst mit einer voll funktionsfähigen Mikroarchitektur kann der entstandene Entwurf leicht modifiziert werden, um die gewünschte partielle und dynamische Rekonfiguration einbringen zu können. Bei dieser letzten Aufgabe ist weniger der

geistige Scharfsinn gefordert; vielmehr ist der Einsatz anspruchsvoller und hoch technologischer Entwurfsmethoden und Tools von größerer Bedeutung.

Die im Laufe dieser Arbeit entwickelte Mikroarchitektur implementiert den Ressourcen-schonenden ARM-*Thumb*-Befehlssatz mit einer superskalaren Pipeline. Diese wird in fünf Phasen (Stufen) aufgeteilt, wie sie in der Abbildung 4.1 aufgezeigt werden. Es sind die aus Abschnitt 2.4.5 bekannten Pipeline-Phasen:

- Befehlsholphase (*Fetch Stage*)
- Decodierphase (*Decode Stage*)
- Zuordnungsphase (*Issue Stage*)
- Ausführungsphase (*Execute Stage*)
- Rückschreibphase (*Write Back Stage*)

Nachfolgend wird erörtert, wie die einzelnen Pipeline-Phasen tatsächlich in Hardware umgesetzt wurden. Abbildung 4.1 zeigt die Mikroarchitektur auf einen Blick.

4.2.1 Befehlsholphase

Bei dieser ersten Phase geht es bekanntlich darum, die einzelnen Befehle eines ausführbaren Programms der Prozessor-Pipeline taktweise zuzuführen. Abbildung 4.2 zeigt, aus welchen Mikroarchitektur-Komponenten solche Befehle abgeholt werden können und wie sie letztendlich in die Pipeline gelangen. Zu diesem Zweck besteht die übliche Lösung darin, einen Befehlsspeicher als Teil des Arbeitsspeichers einzusetzen, aus welchem die Befehle nach erfolgreichem Laden eines bestimmten Programms entnommen werden. Standardmäßig wird dabei der aktuelle Befehlszähler (PC) für die Generierung der benötigten Speicheradresse verwendet.

Für die physikalische Implementierung eines solchen Befehlsspeichers bieten sich zahlreiche Möglichkeiten an. Die ausgewählte Ziel-Plattform, das *Virtex-II*-FPGA, beinhaltet zwei Arten von Speicherlösungen. Zum einen

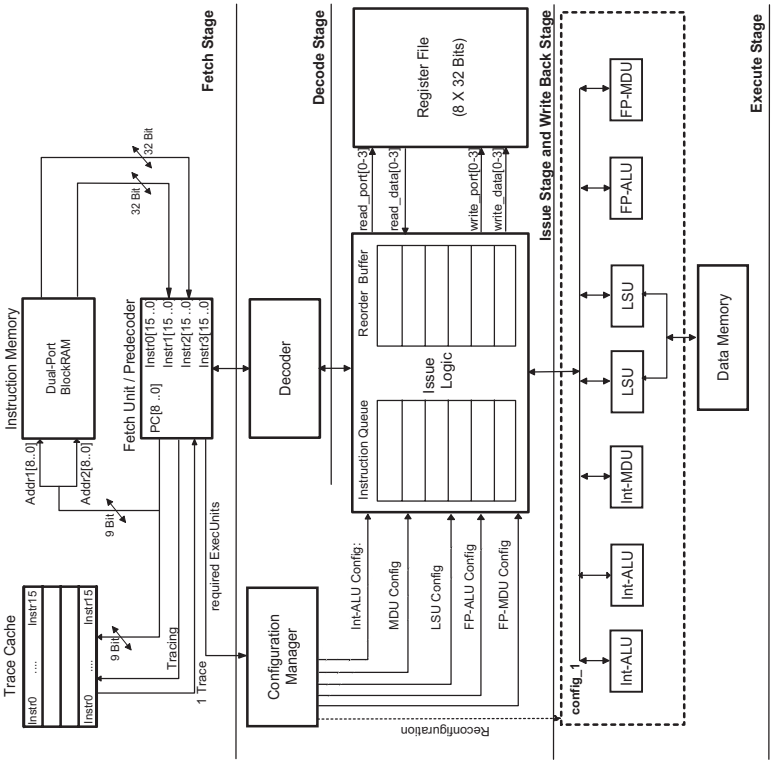

Abbildung 4.1: *Pipeline-Phasen der superskalaren Mikroarchitektur*

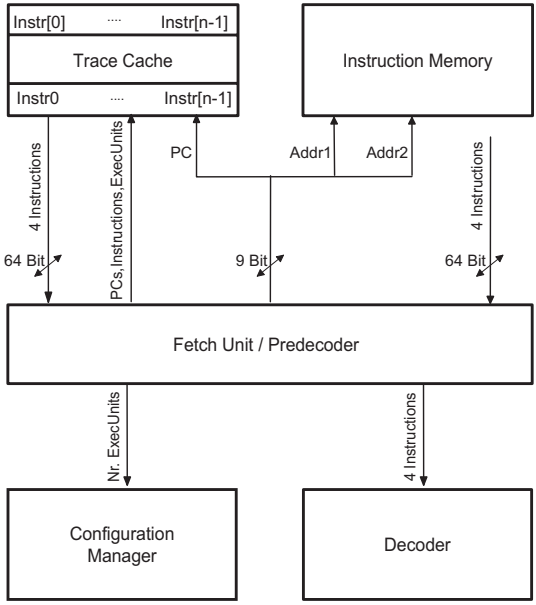

Abbildung 4.2: *Fetch Stage*

kann jeder *Configurable Logical Block* (CLB), wie bereits im Abschnitt 2.1.2 erwähnt, als ein 16 x 1-Bit Speicherelement konfiguriert werden. Daraus ergeben sich unterschiedliche Speichergrößen in Abhängigkeit von der Anzahl der CLBs, aus welchen ein konkretes FPGA dieser Familie aufgebaut ist. Darüber hinaus lässt sich ein einzelnes Speicherelement als *single port*- oder *dual port*-Speicher betreiben. Mit einer geschickten Zusammenlegung dieser einfachen Speicherelemente lassen sich beispielsweise aus dem Virtex-II-XC2V8000 bis zu 1456 Kbit ($\approx$ 186 KByte) als *distributed* RAM herstellen. Die für die Realisierung dieser Art von Speicher verwendeten CLBs stehen selbstverständlich nicht mehr zur Verfügung, um weitere logische Funktionen zu implementieren. Aufgrund des erwarteten hohen Bedarfs an CLBs, um die superskalare Mikroarchitektur vollständig zu implementieren, wurde auf diese Art von CLB-basiertem Speicher verzichtet.

Stattdessen wurde die zweite Speicherlösung bevorzugt, die auf dem 18-Kbit *Block SelectRAM* (Abb. 2.2) aufgebaut ist. Solche Speicherblöcke lassen

sich genauso wie der *distributed RAM* als *single port*- oder *dual port*-Speicher konfigurieren. Durch Kaskadierung ermöglichen sie beim XC2V8000-Baustein die Errichtung eines Speichers mit bis zu 3024 Kbit ($\approx$ 387 KByte) Speicherkapazität. *SelectRAM*-Blöcke befinden sich außerhalb von CLBs, genau genommen zwischen den CLB-Spalten, und beeinträchtigen somit in keiner Weise die Verwendung der gesamten CLB-Ressourcen, um die restlichen Funktionsblöcke der Mikroarchitektur zu implementieren.

Damit von Anfang an gleichzeitig eine hohe Anzahl an Befehlen in die Prozessor-Pipeline eingeleitet werden kann, wird die gebotene Möglichkeit eines *dual port*-Speichers wahrgenommen. Somit wird sichergestellt, dass die vielen parallelen Ausführungseinheiten mit auszuführenden Befehlen versorgt werden können, abgesehen von den möglichen Daten- und Kontroll-*Hazards*. Der Befehlsspeicher wurde für eine Wortlänge von 32 Bit konzipiert. Diese Zahl ergibt sich aus der ursprünglichen Überlegung, einen Speicher zu implementieren mit der Fähigkeit, die üblichen 32-Bit-Befehlsformate ohne zusätzlichen Hardware-Aufwand zu unterstützen. Im Falle des auf 16 Bit reduzierten Befehlsformats des ARM-*Thumb*-Befehlssatzes enthält eine solche Wortlänge bereits zwei Befehle. Aufgrund der in Anspruch genommenen *dual port*-Eigenschaft des *Virtex-II*-Speichers werden außerdem gleichzeitig über zwei Adressleitungen dementsprechend zwei Speicherworte mit jeweils zwei Befehlen (2×32 Bit) ausgelesen. Somit ergibt sich eine Fetch-Bandbreite von vier Befehlen pro Takt.

Wie vorher erwähnt, liefert die *Fetch-Unit* für die Ansteuerung des Befehlsspeichers die notwendigen Speicheradressen. Zum einen wird der aktuelle Stand des Befehlszählers (PC) als direkte Adresse für einen der beiden Ports des Speichers verwendet. Zum anderen erhält der zweite Port den um eins inkrementierten Befehlszählerstand aufgrund des sequentiellen Programmablaufs (von-Neumann-Prinzip, s. Abschn. 2.4.2). Somit werden aus dem Befehlsspeicher stets zwei benachbarte Speicherworte geholt, welche wiederum vier aufeinander folgende *Thumb*-Befehle beinhalten.

Neben dem regulären Befehlsspeicher wird in dieser Mikroarchitektur ein *Trace Cache* eingesetzt, welcher als zusätzlicher prozessorinterner Speicher realisiert wird. Dieser Speicher erhält — genauso wie der Befehlsspeicher

— bei jedem Takt den aktuellen Befehlszählerstand. Es findet dort ein Vergleich zwischen den darin gespeicherten Befehlsadressen mit dem gerade erhaltenen Befehlszählerstand statt. Im Falle einer Übereinstimmung werden die dadurch adressierten, aufeinander folgenden vier Befehle aus dem *Trace Cache* geholt und der *Fetch-Unit* zugeführt. Darüber hinaus erfährt der *Configuration Manager* über weitere Signale den zu diesen Befehlen gehörigen Bedarf an Ausführungseinheiten. Daraufhin entscheidet dann der *Configuration Manager*, welche Konfiguration für die Ausführung der anstehenden Befehle am besten geeignet ist. Falls die ausgewählte Konfiguration gerade nicht aktiv ist, so wird eine partielle Rekonfiguration des FPGA in die Wege geleitet, um damit dem dynamischen Verhalten des ausgeführten Programms gerecht zu werden. Bei einem Suchtreffer im *Trace Cache*, welcher über ein zusätzliches Signal der *Fetch-Unit* bekannt gegeben wird, wird vorzugsweise die Pipeline — und zwar der Decoder mit den Befehlen aus dem *Trace Cache* — aufgefüllt. Die gleichzeitig aus dem Befehlsspeicher kommenden Befehle werden in diesem Fall verworfen.

Die Voraussetzung dafür, dass das Befehlsholen sowie die Voraussage der benötigten Ausführungseinheiten mit Hilfe des *Trace Cache* richtig funktioniert, besteht darin, dass die dafür notwendigen Informationen zuvor mitprotokolliert werden. Dies geschieht einerseits, indem neben den Befehlen selbst laufend die dazu gehörigen Speicheradressen im *Trace Cache* gespeichert werden. Somit kann ein späterer Vergleich zwischen dem aktuellen Befehlszählerstand und den Befehlsadressen im *Trace Cache* zum Erfolg führen. Andererseits findet eine Vordecodierung der Befehle statt. Durch den *Predecoder* wird für jeden abgeholten Befehl festgestellt, welche Art von Operation darin enthalten ist und welche Ausführungseinheit dafür benötigt wird. Diese Angabe wird zusätzlich zu jedem Befehl in den *Trace Cache* aufgenommen.

Während der Programmausführung kann es vorkommen, dass die Pipeline teilweise zum Halten gezwungen wird (Pipeline-*Stall*). Ein solcher Zustand entsteht beispielsweise, wenn eine benötigte Hardware-Ressource gerade nicht zur Verfügung steht. Für die Behandlung dieses Problems wird bei jeder betroffenen Pipeline-Phase ein Puffer eingerichtet, welcher die nicht

übertragenen Signalzustände temporär speichert, bis die Pipeline wieder in voller Länge läuft. Ein solcher Puffer wird ebenfalls in der Fetch-Phase benötigt. Er speichert die zum Decoder ausgehenden Befehle, wenn aufgrund eines Pipeline-*Stall* der Decoder nicht in der Lage ist, die ankommenden Befehle aufzunehmen. Nach der Aufhebung des Pipeline-*Stall* werden zunächst die in diesem Puffer gespeicherten Befehle weitergereicht, bevor das Befehlsholen aus dem *Trace Cache* oder aus dem regulären Befehlsspeicher fortgesetzt werden kann.

4.2.2 Decodierphase

Der Decoder erhält bei jedem Taktzyklus einen Block von vier *Thumb*-Befehlen zusammen mit den korrespondierenden Inhalten des Befehlszählers (PC). Aus jedem dieser Befehle ermittelt er die folgenden Angaben :

- Die auszuführende Operation (Opcode)
- Die Register, welche die notwendigen Operanden enthalten (Operanden-Register)
- Den gegebenenfalls vorhandenen direkten Operanden (*immediate operand*)
- Das Ziel-Register für die anstehende Operation (*destination register)*

Zu jedem auszuführenden Befehl wird der entsprechende Befehlszählerstand benötigt, da sich für die Ausführung einiger Sprungbefehle die Sprungadresse aus dem aktuellen Stand des PC und einem im Befehl enthaltenen *Displacement* ergibt. Eine Alternativlösung für die Bereitstellung des erforderlichen PC bei solchen Sprungbefehlen besteht darin, ein spezielles Register zu verwenden, welches stets den aktuellen Befehlszählerstand zum gerade ausgeführten Befehl enthält. Allerdings ist eine solche Lösung bei einer superskalaren und *out-of-order*-Ausführung nur mit erheblichem Mehraufwand zu realisieren.

Für die Decodierung der vier Befehle bieten sich zwei Möglichkeiten an. Zum einen kann die Decodierung der einzelnen Befehle sequentiell erfolgen.

Daraus ergibt sich der Vorteil eines kurzen Taktzyklus — entsprechend einer hohen Taktfrequenz, da die Decodierung eines einzigen Befehls sicherlich schneller erledigt wird als die von vier Befehlen. Dagegen wird dadurch die erhoffte Parallelausführung von Befehlen aufgegeben und es kann nicht mehr die Rede von Superskalarität sein. Zum anderen können alle vier Befehle parallel decodiert und anschließend für eine Parallelausführung freigegeben werden. Der Nachteil dabei ist, dass eine derartige Decodierung viel Zeit in Anspruch nimmt. Daraus folgt dementsprechend eine niedrigere Taktfrequenz als beim ersten Ansatz. Aufgrund der Tatsache, dass diese Art von Decodierung dennoch nicht den kritischen Pfad für die gesamte Pipeline-Implementierung darstellt, sondern die darauf folgende Zuordnungsphase die komplexere ist, wurde die letzte Variante realisiert. Abbildung 4.3 veranschaulicht diesen Sachverhalt noch einmal in einer grafischen Darstellung.

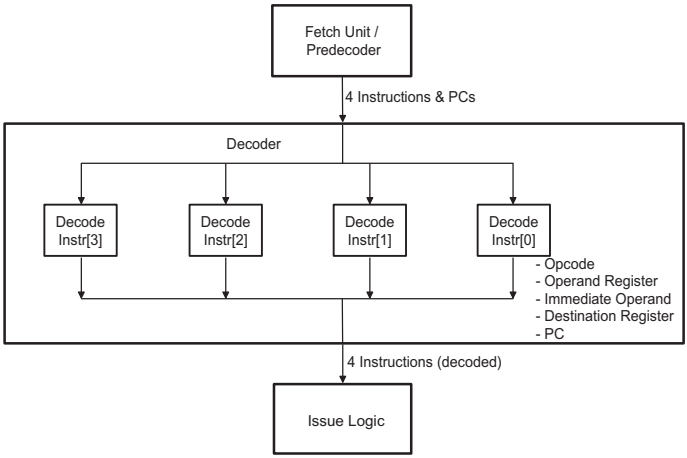

Abbildung 4.3: *Decode Stage*

An dieser Stelle soll ergänzend erwähnt werden, dass — genau wie bei der Befehlsholphase — auch hier ein Puffer eingebaut wurde, um im Fall eines Pipeline-*Stall* die gerade decodierten Befehle aufzufangen. Dementsprechend werden sie bei der Fortsetzung der Pipeline-Verarbeitung der nächsten Verarbeitungsphase zuerst übergeben.

4.2.3 Zuordungsphase

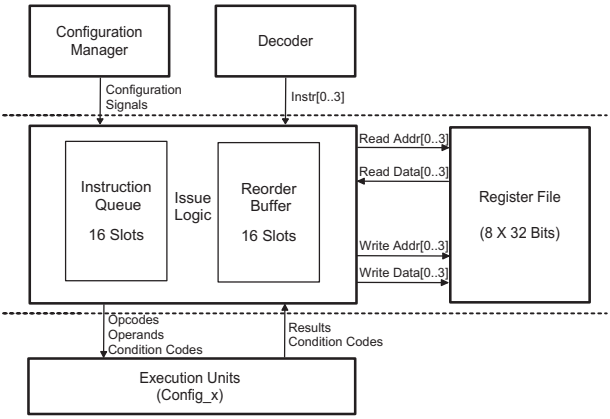

Abbildung 4.4: *Issue Logic*

Die Zuordnungsphase wird durch die sog. *Issue Logic* oder den *Scheduler/Dispatcher* realisiert. Abbildung 4.4 zeigt das Zusammenspiel der dabei beteiligten Komponenten. Die Realisierung einer solchen Phase bildet das Herz eines jeden Mikroprozessors [SS98]. In dieser wichtigen Pipeline-Phase gilt es, eine Reihe komplexer Aufgaben zu bewerkstelligen. Nachfolgend werden die wichtigsten davon erläutert:

- **Aufnahme der decodierten Befehle in die *Instruction Queue***: Als Erstes müssen die taktweise aus dem Decoder ankommenden Befehle in geeigneter Art und Weise zwischengespeichert werden. Diese Zwischenspeicherung wird benötigt, um eine superskalare und *out-of-order*-Ausführung zu gewährleisten. Denn bei dieser Art von Ausführung ist im Voraus nicht bekannt, wann ein bestimmter Befehl tatsächlich ausgeführt wird. Das hängt von der Verfügbarkeit der Operanden sowie der entsprechenden Ausführungseinheit ab. Daher wird eine *Instruction Queue* eingerichtet, welche die Rolle einer zentralen *Reservation Station* (vgl. Abschn. 2.4.5.3) übernimmt. Darin werden Befehle in decodiertem Format während ihres gesamten restlichen Aufenthalts in der Pipeline aufbewahrt. Sie erhalten bei ihrer Aufnahme

eine Identifikationsnummer, welche der jeweiligen Position in der *Instruction Queue* entspricht.

Die Realisierung dieser Subkomponente sieht zunächst eine maximale Anzahl an verfügbaren Plätzen von 16 *Slots* vor. Diese Zahl kann jedoch jederzeit leicht über einen globalen Parameter geändert werden. Außerdem werden bereits vollständig ausgeführte Befehle (s. Abschn. 4.2.5) mit neu ankommenden Befehlen überschrieben, um somit stets möglichst viele Befehle in der *Instruction Queue* bereitzuhalten. Falls aus Platzmangel die gleichzeitig decodierten Befehle nicht auf einmal aufgenommen werden können, werden sie zunächst in einem temporären Puffer zwischengespeichert. Dadurch wird sichergestellt, dass sie nicht verloren gehen, und sie bleiben dort, bis wieder genügend Platz in der *Instruction Queue* zur Verfügung steht. Zugleich setzt die *Issue Logic* das Pipeline-*Stall*-Signal, welches einen Takt später von der *Fetch-Unit* und dem Decoder erhalten wird. Somit verhindert man, dass weitere Befehle geliefert werden. Die bereits in der Pipeline befindlichen Befehle warten, wie schon erwähnt, in den dafür vorgesehenen Puffern auf die Fortsetzung ihrer Abarbeitung. Das *Stall*-Signal bleibt solange aktiv, bis die *Instruction Queue* wieder neue Befehle aufnehmen kann. Dies ist dadurch sichergestellt, dass trotz des *Stall*-Signals die vorhergehenden Befehle weiterhin ausgeführt werden, so dass sie anschließend aus der *Instruction Queue* entfernt bzw. überschrieben werden können. Dadurch werden Plätze wieder frei gesetzt und das *Stall*-Signal kann aufgehoben werden. Daraufhin können alle zuvor angehaltenen Komponenten ihre Arbeit fortsetzen. Dabei werden die in den verschiedenen Puffern gespeicherten Befehle vorrangig behandelt.

Eine besondere Behandlung erfahren die unbedingten Sprungbefehle, welche die dazu gehörigen Sprungadressen aus dem Decoder mitliefern. Statt diese Art von Befehlen in der *Instruction Queue* unterzubringen, werden sie sofort ausgeführt. Dabei wird die mitgelieferte Sprungadresse unmittelbar zur *Fetch-Unit* geleitet, damit die

Programmausführung ohne Verzug an der richtigen Stelle fortgesetzt werden kann.

- **Auslesen der Operanden aus dem *Register File***:
  Für die in der *Instruction Queue* auf ihre Ausführung wartenden Befehle müssen die entsprechenden Operanden bereit gestellt werden. Falls diese nicht direkt im Befehl mit angegeben werden, müssen sie zunächst aus den Registern geholt werden. Aus den vom Decoder gelieferten Angaben ist lediglich zu erkennen, in welchen Registern sich die benötigten Operanden befinden. Die *Issue Logic* sorgt dafür, dass diese Operanden tatsächlich ausgelesen werden. Zu diesem Zweck sowie zur Speicherung der berechneten Ergebnisse wurde ein *Register File* mit vier Lese- und vier Schreib-Ports implementiert. Diese hohe Anzahl an Ports ergibt sich aus dem Anliegen, möglichst viele Befehle pro Takt ausführen zu können, andernfalls kann man von der gebotenen Superskalarität nicht profitieren. Mit diesem eingeschlagenen Lösungsweg können gleichzeitig bis zu vier Operanden gelesen sowie vier Ergebnisse geschrieben werden.

Einen weiteren Sonderfall bilden neben den unbedingten Sprungbefehlen die sog. *move*-Befehle mit unmittelbaren Operanden (Konstanten). Diese Befehle beinhalten beispielsweise Zahlenwerte, welche ohne zusätzliche Verarbeitung in die angegebenen Ziel-Register geschrieben werden sollen. Bei ihrem Eintreffen in der *Issue Logic* werden sie ebenfalls unverzüglich ausgeführt. Mit dieser *out-of-order*-Ausführung erweisen sich die vier Schreib-Ports als sehr vorteilhaft, und somit stehen die benötigten Operanden für eventuell nachfolgende Befehle schnell zur Verfügung.

Die Anzahl der tatsächlich vorhandenen Register wird auch hier über einen globalen Parameter flexibel gestaltet (zwischen 8 und 32). Im Übrigen fiel die Entscheidung auf eine Registerbreite von 32 Bit, da generell die *Thumb*-Befehle auf 32-Bit-Daten operieren.

- **Zuordnung der Befehle zu den verschiedenen Ausführungseinheiten**:
  Die Kernaufgabe der *Issue Logic* besteht darin, die Befehle zügig den verfügbaren Ausführungseinheiten zuzuordnen. Um dieser Anforderung gerecht zu werden, wird in jedem Takt versucht, aus der *Instruction Queue* die höchst mögliche Anzahl an Befehlen zur Ausführung zu bringen. Dies setzt jedoch die Erfüllung einiger Bedingungen voraus. Eine davon ist die angesprochene Bereitstellung der notwendigen Operanden. Zusätzlich muss eine geeignete Ausführungseinheit gefunden werden. Dafür fragt die *Issue Logic* die vom *Configuration Manager* kommenden Signale ab, um den aktuellen Stand der physikalisch verfügbaren Ausführungseinheiten erfahren zu können. Darüber hinaus werden nur solche Einheiten mit auszuführenden Befehlen versorgt, welche nicht gerade mit der Ausführung vorhergehender Befehle beschäftigt sind. Erst wenn alle diese grundlegenden Bedingungen erfüllt sind, steht der leistungssteigernden superskalaren und *out-of-order*-Ausführung nichts mehr im Wege.

  Damit die auszuführenden Befehle nicht zu lange auf ihre Operanden warten müssen, wurde zusätzlich die Möglichkeit eines Operanden-*Forwarding* (s. Abschn 2.4.4) realisiert. Bevor ein Operand aus dem *Register File* abgeholt werden muss, wird zunächst geprüft, ob es nicht einen gerade abgeschlossenen Befehl gibt, welcher den benötigten Operanden bereitstellt. Dies geschieht durch Vergleichen der *Source*-Register eines wartenden Befehls mit den Ziel-Registern der gerade fertiggestellten Befehle. Dadurch lassen sich möglicherweise die Wartezeiten für einige Befehle reduzieren, was ebenfalls zur Leistungssteigerung - wenn beispielsweise die Anzahl der pro Takt ausgeführten Befehle als Leistungsparameter gesetzt wird - beiträgt.

- **Aufnahme der Ergebnisse aus den Ausführungseinheiten in den *Reorder-Buffer***:
  Trotz der *out-of-order*-Ausführung müssen die Befehle ihre Ergebnisse gemäß ihrer ursprünglichen Programmreihenfolge in die jeweiligen Ziel-Register zurückschreiben. Dies ist erforderlich, um eine fehler-

freie Fortsetzung des Programmablaufs sicherzustellen. Für die Erfüllung dieser Anforderung wird ein *Reorder Buffer* eingesetzt, in dem zunächst alle Ergebnisse aus den verschiedenen Ausführungseinheiten zwischengespeichert werden. Mit Hilfe der festen Identifikationsnummer eines jeden Befehls und anhand eines weiteren Steuersignals kann die Reihenfolge der Befehle wieder rekonstruiert werden. Danach werden die Ergebnisse während der abschließenden Rückschreibphase tatsächlich in das *Register File* zurück geschrieben. Außerdem wird mit Hilfe des gleichen *Reorder Buffer* das zuvor erwähnte Operanden-*Forwarding* realisiert.

4.2.4 Ausführungsphase

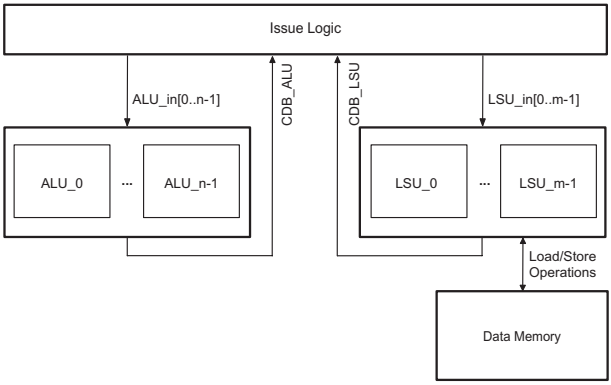

Abbildung 4.5: *Execution/Memory Stage*

Die Ausführungsphase wird über unterschiedliche Pfade implementiert, die sich an den verschiedenen Befehlsarten und den dazu gehörigen Ausführungseinheiten orientieren. Da das ursprüngliche Modell der rekonfigurierbaren Mikroarchitektur bis zu fünf Arten von Ausführungseinheiten vorsieht (s. Abschn. 3.2), wären dementsprechend fünf parallele Ausführungspfade zu implementieren. Für die als *Prototyp* gedachte vorliegende Implementierung wurden zunächst aus Aufwandsgründen lediglich die einfachen und am häufigsten verwendeten Einheiten realisiert: die Integer-ALU für die

Ausführung von arithmetischen und logischen Operationen sowie die unumgängliche *Load/Store*-Einheit für die Behandlung von Speicherbefehlen.

Wie aus Abbildung 4.5 deutlich wird, werden ALU- und Speicherbefehle auf parallelen Leitungen zur Ausführung freigegeben. Ebenfalls werden die Ergebnisse aus den beiden Arten von Ausführungseinheiten über zwei getrennte Busse zur *Issue Logic* zurück transportiert. Dadurch wird eine parallele Ausführung voneinander unabhängiger Befehle ermöglicht.

Aufgrund der Superskalarität kann jede vorgesehene Art von Ausführungseinheiten mehrfach vorhanden sein. Dies wird in Abbildung 4.5 durch n ALUs und m LSUs dargestellt. Für die Ausführung eines Befehls wird die erste passende Einheit genommen, die gerade nicht beschäftigt ist. Die Signale zwischen der *Issue Logic* und den Ausführungseinheiten werden derart dimensioniert, dass bei Bedarf alle verfügbaren Einheiten ohne zusätzlichen Aufwand bedient werden können. Somit kann auf eine aufwendige Bus-Arbitrierung verzichtet werden.

Die Ausführung von Speicherbefehlen erfordert den Zugriff auf einen Datenspeicher, um Operanden auszulesen (*Load*-Befehle) oder Ergebnisse zu speichern (*Store*-Befehle). Daher spricht man in diesem Fall von einer Speicherzugriffsphase (*Memory Stage*) der Pipeline-Ausführung. Für die Realisierung dieser Phase wurde zusätzlich ein RAM-Speicher implementiert, dem genau wie beim zuvor genannten Befehlsspeicher BlockRAMs der Virtex-II-Familie zugrunde liegen.

4.2.5 Rückschreibphase

Die Implementierung der Rückschreibphase wurde aus praktischen Gründen in der selben Komponente wie die Zuordnungsphase realisiert (s. Abb. 4.4), nämlich in der *Issue Logic*. Dies ist darauf zurückzuführen, dass Signale benötigt werden, die im selben Takt aus den beiden Pipeline-Phasen gelesen und geschrieben werden müssen. Insbesondere hat es sich aus der Sicht der technologiegebundenen Hardware-Beschreibung als praktisch erwiesen, wenn die Lese- und Schreib-Zugriffe auf dem *Register File* ausschließlich von einem einzigen Modul ausgehen. Dabei wird einerseits ein Lese-Zugriff

während der Zuordnungsphase für die Bereitstellung der Operanden benötigt. Andererseits erfolgt im selben Takt ein Schreib-Zugriff von der Rückschreibphase aus, um die gerade fertig gestellten Ergebnisse in die jeweiligen Ziel-Register zu schreiben.

Während des Rückschreibvorgangs wird der gesamte *Reorder Buffer* nach Befehlen durchsucht, deren Ergebnisse zum Rückschreiben bereitstehen. Die Suche orientiert sich an den befehlsbezogenen Identifikationsnummern sowie an einem 1-Bit-Signal, welches stets angibt, ob für einen bestimmten Befehl das Zurückschreiben durchgeführt werden kann oder nicht (*ready-for-write-back*). Dieses Signal wird gesetzt, wenn ein Ergebnis aus den Ausführungseinheiten erfolgreich im *Reorder Buffer* aufgenommen wird. Es wird zurückgesetzt, sobald das Resultat dem *Register-File* zur Verfügung gestellt wird, da auf ein *Acknowledge*-Signal von Seiten des *Register File* verzichtet wurde. Daraufhin wird der entsprechende Befehl als vollständig ausgeführt markiert und kann in der *Instruction Queue* gelöscht bzw. überschrieben werden (*ready-to-delete*).

4.2.6 Integration und Test der Mikroarchitektur

Die fünf Pipeline-Phasen wurden zu einer einheitlichen Mikroarchitektur integriert und mit Hilfe moderner CAE-Werkzeuge auf einem FPGA der Firma Xilinx (*Virtex-II*-XC2V8000) implementiert. Abbildung 4.6 zeigt das Ergebnis der Implementierung, nachdem der Hardware-Entwurf erfolgreich platziert und verdrahtet wurde (Place&Route). Daraufhin wurde ein Bitstrom erstellt, welcher über eine geeignete Schnittstelle (JTAG oder SelectMAP) zur Programmierung des auf einem Entwicklungsboard befindlichen FPGA verwendet wird. Mit Hilfe eines *Logic Analyzer* und/oder eines Oszilloskops lässt sich das dynamische Verhalten des FPGA beobachten und somit ein *In-System*-Debugging durchführen. Um die Funktionalität des nun entstandenen Mikroprozessors testen zu können, muss der Befehlsspeicher mit einem bestimmten auszuführenden Programm initialisiert werden. Es wird zusammen mit dem Bitstrom in das FPGA übertragen und anschließend von dort durch die implementierte Mikroarchitektur ausgeführt. Dabei

können ausgewählte Signale mit Hilfe der genannten Debugging-Werkzeuge analysiert werden. Des Weiteren besteht die Möglichkeit, die Signalverläufe anhand eines Software-Tools, das *ChipScope*, zu beobachten. Das Tool liest zur Laufzeit die Signalzustände und stellt die entsprechenden Verläufe auf dem Entwicklungsrechner grafisch dar.

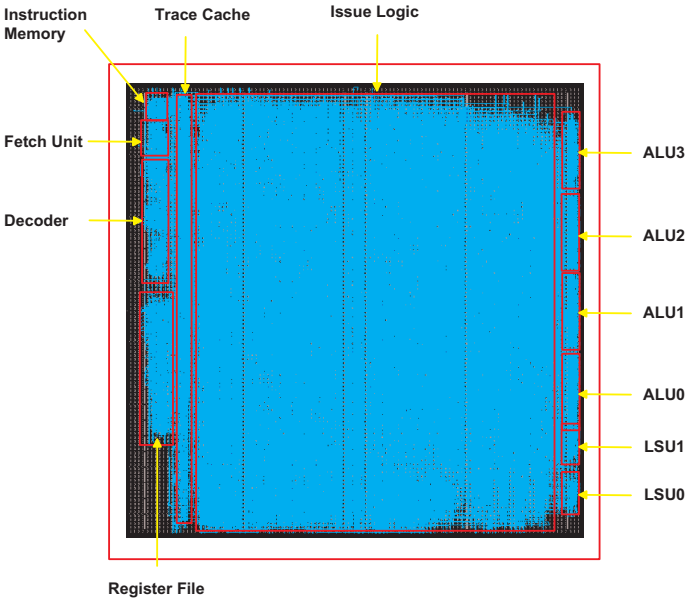

Abbildung 4.6: *Abbild des Mikroprozessors nach Place&Route*

Für einfache Tests wurden zunächst überschaubare Testprogramme (z.B. Zählscheifen) herangezogen. Diese wurden entsprechend dem ARM-*Thumb-Befehlsformat* manuell codiert und gemäß dem Aufbau des Programmspeichers (vgl. Abschn. 4.2.1) dort eingetragen. Dagegen müssen für umfangreiche Testläufe vollständige Anwendungen mit Hilfe eines mit dem ARM-*Thumb*-Befehlssatz kompatiblen Compilers übersetzt werden. Ein solcher Compiler ist zwar leicht erhältlich, dennoch muss das damit erstellte ausführbare Programm zusätzlich in einer geeigneten Art und Weise in die getrennt implementierten Befehls- und Datenspeicher geladen werden. Bei herkömmlichen Rechensystemen ist dies eine der Aufgaben des Betriebs-

systems. Beim Laden eines ausführbaren Programms extrahiert das Betriebssystem aus dem entsprechenden Objektcode die durch den Compiler erzeugten Programm- und Datensegmente und überträgt diese in die dafür reservierten Teile des Arbeitsspeichers. Dieser Ladevorgang erfolgt entweder statisch vor Beginn der Programmausführung oder dynamisch in Abhängigkeit vom Programmablauf. Die Implementierung einer derartigen Funktionalität mit der dazu gehörigen Speicherverwaltung stellt zwar eine notwendige, aber auch anspruchsvolle Aufgabe dar, die den Rahmen dieser Arbeit übersteigen würde.

4.3 Anwendung der partiellen und dynamischen Rekonfiguration

Die laut den vorangegangenen Abschnitten implementierte Mikroarchitektur kann als Basis-Mikroarchitektur betrachtet werden. Sie bildet die Grundvoraussetzung für die Verwirklichung des in dieser Arbeit erklärten Zieles einer partiell und dynamisch rekonfigurierbaren Mikroarchitektur. Zu diesem Zweck wird eine Design-Technik eingesetzt, welche sich von der Seite der Technologie-Anbieter (Xilinx) noch in einer äußerst frühen Phase der Entwicklung befindet. *Virtex-II*-FPGAs lassen sich prinzipiell partiell und dynamisch rekonfigurieren, aber es fehlt immer noch an ausgereiften Werkzeugen zur Durchführung eines solchen Vorhabens. Lediglich anhand eines sehr einfach gehaltenen Beispiels und mit vielen Restriktionen obendrein zeigt Xilinx auf, welche Schritte dabei einzuhalten sind [Xil03]. Im Rahmen einer gemeinsamen Veröffentlichung mit einem der Xilinx-Ingenieure, die diese Technologie zurzeit weiter entwickeln, wurde zwar die partielle Implementierung einer realen Anwendung dargestellt, jedoch sind bis heute keine weiteren unterstützenden Tools für die Öffentlichkeit freigegeben worden [BBHN04].

Von Anfang an wurde die Implementierung der Mikroarchitektur in verschiedenen Hardware-Modulen zweckmäßigerweise hierarchisch organisiert, so dass sich die Umsetzung der sog. *Module Based Partial Reconfiguration*

mit vertretbarem Aufwand durchführen ließ. Im Einklang mit den Xilinx-Richtlinien und Empfehlungen wurden die verschiedenen Module — wie in Abbildung 4.7 dargestellt — in zwei Hauptmodulen zusammengefasst: Ein erstes festes Modul enthält alle Mikroarchitektur-Komponenten, die nicht dynamisch rekonfiguriert werden. Dazu zählen die folgenden Komponenten:

- Befehlsspeicher (*Instruction Memory*)
- *Fetch-Unit*
- *Trace Cache*
- *Configuration Manager*
- Decoder
- *Issue Logic*
- *Register File*
- Datenspeicher (*Data Memory*)

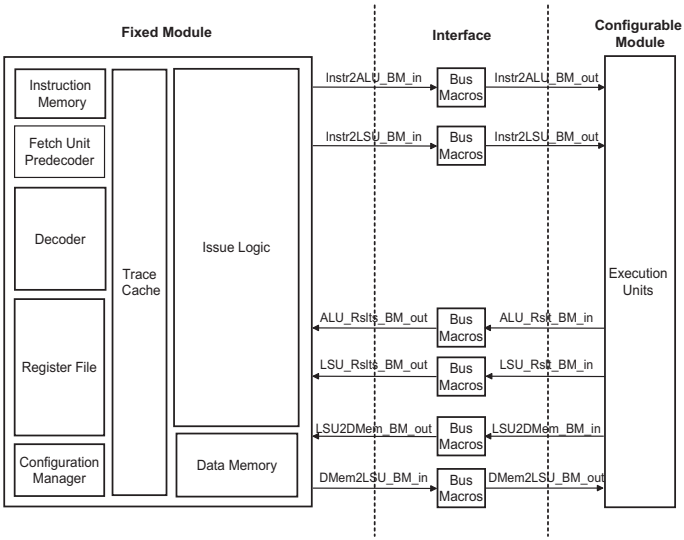

Abbildung 4.7: *Aufteilung der Mikroarchitektur in ein festes und ein rekonfigurierbares Modul*

Das zweite Modul beinhaltet die verschiedenen Ausführungseinheiten, welche in austauschbaren Konfigurationen zusammengefasst werden. Durch die zu implementierende partielle Rekonfiguration werden die vorgesehenen Konfigurationen (s. Abschn. 3.2) während der Laufzeit in Abhängigkeit von den Software-Anforderungen gegeneinander ausgetauscht. Das erste Modul bleibt dabei unberührt.

Um bei der partiellen und dynamischen Rekonfiguration die Signalintegrität insbesondere zwischen dem festen Modul und dem rekonfigurierbaren zu gewährleisten, ist der Einsatz einer besonderen Hardware-Komponente unabdingbar. Dazu stellt die Firma Xilinx ein sog. *Bus-Macro* zur Verfügung, welches speziell während und nach der partiellen Rekonfiguration zur Laufzeit die Signalintegrität sicherstellt. Dies ist für die implementierte Mikroarchitektur umso wichtiger, als dass die beiden Module über zahlreiche Signale verbunden werden. Zum einen werden Signale benötigt, über die Befehle und Operanden von dem festen Modul zu dem rekonfigurierbaren Modul, welches die entsprechenden Ausführungseinheiten beinhaltet, übertragen werden. Zum anderen gelangen die Ergebnisse aus den rekonfigurierbaren Ausführungseinheiten über zusätzliche Signale zum festen Modul, wo sie durch die *Issue Logic* zu den jeweiligen Ziel-Registern weitergeleitet werden.

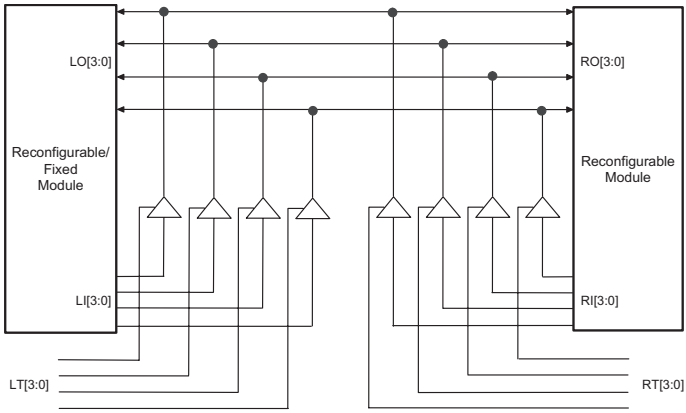

Abbildung 4.8: *Aufbau eines Bus-Macro [Xil03]*

Neben dieser vorteilhaften Eigenschaft eines *Bus-Macro* muss man leider feststellen, dass es technologiebedingt lediglich ein Signal einer Breite von nur 4 Bit übertragen kann. Abbildung 4.8 zeigt den Aufbau dieser speziellen Komponenten. Darüber hinaus empfiehlt Xilinx, dass trotz der vorhandenen Möglichkeit, Signale in beiden Richtungen - von links nach rechts sowie von rechts nach links - beim Einsatz eines konkreten *Bus-Macro* die Übertragungsrichtung lediglich auf eine Richtung beschränkt wird. Somit ist es praktisch nicht möglich, z.B. dasselbe *Bus-Macro* für die Übertragung eines Befehls sowie für die Rückführung des errechneten Ergebnisses zu verwenden, da die dazu gehörigen Signale zwischen den beiden definierten Modulen in unterschiedlichen Richtungen verlaufen. Aufgrund dieser Einschränkungen und angesichts der erheblichen Datenbreite der Signale zwischen dem festen und dem rekonfigurierbaren Modul ist die Zahl der verwendeten *Bus-Macros* sehr hoch (knapp 100). Darüber hinaus ist die Platzierung der *Bus-Macros* nur an bestimmten Stellen des FPGA, den sog. *Tristate-Buffers*, zulässig.

Abbildung 4.9 zeigt das Ergebnis der Implementierung, nachdem die beiden Module und die entsprechenden *Bus-Macros* auf einem Xilinx FPGA (XC2V8000) manuell platziert wurden. Die benötigten Hardware-Ressourcen mussten zuvor für die einzelnen Module mit Hilfe von Software-Tools abgeschätzt werden. Darauf aufbauend erfolgte eine entsprechende Platzierung auf dem FPGA anhand von sog. *Area Constraints*. Ebenso musste für jedes Modul im Anschluss darauf ein partieller Bitstrom nach strengen Vorschriften erstellt werden. Zum Schluss wurden die partiellen Bitströme nach gewünschten Kombinationen zusammengefügt und somit die unterschiedlichen dynamisch austauschbaren Konfigurationen erhalten.

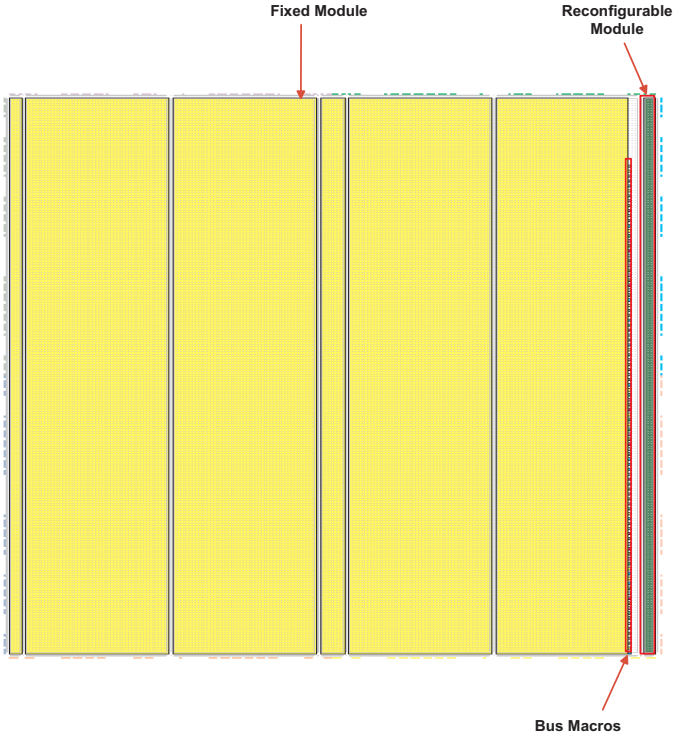

Abbildung 4.9: *Abbild des FPGA nach Platzierung von Bus Macros zwischen dem festen und dem rekonfigurierbaren Modul*

5 Schlussbetrachtung

5.1 Bewertung

Bevor diese Arbeit abgeschlossen wird, soll kurz diskutiert werden, in welcher Weise die erzielten Ergebnisse evaluiert werden können. Als Kriterium eignet sich sicherlich nicht einzig die bloße Angabe der nach der Implementierung der modellierten Mikroarchitektur erreichten Taktfrequenz, wie es oft in der Werbung für konventionelle Mikroprozessoren geschieht. Vielmehr bietet sich dafür eine Gegenüberstellung der am Anfang definierten Ziele und der zum Schluss erzielten Ergebnisse.

Das ursprüngliche Ziel dieser Arbeit bestand darin, die Einsatzmöglichkeit rekonfigurierbarer Hardware in Prozessorarchitekturen zu untersuchen. In den vorangegangenen Abschnitten wurde an vielen Stellen dargelegt, dass in der Literatur bereits zahlreiche Arbeiten durchgeführt worden sind, welche die Vorteile dieser Technologie bewiesen haben. Außerdem existieren auf dem Gebiet der Mikroprozessoren nicht nur Forschungsarbeiten, sondern auch eine Vielzahl von kommerziellen Produkten. Der Mehrwert dieses Beitrags liegt daher in dem aufgezeigten neuen Konzept, wie die speziellen Eigenschaften der partiellen und dynamischen Rekonfiguration für die Entwicklung zukünftiger Mikroprozessoren eingesetzt werden können. Die ausführlich beschriebene Implementierung bildet ein Beispiel für eine entsprechende Umsetzung (*proof-of-concept*). Die dabei verwendete Implementierungstechnik, welche sich selbst noch in einer frühen Entwicklungsphase befindet, führt möglicherweise nicht unbedingt zu der erhofften Leistungssteigerung; dennoch zahlt sich später eine solche parallele Entwicklung von Technologie und Einsatzmöglichkeiten aus.

Eine umfassende Bewertung der entstandenen Mikroarchitektur, insbesondere der erzielten Leistungsfähigkeit müsste, genau wie bei der Software-Simulation, durch Ausführung von üblichen *Benchmarks* erfolgen. Damit

könnten exaktere Leistungsdaten ermittelt werden. In Anbetracht der zuvor angesprochenen Problematik des Ladevorgangs wurden im Rahmen dieser Arbeit derartige *Benchmarks* aus Zeitgründen nicht ausgeführt. Die Durchführung einer solchen Leistungsbewertung muss daher dem Ausblick vorbehalten bleiben. Das gilt ebenfalls für das Heranziehen weiterer Leistungskriterien wie Hardware-Verbrauch, Stromverbrauch, Wiederverwendbarkeit, Kompatibilität etc., was nötig wäre, um eine quantitativ und praxisrelevante Bewertung vornehmen zu können.

5.2 Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Untersuchung der Einsatzmöglichkeiten rekonfigurierbarer Hardware in Prozessorarchitekturen. Zur Durchführung dieses Vorhabens wurden zunächst das Gebiet des *Reconfigurable Computing* umfassend analysiert. Ziel dabei war zum einen, sich einen Überblick über die Forschungsarbeiten auf diesem Gebiet zu verschaffen. Zum anderen erfolgte dadurch eine klare Abgrenzung dieser Arbeit gegenüber den bereits zahlreich existierenden Ansätzen zu rekonfigurierbaren Architekturen.

Im nächsten Schritt ging es darum, die auf diesem Gebiet verfügbaren Technologien zu bewerten, um anschließend eine Auswahl der Werkzeuge (Hardware und Software) treffen zu können, die für die praktische Umsetzung dieser Arbeit notwendig sind. Denn es sollte ein Ansatz entwickelt werden, dessen Vor- und ggf. Nachteile sich mit vertretbarem Aufwand nachweisen lassen. Darüber hinaus sollte sich die Eigenentwicklung zusätzlicher Werkzeuge in einem begrenzten Rahmen bewegen. Zu diesem Zweck musste zuerst ein vertieftes Verständnis der rekonfigurierbaren Hardware, des FPGA — insbesondere seines Aufbaus und seiner spezifischen Eigenschaften — gewonnen werden. Erst dann waren die Voraussetzungen dafür geschaffen, sich Gedanken über mögliche Einsatzmöglichkeiten zu machen. Nicht zuletzt musste das *know-how* über die geltende Entwurfsmethodik und die dazu gehörigen Entwurfswerkzeuge angeeignet werden.

Um den Einsatz rekonfigurierbarer Hardware in Prozessorarchitekturen untersuchen zu können, bedarf es auch erweiterter Kenntnisse über Prozessorarchitekturen. Dies bildet die Voraussetzung dafür, dass mögliche Schwachstellen erkannt und Verbesserungsvorschläge gemacht werden können.

Auf der Grundlage dieser Vorleistungen konnte ein Modell einer rekonfigurierbaren Mikroarchitektur entwickelt werden, welche auf den speziellen Eigenschaften der partiellen und dynamischen Rekonfiguration einiger FPGAs basiert. Das Modell stützt sich auf die Beobachtung der jüngsten Entwicklungen auf dem Gebiet der Mikroprozessortechnik dahingehend, dass eine Vielzahl von parallel arbeitenden Ausführungseinheiten vorgehalten werden. Solche Einheiten belegen bei konventionellen superskalaren Mikroprozessoren stets Hardware-Ressourcen unabhängig davon, ob sie gerade benötigt werden oder nicht. Aus der Vielfalt der Anwendungen folgt, dass unterschiedliche Anwendungen auch unterschiedliche Ausführungseinheiten beanspruchen. Der neue Ansatz besteht im Wesentlichen darin, die verfügbaren Hardware-Ressourcen durch partielle und dynamische Rekonfiguration und zwar während der Programmausführung an die Software-Anforderungen anzupassen. Dabei werden die nicht verwendeten Ausführungseinheiten durch die aktuell benötigten ersetzt.

Dieser Ansatz wurde zunächst mit Hilfe einer Software-Simulation verifiziert. Dadurch sollte vor allem der Einfluss der partiellen und dynamischen Rekonfiguration auf den normalen Betrieb eines Mikroprozessors erfasst werden. Darüber hinaus ermöglichte die durch Software gebotene Flexibilität die Erprobung unterschiedlicher Arten von Hardware-Rekonfiguration. Durch die simulierte Ausführung einiger Anwendungen aus dem SPEC-*Benchmark* konnte die Leistungsfähigkeit der erprobten Mikroarchitekturen untereinander verglichen werden. Daraus ergab sich, dass die zuvor modellierte rekonfigurierbare Mikroarchitektur ein Potenzial zur Leistungssteigerung aufweist. Die Untersuchung wurde auf Basis von rekonfigurierbarer Hardware (FPGA) fortgesetzt, um den Ansatz auf eine reale Umsetzbarkeit hin zu überprüfen.

Den abschließenden Teil dieser Arbeit bildet der Versuch, die modellierte und simulierte Einsatzmöglichkeit rekonfigurierbarer Hardware durch eine

beispielhafte Implementierung der daraus resultierenden Mikroarchitektur zu verwirklichen (*Prototyping*). Zu diesem Zweck wurde eine superskalare Mikroarchitektur auf einer FPGA-Plattform implementiert, welche den ARM-*Thumb*-Befehlssatz ausführt. Durch Anwendung einer noch in Entwicklung befindlichen Entwurfstechnik wurde aus der Basis-Implementierung eine partiell und dynamisch rekonfigurierbare Mikroarchitektur realisiert.

5.3 Ausblick

Die vorliegende Arbeit kann als Grundlage für eine Reihe von Folgeuntersuchungen verstanden werden. Den Ausgangspunkt bildet dabei das neu aufgestellte Konzept einer rekonfigurierbaren Mikroarchitektur, welche — entgegen den meisten bereits existierenden Ansätzen — sich den Software-Anforderungen automatisch anpasst. Die anhand von Software-Simulationen erzielten Ergebnisse sowie die erfolgreich durchgeführte Hardware-Implementierung beweisen die Standfestigkeit dieses Ansatzes.

Darüber hinaus bietet das entwickelte und exemplarisch erprobte Konzept viele Erweiterungs- und Optimierungsmöglichkeiten. Zum Beispiel besteht die implementierte superskalare Mikroarchitektur hauptsächlich aus Komponenten, welche für die Durchführung dieser Arbeit unbedingt notwendig sind, und es wurde aus Zeitgründen auf die Implementierung einer Gleitkomma-Einheit verzichtet. Auch müsste die Ausführung sämtlicher Befehle des ARM-*Thumb*-Befehlssatzes noch in Hardware umgesetzt werden, um eine uneingeschränkte Kompatibilität mit einem herkömmlichen ARM-Compiler zu gewährleisten.

Weitere Ansätze zur Optimierung im Hinblick auf die Funktionalität, den Verbrauch von Hardware-Ressourcen und den damit verbundenen Stromverbrauch, die erzielbare Taktfrequenz etc. sind für bestehende Komponenten denkbar. Beispielsweise wurde bereits eine auf dieser Arbeit basierende Optimierung des *Configuration Manager* (vgl. Abschn. 3.2, 4.2.1) vorgeschlagen [VAT05]. Erst nach derartigen Erweiterungen und Optimierungsarbeiten kann eine abschließende und praxisrelevante Bewertung hinsichtlich

der Leistungsfähigkeit der in dieser Arbeit entwickelten rekonfigurierbaren Mikroarchitektur getroffen werden.

Auf der Compiler-Ebene sind weiterführende Arbeiten denkbar, welche eine Leistungssteigerung prognostizieren lassen. Untersuchungen mit Hilfe eines angepassten Compilers würden beispielsweise dazu führen, dass programmabhängige Hardware-Konfigurationen erstellt werden und somit die Programmausführung deutlich beschleunigt werden kann. Dies ist bereits in der realisierten Software-Simulation erkennbar (siehe Abschn. 3.3.3.2), obwohl dort keine Änderung des Compiler-Vorgangs stattgefunden hat.

Literaturverzeichnis

[Alb98] Albonesi, D. H.. *'Dynamic IPC/Clock rate Optimization'*. 25th Int. Symp. on Computer Architecture, S. 282–293, June 1998.

[Alt05] Altera. http://www.altera.com, 1995-2005.

[ARM00] ARM. *'ARM Architecture Reference Manual'*, 2000.

[Aue94] Auer, A. *'Programmierbare Logic-IC: Eigenschaften, Anwendungen, Programmierung von PLD und FPGA'*. Hüthig, Heidelberg, 2., überarb. Ausgabe, 1994.

[BAB97] Burger, D., Austin, T., Bennett, S. *'Evaluating Future Microprocessors: The Simplescalar Tool Set, Version 2.0'*. Technical Report 1342, University of Wisconsin - Madison, June 1997.

[BBHN04] Blodget, B., Bobda, C., Hübner, M., Niyonkuru, A. *'Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs'*. In: Becker, J., Plazner, M., Vernade, S., (Hrsg.), 'Proc. 14th Int. Conf. on Field-Programmable Logic and Applications (FPL'04), Antwerp, Belgium, Aug./Sept. 2004', Lecture Notes in Computer Science, Vol. 3203, S. 801–810. Springer, Berlin, 2004.

[BDH+97] Burns, J., Donlin, A., Hogg, J., Singh, S., de Wit, M. *'A Dynamic Reconfiguration Run-Time System'*. In: 'Proc. 5th Symp. on FPGA-based Custom Computing Machines (FCCM'97)'. IEEE, Washington, DC, USA, May 1997.

[BDM05] Becker, B., Drechsler, R., Molitor, P. *'Technische Informatik'*. Pearson Studium, München, 2005.

[BKS04] Braunes, J., Köhler, S., Spallek, R. G. *'RECAST: An Evaluation Framework for Coarse-Grain Reconfigurable Architectures'*. In: Müller-Schloer, C., Ungerer, T., Bauer, B., (Hrsg.), 'Proc. Int. Conf. on Architecture of Computing Systems (ARCS2004), Augsburg, Germany, March 2004', Lecture Notes in Computer Science, Vol. 2981, S. 156–166. Springer, Berlin, 2004.

[BL00] Barat, F., Lauwereins, R. *'Reconfigurable Instruction Set Processors: A Survey'*. In: 'Proc. 11th Int. Work. on Rapid System Prototyping', S. 168–173. IEEE, Paris, June 2000.

[Bär02] Bäring, H. *'Mikrorechnertechnik'*. Springer, Berlin, 3. Ausgabe, 2002.

[BU02] Brinkschulte, U., Ungerer, T. *'Mikrocontroller und Mikroprozessoren'*. Springer, Berlin, 2002.

[CAP05] CAP. http://www.ccs.rochester.edu/projects/cap/, 2005.

[CCH+00] Caspi, E., Chu, M., Huang, R., Yeh, J., Wawryzynek, J., Dehon, A. *'Stream Computations Organized for Reconfigurable Execution (SCORE)'.* In: Hartenstein, R. W., Grunbacher, H., (Hrsg.), 'Proc. 10th Int. Conf. on Field-Programmable Logic and Applications (FPL'00), Villach, Austria, Aug. 2000', Lecture Notes in Computer Science, Vol. 1896, S. 605–614. Springer, Berlin, 2000.

[Cel05] Celoxica. http://www.celixica.com, 2001-2005.

[CH99] Compton, K., Hauck, S. *'Configurable Computing: A Survey of Systems and Software'.* Technical Report 1, Dept. of ECE, Northwestern University, 1999.

[Dal99] Dales, M. *'The Proteus Processor - A Conventional CPU with Reconfigurable Funtionality'.* In: Laysaght, P., Irvine, J., Hartenstein, R., (Hrsg.), 'Proc. 9th Int. Work. on Field-Programmable Logic and Applications (FPL'99), Glasgow, Schotland, UK, Aug./Sept. 1999', Lecture Notes in Computer Science, Vol. 1673. Springer, Berlin, 1999.

[Dal01] Dales, M. *'Initial Analysis of the Proteus Architecture'.* In: Brebner, G., Woods, R., (Hrsg.), 'Proc. 11th Int. Conf. on Field-Programmable Logic and Applications (FPL'01), Belfast, Northern Irland, UK, Aug. 2001', Lecture Notes in Computer Science, Vol. 2147. Springer, Berlin, 2001.

[Dal03] Dales, M. *'Managing a Reconfigurable Processor in a General Purpose Workstation Environment'.* In: 'Int. Conf. on Design Automation and Testing in Europe (DATE)'. IEEE, March 2003.

[DeH96] DeHon, A. *'Reconfigurable Architecture for General-Purpose Computing'.* Dissertation, Massachusetts Institute of Technology, Oct. 1996.

[Ele04] Elektronik. *'Befehlserweiterungen in programmierbarer Logik'.* Elektronik, Dec. 2004.

[FL98] Flik, T., Liebig, H. *'Mikroprozessortechnik'.* Springer, Berlin, 5. Ausgabe, 1998.

[FPP97] Friendly, D. H., Patel, S. J., Patt, Y. N. *'Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism'.* Proc. 30th Int. Symp. Microarchitecture, Dec. 1997.

[GSM+99] GoldStein, S. C., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R. R., Laufer, R. *'PipeRench: A Coprocessor for Streaming Multimedia Acceleration'.* In: '26th Int. Symp. on Computer Architecture (ISCA)', S. 28–39. IEEE, Atlanta, Georgia, USA, May 1999.

[Har01] Hartenstein, R. W. *'A Decade of Reconfigurable Computing: a Visionary Retrospective'.* Int. Conf. on Design Automation and Testing in Europe (DATE), March 2001.

[HB02] Hertwig, A., Brück, R. *'Entwurf digitaler Systeme'*. Carl Hanser, München, 2002.

[HHKW90] Hartenstein, R. W., Hirschbiel, A., K.Schmidt, Weber, M. *'A Novel Paradigm of Parallel Computation and Its Use to Implement Simple High-Performance Hardware'*. Interner Bericht 198, Fachbereich Informatik, Universität Kaiserslautern, 1990.

[HHW89] Hartenstein, R., Hirschbiel, A., Weber, M. *'Xputers: An Open Family of Non-von-Neumann Architectures'*. Interner Bericht 195, Fachbereich Informatik, Universität Kaiserslautern, 1989.

[HW97] Hauser, J. R., Wawryzynek, J. *'Garp: A MIPS Processor with a Reconfigurable Coprocessor'*. In: 'Work. on FPGAs for Custom Computing Machines', S. 12–21. IEEE, Chicago, 1997.

[Int02] Intel. *'A Detailed Look Inside the Intel NetBurst Microarchitecture of the Intel Pentium 4 Processor'*, May 2002.

[JM02] Jamin, A., Mähönen, P. *'FPGA Implementation of the Wavelet Packet Transform for High Speed Communications'*. In: Glesner, M., Zipf, P., Renovell, M., (Hrsg.), 'Proc. 12th Int. Conf. on Field-Programmable Logic and Applications (FPL'02), Montpellier, France, Sept. 2002', Lecture Notes in Computer Science, Vol. 2438, S. 212–221. Springer, Berlin, 2002.

[KL02] KleinOsowski, A. J., Lilja, D. L. *'MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research'*. Computer Architecture Letters, (1), Juni 2002.

[Koc02] Koch, A. *'Compilation for Adaptive Computing Systems Using Complex Parameterized Hardware Objects'*. Journal of Supercomputing, S. 179–190, 2002.

[LLVC04] Lee, D., Luk, W., Villasenor, J. D., Cheung, P. *'A Gaussian Noise Generator for Hardware-Based Simulations'*. IEEE TC, 53 (12), Dec. 2004.

[Lud97] Ludwig, S. H.-M. *'Hades - Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor'*. Dissertation, Eidgenössische Technische Hochschule (ETH) Zürich, 1997.

[LWS94] Lehmann, G., Wunder, B., Selz, M. *'Schaltungsdesign mit VHDL'*. Franzis, Poing, 1994.

[Mar04] Maruyama, T. *'Real-Time Computation of Hough Transform'*. In: Becker, J., Plazner, M., Vernade, S., (Hrsg.), 'Proc. 14th Int. Conf. on Field-Programmable Logic and Applications (FPL'04), Antwerp, Belgium, Aug./Sept. 2004', Lecture Notes in Computer Science, Vol. 3203, S. 980–985. Springer, Berlin, 2004.

[MBV+02] Marescaux, T., Bartic, A., Verkest, D., Vernade, S., Lauwereins, R. *'Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking'*.

In: Glesner, M., Zipf, P., Renovell, M., (Hrsg.), 'Proc. 12th Int. Conf. on Field-Programmable Logic and Applications (FPL'02), Montpellier, France, Sept. 2002', Lecture Notes in Computer Science, Vol. 2438, S. 795–805. Springer, Berlin, 2002.

[Möl03] Möller, D. P. F. *'Rechnerstrukturen'*. Springer, Berlin, 2003.

[Moo65] Moore, G. E. *'Cramming more components onto integrated circuits'*. http://www.intel.com/research/silicon/mooreslaw.htm, April 1965.

[MSH97] Mangione-Smith, W., Hutchings, B. *'Configurable Computing: The Road Ahead'*. In: Hartenstein, R., Prasanna, V., (Hrsg.), 'Reconfigurable Architectures: High Performance by Configware', S. 81–96. IT Press, Chicago, USA, 1997.

[MZ97] Merzenich, W., Zeidler, H. Ch. *'Informatik für Ingenieure'*. Teubner, Stuttgart, 1997.

[OV03] Oberschelp, W., Vossen, G. *'Rechneraufbau und Rechnerstrukturen'*. Oldenbourg, München, 9. Ausgabe, 2003.

[PAC05] PACT. *'PACT XPP'*. http://www.pactcorp.com, 2002-2005.

[PB99] Purna, K. G., Bhatia, D. *'Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers'*. IEEE TC, 48 (6) : 579–590, Juni 1999.

[PH96] Patterson, D. A., Hennessy, J. L. *'Computer Architecture: A Quantitative Approach'*. Morgan Kaufmann, San Francisco, USA, 2. Ausgabe, 1996.

[PW04] Pramstaller, N., Wolkerstorfer, J. *'A Universal and Efficient AES Co-Processor for Field Programmable Logic Arrays'*. In: Becker, J., Plazner, M., Vernade, S., (Hrsg.), 'Proc. 14th Int. Conf. on Field-Programmable Logic and Applications (FPL'04), Antwerp, Belgium, Aug./Sept. 2004', Lecture Notes in Computer Science, Vol. 3203, S. 565–574. Springer, Berlin, 2004.

[RBS96] Rothenberg, E., Bennett, S., Smith, J. E. *'Trace Cache: a low latency approach to high bandwidth instruction fetching'*. Proc. 29th Int. Symp. Microarchitecture, S. 24–34, Dec. 1996.

[RBS99] Rothenberg, E., Bennett, S., Smith, J. E. *'A Trace Cache Microarchitecture and Evaluation'*. IEEE TC Special Issue on Cache Memory, Febr. 1999.

[RJSS97] Rothenberg, E., Jacobson, Q., Sazeides, Y., Smith, J. E. *'Trace Processors'*. Proc. 30th Int. Symp. Microarchitecture, S. 138–148, Dec. 1997.

[SFK97] Sima, D., Fountain, T., Kacsuk, P. *'advanced computer architectures A DESIGN SPACE APPROACH'*. Addison-Wesley, Harlow, England, 1997.

[Sik01] Sikora, A. *'Programmierbare Logikbauelemente'*. Carl Hanser, München, Wien, 2001.

[Sim05] SimpleScalar. http://www.simplescalar.com, 2005.

[SLC00] Seng, S., Luk, W., Cheung, P. *'Flexible Instruction Processors'*. Proc. CASES, Febr. 2000.

[SLC02] Seng, S., Luk, W., Cheung, P. Y. K. *'Runtime Adaptive Flexible Instruction Processors'*. In: Glesner, M., Zipf, P., Renovell, M., (Hrsg.), 'Proc. 12th Int. Conf. on Field-Programmable Logic and Applications (FPL'02), Montpellier, France, Sept. 2002', Lecture Notes in Computer Science, Vol. 2438, S. 545–555. Springer, Berlin, 2002.

[SLL+00] Singh, H., Lee, M.-H., Lu, G., Kurdahi, F. J., Bagherzadeh, N., Filho, E. C. *'MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications'*. IEEE TC, 49 (5), May 2000.

[SPE05] SPEC. *'SPEC Benchmark Suite'*. http://www.spec.org, 1995-2005.

[SS98] Shriver, B., Smith, B. *'The Anatomy of a High-Performance Microprocessor: A Systems Perspective'*. IEEE, USA, 1998.

[Sta03] Stallings, W. *'Computer Organization and Architecture'*. Prentice Hall, New Jersey, USA, 6. Ausgabe, 2003.

[Str04] Strelzoff, A. *'Functional Programming for Reconfigurable Computing'*. Proc. Reconfigurable Architectures Work. (RAW04)/ 18th Int. Parallel and Distributed Processing Symp. (IPDPS 2004), April 2004.

[Str05] Stretch. http://www.stretchinc.com, 2004-2005.

[Ten05] Tensilica. http://www.tensilica.com, 2004-2005.

[VAT05] Veale, B. F., Antonio, J. K., Tull, M. P. *'Configuration Steering for a Reconfigurable Superscalar Processor'*. In: 'Proc. Reconfigurable Architectures Work. (RAW05)/ 19th Int. Parallel and Distributed Processing Symp. (IPDPS 2005), Denver, Colorado, USA, April 2005'. IEEE, 2005.

[VMS97] Villasenor, J., Mangione-Smith, W. H. *'Configurable Computing'*. Scientific American, 276 : 54–59, 1997.

[vRU99] Šilc, J., Robič, B., Ungerer, T. *'Processor Architecture'*. Springer, Berlin, 1999.

[Wan98] Wannemacher, M. *'Das FPGA Kochbuch'*. International Thomson Publishing, Bonn, 1998.

[WH95] Wirthlin, M. J., Hutchings, B. H. *'A Dynamic Instruction Set Computer'*. In: 'Work. on FPGAs for Custom Computing Machines', S. 99–107. IEEE, Los Alamitos, CA, USA, April 1995.

[WK01] Wigley, G., Kearney, D. *'The Development of an Operating System for Reconfigurable Computing'.* In: 'Symp. on Field-Programmable Custom Computing Machines (FCCM)'. IEEE, Los Alamitos, CA, USA, 2001.

[WP03] Walder, H., Platzner, M. *'Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations'.* In: '3th Int. Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)', S. 284–287. CSREA Press, June 2003.

[XA00] Xu, B., Albonesi, D. H. *'Runtime Reconfiguration for Efficient General Purpose Computation'.* Proc. Design & Test of Computers, Special Issue on Configurable Computing, S. 42–52, March 2000.

[Xil02] Xilinx. *'Virtex-II Platform FPGA Handbook'.* http://www.xilinx.com, 2002.

[Xil03] Xilinx. *'Two Flows for Partial Reconfiguration: Module Based or Difference Based'.* http://www.xilinx.com, 2003. Application Note 290.

[Xil04] Xilinx. *'Virtex-II Platform FPGA User Guide'.* http://www.xilinx.com, 2004. Version 1.8.

[Xil05] Xilinx. http://www.xilinx.com, 1994-2005.

[YMH+02] Ye, Z. A., Moshovos, A., Hauck, S., Shenoy, N., Banerjee, P. *'CHIMAERA: Integrating a Reconfigurable Functional Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor'.* In: 'IEEE Transactions on VLSI Systems'. IEEE, May 2002.

[ZM00] Zhou, X., Martonosi, M. *'Augmenting Modern Superscalar Architectures with Configurable Extended Instructions'.* In: 'Reconfigurable Architectures Workshop (RAW00)'. IEEE, Mexico, May 2000.

Liste eigener Veröffentlichungen

In der folgenden Auflistung sind die eigenen Veröffentlichungen in chronologischer Reihenfolge aufgeführt.

Eggers, G., Niyonkuru, A., Zeidler, H. Ch. *'A Reconfigurable Processor Architecture'.* Proc. 12th Int. Conf. on Field-Programmable Logic and Applications FPL 2002, Montpellier, France, Sept. 2002.

Niyonkuru, A., Zeidler, H. Ch. *'Evaluation of Run-Time Reconfiguration for General-Purpose Computing'.* Work. Proc. ARCS 2004 Organic and Pervasive Computing, Augsburg, Germany, March 2004.

Niyonkuru, A., Zeidler, H. Ch. *'Designing a Runtime Reconfigurable Processor for General Purpose Applications'.* Proc. Reconfigurable Architectures Work. RAW04/ 18th Int. Parallel and Distributed Processing Symp. IPDPS 2004, Santa Fe, New Mexico, USA, April 2004.

Blodget B., Bobda C., Hübner M., Niyonkuru A. *'Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs'.* Proc. 14th Int. Conf. on Field-Programmable Logic and Applications FPL 2004, Antwerp, Belgium, Aug./Sept. 2004.

A Simulationsergebnisse

	sim-ref	sim-predef	sim-dynrec	sim-statrec1	sim-statrec2
gzip	1,8488	1,8814	1,8814	1,8814	1,6989
place	1,0453	1,0528	1,0696	1,0526	1,0247
route	1,6018	1,6087	1,7802	1,6087	1,5138
gcc	0,9899	1,0012	1,0024	1,0012	0,9448
equake	1,4718	1,4953	1,6666	1,4799	1,4172
ammp	0,8000	1,0107	1,0824	0,7937	0,9882

Tabelle A.1: *Instructions Per Cycle*

	sim-ref	sim-predef	sim-dynrec	sim-statrec1	sim-statrec2
gzip	0,5409	0,5315	0,5315	0,5315	0,5886
place	0,9566	0,9498	0,9349	0,9501	0,9759
route	0,6243	0,6216	0,5617	0,6216	0,6606
gcc	1,0102	0,9988	0,9976	0,9988	1,0584
equake	0,6795	0,6688	0,6000	0,6757	0,7056
ammp	1,2500	0,9894	0,9239	1,2599	1,0120

Tabelle A.2: *Cycles Per Instruction*

	sim-ref	sim-predef	sim-dynrec	sim-statrec1	sim-statrec2
gzip	1,8488	1,8814	1,8814	1,8814	1,6989
place	1,0453	1,0528	1,0696	1,0526	1,0247
route	1,6018	1,6087	1,7802	1,6087	1,5138
gcc	0,9899	1,0012	1,0024	1,0012	0,9449
equake	1,4718	1,4952	1,6666	1,4799	1,4172
ammp	0,8000	1,011	1,0824	0,7937	0,9885

Tabelle A.3: *Instructions Per Cycle bei erweiterten Hardware-Ressourcen*

	sim-ref	sim-predef	sim-dynrec	sim-statrec1	sim-statrec2
gzip	0,5409	0,5315	0,5315	0,5315	0,5886
place	0,9566	0,9498	0,9349	0,9501	0,9759
route	0,6243	0,6216	0,5617	0,6216	0,6606
gcc	1,0102	0,9988	0,9976	0,9988	1,0583
equake	0,6795	0,6688	0,6000	0,6757	0,7056
ammp	1,2500	0,9891	0,9239	1,2599	1,0117

Tabelle A.4: *Cycles Per Instruction mit erweiterten Hardware-Ressourcen*

	Int-ALU	int-MDU	LSU	FP-ALU	FP-MDU
gzip	4	2	4	4	1
place	4	1	4	4	2
route	4	1	4	4	2
gcc	4	3	4	4	1
equake	4	4	4	4	4
ammp	4	4	4	4	4

Tabelle A.5: *Anforderung der verschiedenen Ausführungseinheiten*

Index

Abstraktionsebene, 26, 28
Algorithmenebene, 27
ALU, 6, 35, 50, 67, 83, 99
Anforderung
    Software-, 55, 111
Ansatz
    evolutionär, 5, 55
Arbeitsspeicher, 34, 36, 39, 62, 66, 69, 88, 103
Architektur
    rekonfigurierbar, 109
Art der Kopplung, 19, 54
ASIC, 2, 4, 10, 54
Aufwand
    Hardware-, 55
    Software-, 55
Ausführungseinheit, 6, 50, 66, 70, 76, 77, 80, 81, 91, 92, 95, 98–101, 105, 110
Ausführungsphase, 37, 39, 44, 50, 88, 99

Basiszelle, 10, 13, 15
Befehlsdurchsatz, 37, 41, 45
Befehlsfolge, 63, 64, 76, 77
Befehlsformat, 33, 34, 75, 87, 91, 102
Befehlsholphase, 37, 39, 45, 60, 62, 69, 88
Befehlsparallelisierung, 63, 82
Befehlssatz, 9, 19, 33, 34, 59, 75, 86–88, 91, 111
Befehlsspeicher, 60, 88, 91, 93, 100–102
Befehlszähler, 36, 39, 88, 91
Befehlszählerstand, 60, 77, 91–93
Befehlszuordnung, 48
Benchmark, 77
Benchmark, 78
Bitstrom, 32, 101
    partiell, 106
Bus-Macro, 23, 105, 106

CISC, 33
CLB, 14
Co-Prozessor, 2, 20
Compiler, 58
Configuration Manager, 70, 75, 76, 92, 98, 111
CPI, 74, 81
CPU, 35

Datenspeicher, 70, 100, 102
Decoder, 64, 65, 92, 93, 95
Decodierphase, 39, 47, 88, 93
Decodierung, 33, 93
Design Flow, 25, 28
Dispatcher, 65
DNF, 11

Echtzeitanwendung, 2
Entscheidungsbaum
    binär, 17
Entwicklungsplattform, 54
Entwurfsmethode, 88
Entwurfsmethodik, 25, 109
EPIC, 51

Flexibilität, 2
Forwarding, 48
FPGA, 2, 87, 90, 101, 106, 109, 110
    partiell und dynamisch rekonfigurierbar, 53
    Programmierung, 4
    Struktur, 14
FPU, 6, 50, 67, 83
Full-Custom-Entwurf, 11
Funktionseinheit, 5, 6, 50
    rekonfigurierbar, 19
Funktionsgenerator, 14

General Purpose Computing, 3, 59
Gleitkomma-Arithmetik, 6

Granularität, 18, 54

Hardware
  Beschreibung, 58, 100
  Beschreibungssprache, 29, 58
  dynamisch rekonfigurierbar, 68
  Entwicklungstools, 58
  Entwicklungsumgebung, 87
  Entwurf, 25, 101
  fest verdrahtet, 10
  Implementierung, 4, 53, 86, 87, 111
  Konfiguration, 7, 68
  Lösung, 2
  Plattform, 2
  Rekonfiguration, 66, 80, 87, 110
  rekonfigurierbar, 2, 3, 5, 8, 19, 53–55, 60, 84, 108–110
  Ressource, 6, 81, 92, 106, 110, 111
Hazard, 41, 49, 65, 91
HDL, 29

IC, 10
ILP, 51, 82
Instruction-Level Parallelism, 6
Integer-Operation, 6
Interface, 18
IPC, 74, 81, 82
ISA, 33
Issue Logic, 65, 66, 95–98, 100, 105

JTAG, 32, 101

Kompatibilität, 55, 57, 59
Konfiguration, 22, 92, 105, 106
Kontext, 22

Leistungsaufnahme, 5
Leistungsbewertung, 77, 109
Leistungsfähigkeit, 108, 110
Leistungsparameter, 80, 81
Logikebene, 27
LSU, 6, 50, 67, 70, 83, 100
LUT, 16

Maschinenzyklus, 38
Mikroarchitektur, 8, 19, 33–35, 57–59, 68, 75, 81, 86, 91, 101, 103, 108, 110, 111
  anwendungsspezifisch, 3
  dynamisch rekonfigurierbar, 53, 83
  partiell und dynamisch rekonfigurierbar, 8, 85, 87, 103, 111
  rekonfigurierbar, 3, 4, 8, 54, 59, 60, 71, 75, 77, 79, 86, 99, 110–112
  simuliert, 77, 80
  superskalar, 65, 90, 111
Mikrocontroller, 10
Mikroprogrammierung, 33
Mikroprozessor, 10, 33, 108, 110
  fest verdrahtet, 53
  rekonfigurierbar, 55
  superskalar, 43, 87, 110
  universell, 1
Mikroprozessortechnik, 5, 33, 37, 72, 84, 110
Modell, 8, 60, 79, 99, 110
Moore'sches Gesetz, 1
Multithreading, 1

Operand, 40
Operationsprinzip, 37

Parallelausführung, 94
Parallelisierung, 54
Partition, 7
Partitionierung, 55
Pipeline, 44, 45, 60, 93, 96, 101
Pipelining, 1, 8, 34, 37, 47, 75
Place&Route, 31
Plattform
  Ziel-, 87, 88
PLD, 12
  Klassifikation, 13
Programmierbare Logikbausteine, 11, 13
Programmierbarkeit, 18, 54
Programmiermodell, 8, 33

Prozessor, 19, 33–35, 66
    fest verdrahtet, 3
    rekonfigurierbar, 19
    universell, 4
Prozessorarchitektur, 5, 33–35, 60, 86, 108–110
Prozessorentwicklung, 53
Prozessorleistung, 1

Rückschreibphase, 39, 51, 88, 99, 100
Rapid Prototyping, 4, 54
Rechenoperation, 40
Rechensystem, 36
    rekonfigurierbar, 18, 55
Rechenwerk, 35
Reconfigurable Computing, 3–5, 7, 18, 54, 109
Register, 40
Registersatz, 65, 66, 97, 98, 100
Rekonfiguration, 57
    dynamisch, 3, 22, 67, 80
    partiell, 22, 105
    partiell und dynamisch, 75, 103, 105, 108, 110
    statisch, 21, 76, 80
Rekonfigurationszeit, 84
Rekonfigurierbarkeit, 18, 54
Reorder Buffer, 51, 65, 99, 101
RISC, 34
RTL-Ebene, 27

Schaltkreisebene, 27
Schaltnetz, 14
    zweistufig, 12
Schaltplan, 26
Schaltung
    digital, 10
    integriert, 1, 10
Schaltwerk, 14
Scoreboard, 49
Sea-of-Gates, 17
Semi-Custom-Entwurf, 11
Signalintegrität, 105
SimpleScalar, 72–74, 77, 78, 86
Simulation, 9, 71, 84
    funktional, 30, 71
    Software-, 53, 108, 110–112
Simulationsergebnis, 53, 84
Simulationszeit, 74
Simulator, 73–75
    erweitert, 75, 77
    Prozessor-, 71
Software
    Implementierung, 76
SPEC, 6, 77, 78, 110
Speicherzugriff, 6, 40
Speicherzugriffsphase, 39
Sprungbefehl, 44, 64, 77, 93, 96, 97
Steuerwerk, 35, 36
Superskalarität, 8
Superskalarität, 1, 72, 75, 94, 97, 100
Superskalartechnik, 44, 48, 50, 51
Synthese, 31, 58
Systembus, 34
Systemebene, 26
Systemverhalten, 27, 30, 71

Taktzyklus, 34, 38, 47, 74, 81, 93
Trace Cache, 46, 60–63, 69, 75, 76, 91–93

Verarbeitungszeit, 38
Verbindungsglied
    programmierbar, 14
Verbindungsstruktur, 36
Verdrahtung
    fest, 12
Verhaltensbeschreibung, 25
Verlustleistung, 84
Virtex, 87, 88, 91, 101, 103
VLIW, 1, 43, 51
von-Neumann-
    Prinzip, 35, 37, 60, 91
    Rechner, 35, 37

Werkzeug, 55, 58, 59, 102, 106, 109

grafisch, 29
Hardware-, 4
Software-, 4

Zeitverhalten, 32
Zuordnung, 44, 49
Zuordnungsphase, 48, 51, 88, 94, 95, 100

Lebenslauf

Name		Niyonkuru
Vorname		Adronis
Geburtstag		28. Oktober 1969
Geburtsort		Burambi, Burundi
Familienstand		verheiratet
Schulbildung/ Ausbildung	1976 - 1983	Grundschule in Murago, Burundi
	1983 - 1987	Sekundäre Schule in Buta, Burundi
	1987 - 1990	Gymnasium in Rutovu, Burundi
	1990 - 1992	Offizierschule in Bujumbura, Burundi
	1992 - 1992	Lehrgang am Bundessprachenamt in Hürth Deutschland
	1992 - 1993	Studienkolleg bei den wissenschaftlichen Hochschulen des Freistaates Bayern
	1993 - 1997	Studium an der Universität der Bundeswehr München Fakultät für Informatik Abschluss Diplom-Informatiker
Berufstätigkeit	1997 - 2000	IT-Spezialist im burundischen Verteidigungsministerium Assistent und Consultant an der staatlichen Universität von Burundi
	seit 2000	Wissenschaftlicher Mitarbeiter an der Professur für Technische Informatik der Helmut-Schmidt-Universität/ Universität der Bundeswehr Hamburg