Softwareframework für Prozessoren mit variablen Befehlssatzarchitekturen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Inform. Timo Stripf

geboren in Karlsruhe

Tag der mündlichen Prüfung:
11.12.2013

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Reiner Hartenstein

Erklärung

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebene Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, den 6. November 2013

_____

Timo Stripf

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Instituts für Technologie (KIT). Ich danke meinem Doktorvater, Prof. Dr.-Ing. Dr. h. c. Jürgen Becker, für die Betreuung meiner Arbeit sowie seine wissenschaftliche Führung, sein Vertrauen und die Freiheiten, die diese Arbeit überhaupt erst möglich gemacht haben. Weiterhin möchte ich Prof. Dr.-Ing. Reiner Hartenstein für die Übernahme des Korreferats sowie für sein Feedback und die hilfreichen Diskussionen danken.

Bedanken möchte ich mich bei allen Kolleginnen und Kollegen für die schöne Arbeitsatmosphäre, die zahlreichen Kicker-Spiele, die Hilfsbereitschaft und Zusammenarbeit und zu guter Letzt für den schönen Doktorhut, den ich bei meiner Verteidigung bekommen habe. Namentlich hervorheben möchte ich insbesondere meinen ehemaligen Kollegen und Freund Ralf König. Dank ihm habe ich am ITIV meine Promotion begonnen und kann auf Jahre der erfolgreichen Zusammenarbeit im Kahrisma-Projekt mit vielen anregenden und bereichernden Diskussionen zurückblicken. Durch seine ursprüngliche Idee einer rekonfigurierbaren Prozessorarchitektur mit unterschiedlichen Befehlssätzen wurde der Grundstein unserer beider Dissertationen gelegt. Weiterhin möchte ich noch Michael Rückauer, Jan Heißwolf, Oliver Oey, Thomas Bruckschlögl sowie Jens Becker für die schöne Zeit und den regen wissenschaftlichen und nichtwissenschaftlichen Austausch danken. Ich wünsche allen Kolleginnen und Kollegen weiterhin viel Erfolg für ihre Promotion.

Ich möchte mich bei meinem Freund David Kramer für die wöchentlichen Mensa-Treffen sowie die interessanten Diskussionen bedanken. Wir haben uns in der O-Phase des Informatikstudiums kennengelernt. David ist der Informatik treu geblieben und hat darin seine Promotion erfolgreich abgeschlossen.

Ferner möchte ich mich bei allen Studenten, die im Rahmen ihrer Abschlussarbeiten wertvolle Implementierungsarbeiten geleistet haben, bedanken. Namentlich möchte ich hier Patrick Rieder und Achim Lösch erwähnen.

Ich möchte meinen Eltern und meinem Bruder für ihre Unterstützung meiner Orientierung zur Elektrotechnik und Informatik danken. Sie haben mich mit Hilfe von KOSMOS-Elektronik-Experimentierkästen bzw. von QuickBASIC schon frühzeitig in die Welt der Elektrotechnik bzw. Informatik eingeführt und somit meine Leidenschaft geweckt. Ich danke meinen Eltern auch für die Unterstützung und Förderung meines späteren Werdegang über das Technische Gymnasium, das Informatikstudium bis hin zur Promotion in Elektrotechnik.

Weiterhin möchte ich meiner Frau für ihre Unterstützung während meiner Promotion danken. Vier Wochen vor meiner Verteidigung hat sie mir die erfreuliche Nachricht übermittelt, dass wir ein Baby erwarten. Da ich sowieso den Plan hatte sie nach meiner Dissertation um ihre Hand anzuhalten, habe ich dies dann vier Tage nach meiner Verteidigung verwirklicht und wir haben im März geheiratet.

Ich widme diese Arbeit meiner Tochter Lucy und meinen Neffen Henry und wünsche ihnen viel Erfolg für die Zukunft.

Karlsruhe, im August 2014

Timo Stripf

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

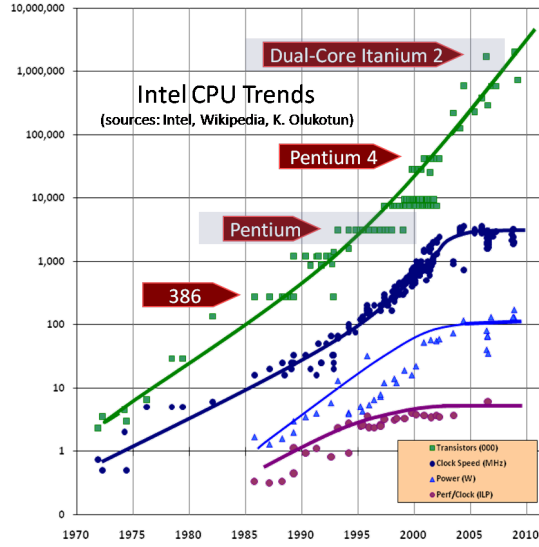


Abbildung 1.1: Charakteristiken von Intel-Prozessoren nach Erscheinungsjahr
Quelle: [1]

Das Mooresche Gesetz (engl. *Moore's Law*) [2] sagt seit 1965 die regelmäßige Verdoppelung der Komplexität integrierter Schaltkreise voraus. Damit einhergehend hat sich auch die Integrationsdichte und somit die Anzahl der Transistoren, die für eine Prozessorarchitektur verwendet werden können, regelmäßig verdoppelt. Mit zunehmender Integrationsdichte ging auch eine Steigerung der Performanz von Prozessorkernen mittels (1) höherer Taktraten, (2) Optimierung der Ausführung und (3) Caches

einher. Eine Steigerung der Taktrate hat sich dabei automatisch durch eine Reduktion des Delays durch kürzere Leitungen und schnellere Transistoren infolge von höheren Integrationsdichten ergeben. Durch die zusätzliche Transistoren konnten zusätzliche Mikroarchitekturtechniken, wie z.B. Pipelining, Superskalarität, out-of-order Ausführung, Registerumbenennung, spekulative Ausführung und Sprungvorhersage, umgesetzt werden. Darüber hinaus konnten Erweiterungen der *Befehlssatzarchitektur* (ISA, engl. *Instruction Set Architecture*) (wie Single Instruction, Multiple Data) eingebaut werden. Dadurch konnte die Ausführung optimiert und somit die Rechenleistung pro Taktzyklus gesteigert werden. Gleichzeitig wurden die zusätzlichen Transistoren in größere Caches investiert, um die Daten aus dem Hauptspeicher näher an die verarbeitenden Einheiten zu bringen und somit konnte sowohl die Latenz als auch der Durchsatz von Datenzugriff optimiert werden. [1, 3]

Obwohl das Mooresche Gesetz weiterhin seine Gültigkeit hat, konnte in jüngerer Zeit die Performanz von Einprozessorsystemen durch höhere Taktraten und durch eine höhere *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) davon nicht in gleichem Maße profitieren. Abbildung 1.1 zeigt die Entwicklung der Transistoranzahl, Taktfrequenz und Performanz pro Takt am Beispiel der Intel-Universalprozessoren. Darin ist zu erkennen, dass bis 2003 das exponentielle Wachstum auch mit einem exponentiellen Wachstum des Energieverbrauchs korrelierte. Ab 2003 wurde allerdings die Leistungsgrenze (engl. *Power Wall*) erreicht und infolge dessen konnte die Taktrate nicht weiter gesteigert werden, da die Verlustleistung effektiv quadratisch mit dieser zusammenhängt. Ebenfalls wurde durch die Energiegrenze die Optimierung der Ausführung beeinflusst. Nach der Regel von Pollack (engl. *Pollack's Rule*) [4] ist die Performanzsteigerung durch Mikroarchitekturtechniken ungefähr proportional zur Quadratwurzel der Skalierung der Transistoranzahl. Somit haben komplexere Prozessorkerne eine schlechtere Energieeffizienz (Energieverbrauch pro Rechenleistung) als einfachere.

Getrieben durch die Energiegrenze und der Notwendigkeit nach immer höherer Rechenleistung hat sich daher ein Paradigmenwechsel bei Universalprozessoren von Einkern- zu Mehrkernprozessoren eingestellt. Aus Hardwaresicht bieten Mehrkernprozessoren den Vorteil, dass sie bei gleicher Chipfläche energieeffizienter sind als Einkernprozessoren, da pro Kern der Energieverbrauch pro Rechenleistung niedriger ist und die theoretisch Rechenleistung proportional mit der Anzahl der Kerne steigt. Aus Softwaresicht war es bei Einprozessorsystemen die Aufgabe der Prozessordesigner die Parallelität der Hardware vor dem Softwareentwickler zu verstecken und sequentielle Anwendungen unter Ausnutzung der feingranularen räumlichen Parallelität im Befehlsstrom und der zeitlichen Parallelität in der Hardware zu parallelisieren. Bei Mehrkernprozessoren wurde allerdings die Aufgabe der Parallelisierung zusätzlich dem Softwareentwickler übertragen, der eine Anwendungen durch Aufteilung in mehrere Threads parallelisieren muss. Die Programmierung von parallelen Anwendungen erfordert in der Regel einen erheblichen Mehraufwand im Vergleich zu Single-Thread-Anwendungen. Dies ist durch die zusätzliche Threadpartitionierung, -synchronisation und -kommunikation, der schlechteren Debugbarkeit und

des effektiven Nichtdeterminismus bei der Ausführung begründet. Praktisch hängt die Performanzsteigerung durch Mehrkernprozessoren von der Anwendung ab, wie viel *Parallelität auf Threadebene* (TLP, engl. *Thread-Level Parallelism*) diese ausnutzt und überhaupt theoretisch vorhanden ist. So existieren serielle Anwendungen, wie z.B. die Entropiekodierung, bei der die Berechnung von einem Ergebnis direkt vom vorherigen abhängt und somit eine Parallelisierung des Algorithmus nicht möglich ist.

Somit hängt es von der Anwendung an sich ab, auf welcher Prozessorarchitektur diese effizient ausgeführt werden kann. Bei Anwendungen mit viel Parallelität auf Befehlsebenen und wenig auf Threadebene würde man die Transistoren eher in komplexere Kerne anstatt in die Erhöhung der Anzahl der Kerne investieren. Im Gegensatz dazu würde eine Anwendung mit wenig Parallelität auf Befehlsebene und hoher Parallelität auf Threadebene auf einer Prozessorarchitektur mit vielen einfachen Kernen effizienter ausgeführt werden. Somit ist es eigentlich bei heutigen statischen Mehrkernprozessoren unmöglich, eine effiziente Architektur für eine breite Masse an Anwendungen zu realisieren.

Zur Lösung dieses Problems wurde daher die Kahrisma Prozessorarchitektur entwickelt, die eine dynamische Ausnutzung von TLP und ILP ermöglicht. Die Architektur besteht aus verschiedenen Arten von Pipelineressourcen, wie z.B. Befehlsholeinheiten, Registerumbenennungseinheiten und eingebetteten Datenpfadeinheiten, die zur Laufzeit rekonfiguriert und dynamisch zusammengeschaltet werden können, um virtuelle Prozessorinstanzen zu erzeugen. Die virtuellen Prozessorinstanzen können dann jeweils einen Thread ausführen und sich im Ressourcenverbrauch, maximalen Performanz und im Energieverbrauch unterscheiden. Je mehr Ressourcen für eine Instanz verwendet werden, umso mehr Befehle können in dem jeweiligen Thread parallel ausgeführt werden. Allerdings können dadurch auch insgesamt weniger Threads erzeugt werden, da die Ressourcen in der Kahrisma Prozessorarchitektur konstant sind. Die Aufteilung der Prozessorpipeline in verschiedene rekonfigurierbare und zusammenschaltbare Ressourcen hat eine neue ISA erfordert. Die ISA zeichnet sich dadurch aus, dass sie nicht fest ist sondern sich flexibel zur Laufzeit in ihrer jeweiligen Ausprägung je nach virtueller Prozessorinstanz und der Anzahl an verwendeten Ressourcen verändern kann. In der komplexesten Ausprägung lässt sich die ISA in die Klasse der geclusterten VLIW-ISAs einordnen.

Für den Erfolg neuer Prozessorarchitekturen spielt nicht nur die theoretische Rechenleistung oder der Energieverbrauch eine Rolle. Mindestens genauso wichtig ist deren Programmierbarkeit in einer weit verbreiteten Programmiersprache. Nur dann müssen Anwendungen für die Architektur nicht neu entwickelt und eine Wiederverwendung von existierenden Implementierungen kann gewährleistet werden. Die rekonfigurierbare Prozessorarchitektur und die damit einhergehende neue rekonfigurierbare ISA erfordert daher ein neues Softwareframework, das die speziellen Eigenschaften der Prozessorarchitektur unterstützt und eine effiziente, werkzeuggestützte Softwareentwicklung in einer Standard-Programmiersprache ermöglicht.

1.2 Ziele dieser Arbeit

Ziel dieser Arbeit ist die Konzeption und Realisierung eines mixed-ISA Softwareframeworks, das sich durch die Unterstützung und effiziente Ausnutzung von verschiedenen ISAs mit unterschiedlichen Charakteristiken in Bezug auf Performanz, Ressourcen- und Energieverbrauch auszeichnen. Der Begriff „mixed-ISA“ bedeutet hierbei, dass das Softwareframework mit mehreren ISAs gleichzeitig umgehen kann. Als Programmiersprachen für den Endbenutzer verwendet das Softwareframework die weit verbreitete C/C++-Programmiersprache, die um Konzepte für die Programmierung von mixed-ISA Anwendungen erweitert wurde und somit eine weiche Migration von bestehenden Anwendungen ermöglicht. Fokus des Softwareframeworks ist dabei nicht nur die Unterstützung der verschiedenen ISAs, wie es z.B. von retargierbaren Compilern ermöglicht wird, sondern ebenfalls das dynamische Umschalten der Befehlssatzarchitektur während der Laufzeit einer Anwendung. Dadurch kann sich die Architektur dynamisch an wechselnden Anforderungen der Anwendungen zur Optimierung der Performanz, des Ressourcen- und des Energieverbrauchs anpassen.

Die Unterstützung der Programmiersprache im Softwareframework wird dabei durch eine klassische Einteilung in Compiler, Assembler und Linker erfolgen. Zusätzlich ist zum Testen und der Evaluation der Ergebnisse bezüglich Korrektheit, Performanz und Ressourcenverbrauch ein Befehlssatzsimulator notwendig. Sämtliche Komponenten müssen dabei mixed-ISA-fähig sein und somit sowohl die verschiedenen ISAs als auch deren Wechsel während der Ausführung einer Anwendung unterstützten. Zu diesem Zweck basieren sämtliche Komponenten auf einer *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*), die sämtliche ISAs einer Zielarchitektur verhaltensorientiert beschreiben kann und somit die Retargierbarkeit des Frameworks ermöglicht. Komplettiert wird das Softwareframework durch ein ISA-Partitionierungswerkzeug, das für eine gegebene Anwendung und eine gegebene Prozessorarchitektur (ebenfalls spezifiziert innerhalb der ADL) eine Partitionierung der Anwendung bezüglich ISAs zur Optimierung der Performanz, des Ressourcen- und des Energieverbrauchs ermöglicht.

1.3 Gliederung

Diese Arbeit ist wie folgt gegliedert:

Kapitel 2 beschreibt die Grundlagen sowie den Stand der Forschung im Bereich Befehlssatzarchitekturen, Mikroarchitekturen und Architekturbeschreibungssprachen von Prozessoren sowie im Bereich Compiler für Prozessoren.

Kapitel 3 stellt die Kahrisma-Architektur zur dynamischen Ausnutzung von ILP und TLP vor. Dabei werden insbesondere die Ziele vorgestellt, die nur durch ein Hardware/Software-Codesign umgesetzt werden konnten. Die Hardware-Komponente der

Architektur wird in diesem Kapitel beschrieben während sich die nachfolgenden Kapitel um die Umsetzung des mixed-ISA Softwareframeworks beziehen.

Kapitel 4 stellt das Konzept des Softwareframeworks vor, das die Programmierbarkeit der Kahrisma-Architektur gewährleistet. Dabei werden aus den Zielen der Gesamtarchitektur die Ziele und Anforderungen an das Softwareframework abgeleitet sowie die Konzepte zur Erfüllung der Anforderungen vorgestellt.

Kapitel 5 beschreibt die mixed-ISA Architekturbeschreibungssprache des Softwareframeworks. Diese wird zur Unterstützung der Designzeit-Flexibilität sowie der rekonfigurierbaren Befehlssatzarchitektur für eine dynamische Ausnutzung von ILP und TLP benötigt.

Kapitel 6 enthält die Realisierung des mixed-ISA Softwareframeworks. Dabei wird im Detail auf die Umsetzung der jeweiligen Komponenten des Frameworks eingegangen. Dies beinhaltet die mixed-ISA Erweiterung der C/C++-Programmiersprache, das CoreGen-Werkzeug, den mixed-ISA LLVM-Compiler, die mixed-ISA Binärwerkzeuge, den Core-Simulator, den System-Simulator und den ISA-Partitionierer.

Kapitel 7 enthält die Ergebnisse und Charakterisierung der Realisierung des mixed-ISA Softwareframeworks.

Kapitel 8 gibt einen Ausblick auf weiterführende und darauf aufbauende Arbeiten.

Kapitel 9 fasst die Arbeit zusammen.

2 Grundlagen

2.1 Befehlssatzarchitekturen

2.1.1 Überblick

Eine ***Befehlssatzarchitektur*** **(ISA, engl. *Instruction Set Architecture*)** beschreibt eine Menge von Eigenschaften eines Mikroprozessors. Dazu zählt insbesondere der Befehlssatz (engl. *Instruction Set*) aber auch die Unterbrechungsbehandlung (engl. *Interrupt*) oder virtuelle Speicherverwaltung. Ein Mikroprozessor implementiert eine ISA, wenn alle Spezifikationen der ISA erfüllt sind und somit sämtliche Programme in der ISA korrekt ausgeführt werden. Der interne Aufbau des Prozessors (z.B. Pipelinelänge, Schattenregister) sowie der Peripheriegeräte (z.B. DMA) gehören nicht zur ISA.

2.1.2 Instruktionen und Operationen

In dieser Arbeit wird zwischen Instruktionen und Operationen unterschieden. Es gilt folgende Definition:

Operation oder Befehl ist eine Rechenvorschrift, die angewandt wird, sobald die Operation angewandt wird. Ein Beispiel einer Operation ist eine Addition zweier Register.

Instruction ein oder mehreren Operationen. Bei RISC und CISC (siehe Abschnitt 2.1.9) ist eine Instruktion identisch zu einer Operation. Bei VLIW und EPIC hingegen kann eine Instruktion mehrere Operationen enthalten.

2.1.3 Bitbreite

Die Bitbreite einer ISA legt die Breite der Register fest. So sind in der Regel alle Operationen von der Geschwindigkeit und Kodierung im Befehlssatz für diese Bitbreite optimiert. Operationen auf einer niedrigeren Bitbreite werden in der Regel nicht schneller ausgeführt oder brauchen sogar mehr Platz, weil die Kodierung ineffizienter wird (CISC). Operationen mit höherer Bitbreite brauchen wesentlich länger und müssen manchmal durch mehrere Operationen kleinerer Bitbreite (zwei Additionen mit Übertragsbit (engl. *Carry-Bit*)) generiert werden.

2.1.4 Registeranzahl

Die Registeranzahl beschreibt die Anzahl der frei verwendbaren *Allzweckregister* (GPR, engl. *General Purpose Register*), auf welche die Befehle zugreifen können. Aus Kodierungsgesichtspunkten genügt meist die Registeranzahl 2^b, wobei b die Anzahl der Bits zur Auswahl der Register im Instruktionsformat ist.

Eine Erhöhung der Registeranzahl wirkt sich auf zwei Arten auf die Architektur aus. Zum einen werden die Zugriffe auf den Speicher zwecks der Auslagerung von Registerwerten reduziert und es können Befehle und Speicher eingespart werden. Zum anderen wird die Anzahl an Bits erhöht, die für die Kodierung eines Befehls notwendig sind. Bei Pipelining und Superskalarität hat eine höhere Registeranzahl noch den Vorteil, dass dadurch Namenskonflikte bereits im Compiler leichter verhindert werden können.

Manche Architekturen unterscheiden zusätzlich zwischen statischen und rotierenden Registern. Dabei sind statische Register die klassische Form. Jedes Register wird mit einer festen Nummer angesprochen. Bei rotierenden Registern sind mehr physikalische Register vorhanden, als logisch ansteuerbar sind. Stattdessen wird ein Registerfenster so verschoben, dass z.B. alle Register 0-7 oder 10-17 gleichzeitig verwendet werden können. Soll jetzt auf ein Register außerhalb des Fensters zugegriffen werden, muss dieses durch einen speziellen Befehl verschoben werden. Zum Beispiel hat die SPARC-ISA [5] 32 logische Register, wobei 8 statisch und 24 rotierend sind. Bei einem Funktionsaufruf wird das Registerfenster der rotierenden Register um 16 Register verschoben. Wenn keine physikalischen Register mehr vorhanden sind, dann löst der Prozessor eine Exception aus, damit die Software die physikalischen Register leeren kann.

2.1.5 Instruktions- und Operationsformat

Das Instruktionsformat (engl. *Instruction Format*) definiert die Kodierung des Instruktion. Je nach Instruktionsformat kann eine Instruktion aus mehreren Operationen oder keiner Operation (NOP) bestehen. In klassischen Architekturen, wie z.B. RISC und CISC, besteht ein Befehl immer aus genau einer Operation.

In dieser Arbeit wird zwischen Instruktionsformat und Operationsformat unterschieden:

- Das **Instruktionsformat** definiert den Aufbau einer Instruktion. Eine Instruktion kann je nach ISA aus mehreren Operationen bestehen. Für jede Operation muss die Stelle im Instruktionsformat bekannt sein, allerdings zählt der Aufbau des Operationsformats nicht mehr zum Instruktionsformat. Zusätzlich kann es noch Informationen beinhalten, die Abhängigkeiten zwischen Operationen oder Instruktionen betreffen. Bspw. fällt die Information, welche Operationen parallel ausgeführt werden können, in diese Kategorie.

- Das **Operationsformat** beschreibt das Format zur Kodierung von genau einer Operation. Nicht enthalten sind Informationen, die das Zusammenspiel zwischen mehreren Operationen regeln.

2.1.6 Instruktionsformat

Das Instruktionsformat definiert den Aufbau einer Instruktion. Dabei setzt es auf den Operationsformaten auf. Abbildung 2.1 zeigt einen möglichen Aufbau eines Instruktionsformats.

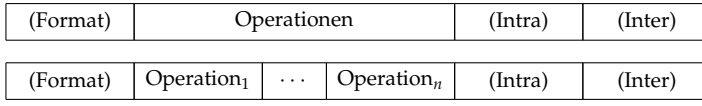


Abbildung 2.1: Allgemeines Instruktionsformat

Die Felder beschreiben eine Obermenge an möglichen Feldern. Bis auf die Felder der Operationen kann jedes Feld weggelassen werden. Die Felder können beliebig im Instruktionsformat verteilt und gesplittet sein. Die Anordnung in der Abbildung dient der Übersichtlichkeit.

Bei RISC- und CISC-Architekturen besteht das Befehlsformat nur aus einem Operationen-Feld mit genauer einer Operation.

Format (optional) Das Format-Feld des Befehlsformates ist ähnlich dem Opcode Feld des Operationsformates. Es beschreibt den allgemeinen Aufbau des restlichen Befehlsformates und kann z.B. bei VLIW (siehe Abschnitt 2.1.9.3) steuern, welche Operationsarten und -formate vorhanden und kodiert sind.

Operationen Das Operationenfeld repräsentiert den Platz im Befehlsformat, der für die Beschreibung der einzelnen Operationen verwendet wird. Es besteht aus einer oder mehreren Operationen, die alle ein unterschiedliches Operationsformat aufweisen können.

Intra(abhänigkeiten) (optional) Die Intraabhängigkeiten beschreiben Abhängigkeiten zwischen mehreren Operationen innerhalb des Befehles. Ein typisches Beispiel ist die Information bei VLIW, welche Operationen parallel ausgeführt werden können.

Inter(abhänigkeiten) (optional) Die Interabhängigkeiten beschreiben Abhängigkeiten zwischen mehreren Befehlen. So ist die Information, dass dieser Befehl parallel zum nächsten Befehl ausgeführt werden kann, als Interabhängigkeit zu verstehen.

2.1.7 Operationsformat

Das Operationsformat besteht aus verschiedenen Feldern. Optionale Felder müssen in einer praxisrelevanten Architektur nicht vorhanden sein. In der Theorie ist jedes Feld optional.

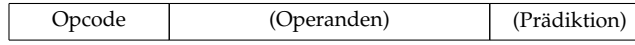


Abbildung 2.2: Allgemeines Operationsformat

2.1.7.1 Opcode

Der wichtigste Teil des Operationsformates ist der Opcode. Er repräsentiert nicht nur die Operation, die ausgeführt werden soll, sondern legt auch den Aufbau des restlichen Operationsformates fest. In der Theorie kann jeder Opcode ein unterschiedliches Operationsformat haben. In der Praxis versucht man – besonders bei modernen Architekturen – möglichst wenig verschiedene Operationsformate zu erlauben, um den Dekodiervorgang effizienter bezüglich Chipfläche und Stromverbrauch gestalten zu können.

Das Opcode-Feld muss nicht immer gleich lang sein, sondern es können bei verschiedenen Gruppen von Befehlen mehr Bits zur Verfügung stehen. Zum Beispiel bietet es sich an bei Operationen, die nicht alle Operandenfelder benötigen, diesen Platz für die Verlängerung des Opcode-Felds zu nutzen.

Bei einem theoretischen *Eininstruktionscomputer* (SIC, engl. *Single Instruction Computer*) [6] ist das Opcode-Feld ebenfalls optional.

2.1.7.2 Operanden

Die Operanden im Operationsformat legen fest, woher die *Quelloperanden* (src, engl. *Source Operand*) geladen und wohin die *Zieloperanden* (dest, engl. *Destination Operand*) geschrieben werden sollen. Für jeden Operanden wird eine Adressierungsart verwendet.

Je nach Befehl unterscheidet sich die Anzahl der Operanden. Dabei wird zwischen impliziten Operanden und expliziten Operanden unterschieden:

Implizite Operanden verbrauchen keinen Platz im Operationsformat. Es besteht quasi keine Wahlmöglichkeit in der Kodierung, der Operand ist bereits durch den Opcode festgelegt. So wird z.B. bei einer Akkumulatorarchitektur implizit der Akkumulator als ein Quell- und Zieloperand festgelegt. Die Operationsflags, die häufig bei arithmetisch-logischen Operationen gesetzt werden, lassen sich

bei vielen Architekturen nicht steuern und werden ebenfalls als impliziter Zieloperand angesehen.

Explizite Operanden hingegen verbrauchen Platz im Operationsformat. Durch die Operationskodierung ist eine Wahlmöglichkeit gegeben, welche als Quell- oder Zieloperand verwendet werden soll. Bei Zieloperanden kann es auch vorkommen, dass diese bei einer bestimmten Kodierung deaktiviert werden und das Teilergebnis dann verworfen wird. So bieten z.B. die ARM ISA bei bestimmten Befehlen die Möglichkeit, die Flags wahlweise in das Flagsregister zu übernehmen.

Bei einem Kellerautomaten (engl. *Pushdown Automaton*) ist das Operandenfeld nicht vorhanden.

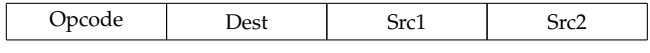
2.1.7.3 Prädikation (optional)

Das Prädikation-Feld bietet die Möglichkeit die Ausführung eines Befehls an bestimmte Bedingungen zu knüpfen. Durch diese Technik ist es möglich Sprünge einzusparen, die besonders bei Pipelining und Superskalarität nur mit Sprungvorhersageeinheiten schnell ausgeführt werden können. Zusätzlich kann dadurch die *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) eines Programmes gesteigert werden. Das Prädikation-Feld wertet dabei z.B. die Flags eines der letzten Befehle aus.

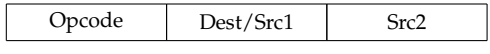
2.1.7.4 Klassifikation der ISA anhand des Operationsformates

Je nach Art, wie arithmetisch-logische Befehle ihre Operanden adressieren, kann man die Architektur klassifizieren:

- Das **Dreiadressformat** besteht aus dem Opcode, zwei Quell- und einem Zieloperandenbezeichner.



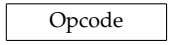
- Das **Zweiadressformat** besteht aus dem Opcode, einem Quell- und einem Quell-/Zieloperandenbezeichner. D.h. der Operandenbezeichner wird parallel als Quell- und Zieloperand verwendet.



- Das **Einadressformat** besteht aus einem Opcode und einem Quelloperandenbezeichner.

Dabei kommt einem Register, dem sog. Akkumulator, eine besondere Bedeutung zu. Er wird implizit als zweiter Quelloperand und als Zieloperand verwendet.

- Das **Nulladressformat** besteht nur aus dem Opcode.

  Das Nulladressformat kann nur von einer Kellerarchitektur verwendet werden.

2.1.8 Befehlssatz

Der **Befehlssatz** (engl. *Instruction Set*) beschreibt die Menge an Operationen, welche die Architektur ausführen kann. Dabei ist die Anzahl an Operationen nahezu beliebig. Das theoretische Minimum ist eine Operation und dem Maximum sind keine Grenzen gesetzt, da beliebige Befehlsfolgen zu einem einzigen Befehl kombiniert werden können.

Grundsätzlich kann man den Befehlssatz in folgende Befehlsarten einteilen:

Transferbefehle (engl. *Data Movement Operations*) übertragen Daten von einer Speicherstelle zu einer anderen. Darunter fallen besonders die Move- oder Load/-Store-Befehle bei einer Lade-/Speicherarchitektur (engl. *Load-Store Architecture*) aber auch Ein- und Ausgabebefehle, die – falls vorhanden – auf einem getrennten E/A-Adressraum arbeiten.

Arithmetisch-logische Befehle (engl. *Arithmetic and Logical Operations*) werden zum Berechnen von Daten verwendet. Sie bestehen häufig aus einem, zwei oder drei Operanden. Dabei ist das Datenformat der Operanden im Befehl selbst kodiert. Gängige Datenformate sind vorzeichenlose bzw. vorzeichenbehaftete Ganzzahlen im Zweierkomplement, Festpunkt- und Gleitkommadatenformate.

Die Gruppe der arithmetisch-logischen Befehle kann weiter in folgende Untergruppen unterteilt werden:

Integer arithmetische Befehle (engl. *Integer Arithmetic Operations*) führen arithmetische Operationen wie Addition, Subtraktion, Multiplikation, Division, Modulo oder Vergleich zweier Operanden aus. Bei der Addition/-Subtraktion existieren zusätzliche Befehle mit Übertrag, um durch Kombination mehrere Additions-/Subtraktionsbefehle größere Zahlen berechnen zu können, als es mit einem Befehl möglich wäre. Bei Multiplikation, Division, Modulo und Vergleich muss jeweils zwischen vorzeichenbehafteten und vorzeichenlosen Befehlen unterschieden werden.

Logische und Bitmanipulationsbefehle (engl. *Logical and Bit Manipulation Operations*) erlauben das bitweise und logische Verknüpfen von Operanden, das Setzen, Löschen oder Invertieren einzelner Bits sowie die Verschiebung

und Rotation von Bitstellen. Befehle zum Zählen von Bits lassen sich ebenfalls in diese Kategorie zuordnen.

Bei logischen und bitweisen Operationen stehen meistens Negation, UND, ODER und XOR Verknüpfungen zur Verfügung. Bitweise Operationen verknüpfen jedes Bit der Operanden einzeln miteinander. Bei logischen Operationen hingegen wird ein Operand von Null als falsch und ein Operand von ungleich Null als wahr angenommen und nur eine Verknüpfung durchgeführt. Als Ergebnis kann nur eine Null oder Eins auftreten.

Schiebe- und Rotationsbefehle (engl. *Shift and Rotate Operations*) schieben/rotieren die Bits eines Operanden um eine Anzahl von Stellen (n) entweder nach links oder rechts. Das Links- bzw. Rechtschieben entspricht dabei einer Multiplikation bzw. Division mit 2^n im Zweierkomplement oder Zweiersystem. Unterschieden wird zwischen logischen (vorzeichenlosen) und arithmetischen (vorzeichenbehafteten) Schiebeoperationen.

Befehle zum Zählen von Bits (engl. *Bit Counting Operations*) zählen entweder das Vorhandensein von Null/Eins Bits (engl. *Population Count*) oder die Anzahl der führenden/nachfolgenden Null/Eins Bits (engl. *Count Leading/Tailing Zero/Ones*).

Festkommabefehle (engl. *Fixed-Point Operations*) sind Befehle, die die Berechnung von Festkommaarithmetik (engl. *Fixed-Point Arithemtic*) unterstützen.

Gleitkommabefehle (engl. *Floating-Point Operations*) bestehen aus arithmetischen Befehlen, die auf der Gleitkommaarithmetik (engl. *Floating-Point Arithmetik*) aufsetzen. Als Datenformat wird meisten der IEEE 754 [7] Standard genommen, der die vier Zahlenformate single (32 Bits), single extended (> 42 Bits), double (64 Bits) und double extended (> 78 Bits) spezifiziert.

Neben den arithmetischen Befehlen Addition, Subtraktion, Multiplikation, Division und Vergleich zählen noch weitere komplexere mathematische Befehle, z.B. zum Berechnen von Wurzeln, Exponenten, Exponentialfunktion, Kosinus, Tangens etc., zu dieser Gruppe.

SIMD-Befehle (engl. *Single Instruction, Multiple Data*) oder Multimediabefehle (engl. *Multimedia Instructions*) führen gleichzeitig dieselbe Operation auf mehreren Teiloperanden innerhalb eines Operanden aus und wurden nach der Flynnschen Klassifikation [8] benannt. Sie dienen hauptsächlich der Beschleunigung von Multimedia-Anwendungen, wie Videowiedergabe und Bildbearbeitung. Die Teiloperanden können je nach Befehl Ganzzahlen oder Gleitkommazahlen darstellen. Häufig können die Operationen wahlweise mit Sättigungsarithmetik berechnet werden.

Programmsteuerbefehle (engl. *Control Transfer Operations*) sind alle Befehle, die den

Programmablauf direkt ändern, also die bedingten und unbedingten Sprungbefehle, Unterprogrammaufruf und -rückkehr sowie Unterbrechungsaufruf und -rückkehr.

Systemsteuerbefehle (engl. *System Control Operations*) erlauben es, in manchen Befehlssätzen direkten Einfluss auf Prozessor- oder Systemkomponenten, wie z.B. den Daten-Cache-Speicher oder die Speicherverwaltungseinheit, zu nehmen. Weiterhin gehören der HALT-Befehl zum Anhalten des Prozessors und Befehle zur Verwaltung der elektrischen Leistungsaufnahme zu dieser Befehlsgruppe, die üblicherweise nur vom Betriebssystem genutzt werden dürfen.

Synchronisationsbefehle ermöglichen es, Synchronisationsoperationen zur Prozess-, Thread- und Ausnahmebehandlung zu implementieren. Diese Operationen setzen sich aus einzelnen einfacheren Befehlen zusammen, die dabei aber atomar, d.h. ohne Unterbrechung, ausgeführt werden müssen. Ein Beispiel hierfür ist der Swap Befehl, der einen Speicherwert mit einem Register vertauscht oder der Test-And-Set Befehl, der einen Speicherwert erst lädt, auf einen bestimmten Wert testet und abhängig davon einen anderen Wert zurückschreibt. Dies wird für die korrekte Implementierung von Semaphoren bei Multiprozessorsystemen verwendet.

2.1.9 ISA-Typen

2.1.9.1 CISC

Complex Instruction Set Computers (CISC) wurden in den 60er und 70er Jahren bei der Entwicklung von Großrechnern eingesetzt. Damals haben teure und langsame Hauptspeicher (es gab noch keine Cache-Speicher) dazu geführt, dass immer komplexere Maschinenbefehle entworfen wurden, die mehrere Operationen pro Opcode kodieren, um den Hauptspeicher effizient auszunutzen und das Rechenwerk zu beschäftigen.

CISC-Prozessoren charakterisieren sich durch mächtige Maschinenbefehle, umfangreiche Befehlssätze, viele Befehlsformate, Adressierungsarten und spezialisierte Register.

Heutige CISC-Prozessoren – besonders die auf dem Desktoprechner vorherrschende *32 bit Intel Architecture* (IA-32) [9] oder x86-64 [10] Architekturen – wandeln vor der Ausführung die komplexen Befehle in einfache RISC-Befehle um, die Micro-Op oder μOp genannt werden. Sie können deswegen als (superskalare) RISC-Prozessoren mit Codekompression verstanden werden.

2.1.9.2 RISC

Das *Reduced Instruction Set Computer* (RISC) [11] Architekturkonzept entwickelte sich Anfang der 80er Jahre und ist als gegenläufiger Trend zu den damals vorherrschenden CISC-Architekturen zu verstehen. Dies geschah aus folgenden Gründen:

- Manche Befehle werden fast nie verwendet
- Komplexe Befehle lassen sich durch eine Folge einfacherer Befehle ersetzen
- Einfachere Codegenerierung durch den Compiler
- Einfachere Steuerwerke benötigen weniger Chipfläche und Entwicklungszeit
- Verwendung von Pipelining

Frühere RISC-Architekturen lassen sich durch folgende Eigenschaften charakterisieren:

- Kleiner Befehlssatz ($\leq$ 128)
- Wenige Befehlsformate ($\leq$ 4)
- Wenige Adressierungsarten ($\leq$ 4)
- Einheitliche Befehlslänge (üblicherweise 32 Bits)
- Eine große Registeranzahl von mindestens 32 GPR
- Lade-/Speicherarchitektur

Skalare RISC-Prozessoren bezeichnen RISC-Prozessoren, die maximal eine Befehlsausführung pro Takt erreichen. Häufig ist bei solchen Prozessoren das Entwurfsziel, die durchschnittliche Befehlsausführung pro Takt möglichst nah dem Maximum anzunähern.

Superskalare RISC-Prozessoren werden als Steigerung der skalaren RISC-Prozessoren verstanden, weil sie die räumliche Parallelität ausnutzen und mehrere parallele Ausführungseinheiten enthalten. Pro Takt können dabei mehrere Befehle den Ausführungseinheiten zugeordnet und beenden werden. Verschiedene Mikroarchitekturen von superskalaren Prozessoren werden in Abschnitt 2.2.5 behandelt.

2.1.9.3 VLIW

Very Long Instruction Word (VLIW) [12] bezeichnet eine Gruppe von Architekturen, die sich durch ein besonders breites Instruktionsformat auszeichnen, das durch eine meist feste Anzahl von einfachen, voneinander unabhängigen Operationen besteht. Im Vergleich zu superskalaren RISC-Prozessoren wird die Befehlsparallelisierung nicht zur Laufzeit sondern zur Compilezeit durchgeführt. Allgemein liegt dem

Designkonzept zu Grunde, die Hardware möglichst einfach zu halten indem möglichst viele Berechnungen von der Hardware in den Compiler zu verlagern werden. Diese Berechnungen müssen dann in der ISA kodiert werden.

Dafür wird das Programm im Compiler parallelisiert und teilweise sogar die komplette Zuteilung der Operationen zu den einzelnen Funktionseinheiten vorgenommen. Dadurch verringert sich die Chipfläche und Komplexität des Prozessors, aber es wird auch das Instruktionsformat bezüglich der Codegröße in der Regel ineffizienter. Dies kann durch den Einsatz von Codekompressionstechniken kompensiert werden [13]. Ebenso reagiert ein VLIW-Prozessor ineffizienter auf dynamische Ereignisse zur Laufzeit, wie z.B. Cache-Fehlzugriffe, weil diese nicht statisch im Compiler berücksichtigt werden können. Dynamische Ereignisse werden bei statischen VLIW-Prozessoren durch das Anhalten der nachfolgenden Pipelinestufen behandelt während superskalare Architekturen mit dynamischen Scheduling in diesem Fall durch dynamische Umsortieren der Operationen eine Performanzvorteil haben.

Charakterisierung von VLIW:

- Sehr langes Instruktionsformat (meistens zwischen 128 und 1024 Bits)
- Häufig festes Instruktionsformat
- Instruktionsformat ist aus mehreren einfacheren Operationen aufgebaut
- Parallelisierung und Abhängigkeitsauflösung findet im Compiler statt

2.1.9.4 EPIC

Explicit Parallel Instruction Computing (EPIC) [14] ist ein von HP und Intel entwickeltes Befehlsformat, das die Einfachheit von VLIW-Prozessoren mit den Vorteilen des dynamischen Schedulings verbindet.

Ähnlich wie bei VLIW erlaubt EPIC dem Prozessor die Befehlsparallelität im Instruktionsformat mitzuteilen, aber es wird keine direkte Zuteilung zu den Funktionseinheiten vorgenommen. Eine out-of-order Ausführung ist nicht möglich. Vielmehr wird durch einen intelligenten Befehlssatz versucht, die ILP zu erhöhen und die Reaktion auf dynamische Ergebnisse zu verbessern. So gibt es spezielle spekulative Lese- und Schreibzugriffe, die vor dem eigentlichen Speicherzugriff stattfinden und dem Prozessor erlauben, vorzeitig auf Cache-Fehlzugriffe zu reagieren.

Abbildung 2.3 zeigt das EPIC-Befehlsbündel der *64 Bit Intel Architecture* (IA-64), das immer drei Operationen zusammen mit den sog. Template Bits in einem Befehlsformat kombiniert. Die Template Bits kodieren dabei die Inter- und Intraabhängigkeiten der Operationen.

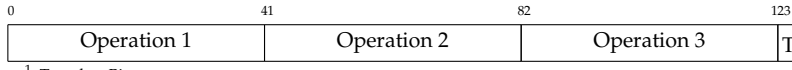


Abbildung 2.3: IA-64 Befehlsbündel

2.1.10 Interrupts und Exceptions

Unter einem Interrupt oder Exceptions versteht man die Unterbrechung der normalen Programmausführung. In der Literatur ist die Verwendung des Begriffs Interrupt und Exception nicht einheitlich geregelt. In dieser Arbeit wird als Interrupt eine Unterbrechung bezeichnet, die von Hardware außerhalb des Prozessors kommt. Eine Exception hingegen wird durch den Prozessor ausgelöst, z.B. durch eine Division durch 0.

Ein Interrupt/Exception, die den Prozessor in einem wohldefinierten Systemzustand hinterlässt, wird als präziser Interrupt (engl. *precise interrupt*) bezeichnet. Solch ein Interrupt/Exception hat vier Eigenschaften:

1. Der *Befehlszeiger* (IP, engl. *Instruction Pointer*) wird an einem bekannten Ort gespeichert
2. Alle Instruktionen vor dem IP müssen komplett ausgeführt worden sein
3. Es darf keine Instruktion nach dem IP ausgeführt worden sein
4. Der Ausführungszustand der Instruktion des IPs muss bekannt sein

2.2 Mikroarchitekturen

Die Mikroarchitektur setzt Befehlssatzarchitektur in einem Prozessor um. Dabei bildet die ISA als Spezifikation die Schnittstelle zur Software, indem sie das Verhalten des Prozessors festlegt. Die Realisierung des Verhaltens in der Hardware wird dann in der Mikroarchitektur beschrieben. Dabei ist die zeitliche Abarbeitung eines Befehls (z.B. wie viele Zyklen dieser benötigt) nicht von der ISA festgelegt sondern abhängig von der Implementierung in der Mikroarchitektur. So wird z.B. die x86 Befehlssatzarchitektur von zahlreichen Prozessoren mit unterschiedlichen Mikroarchitekturkonzepten implementiert: Intel i486 verwendet zur Realisierung der x86 ISA eine skalare Pipeline, der Pentium Prozessor eine superskalare duale in-oder Pipeline und der Pentium Pro eine superskalare out-of-order Pipeline mit spekulativer Ausführung und Registerumbenennung. Der Vorteil in der Trennung zwischen Mikroarchitektur und ISA liegt in der Binärkompatibilität zwischen sämtlichen Prozessoren, die die gleiche ISA verwenden. Diese erlaubt nicht nur die ausführbare Datei auf verschiedenen Prozessoren auszuführen sondern erleichtert ebenfalls die Softwareentwicklung, da nicht für jeden neuen Prozessor ein angepasster Compiler und Binärwerkzeuge (engl. *Binary Utilities*)

entwickelt und zur Verfügung gestellt werden müssen. [15]

Im Folgenden werden gängige Mikroarchitekturkonzepte vorgestellt, die in heutigen Mikroprozessoren zur Steigerung der Performanz eingesetzt werden.

2.2.1 Befehlszyklus

Der Befehlszyklus (engl. *Instruction Cycle*) beschreibt den Ablauf der Abarbeitung einer Instruktion innerhalb eines Prozessors. Der Befehlszyklus wird im Prozessor vom Starten bis zum Anhalten kontinuierlich wiederholt. Der Befehlszyklus kann je nach Prozessortyp und Befehlssatz variieren. Im Folgenden wird ein Befehlszyklus exemplarisch beschrieben:

1. *Befehl laden* (IF, engl. *Instruction Fetch*)

   Zunächst muss der Befehl aus dem Hauptspeicher in den Prozessor geladen werden. Dazu wird ein Lesezugriff auf dem Hauptspeicher an der Adresse des Befehlszeigers (engl. *Instruciton Pointer*) initiiert.

2. *Befehl dekodieren* (ID, engl. *Instruction Decode*)

   Der geladene Befehl aus dem Hauptspeicher wird dekodiert und dabei das Befehlsformat, der Opcode, die Operanden und Adressierungsart der Operanden ermittelt.

3. *Operanden laden* (OF, engl. *Operand Fetch*)

   Die Operanden werden aus dem Registerspeicher geladen.

4. *Befehl ausführen* (EX, engl. *Execute*)

   Der Befehl wird anhand des Opcodes ausgeführt.

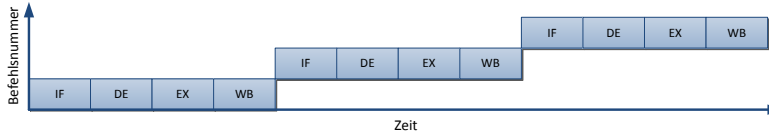
5. *Speicherzugriff* (MEM, engl. *Memory Access*)

   Optional wird bei Lade-/Speicherbefehle der Hauptspeicherzugriff durchgeführt.

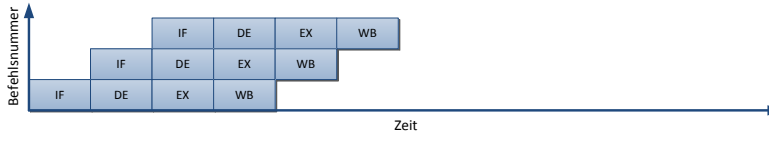
6. *Ergebnis zurückschreiben* (WB, engl. *Write Back*)

   Das Ergebnis der Befehlsausführung wird in den Registerspeicher zurückgeschrieben. Bei einem Sprungbefehl wird der Befehlszeiger entsprechend gesetzt. Ansonsten wird der Befehlszeiger um die Länge des ausgeführten Befehls erhöht.

Der Befehlszyklus kann in einem einfachen Prozessor in mehreren Taktzyklen mittels eines *endlichen Automaten* (FSM, engl. *Finite State Machine*) realisiert werden.



(a) Befehlszyklus mit serieller Abarbeitung



(b) Befehlszyklus mit Pipelining

Legende: Instruction Fetch (IF), Instruction Decode (DE), Execute (EX), Write Back (WB)

Abbildung 2.4: Pipelining und serielle Abarbeitung am Beispiel eines 4-stufigen Befehlszyklus

2.2.2 Pipelining

Bei Pipelining (dt. *Fließbandverarbeitung*) wird der Befehlszyklus im Prozessor nicht mehr mittels einer FSM in mehreren Takten ausgeführt. Stattdessen wird die zeitliche Parallelität (engl. *Temporal Parallelism*) ausgenutzt und der in Teilaufgaben zerlegte Befehlszyklus wird überlappend bearbeitet. Jede Teilaufgabe wird in einer Pipelinestufe ausgeführt und die Menge aller Pipelinestufen bilden dann die gesamte Prozessorpipeline. Dabei befinden sich in der Regel mehrere Befehle in der Pipeline, wobei jeder Befehl sich in einer Pipelinestufe und damit in einem anderen Zustand der Abarbeitung befindet. Eine Performanzsteigerung durch Pipelining wird durch die parallele Abarbeitung mehrerer Befehle erzielt. Dies macht sich durch eine höhere Taktrate und das Fertigstellen von einem Befehl pro Takt bemerkbar. Abbildung 2.4 zeigt die beispielhafte Abarbeitung eines vier-stufigen Befehlszyklus in einem Prozessor mit und ohne Pipelining.

Pipelining nutzt die zeitliche Parallelität der Hardware aus. Die Hardwareressourcen zur Abarbeitung des Befehlszyklus sind bei einer FSM-Realisierung die meiste Zeit ungenutzt, da sich die FSM immer nur in einem Zustand der Abarbeitung befindet. Durch Pipelining wird die zeitliche Verwendung der Hardwareressourcen optimiert. Dies ist allerdings nur mit zusätzlicher Hardware möglich. Zwischen den Pipelinestufen müssen Pipelineregister eingefügt werden und durch die schnellere Verarbeitung müssen auch die Daten schneller in die CPU transportiert werden, was zur Einfüh-

rung von Caches geführt hat.

Eine Pipeline eines Prozessors ist dann besonders effizient, wenn jeder Befehl die Pipeline möglichst gleichmäßig auslastet und jede Pipelinestufe möglichst gleich lang zur Abarbeitung benötigt und somit der Slack der Pipelinestufen möglichst minimal ist. Dies führte Anfang der 80er Jahre zu der Entwicklung von RISC-Prozessoren, da diese besser als CISC für Pipelining geeignet sind. Der CISC-Befehlssatz besteht aus vielen Befehlen mit unterschiedlicher Komplexität und Abarbeitungszeit. So muss eine CISC-Pipeline auf den komplexesten CISC-Befehl ausgelegt sein, dessen Pipelinestufen dann bei einfacheren CISC-Befehlen nicht vollständig benötigt werden. Bei RISC haben dagegen alle Befehle eine möglichst gleiche Komplexität, so dass eine gleichmäßige Auslastung der Pipeline begünstigt wird.

Als Beispiel skalarer Prozessorpipelines ist in Abbildung 2.5 die DLX-Pipeline [15] als Beispiel einer RISC-Pipeline sowie die i486-Pipeline [16] als Beispiel einer CISC-Pipeline aufgeführt.

2.2.3 Datenabhängigkeiten und Pipelinekonflikte

Bei der überlappenden Abarbeitung der Befehle können verschiedene Konflikte entstehen, die die parallele Abarbeitung einschränken:

Datenkonflikte ergeben sich aus Datenabhängigkeiten zwischen Befehlen im Programm. Es wird zwischen echten Datenabhängigkeiten, Gegenabhängigkeiten und Ausgabeabhängigkeiten unterschieden. Bei Pipelining sind häufig nur die echten Datenabhängigkeiten von Bedeutung, bei denen ein Befehl das Ergebnis eines anderen Befehls benötigt. Datenkonflikte können in der Hardware durch Forwarding oder das Erkennen und Anhalten der Pipeline aufgelöst werden.

Steuerkonflikte treten bei Sprungbefehlen auf, die den IP verändern. In der Pipeline muss beim Befehl holen der nachfolgende Befehl vorhergesagt werden. Das Ergebnis eines Sprungbefehls ist allerdings erst am Ende der Pipeline bekannt, so dass bei einem falsch Vorhergesagten Sprung sämtliche nachfolgenden Befehle falsch sind und verworfen werden müssen.

Strukturkonflikte treten auf, wenn Ressourcenkonflikte innerhalb von Befehlen in der Pipeline vorhanden sind, z.B. wenn zwei Pipelinestufen gleichzeitig auf den Registerspeicher zugreifen wollen, aber nur ein Port zur Verfügung steht. Ressourcenkonflikte können durch das Anhalten der Pipeline aufgelöst werden.

2.2.4 Caches

Mit jeder neuen Technologiegeneration steigt die Abarbeitungsgeschwindigkeit des Prozessors stärker als die Zugriffsgeschwindigkeit auf den Hauptspeicher außerhalb

des Chips an. Diese immer stärkere Diskrepanz wird als Memory Wall bezeichnet und führte im Prozessordesign zu der Einführung von Caches. Durch Caches werden Daten aus dem Hauptspeicher näher an die verarbeitenden Einheiten platziert und somit der Delay von Datenzugriffen reduzieren. Caches sind kleiner als der Hauptspeicher und halten immer einen Teil des Hauptspeichers vor. Wenn bei einem Zugriff ein Datum nicht im Cache ist (Cache-Miss) muss das Datum aus dem Hauptspeicher geladen werden und wegen dem begrenzten Platz muss dafür ein anderes Datum aus dem Cache entfernt werden. Je nach Organisation an welchen Stellen im Cache ein Datum abgelegt werden kann, wird zwischen des Caches wird zwischen direct-mapped, n-wege assoziativen und vollassoziativen Caches unterschieden.

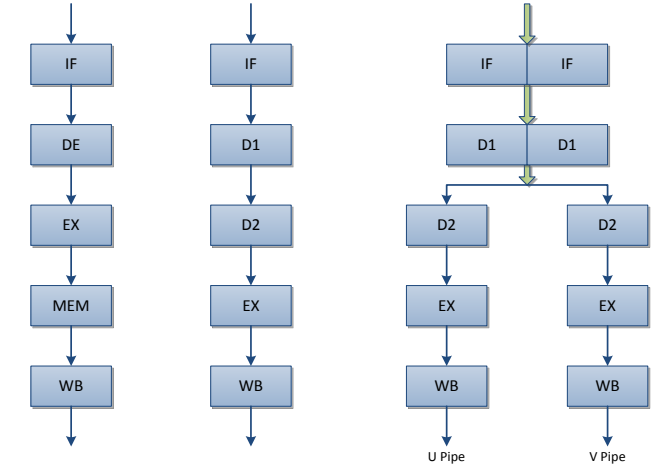
2.2.5 Superskalarität

Bei superskalaren Prozessoren wird zusätzlich zu der zeitlichen Parallelität beim Pipelining noch die räumliche Parallelität (engl. *spatial parallelism*) ausgenutzt. Dazu sind die verarbeitenden Einheiten mehrfach vorhanden und es können – im Gegensatz zu skalaren Prozessoren – mehrere Befehle pro Takt verarbeitet und fertiggestellt werden. Superskalarität im Prozessor kann durch verschiedene Techniken erzielt werden:

- Superskalarität mit statischem Scheduling
- Superskalarität mit dynamischen Scheduling
- *Very Long Instruction Word* (VLIW)

2.2.6 Superskalarität mit statischem Scheduling

Bei superskalaren Prozessoren mit statischem Scheduling ist die Reihenfolge der Befehlsabarbeitung statisch durch den Compiler vorgegeben und wird im Gegensatz zum dynamischen Scheduling nicht durch den Prozessor verändert. Dabei ist die Befehlspipeline mehrfach vorhanden und es werden mehrere aufeinander folgende Befehle gleichzeitig geladen und parallel ausgeführt, sofern die Befehle unabhängig voneinander sind und keine Konflikte entstehen. Abbildung 2.5(c) zeigt als Beispiel die 2-fach superskalare CISC-Pipeline des Pentium Prozessors [17]. Im Vergleich zur Pipeline des i486-Vorgängerprozessors [16] (siehe Abbildung 2.5(b)) wurde die Integer-Pipeline um eine V-Pipeline erweitert, die einfache Befehle parallel zur U-Pipeline ausführen kann. Dabei werden zwei Befehle gleichzeitig geholt (IF) und dekodiert (D1). Während der Dekodierung wird überprüft, ob diese parallel zueinander ausgeführt werden können.



(a) DLX-Pipeline (b) i486-Pipeline (c) Pentium-Pipeline

Legende: Instruction Fetch (IF), Instruction Decode (DE), Execute (EX), Memory Access (MEM), Write Back (WB), First Instruction Decode (D1), Second Instruction Decode (D2)

Abbildung 2.5: Die skalare DLX-Pipeline und i486-Pipeline im Vergleich zur superskalare Pentium-Prozessor-Pipeline

2.2.7 Superskalarität mit dynamischem Scheduling

Bei superskalaren Prozessoren mit dynamischen Scheduling ist die Befehlsausführung in der Hardware unabhängig von der Befehlsreihenfolge, wie sie durch die ISA vorgegeben wird. Unter Einbehaltung der Semantik des kompilierten Programms wird hierbei ein dynamisches Scheduling in der Hardware durchgeführt. Die Technik wird auch *Out-of-Order Execution* (OOE) (dt. *Ausführung in anderer Reihenfolge oder out-of-order Ausführung*) genannt.

Im Vergleich zu Superskalarität mit statischem Schedule kann die Hardware effizienter mit Befehlen mit unterschiedlicher Ausführungszeit umgehen. So müssen Befehle mit einer kurzen Laufzeit (z.B. Integerbefehle) nicht auf Befehle mit einer langen Laufzeit (z.B. Gleitkommabefehle) warten weil die Befehlsreihenfolge eingehalten werden muss. Somit werden insgesamt diversitäre Ausführungseinheiten mit unterschiedlichen Laufzeiten in der Mikroarchitektur begünstigt.

Ein VLIW-Prozessor kann ebenfalls gut mit diversitären Ausführungseinheiten umgehen, da hierbei die Ausführungsdauer eines Befehls bereits zur Compilezeit berücksichtigt werden kann. Allerdings kann eine Pipeline mit OOE besser auf dynamische Ereignisse zur Laufzeit reagieren, die die Ausführungsdauer eines Befehls zur Compilezeit unvorhersehbar machen. Dies gilt insbesondere für Load/Store-Befehle, deren Laufzeit durch den Inhalt von Caches stark beeinflusst werden kann. Da der Inhalt des Caches im Allgemeinen zur Compilezeit nicht vorhersagbar ist, kann auch die Laufzeit eines Load/Store-Befehls in einem statischen Schedule durch den Compiler nicht korrekt berücksichtigt werden und es entsteht zwangsläufig ein ineffizientes Schedule. Durch das dynamische Scheduling kann der Prozessor stattdessen auf dynamische Ereignisse reagieren und z.B. im Falle eines Cache-Misses unabhängige Befehle vorziehen.

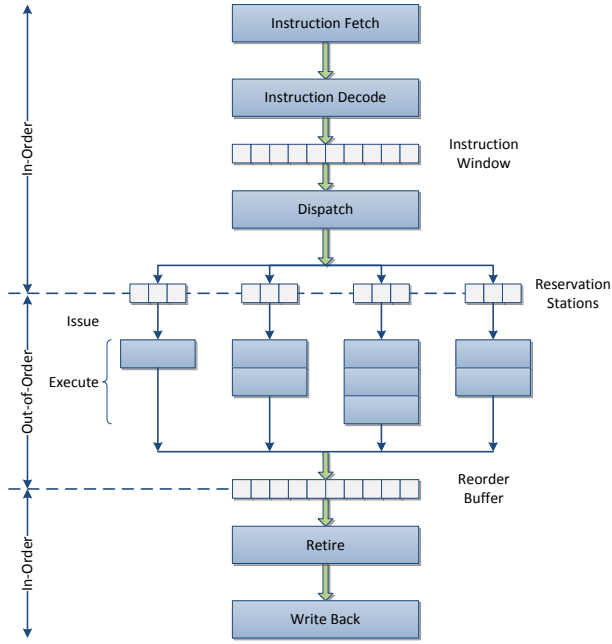


Abbildung 2.6: Beispiel einer superskalaren Pipeline mit dynamischen Scheduling

Das dynamische Scheduling in einem superskalaren Prozessor kann mittels zweier

Techniken realisiert werden: Scoreboarding [18] und der Tomasulo-Algorithmus [19]. Beim Scoreboarding werden in einer zentralen Tabelle, dem Scoreboard, die Datenabhängigkeiten sämtlicher Befehle geloggt. Ein Befehl wird aus dem Scoreboard nur dann zur Ausführung freigegeben, wenn keine Konflikte zu vorherigen in Ausführung befindlichen Befehlen bestehen. Der Nachteil von Scoreboarding ist, dass Namensabhängigkeiten zwischen Befehlen ebenfalls die parallele Ausführung von Befehlen behindern.

Tomasulo löst hingegen Namensabhängigkeiten durch die Verwendung von Registerumbenennung [20] elegant auf. Tomasulo unterscheidet zwischen virtuellen und physikalischen Registern. Die virtuellen Register der ISA werden auf eine größere Anzahl von physikalischen Registern abgebildet und in einer Umrechnungstabelle gespeichert. Bei jedem Schreibzugriff auf ein virtuelles Register wird dieses einem freien physikalischen Register zugewiesen. Das physikalische Register wird erst wieder freigegeben, sobald kein Befehl mehr darauf schreibt oder davon liest. Dadurch wird für die Gültigkeit eines physikalischen Registers dieses genau einmal geschrieben und es können keine Gegen- oder Ausgabeabhängigkeiten zwischen physikalischen Registern mehr auftreten. Außerdem können Datenabhängigkeiten leicht überprüft werden, indem gespeichert wird, ob ein physikalisches Register bereits geschrieben wurde und damit eine validen Wert erhält.

In Abbildung 2.6 ist ein Beispiel einer superskalaren Prozessorpipeline mit dynamischen Scheduling mittels Tomasulo zu sehen. Die Prozessorpipeline ist in in-order und out-of-order Bereiche eingeteilt, in denen jeweils die Befehle innerhalb und außerhalb der Programmreihenfolge abgearbeitet werden. Die ersten beiden Pipelinestufen (Fetch und Decode) sind noch zu superskalaren Pipelines mit statischem Schedule identisch. Danach führt der Dispatcher das dynamische Scheduling durch und weist die Befehle, unter Berücksichtigung der Datenabhängigkeiten zwischen den Befehlen, an die unterschiedlichen Reservation Stations vor den Ausführungseinheiten zu. In den Reservation Stations warten die Befehle bis alle Operanden verfügbar sind und werden danach zur Ausführung angestoßen (engl. *Issue*). Dadurch können sich einzelne Befehle – je nach Operandenverfügbarkeit – überholen und es folgt die out-of-order Ausführung. Nach der Ausführung müssen die Befehle wieder in die richtige Reihenfolge gebracht werden. Dies ist insbesondere zur Vermeidung von Datenkonflikten wegen Ausgabeabhängigkeiten, zur Realisierung von präzisen Interrupts und wegen spekulativer Ausführung durch die Sprungvorhersage notwendig. Dazu werden die Befehle out-of-order in den Reorder Buffer geschrieben. Die Completion Unit holt die Befehle in-order aus dem Reorder Buffer und überprüft dabei, ob der Befehl verworfen werden muss. Falls nicht, wird der Befehl an Retire weitergeleitet und das Ergebnis des Befehls in der richtigen Reihenfolge zurückgeschrieben.

2.2.8 Very Long Instruction Word (VLIW)

Very Long Instruction Word (VLIW) ist eine weitere Möglichkeit zur Realisierung von Prozessoren mit mehreren parallelen Ausführungseinheiten und zeichnet sich durch ein breites Instruktionswort (engl. *Instruction Word*) aus. Dabei bezeichnet VLIW sowohl die Klasse der VLIW-Mikroarchitekturen als auch der VLIW-Befehlssatzarchitekturen, die bereits in Abschnitt 2.1.9.3 vorgestellt wurde. VLIW-Prozessoren werden hauptsächlich im eingebetteten Bereich für datenflussorientierte Anwendungen eingesetzt, die über genügend ILP zur Ausnutzung der parallelen Ausführungseinheiten verfügen.

Im Vergleich zu Superskalarität mit dynamischen Scheduling wird die Parallelisierung der Befehle nicht von der Hardware sondern bereits im Vorfeld vom Compiler durchgeführt und im Instruktionswort kodiert. Dadurch ist bei VLIW-Prozessoren die Kontrolllogik wesentlich einfacher und es kann Chipfläche und Energie eingespart werden. Die Operationen werden immer in-order zur Bearbeitung angestoßen. Wenn eine Operationen z.B. wegen Datenabhängigkeiten warten muss wird die gesamte Verarbeitung der Instruktion angehalten. Dadurch kann ein VLIW-Prozessor im Vergleich eines superskalaren mit dynamischen Scheduling schlechter auf dynamische Ereignisse, die zur Compilezeit nicht vorhersagbar sind, reagieren. Dies gilt insbesondere für Speicherzugriffe, deren Latenz abhängig vom Inhalt des Caches ist. Bei einer Datenabhängigkeit in Folge eines Cache-Misses wird die Verarbeitung der gesamten Instruktion angehalten wodurch VLIW-Prozessoren eine niedrigere Performanz als gleichwertige superskalare Prozessoren mit dynamischen Scheduling haben.

Ein VLIW-Prozessor hat mehrere parallele Issue-Slots. Pro Issue-Slot kann ein VLIW-Prozessor im Idealfall eine Operation pro Taktzyklus zur Bearbeitung anstoßen. Jeder Issue-Slot hat seine eigenen Ausführungseinheiten, wodurch eine parallele Verarbeitung der Befehle ermöglicht wird. Häufig haben VLIW-Prozessoren heterogene Issue-Slots, so dass nicht jeder Slot alle Operationen ausführen kann oder bestimmte Slots für einen Operationstyp spezialisiert ist. Die Position einer Operation im VLIW-Instruktionswort gibt gleichzeitig auch den Issue-Slot an, auf dem die Operation berechnet wird. Abbildung 2.7 zeigt ein Beispiel einer 4-wege VLIW-Pipeline. Eine VLIW-Instruktion mit vier Operationen wird geladen und dekodiert. Danach werden die Operationen in vier getrennten Pipelines ausgeführt und die Ergebnisse zurückgeschrieben. In diesem Beispiel kann nur der erste Issue-Slot auf den Hauptspeicher zugreifen.

Häufig wird unter VLIW auch das Designkonzept verstanden, möglichst viele Berechnungen zur Laufzeit in die Compilezeit zu verlagern. So muss bei einem *Non-Unit Assumed Latencies* (NUAL) VLIW-Prozessor [21] der Compiler nicht nur die Parallelisierung der Befehle sondern auch die Latenz der Befehle berücksichtigen, um ein korrektes Programm zu erzeugen. Dadurch wird die Kontrolllogik in VLIW-Prozessoren weiter reduziert, allerdings kann der VLIW-Prozessor auch schlechter auf Ereignisse zur Laufzeit reagieren, die zur Compilezeit nicht vorhersagbar sind. So muss häufig bei einem Cache-Miss bei einer Ladeoperation die gesamte Pipeline angehalten wer-

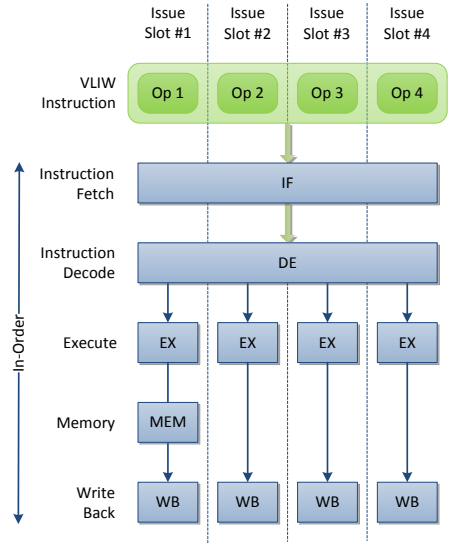


Abbildung 2.7: Beispiel einer VLIW-Prozessor-Pipeline

den, während bei einem dynamischen Scheduling in diesem Fall andere unabhängige Befehle vorgezogen werden könnten.

Im Gegensatz dazu muss ein Compiler bei einem *Unit Assumed Latencies* (UAL) VLIW-Prozessor [21] die Latenz für ein korrektes Scheduling nicht berücksichtigen, sondern kann annehmen, dass jeder Operation vor dem Ausführen der nächsten Instruktion fertiggestellt ist. Dadurch muss der Compiler weniger NOPs verwenden selbst wenn er aus Optimierungsgründen die Latenz der Operationen beim Scheduling berücksichtigt. Allerdings hat ein dynamischer VLIW-Prozessor auch einen höheren Hardwareaufwand da der Prozessor dynamisch das Anstoßen (engl. *Issue*) der Operationen abhängig von den Abhängigkeiten übernimmt.

Abbildung 2.8 zeigt eine Kategorisierung von Prozessoren, die die räumliche Parallelität in Prozessoren und ILP im Befehlssatz zur Performanzsteigerung einsetzen. Darin wird die Ausnutzung von ILP in Aufgaben eingeteilt: die Gruppierung (engl. *Grouping*), das Zuweisen zu funktionale Einheiten (engl. *Function Unit Assignment*) und das Anstoßen (engl. *Issuing*) von Befehlen. In der Abbildung wird gezeigt, dass abhängig von der Architektur die Aufgaben entweder in der Hardware oder Software durchgeführt werden.

	Grouping	Functional Unit Assign	Issuing
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Abbildung 2.8: Kategorien von Prozessoren zur Ausnutzung von ILP
Quelle [22]

2.2.9 Clustered VLIW

Die Skalierbarkeit von VLIW-Prozessoren bezüglich der funktionellen Einheiten wird durch den zentralen multi-ported Registerspeicher und durch das Bypass-Netzwerk limitiert. So steigen die Anzahl der Registerports proportional mit der Anzahl der Issue-Slots an, aber die Fläche von einem Registerspeicher hängt quadratisch mit der Anzahl der Ports zusammen [23] so dass bei einem 8-wege VLIW-Prozessor der multi-ported Registerspeicher unpraktisch groß und langsam wird. Deswegen ist man bei VLIW-Prozessoren mit vielen Issue-Slots dazu übergegangen die monolithische Struktur von VLIW-Prozessoren in kleiner Cluster zu unterteilen, die jeweils über einen eigenen Registerspeicher verfügen und über ein inter-cluster Kommunikationsnetzwerk verbunden sind. Ergebnisse im eigenen Cluster können wie bei unicluster Architekturen schnell weiterverwendet werden. Die Skalierbarkeit von Clustered-VLIW-Architekturen bezüglich der Taktgeschwindigkeit und Fläche wird allerdings dadurch erkauft, dass Ergebnisse von entfernten Clustern zusätzliche Taktzyklen für die Kommunikation benötigen. [24]

2.2.9.1 Inter-cluster Kommunikationsmodelle

Für die Kommunikation zwischen Clustern muss zusätzlich zu unicluster Architekturen *inter-cluster Kommunikation* (ICC, engl. *Inter-Cluster Communication*) eingefügt werden. Hierbei wird grundlegend zwischen architektursichtbarem Clustering (engl. *architecture-visible clustering*) und architekturunsichtbarem Clustering (engl. *architecture-invisible clustering*) unterschieden. Bei architekturunsicherbarem Clustering wird die ICC vor der ISA versteckt und muss automatisch von der Mikroarchitektur eingefügt werden, um den Eindruck eines einzelnen Registerspeichers zu erzeugen. Dies führt häufig zu zusätzlichen Hardwarekosten und Stallzyklen, wenn der Compiler

das versteckte Clustering nicht explizit berücksichtigt.

Stattdessen passt architekturunsichtbares Clustering besser in die Designphilosophie von VLIW-Prozessoren, möglichst viele Berechnungen von der Laufzeit in die Compilezeit zu verlagern. Hierbei muss der Programmierer/Compiler das Clustering bereits während der Compilezeit berücksichtigen. Zu diesem Zweck muss die ISA die Möglichkeiten zur Spezifikation des Mappings von Befehlen zu Clustern und der ICC bieten. Je nachdem, wie ICC in der ISA spezifiziert wird, wird zwischen verschiedenen inter-cluster Kommunikationsmodellen (engl. *Inter-Cluster Communication Models*) [25] unterschieden. Der Compiler für Clustered-VLIW-Architekturen muss dann zusätzlich die Befehle zu Clustering zuweisen und ICC gemäß dem verwendeten ICC-Modell hinzufügen.

In den nächsten Abschnitten werden fünf verschiedenen ICC-Modelle beschrieben und bezüglich der ISA Kodierung, des Registerdrucks, der zusätzlichen Operationen und des ICC-Zeitpunkts bewertet. Dabei wird in den Tabelle 2.1 bis 2.6 jeweils ein Codebeispiel gezeigt, bei dem in Cluster 1 ein Wert erzeugt und dann einmal in Cluster 1 und zweimal in Cluster 2 konsumiert wird.

Explicit-Copy-Modell Die ICC wird diesem Modell mithilfe von speziellen Copy-Operationen realisiert. Diese werden in regulären VLIW-Issue-Slots zusammen mit gewöhnlichen Operationen ausgeführt. Nur die Copy-Operationen können auf entfernte Cluster zugreifen während alle anderen Operationen auf dem lokalen Registerspeicher arbeiten. Tabelle 2.1 zeigt Beispielcode für die ICC im Explicit-Copy-Modell. Das Register r3 aus dem ersten Cluster wird mittels einer Copy-Operation in das Register r1 des zweiten Clusters kopiert.

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r3} \leftarrow r1 + r2$	*	*	*
2	$r5 \leftarrow \mathbf{r3} + r4$	*copy* $\mathbf{r2.1} \leftarrow \mathbf{r3}$	*	*
3	*	*	$r3 \leftarrow \mathbf{r1} + r2$	$r5 \leftarrow \mathbf{r1} + r4$

Tabelle 2.1: Beispielcode zur Verwendung des Explicit-Copy-ICC-Modells
Quelle [118, 24]

Das Explicit-Copy-ICC-Modell hat den Nachteil, dass zusätzliche Operationen für die ICC benötigt werden, die reguläre Operationen verdrängen und somit den Schedule verlängern können. Dafür kann das Modell ohne große Veränderungen an der ISA realisiert werden, weil nur eine neue Operation im Befehlssatz benötigt wird. Ein Vorteil liegt darin, dass der Zeitpunkt der ICC unabhängig vom Erzeuger und Konsument ist und vom Compiler frei gewählt werden kann. Dadurch ist das Schedule flexibler gegenüber Constraints für die ICC, weil ein beliebiger Zeitpunkt zwischen Erzeuger und Konsument für die ICC ausgewählt werden kann.

Dedicated-Issue-Slot-Modell Das Dedicated-Issue-Slot-Modell stellt eine Erweiterung des Explicit-Copy-Modells da. Dabei wird die VLIW-Instruktion um zusätzliche Issue-Slots speziell für ICC-Copy-Operationen erweitert. Tabelle 2.2 zeigt ein Codebeispiel des Dedicated-Issue-Slot-ICC-Modells. Slot 3 und 6 sind in diesem Beispiel für Copy-Operationen reserviert während die restlichen Slots reguläre Operationen ausführen können.

	Cluster 1			Cluster 2		
Cycle	Slot 1	S. 2	Slot 3	Slot 4	Slot 5	S. 6
1	$\mathbf{r3} \leftarrow r1 + r2$	*	*	*	*	*
2	$r5 \leftarrow \mathbf{r3} + r4$	*	*copy* $\mathbf{r2.1} \leftarrow \mathbf{r3}$	*	*	*
3	*	*	*	$r3 \leftarrow \mathbf{r1} + r2$	$r5 \leftarrow \mathbf{r1} + r4$	*

Tabelle 2.2: Beispielcode zur Verwendung des Dedicated-Issue-Slot-ICC-Modells Quelle [118, 24]

Das Dedicated-Issue-Slot-Modell gleicht den Nachteil des Explicit-Copy-ICC-Modells aus, dass regulären Operationen durch Copy-Operationen behindert oder verdrängt werden. Dadurch wird der Schedule durch ICC nicht unnötig verlängert. Der Vorteil wird allerdings durch ein längeres Instruktionswort erkauft, das die zusätzliche Issue-Slots kodieren muss, wodurch insgesamt die Codedichte abnimmt. Wie beim Explicit-Copy-Modell ist die ICC unabhängig vom Erzeuger und Konsumenten.

Extended-Operands-Modell Beim Extended-Operands-ICC-Modell können die Quelloperanden von sämtlichen Clustern lesen während das Ergebnis nur in den lokalen Registerspeicher geschrieben werden kann. Dazu werden die Quelloperanden in der ISA mit einer zusätzlichen Clusteridentifikationsnummer erweitert wodurch das Instruktionswort insgesamt breiter wird. Tabelle 2.2(a) zeigt die Verwendung des Extended-Operands-Modells an einem Codebeispiel. Die erste Operation schreibt ihr Ergebnis in den lokalen Registerspeicher während die zweite Operation im letzten Zyklus direkt das Register r3 vom ersten Cluster lesen kann.

Der Vorteil des Extended-Operands-Modells liegt in der Verwendung von Werten von entfernten Registerspeichern ohne ein lokales Register zu benötigen, wodurch insgesamt der Registerdruck reduziert werden kann. Allerdings ist eine Wiederverwendung des Wertes im Registerspeicher des Konsumenten nicht effizient möglich. Entweder muss der Wert ein zweites Mal kopiert werden, wodurch der ICC-Verkehr erhöht wird und anderen Operationen verzögert werden können, oder der Wert muss erst in ein lokales Register kopiert werden, wodurch der Delay zwischen Erzeuger und Konsument erhöht wird. Die ICC-Transfer ist zeitlich an den Konsumenten gekoppelt. Allerdings kann der Zeitpunkt durch einen zusätzliche Move-Operation frei gewählt werden wodurch allerdings neben den zusätzlich benötigten Befehlen der Delay erhöht wird und ein zusätzliches Register benötigt wird.

(a) Mit einem Konsumenten im Zielcluster

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r3} \leftarrow r1 + r2$	*	*	*
2	$r5 \leftarrow \mathbf{r3} + r4$	*	*	*
3	*	*	$r3 \leftarrow \mathbf{r1.3} + r2$	*

(b) Mit mehreren Konsumenten im Zielcluster

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r3} \leftarrow r1 + r2$	*	*	*
2	$r5 \leftarrow \mathbf{r3} + r4$	*	*	*
3	*	*	$r3 \leftarrow \mathbf{r1.3} + r2$	*
4	*	*	$r5 \leftarrow \mathbf{r1.3} + r4$	*

Tabelle 2.3: Beispielcode zur Verwendung des Extended-Operands-ICC-Modell Quelle [118, 24]

Extended-Results-Modell Das Extended-Results-Modell ist ähnlich dem Extended-Operands-Modell mit dem Unterschied, dass nicht die Quelloperanden auf entfernte Cluster lesen können sondern die Zieloperanden auf entfernte schreiben. Dazu werden die Zieloperanden mit einer Clusteridentifikationsnummer ergänzt. Tabelle 2.3(a) zeigt ein Codebeispiel des Extended-Results-Modells. Hierbei wird das Ergebnis der ersten Operation direkt in Register r3 des zweiten Clusters abgelegt und kann dann von der zweiten Operation lokal gelesen werden.

(a) Mit einem Konsumenten

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r2.1} \leftarrow r1 + r2$	*	*	*
2	*	*	*	*
3	*	*	$r3 \leftarrow \mathbf{r1} + r2$	$r5 \leftarrow \mathbf{r1} + r4$

(b) Mit zwei Konsumenten in unterschiedlichen Clustern

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r3} \leftarrow r1 + r2$	*	*	*
2	$r5 \leftarrow \mathbf{r3} + r4$	$Copy\ \mathbf{r2.1} \leftarrow \mathbf{r3}$	*	*
3	*	*	*	*
4	*	*	$r3 \leftarrow \mathbf{r1} + r2$	$r5 \leftarrow \mathbf{r1} + r4$

Tabelle 2.4: Beispielcode zur Verwendung des Extended-Results-ICC-Modells Quelle [118, 24]

Der Vorteil des ICC-Modells liegt darin, dass das Ergebnis direkt in entfernte Regis-

terspeicher geschrieben werden kann und dadurch der Registerdruck im lokalen Registerspeicher reduziert wird. Allerdings ist das Modell nachteilhaft, wenn der Wert in mehreren Clustern benötigt wird, weil dann zusätzliche Copy-Operationen benötigt werden und dadurch das Ergebnis im zweiten Cluster erst später zur Verfügung steht. Die zusätzliche Copy-Operation befindet sich dann auch im Cluster des Erzeugers, das mit hoher Wahrscheinlichkeit gut ausgelastet ist, sonst würde es keinen Sinn ergeben eine Berechnung in ein anderes Cluster zu verlagern. Tabelle 2.3(b) verdeutlicht den Fall an einem Beispiel. Das Ergebnis wird erst in den lokalen Registerspeicher geschrieben und danach von diesem mittels einer Copy-Operation ins zweite Cluster kopiert, wo es erst im vierten Zyklus dann verwendet werden kann. Der ICC-Transfer ist zeitlich an den Erzeuger gekoppelt. Allerdings kann der Zeitpunkt durch einen zusätzlichen Copy-Befehl frei gewählt werden wodurch allerdings neben dem zusätzlich benötigten Befehl der Delay erhöht wird und ein zusätzliches Register benötigt wird.

Mutlicast-Modell Das Multicast-Modell ist eine Erweiterung des Extended-Results-Modells. Das Ergebnis einer Operation kann dabei nicht nur in ein Register sondern in mehrere Register und Cluster geschrieben werden. Tabelle 2.5 zeigt ein Codebeispiel zum Multicast-Modell. Die erste Operation schreibt das Ergebnis sowohl ins erste als auch ins zweite Cluster. Dadurch kann es ohne zusätzliche Verzögerung in beiden Clustern verwendet werden.

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	**r3**, **r2.1** $\leftarrow$ $r1 + r2$	*	*	*
2	$r5 \leftarrow$ **r3** $+ r4$	*	*	*
3	*	*	$r3 \leftarrow$ **r1** $+ r2$	$r5 \leftarrow$ **r1** $+ r4$

Tabelle 2.5: Beispielcode zur Verwendung des Multicast-ICC-Modells Quelle [118, 24]

Das Multicast-Modell gleicht sowohl den Nachteil des Extended-Results-Modells bei mehreren Konsumente in unterschiedlichen Clustern als auch den Nachteil des Extended-Operand-Modells bei mehreren Konsumenten im entfernten Cluster aus. In beiden Fällen wird weder eine zusätzliche Copy-Operation noch der Delay erhöht. Der Nachteil des Modells liegt in der Kodierung von zusätzlichen Zielregistern in der ISA.

Broadcast-Modell Ein Spezialfall des Multicast-Modells ist das Broadcast-Modell, in dem wahlweise die Ergebnisse an alle Cluster verteilt werden. Dazu wird der Adressraum der Register in globale und lokale Register eingeteilt. Die globalen Register können aus allen Clustern gelesen und geschrieben werden während jedes Cluster nur auf seine eigenen lokalen Register zugreifen kann. In Hardware greifen die einzelnen Cluster nicht auf einen einzelnen globalen Registerspeicher zu sondern jedes

Cluster hat seine eigene Kopie der globalen Register. Bei einem Schreibzugriff auf ein globales Register wird dessen neuer Wert an sämtliche Cluster transferiert, um einen konsistenten Zustand der globalen Register zu gewährleisten. Durch dieses Verfahren ist das Ergebnis im lokalen Cluster früher verfügbar als in den entfernten Clustern. In Tabelle 2.6 ist ein Codebeispiel zu sehen. Sämtliche Register über 64 sind globale Register während die niederwertigen Registernummern lokale Register adressieren. Durch r127 wird auf ein globales Register zugegriffen und somit das Ergebnis an sämtliche Cluster verteilt. Ab den nächsten Zyklus kann im lokalen Cluster auf das Register zugegriffen werden wobei im zweiten Cluster ein Zyklus länger gewartet werden muss.

	Cluster 1		Cluster 2	
Cycle	Slot 1	Slot 2	Slot 3	Slot 4
1	$\mathbf{r127} \leftarrow r1 + r2$	*	*	*
2	$r5 \leftarrow \mathbf{r127} + r4$	*	*	*
3	*	*	$r3 \leftarrow \mathbf{r127} + r2$	$r5 \leftarrow \mathbf{r127} + r4$

Tabelle 2.6: Beispielcode zur Verwendung des Broadcast-ICC-Modells Quelle [118, 24]

Aus Sicht der ISA bietet das Broadcast-Modell eine effiziente Kodierungsmöglichkeit, weil ein beliebiger Teil der Registeradressen für globale Register verwendet werden kann. Der Nachteil des Broadcast-Modells liegt in dem vergleichsweise hohem Registerdruck, da immer alle Cluster geschrieben werden und die Wahlmöglichkeit der Zielregister beschränkt ist. In der Praxis wirkt sich das letztendlich negativ auf die Performanz aus.

2.3 Architekturbeschreibungssprachen

Der Begriff *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*) wird für die Beschreibung von Software- und Hardwarearchitekturen gleichermaßen verwendet.

Software-ADLs spezifizieren das Verhalten von Softwarekomponenten und deren Interaktion untereinander. Dadurch können Softwarearchitekturen repräsentiert und analysiert werden.

Hardware-ADLs beschreiben die Struktur (Hardwarekomponenten und ihren Zusammenhang) und/oder das Verhalten von Hardwarekomponenten (z.B. der Befehlssatzarchitektur von einem Prozessor). Das Konzept, Maschinenbeschreibungssprachen zur Spezifikation von Architekturen zu verwenden, existiert schon länger. Frühe ADLs wurden für die Simulation, Evaluierung und Synthese von Computern und anderen digitalen Systemen verwendet.

Im weiteren Verlauf dieses Kapitels werden Hardware-Architekturbeschreibungssprachen speziell für Prozessoren und Mehrprozessorsystemen [26, 27, 28, 29, 30, 31] behandelt. In Abschnitt 2.3.1 werden zunächst ADLs zu anderen Sprachen abgrenzt. Danach werden in Abschnitt 2.3.2 eine inhaltsbasierte und zielbasierte Klassifikation vorgestellt. Anschließend wird die Prozessorbeschreibung (Abschnitt 2.3.3) und die Systembeschreibung (Abschnitt 2.3.4) von existierenden ADLs im Detail besprochen. Abschnitt 2.3.5 fasst die Architekturbeschreibungssprachen zusammen.

2.3.1 Abgrenzung zu anderen Sprachen

Es existieren keine klaren Kriterien, ob eine Sprache eine ADL ist oder nicht. Daher gestaltet sich die Abgrenzung von ADLs zu anderen Sprachen, wie z.B. Programmiersprachen (z.B. C, C++), Hardwarebeschreibungssprachen (z.B. VHDL, Verilog) und Modellierungssprachen (z.B. UML) gestaltet sich nicht so einfach. Im Folgenden werden Hardware ADLs zu verschiedenen Sprachen abgegrenzt:

Natürliche Sprachen Natürliche Sprachen, wie z.B. Englisch, werden heute noch verwendet, um Spezifikationen zu verfassen. Sie entziehen sich automatischer Analyseverfahren und es kann die Eindeutigkeit, Vollständigkeit und Konsistenz der Spezifikation nicht sichergestellt werden. Diese Probleme können zu verschiedenen Interpretationen einer Spezifikation führen. Somit sollten ADL Spezifikationen eine formale eindeutige Semantik haben.

Formale Sprachen Formale Spezifikationssprachen können – im Gegensatz zu natürlichen Sprachen – für die Verifikation und Analyse verwendet werden. Manche davon sind populär geworden, gerade weil sie als Eingangssprache von mächtigen Verifikationsprogrammen verwendet wurden. Diese Sprachen sind für Designer und Entwickler ungeeignet, da sie keine einfache Umsetzung der Architekturanleitung erlauben.

Programmiersprachen Programmiersprachen haben die Eigenschaft, dass sie Architekturen in Einzellösungen exakt im Detail beschreiben. ADLs hingegen versuchen, ein Modell für die Architekturbeschreibung zu bieten, das eine Reihe von Architekturen abdeckt. Einzellösungen sind hier unerwünscht oder gar nicht möglich.

Im Prinzip bieten Programmiersprachen die Möglichkeit der Beschreibung von Architekturen. So können einzelne Komponenten als Klassen, z.B. in C++, modelliert werden. Durch den hohen Freiheitsgrad können diese aber nur umständlich für automatische Analyseverfahren auf Architekturebene verwendet werden. Ein weiteres Problem bei traditionellen Programmiersprachen ist die fehlende Beschreibung von Hardware-Features wie Parallelität oder Synchronisation.

Modellierungssprachen Modellierungssprachen, wie z.B. *Unified Modeling Language* (UML), können gut für die Beschreibung von zusammenhängenden Kom-

ponenten verwendet werden. Abstrakte Daten, wie z.B. der Befehlssatz eines Prozessors, lassen sich mit ihnen allerdings schwer abdecken.

Hardwarebeschreibungssprachen *Hardwarebeschreibungssprachen* (HDLs, engl. *Hardware Description Languages*) wie *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) [32] oder Verilog [33] sind nicht ausreichend abstrakt für die Beschreibung und Erforschung der Architektur auf Systemebene. Die Extraktion der Struktur der Architektur ist durch Reverse Engineering mit überschaubarem Aufwand möglich. Beim Befehlssatz gestaltet sich eine Extraktion wesentlich schwieriger, besonders wenn kompliziertere Optimierungsmethoden wie Pipelining verwendet wurden.

Viele Sprachen können theoretisch für eine Architekturbeschreibung verwendet werden. Eine Abgrenzung, welche als ADL zählen und welche nicht, ist deswegen nicht möglich. Sprachen, die für die Architekturbeschreibung entwickelt wurden, besitzen wesentliche Vorteile gegenüber Sprachen, die aus anderen Gründen entwickelt wurden. Sie repräsentieren die Informationen in einer möglichst einfachen Form bei der die niedrigere Flexibilität durch ein höheres Maß an Abstraktion ersetzt wird. Abbildung 2.9 zeigt ein Venn-Diagramm, das ADLs im Vergleich zu anderen Sprachen zeigt.

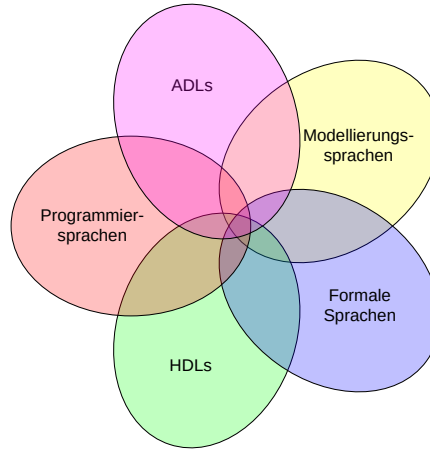


Abbildung 2.9: Venn-Diagramm von ADLs und anderen Sprachen im Vergleich (Quelle [31])

Mögliche Kriterien für Architekturbeschreibungssprachen:

- Formal eindeutige Semantik
- Möglichst einfache Beschreibung der Architektur
- Abstrakte Modellierung von Informationen
- Automatisierte Analyse und Weiterverwendung der Architekturinformationen

2.3.2 Klassifikation

Es gibt mehrere Möglichkeiten, Architekturbeschreibungssprachen für Prozessoren und Mehrprozessorsystemen zu klassifizieren. In dieser Arbeit wird eine Klassifikation per Inhalt und per Ziel vorgenommen.

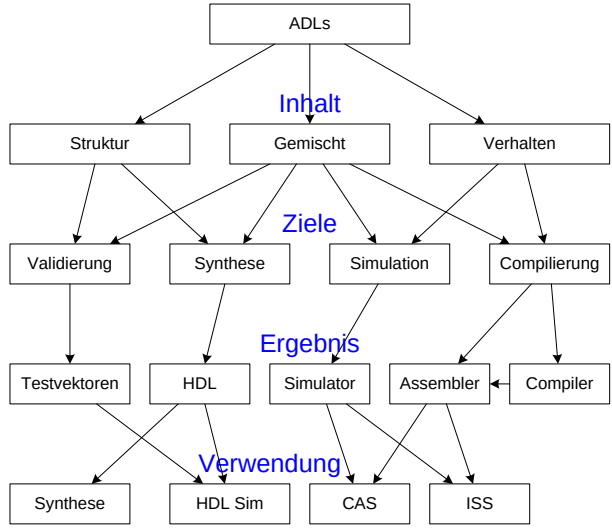


Abbildung 2.10: Inhalt, Ziele, Ergebnisse und Verwendung von Architekturbeschreibungssprachen

2.3.2.1 Inhaltsbasierte Klassifikation

Inhaltlich lassen sich ADLs in folgende Kategorien einteilen:

Prozessorbeschreibende ADLs enthalten eine detaillierte Beschreibung des Prozessors oder Prozessorkern eines Systems. Die Beschreibung eines Prozessors kann durch eine Verhaltensbeschreibung, Strukturbeschreibung oder eine redundante gemischte Beschreibung erfolgen, wodurch diese Klasse von ADLs noch weiter unterteilt werden kann.

Strukturorientierte ADLs (engl. *Structural ADLs*) beschreiben die Struktur in Form von Komponenten und ihrer Verbundenheit. Dabei ist das größte Problem eine geeignete Abstraktionsebene zu finden, um alle Features von verschiedenen Prozessoren zu unterstützen. Im Allgemeinen gilt, je niedriger die Abstraktionsebene, desto allgemeingültiger die Sprache. Häufig wird die *Registertransferebene* (RTL, engl. *Register Transfer Level*) für die strukturelle Beschreibung verwendet.

Verhaltensorientierte ADLs (engl. *Behavioral ADLs*) spezifizieren explizit den Befehlssatz und die semantische Bedeutung der Befehle. Dabei werden detaillierte Hardwarestrukturen ignoriert. Gewöhnlich sind sich verhaltensorientierte ADLs und die Befehlssatz-Handbuch sehr ähnlich.

Gemischte ADLs (engl. *Mixed ADLs*) beinhalten sowohl strukturelle wie verhaltensbeschreibende Details einer Architektur. Entscheidend ist bei gemischten ADLs, dass eine redundante Beschreibung des Prozessors auf zwei Abstraktionsebenen gegeben ist und diese immer vom Benutzer synchron gehalten werden müssen.

Systembeschreibende ADLs enthalten eine detaillierte strukturelle Beschreibung von Prozessorsystemen, die aus ein oder mehreren Prozessoren aufgebaut sind. In der Systembeschreibung ist das exakte Verhalten oder die exakte Struktur der Prozessoren nicht gegeben. Eine ADL kann allerdings sowohl die Systemsicht als auch die Prozessorsicht abdecken, wodurch sie sowohl prozessorbeschreibend als auch systembeschreibend zu klassifizieren ist. Allerdings ist in diesem Fall keine redundante Beschreibung der Architektur wie im Falle von gemischten ADLs gegeben, weshalb die Kombination aus prozessor- und systembeschreibenden ADLs keine eigene Klassifizierung hat.

2.3.2.2 Zielbasierte Klassifikation

Je nach Zielsetzung der ADL kann dieser eine oder mehrere der folgenden Kategorien zugeteilt werden:

Kompilierungsorientierte ADLs (engl. *Compilation-oriented ADLs*) werden als Eingangssprache für einen retargierbaren Compiler (engl. *retargetable Compiler*) verwendet zu werden. Ein Compiler wird als retargierbar bezeichnet, wenn er mit vergleichsweise niedrigem Aufwand an eine neue Zielarchitektur angepasst werden kann. Bei kompilierungsorientierten ADLs wird die Retargierbarkeit ty-

pischerweise dadurch erreicht, dass neben dem Programm eine Maschinenbeschreibung als Eingabe für die Kompilierung mitgegeben wird.

Eine alternative Methode besteht darin, dass Teile des Quellcodes des Compilers mit Hilfe der ADL automatisch generiert werden. Der Vorteil besteht in einer meist schnelleren Kompilierung. Der Nachteil besteht darin, dass der Compiler für eine neue Zielarchitektur erst neu kompiliert werden muss, was zum einen länger dauert und zum anderen Abhängigkeiten von anderen Programmen/-Compilern erzeugt.

Verhaltensorientierte und gemischte ADLs sind für eine retargierbare Compilergenerierung gut geeignet, weil sie die Informationen für einen retargierbaren Compiler direkt enthalten. Bei strukturorientierten ADLs muss zuerst die Befehlssatzarchitektur aus der Strukturbeschreibung extrahiert werden, was allerdings im Allgemeinen nicht immer möglich ist.

Simulationsorientierte ADLs (engl. *Simulation-oriented ADLs*) haben als Ziel, eine möglichst schnelle Simulation der Architektur zu gewährleisten. Dabei kann die Simulation auf verschiedenen Abstraktionsebenen durchgeführt werden und liefert als Ergebnis unterschiedlich akkurate Informationen über aktuellen Zustand der Architektur. Allgemein gilt, je höher die Abstraktionsebene desto höher die Simulationsgeschwindigkeit.

- ***Befehlssatzsimulator*** **(ISS, engl. *Instruction Set Simulator*)** ist ein funktionaler Simulator, der nur das Verhalten des Befehlssatzes simuliert. Die Ausführungsdauer eines Befehls spielt dabei keine oder eine untergeordnete Rolle. ISS ist gut für das Testen und Debuggen von Programmen geeignet. Funktionale Simulatoren können durch verhaltensorientierte ADLs generiert werden.
- ***Taktgenauer Simulator*** **(CAS, engl. *Cycle-Accurate Simulator*)** ist eine Simulator, der möglichst auf den Takt genau einen Prozessor simuliert. Dabei werden im Gegensatz zum Befehlssatzsimulator die Prozessorinterna, wie z.B. die Pipelinestruktur oder Cache-Architektur, bei der Simulation berücksichtigt. CAS können Aufschluss über die Performanz und den ungefähren Energieverbrauch einer Applikation geben. Taktgenaue Simulatoren können mit strukturellen ADLs generiert werden.
- **HDL-Simulation** ist die Simulation basierend auf einer Hardwarebeschreibungssprache. Dabei muss es möglich sein, eine HDL-Beschreibung generieren zu können, die dann für die Simulation verwendet wird. Eine HDL-Simulation gibt Aufschluss über die genauen Timing-Informationen und kann genaue Informationen über den Energieverbrauch liefern.

Syntheseorientierte ADLs (engl. *Synthesis-oriented ADLs*) werden für die Hardwaregenerierung verwendet. Dabei wird aus der ADL eine HDL wie VHDL, Verilog oder SystemC generiert.

Validierungsorientierte ADLs (engl. *Validation-oriented ADLs*) ermöglichen eine automatische Testgenerierung für die funktionale Validierung von eingebetteten Prozessoren oder Systemen.

2.3.3 Prozessorbeschreibung in ADLs

2.3.3.1 Machine Independent Microprogramming Language (MIMOLA)

Machine Independent Microprogramming Language (MIMOLA) [34] ist sowohl eine strukturorientierte ADL als auch eine High-Level-Programmiersprache, die an der Universität Dortmund entwickelt wurde. Ursprünglich war sie für die Beschreibung von Mikroarchitekturen gedacht. Einer der Hauptvorteile von MIMOLA besteht darin, dass sie für die Synthese, Simulation, Testgenerierung und Kompilierung verwendet werden kann. Eine Toolchain bestehend aus dem MSSH-Hardwaresynthetisierer, dem MSSQ-Codegenerator, dem MSST-Selbsttest-Programmcompiler, dem funktionalen MSSB-Simulator, dem MSSU-RTL-Simulator und wurden basierend auf der MIMOLA-Sprache entwickelt. Außerdem wird MIMOLA von dem RECORD-Compiler [35] verwendet.

Die MIMOLA-Beschreibung besteht aus zwei Teilen: Dem Hardwareteil, bestehend aus der Komponentennetzliste, und einem Softwareteil, in dem Programme in einem PASCAL-ähnlichen Syntax beschrieben werden.

Hardware Hardwarestrukturen in MIMOLA werden durch Komponenten/Module und einer darauf aufbauenden Netzliste modelliert. Jedes Modul wird auf der Registertransferebene beschrieben. Folgendes Beispiel zeigt eine einfache arithmetische Einheit:

Quelltext 2.1: MIMOLA: Beispiel einer Modulspezifikation

```
MODULE Alu (
    IN i1, i2: (15:0);
    OUT outp: (15:0));
    IN ctr: (1:0)
);
CONBEGIN
    outp <- CASE ctr OF
        0: i1 + i2 ;
        1: i1 - i2 ;
        2: i1 AND i2 ;
        3: i1 ;
        END AFTER 1;
CONEND;
```

Module werden in einem VHDL-ähnlichen Stil deklariert. Die erste Zeile deklariert das Modul namens *Alu*. Die folgenden drei Zeilen beschreiben den Namen, die Richtung und die Bitbreite der Modulports. Zwischen CONBEGIN und CONEND befindet sich die RTL-Beschreibung. Besteht sie aus mehreren Anweisungen, werden diese gleichzeitig ausgeführt. Das zeitliche Verhalten wird in diesem Fall durch die AFTER-Anweisung beschrieben.

Für eine komplette Netzliste müssen Verbindungen zwischen den einzelnen Modulports definiert werden. Das folgende Beispiel definiert zwei Verbindungen:

Quelltext 2.2: MIMOLA: Beispiel einer Verbindungsspezifikation

```
CONNECTIONS     Alu.outp -> ACCU.inp
                Accu.outp -> alu.i1
```

Der MSSQ-Codegenerator extrahiert Informationen über den Befehlssatz aus der Modulnetzliste, die für die spätere Codegenerierung verwendet werden. Er transformiert die RTL-Hardwarestruktur in einen sog. *Connection Operation Graph* (COG). Die Knoten eines COG repräsentieren Hardwareoperationen oder Modulports, die Kanten bestimmen den Datenfluss durch die Knoten. Der *Befehlsbaum* (I-Tree, engl. *Instruction Tree*), der einen Eintrag für die Befehlskodierung enthält, wird ebenfalls aus der Netzliste und den Modulen, die für die Dekodierung zuständig sind, generiert. Während der Codegenerierung wird der PASCAL-ähnliche Quellcode in eine Zwischendarstellung umgewandelt. Danach werden ein Musterabgleich (engl. *Pattern Matching*) mit dem COD und die Registerzuteilung durchgeführt. Der MSSQ-Compiler gibt direkt den binären Code durch Abfrage des I-Trees aus.

Damit der Compiler wichtige Hardwaremodule finden kann, müssen Verknüpfungsinformationen (engl. *Linkage Information*) bereitgestellt werden. Wichtige Einheiten, wie z.B. Registerbänke, Befehlsspeicher, Befehlszähler etc., werden dadurch identifiziert. In folgendem Beispiel wird der Befehlsspeicher und Befehlszähler spezifiziert. Dabei sind PCReg und IM vorher definierte Module.

Quelltext 2.3: MIMOLA: Beispiel von Verknüpfungsinformationen

```
LOCATION_FOR_PROGRAMCOUNTER PCReg;
LOCATION_FOR_INSTRUCTIONS IM[0..1023];
```

Selbst mit den Verknüpfungsinformationen ist es, wegen der Flexibilität der RTL-Beschreibung, immer noch sehr schwierig, den COG und I-Trees zu extrahieren. Daher mussten zusätzliche Einschränkungen verhängt werden, damit der MSSQ-Codegenerator richtig arbeiten kann.

Software Das MIMOLA-Softwareprogrammiermodell ist eine Erweiterung zu PASCAL. Es unterscheidet sich von anderen High-Level-Programmiersprachen dadurch,

dass es dem Programmierer erlaubt, Variablen an bestimmte Register zu binden und Hardwarekomponenten wie Funktionsaufrufe zu verwenden. Um z.B. eine Operation, die durch das Modul *Simd* durchgeführt wird, zu verwenden, kann ein Programmierer einfach folgendes schreiben:

Quelltext 2.4: MIMOLA: Beispiel einer Modulverwendung in der Software

```
x: = Simd(y,z);
```

Dieses Feature hilft dem Programmierer, die Codegenerierung zu kontrollieren und spezielle Befehle direkt zu verwenden.

2.3.3.2 Unified Design Language (UDL/I)

Unified Design Language (UDL/I) [36] ist eine strukturelle ADL. Sie wurde als Hardwarebeschreibungssprache an der Kyushu Universität in Japan entwickelt und wird für die Compilergenerierung in der COACH-ASIP-Entwicklungsumgebung eingesetzt. UDL/I wird für die Beschreibung von Prozessoren auf der RTL-Ebene auf einer zyklenbasierten Basis verwendet. Der Befehlssatz wird automatisch aus der UDL/I-Beschreibung extrahiert und danach für die Generierung eines Compilers und Simulators verwendet. COACH setzt einen einfachen RISC-Prozessor voraus und unterstützt weder ILP noch Prozessorpipelines. Die Prozessorbeschreibung wird mit Hilfe des UDL/I-Synthesesystems synthetisiert. Der Hauptvorteil des COACH-Systems liegt darin, dass nur eine einzige Beschreibung für Synthese, Simulation und Kompilierung benötigt wird. Allerdings muss der Designer dem System Hinweise geben, damit dieser wichtige Maschinenzustände, wie den Programmzähler oder die Registerbänke, finden kann. Wegen der Schwierigkeit der *Befehlssatzextraktion* (ISE, engl. *Instruction Set Extraction*) wird diese für VLIW oder superskalare Architekturen nicht unterstützt.

2.3.3.3 nML

nML [37] ist eine verhaltensorientierte ADL, die den Fokus auf die Beschreibung des Befehlssatzes legt. Sie wurde anfänglich von der technischen Universität Berlin entwickelt und später von IMEC und Target Compiler Technologies erweitert. nML wurde von den Codegeneratoren CBC [38] und CHESS [39] sowie von den Simulatoren Sigh/Sim [40] und CHECKERS verwendet. Gegenwärtig wird eine CHESS/CHECKERS-Umgebung [41]für die automatische und effiziente Softwarekompilierung und Befehlssatzsimulation benutzt.

Befehlssatzbeschreibung In nML kann der Befehlssatz mit einer hierarchischen Baumstruktur beschrieben werden. Dabei wird eine top-down Entwurfsmethologie

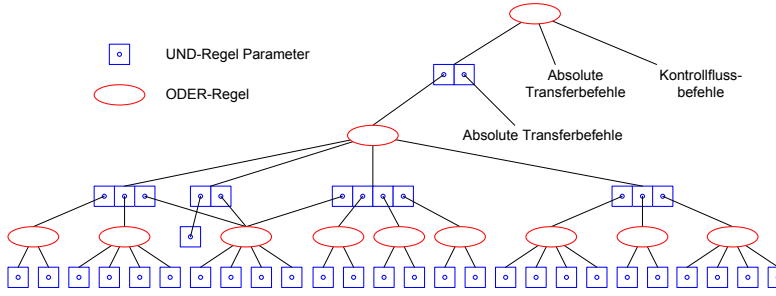


Abbildung 2.11: Hierarchische Beschreibung in nML
(Quelle [37])

vorgeschlagen. Die Baumknoten bestehen aus so genannten *partiellen Befehlen* (PI, engl. *Partial Instructions*), die – zusammen mit ihren Kindern – einen Teil des Befehlssatzes beschreiben. Die Wurzel repräsentiert den gesamten Befehlssatz. PIs können aus einer von zwei gegensätzlichen Arten von Regeln bestehen:

1. ODER-Regeln (engl. *OR-Rules*) zählen alle Alternative für einen Teil eines Befehls auf.
2. UND-Regeln (engl. *AND-Rules*) beschreiben und setzen einen Teilbefehl aus mehreren anderen Teilbefehlen zusammen.

Diese Regeln bilden eine Grammatik, in der jede mögliche Ableitung einen Befehl repräsentiert. Bei einer UND-Regel ist die Anzahl der Ableitungsmöglichkeiten gleich dem Produkt der Ableitungsmöglichkeiten der Kinder. Bei einer ODER-Regel ergeben sich die Ableitungsmöglichkeiten aus der Summe der Ableitungsmöglichkeiten der Kinder. Die Blätter des Baumes bestehen immer aus UND-Regeln und haben eine Ableitungsmöglichkeit. Abbildung 2.11 zeigt ein Beispiel eines nML-Baumes.

Die Eigenschaften eines partiellen Befehls werden in den UND-Regeln beschrieben. Diese haben sogenannte Attribute, die das Verhalten (action), die Kodierung (image) und den Assemblersyntax (syntax) eines Teilbefehls definieren. In der Attributbeschreibung können die Attribute der Kinder verwendet werden. So kann bspw. ein Befehlsformat mit einem Opcode und zwei Operanden in einer UND-Regel festgehalten werden, wobei sich die Bitkodierung als Konkatenation der Bitkodierungen der Kinder definieren lässt.

Beispiel Die folgende nML-Beschreibung zeigt ein Beispiel einer Befehlsspezifikation:

Quelltext 2.5: nML: Beispiel einer Befehlsspezifikation

```
op numeric_instruction(a:num_action, src:SRC, dst:DST)
action {
    temp_src = src;
    temp_dst = dst;
    a.action;
    dst = temp_dst;
}

op num_action = add | sub

op add()
action = {
    temp_dst = temp_dst + temp_src
}
```

Der Quellcode beinhaltet die Definitionen von drei Regeln:

numeric_instruction beschreibt eine UND-Regel, die sich aus den drei partiellen Befehlen (PI) num_action, SRC und DST zusammensetzt.

num_action verwendet die ODER-Regel, um die beiden erlaubten PIs add und sub zu beschreiben.

add beschreibt eine UND-Regel ohne Element. Es wird das *action* Attribut definiert, das in der numeric_instruction durch a.action verwendet wird.

In diesem Beispiel wurde mit den verschiedenen *action* Attributen die hierarchische Definition des Verhaltens der Befehle gezeigt.

Strukturelemente nML bietet ebenfalls die Möglichkeit strukturelle Elemente mit in die Beschreibung zu integrieren. Diese werden durch die Definition von Speicherelementen durchgeführt. Die Verhaltensbeschreibung definiert dabei prinzipiell den Registertransfer zwischen den einzelnen Speicherelementen.

Speicherelemente werden grundsätzlich durch ihren Namen, Größe und Elementtyp beschrieben. Es wird zwischen Registern (*reg*), Speichern (*mem*) und Transitionen (*trn*) unterschieden. Letztere sind flüchtige Speicher, die ihren Speicherinhalt nach einer bestimmten Zeit wieder verlieren. Sie wurden zur Modellierung von Pipelinestufen eingeführt. Die Dauer eines Zugriffs kann mit dem *delay*-Schlüsselwort angegeben werden. Ansonsten wird eine Verzögerung von Null angenommen.

Im folgenden Quelltext werden ein Speicher, ein Register und zwei Transitionen definiert. Der Speicher umfasst 1024 Elemente bestehend aus 16 Bit Integerwerten. Das Register hat vier 32 Bit Festpunktelemente. Die Transitionen werden mit einer Verzögerung von eins definiert. Beide werden synchronisiert und unterteilen so den Datenpfad in zwei Pipelinestufen.

Quelltext 2.6: nML: Beispiel einer Speicherspezifikation

```
mem m[1024,int(16)]
reg r[4,fix(1,31)]

trn as[1,int(16)] delay=1
trn ms[1,int(32)] delay=1 sync=as
```

2.3.3.4 Instruction Set Description Language (ISDL)

Instruction Set Description Language (ISDL) [42] ist eine verhaltensorientierte ADL, die am MIT entwickelt wurde und von dem AVIV-Compiler [43] und GENSIM-Simulatorgenerator [44] verwendet wird. Das Problem der Abhängigkeitsmodellierung wird in ISDL durch explizite Spezifikation vermieden. ISDL zielt hauptsächlich auf die Entwicklung von VLIW-Prozessoren.

Eine Beschreibung besteht aus hauptsächlich fünf Sektionen. Zusätzlich dazu existiert noch eine Sektion, die Informationen für den Compiler enthält, um dessen maschinenabhängige Optimierung mit Hilfe von architekturspezifischen Tipps verbessern zu können.

Befehlsformat (Instruction Word Format) Die Instruktion-Word-Format-Sektion beschreibt das Befehlsformat der Architektur. Es ist unterteilt in verschiedene Felder, die jeweils aus einem oder mehreren Teilfeldern bestehen. Von jedem Teilfeld ist die Bitbreite angegeben. Das Befehlsformat ergibt sich aus der Aneinanderreihung sämtlicher Teilfelder.

Quelltext 2.7: ISDL: Beispiel einer Befehlswortformatspezifikation

```
Section Format

DBM = OP[8], MODE[8];
Main = OP[8];
```

Dieses Beispiel beschreibt einen 24 Bit Befehl, bestehend aus drei Teilfeldern: DBM.OP, DBM.MODE und Main.OP. Jeder Teilfeld ist acht Bits breit. DBM.OP ist das *Most Significant Byte* (MSB) und Main.OP das *Least Significant Byte* (LSB).

Globale Definitionen (Global Definitions) Diese Sektion beinhaltet eine Liste von Definitionen, die in den späteren Sektionen verwendet werden. Sie helfen bei der Assemblergenerierung und bestehen aus den folgenden drei Hauptgruppen:

Tokens dienen der symbolischen Repräsentation des Assemblersyntaxes innerhalb des Parsers. Sie können Namen von Registern oder Registerbänken, Immediatewerte etc. beschreiben. Zusätzlich gibt es die Möglichkeit der Gruppierung von syntaktisch verwandten Tokens. Um zwischen verschiedenen Elementen einer Gruppe unterscheiden zu können, geben diese einen Wert zur Identifikation zurück. Auf diese Weise können z.B. die Registernamen R0 bis R15 durch ein Token abgekürzt werden, dessen Wert mit einer Registernummer übereinstimmt. Ein Beispiel einer Token-Definition eines binären Operands lautet:

Quelltext 2.8: ISDL: Beispiel einer Tokenspezifikation

```
Token X[0..1] X_R ival {yylval.ival = yytext[1] - '0';}
```

Dabei folgt dem *Token*-Schlüsselwort das Assemblerformat des Operands. *X_R* ist der symbolische Name des Tokens, der für Referenzen verwendet wird. *ival* beschreibt den Rückgabewert des Tokens. Danach folgt die Berechnung des Wertes. In diesem Beispiel wird der Assemblersyntax *X0* und *X1* für das Token *X_R* definiert und der Rückgabewert 0 bzw. 1 dafür festgelegt (vom zweiten Zeichen des Syntaxes wird das '0'-Zeichen abgezogen).

Non-terminals sind im Gegensatz zu Tokens Nichtterminalsymbole. Sie bestehen aus einer Liste von Tokens oder Non-terminals. Sie können dazu verwendet werden mehrere syntaktisch unabhängige Symbole in einer Gruppe zusammenzufassen. Folgendes Beispiel zeigt eine Definition eines Non-terminals:

Quelltext 2.9: ISDL: Beispiel einer Nichtterminalsymbolspezifikation

```
Non_Terminal ival XYSRC:
    X_D {$$ = 0;} |
    Y_D {$$ = Y_D + 1;};
```

Die Definition von *XYSRC* besteht aus dem Schlüsselwort *Non_Terminal*, dem Typ *ival* des Rückgabewertes, einem symbolischen Namen für Referenzzwecke und ein Aktionsfeld, das alle möglichen Kombinationen von Tokens oder Nonterminals beschreibt und jedem einen Rückgabewert zuweist.

In diesem Beispiel verweist *XYSRC* auf die beiden Tokens *X_D* und *Y_D* als Auswahlmöglichkeiten. Es gibt den Wert 0 für *X_D* oder den um eins inkrementierten Rückgabewert von Y_D zurück.

Split functions sind spezielle Funktionen, die ein langes Bitfeld als Eingabe haben und dieses in mehrere existierende Teilfelder des Befehlswortes aufteilen.

Speicherressourcen (Storage Resources) Diese Sektion listet alle Speicherressourcen auf, die für den Programmierer sichtbar sind. Es werden der Name und die Größe

von Speichern, Registerbänken und Spezialregistern definiert. Diese Sektion wird vom Compiler für die Bestimmung der verfügbaren Ressourcen verwendet.

Es folgt eine Auflistung aller möglicher Speicherressourcen und ihrer Definitionen:

Memory(depth, width)	Speicher
RegFile(depth, width)	Registerbank
Register(width)	Einzelnes Register
CRegister(width)	Kontroll- und Statusregister
Stack(depth, width)	Stack
ProgramCounter(width)	Befehlszähler
Wire(width)	Eine Verbindung für den Datenpfad

Befehlssatz (Instruction Set) Die Befehlssatzsektion ist in verschiedene Felder zerteilt, die jeweils eine unterschiedliche Operation beschreiben, die in einem Befehl parallel abgearbeitet werden kann. Dadurch wird die Beschreibung von VLIW-Architekturen ermöglicht. Einige Felder sind optional. Jedes Feld beinhaltet eine Liste von erlaubten Operationen.

Die Beschreibung einer Operation besteht aus folgenden Elementen:

Operation Name ist das Assemblerkürzel der Operation.

Operation Parameters ist eine Parameterliste, bestehend aus Tokens oder Non-Terminals.

Bitfield Assignment Commands beinhaltet eine Reihe von Befehlen zur Manipulation des Bitfelds.

RTL Description beschreibt das Verhalten der Operation bezogen auf die Speicherressourcen. Der Compiler verwendet die RTL-Beschreibung für die Auswahl der Operationen.

Costs beschreibt eine oder mehrere Kostenfaktoren der Operation. Dies können z.B. Ausführungszeit, Codegröße, Kosten durch Ressourcenkonflikte, etc. sein.

Timing beschreibt, wann die Änderungen der Speicherressourcen durch die RTL-Beschreibung stattfinden. Damit kann man z.B. die Verzögerungen durch Pipelining angeben.

Quelltext 2.10: ISDL: Beispiel einer Assemblerspezifikation

```
Section Assembly

Field Main:
    ADC XYSRC, ACC
        { Main.OP = 0x21|(ACC<<3)|(XYSRC<<4); }
        { ACC <- ACC + XYSRC + CCR[0]; }
```

```
        { cycle = 2 + dbm; size = 1 + dbm; }
        { latency = 1; }
```

Das *Field*-Schlüsselwort kennzeichnet Operationen, die parallel abgearbeitet werden können. In diesem Beispiel hat die ABC-Operation zwei Parameter. Das *Main.OP*-Bitfeld wird auf das Ergebnis von $0x21|(ACC << 3)|(XYSRC << 4)$ gesetzt.

In der zweiten geschweiften Klammern befinden sich Operationen, die jeweils eine RTL-Beschreibung beinhalten. Bei der ADC-Operation wird der Inhalt des Akkumulators *ACC* auf die Quelle *X* oder *Y* und dem Carry-Bit *CCR[0]* addiert. Das Ergebnis wird zurück in den Akkumulator geschrieben.

In der dritten geschweiften Klammer befinden sich Operationen, die die Kosten der Operationen beschreiben. In diesem Beispiel sind zwei Kosten vorhanden: Zyklenanzahl *Cycle* und Codegröße *size*. Die *ADC*-Operation benötigt zwei Zyklen und ein Befehlswort so lange sie nicht mit einer parallelen Operation gruppiert ist, die zusätzliche Zyklen oder Befehlswörter benötigt.

Die letzten geschweiften Klammern beinhalten Timing-Informationen.

Abhängigkeiten (Constraints) Die Befehlssatzsektion beschreibt eine Anzahl von Feldern, die prinzipiell alle parallel ausgeführt werden können. Wegen Ressourcenabhängigkeiten erlaubt die Hardware nicht das Ausführen beliebiger Kombinationen.

Die Constraints-Sektion dient der Definition der unerlaubten Kombinationen, damit der Compiler diese bei der Synthese nicht erzeugt.

Abhängigkeiten werden in ISDL mit Hilfe einer Menge von booleschen Regeln modelliert. Alle Regeln müssen für einen Befehl erfüllt sein, damit dieser zugelassen ist. Es können auch zeitversetzte Abhängigkeiten modelliert werden, um Konflikte zwischen Befehlen, die zu unterschiedlichen Zyklen angestoßen wurden, aufzuzeigen. Zur Vereinfachung können auch Platzhalter (engl. *Wildcards*) verwendet werden.

Quelltext 2.11: ISDL: Beispiel einer Abhängigkeitsspezifikation

```
Section Constraints

~(( REP *) & ([1] DO *,*))
```

In diesem Beispiel wird definiert, dass die *DO*-Operation nicht erlaubt ist, wenn sie nach einer *REP* ausgeführt wird. Durch $[1]$ wird die Zeitverzögerung von einer Befehlsaufnahme (engl. *instruction fetch*) für die *DO*-Operation angedeutet.

Optionale Architekturdetails (Optional Architectural Details) Die ISDL-Beschreibung kann dem Compiler Informationen über die Hardwarearchitektur bereitstellen,

damit dieser bessere maschinenabhängige Codeoptimierungen durchführen kann. Diese Sektion ist nicht notwendig, damit der Compiler guten Code erzeugen kann. Die Informationen in dieser Sektion unterstützen vielmehr den Compiler, ein paar Optimierungen zu finden, um besseren Code zu erzeugen. Ein Beispiel von solchen Optimierungen ist die Verwendung von Branch-Delay-Instruktionen oder Vorschläge zur Sprungvorhersage.

2.3.3.5 HMDES

Die Maschinenbeschreibungssprache **HMDES** [45] gehört zu der Kategorie der gemischten ADLs. Sie wurde an der University of Illinois at Urbana-Champaign für den IMPACT-Forschungscompiler entwickelt. Der IMPACT-Compiler hat als Schwerpunkt die Erforschung der *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*).

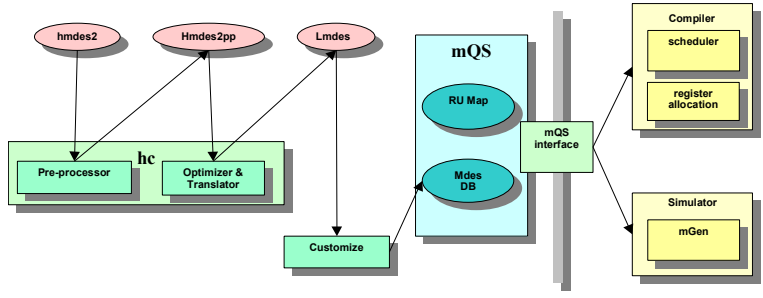


Abbildung 2.12: MDES in Trimaran
(Quelle [46])

Die Sprache bietet C-ähnliche Präprozessorfähigkeiten, wie Einfügen von Dateien, Makroexpansion und bedingtes Einfügen. Zusätzlich werden noch komplexere Strukturen, wie das Expandieren von Schleifen oder Zahlenbereichen, unterstützt. Eine *High-Level Machine Description* (HMDES)-Beschreibung ist die Eingabe für das *Maschinenbeschreibungssystem* (MDES, engl. *Machine Description System*) der Trimaran-Compilerinfrastruktur, die den IMPACT- und den Elcor-Forschungscompiler von HP Labs beinhaltet (siehe Abschnitt 2.4.2.4). Die Beschreibung wird zuerst vorverarbeitet, optimiert und in eine Low-Level-Darstellung namens LMDES überführt. Eine Maschinendatenbank (*Mdes DB*) liest die Low-Level-Dateien und stellt die Informationen dem Compiler-Backend über ein vordefiniertes Interface (*mQS*) zur Verfügung [47].

MDES beinhaltet die Struktur- und Verhaltensbeschreibung des Zielprozessors. Die Informationen sind in kleine Sektionen, wie Format, Ressourcenverwendung, Latenz, Operationen und Register, eingeteilt. Obwohl es keine explizite Beschreibung von Operationsbündeln gibt, kann der VLIW-Zuteilungsprozess mit Hilfe von VLIW-Slots spezifiziert werden.

Als Beispiel beschreibt der folgende Code die Register und Registerbänke. Er enthält 64 Register. Die Registerbank beinhaltet neben der Breite der Register weitere optionale Felder wie den generellen Registertyp *virtual*, sowie spekulative, statische und rotierende Register. Der Wert „1“ definiert ein spekulatives und „0“ ein nicht-spekulatives Register.

Quelltext 2.12: HMDES: Beispiel einer Registerspezifikation

```
SECTION Register {
    R0(); R1(); ... R63();
    'R[0]'(); ... 'R[63]'();
    ...
}
SECTION Register_File {
    RF_i(width(32) virtual(i) speculative(1) static(R0...R63) rotating('R
          [0]'...'R[63]'));
    ...
}
```

Ein besonderer Wert wird auf die Beschreibung der Reservierungstabellen (engl. *Reservation Tables*) gelegt. IMPACT ist besonders interessant für Architekturen, die viele Operationen gleichzeitig zuteilen können und es daher viele alternative Schedulingmöglichkeiten gibt. Z.B. kann bei einer Architektur, die gleichzeitig acht Operationen verarbeiten kann, eine Operation durch eine der acht Dekodiereinheiten und eine der acht funktionalen Einheiten bearbeitet werden. Dadurch ergeben sich insgesamt 64 Schedulingmöglichkeiten. Um eine mühselige Auflistung zu verhindern, wird in MDES eine Und-Oder-Baumstruktur für die Reservierungstabellenbeschreibung verwendet. Abbildung 2.13 illustriert die hierarchische Beschreibung der Reservierungstabellen. Jedes Blatt enthält den Ressourcenverbrauch (engl. *ressource usage*) als Tuple (Ressource, Zeit).

MDES erlaubt nur eine eingeschränkte Retargierbarkeit des taktgenauen Simulators für die HPL-PD-Prozessorfamilie. MDES erlaubt zwar die Beschreibung von Speichersystemen ist aber auf die traditionelle Hierarchie von Registern, Caches und Hauptspeicher limitiert.

2.3.3.6 EXPRESSION

EXPRESSION [48] ist eine gemischte Architekturbeschreibungssprache, die an der University of California, Irvine entwickelt wurde. Sie wird von dem gleichnamigen

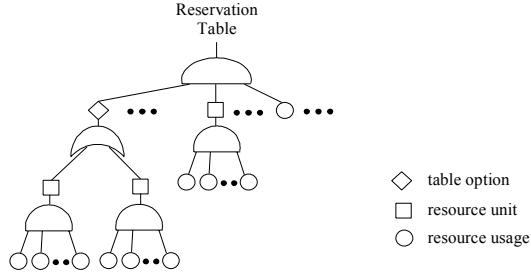


Abbildung 2.13: HMDES Reservation Table Hierarchie
(Quelle [26])

Toolkit verwendet [49]. Dieses umfasst den retargierbaren C-Compiler EXPRESS und den taktgenauen Simulator SIMPRESS. Ein Prozessor wird durch eine Netzliste, die verschiedene Einheiten und Speicher miteinander verbindet, beschrieben. Dadurch bietet EXPRESSION die Möglichkeit, aus der Netzliste automatisch *Reservierungstabellen* (RT, engl. *Reservation Tables*) zu generieren.

Die EXPRESSION-Beschreibung verwendet eine LISP-ähnlichen Syntax. Sie ist in die zwei Hauptsektionen Verhalten (Befehlssatz) und Struktur unterteilt. Die Verhaltenssektion besteht aus den drei Untersektionen Operationen, Befehle und Operationsabbildung (engl. *Operation Mapping*). Weiterhin kann die Struktursektion in die drei Untersektionen Komponenten, Pipeline & Datentransferpfad (engl. *Data Transfer Path*) und Speichersystem unterteilt werden.

Operationen Die Operationenuntersektion beinhaltet den Befehlssatz des Prozessors. Jede Operation besteht aus ihrem Opcode und Operanden. Bei jedem Operanden kann der Typ und die möglichen Ziele angegeben werden. Ein nützliches Feature sind die Operationengruppen. Damit ist es möglich, Operationen in einer Gruppe zusammenzufassen, die später als Ganzes einfach referenziert werden können.

Zum Beispiel zeigt der folgende Codeabschnitt eine Operationengruppe *alu_ops*, bestehend aus den zwei ALU-Operationen *add* und *sub*.

Quelltext 2.13: EXPRESSION: Beispiel einer Operationenspezifikation

```
(OP_GROUP alu_ops
    (OPCODE add
        (OPERANDS (SRC1 reg) (SRC2 reg/imm) (DEST reg))
        (BEHAVIOR DEST = SRC1 + SRC2)
    ...)
```

```
    (OPCODE sub
        (OPERANDS (SRC1 reg) (SRC2 reg/imm) (DEST reg))
        (BEHAVIOR DEST = SRC1 - SRC2)
    ...)
)
```

Befehle Die Befehlsuntersektion beschreibt die Parallelität, die in der Architektur verfügbar ist. Jeder Befehl beinhaltet eine Liste von Slots. Jeder Slot wird durch eine Operation gefüllt und ist mit einer funktionalen Einheit (engl. *Functional Unit*) verbunden.

Quelltext 2.14: EXPRESSION: Beispiel einer Befehlsspezifikation

```
(INSTR
    (SLOTS
        (UNIT ALU)
        (UNIT MULT)
        (UNIT AGU1)
        (UNIT AGU2)
    )
)
```

In diesem Beispiel wird ein Befehl *INSTR* mit vier Slots definiert (ALU, Mult und zwei parallele Transferbefehle).

Operationsabbildung Die Untersektion Operationsabbildung wird zur Beschreibung von Informationen verwendet, die der Compiler für die Befehlsauswahl und architekturspezifischen Optimierungen benötigt.

Das folgende Beispiel zeigt die Abbildung des allgemeinen Assemblerbefehls *iadd* auf die Operation *add* der Zielarchitektur. Eine Multiplikation mit zwei wird automatisch durch eine Addition ersetzt.

Quelltext 2.15: EXPRESSION: Beispiel einer Operationsabbildungsspezifikation

```
(OP_MAPPING
    (
        (GENERIC(iadd src1 src2 dst))
        (TARGET(add src1 src2 dst))
    )
    (
        (GENERIC(mult src1 #2 dst))
        (TARGET(add src1 src1 dst))
    )
)
```

Komponentenspezifikation Diese Untersektion beschreibt Komponenten der Architektur auf Registertransferebene. Es können Pipelineeinheiten, funktionale Einheiten, Speicherelemente, Ports, Verbindungen oder Busse sein. Für Multi-Cycle oder Pipelineeinheiten kann zusätzlich noch das zeitliche Verhalten angegeben werden.

In folgendem Beispiel wird eine ALU-Komponente spezifiziert. Sie unterstützt alle Operationen der Operationengruppe *alu_ops*, die in der ersten Untersektion definiert wurden.

Quelltext 2.16: EXPRESSION: Beispiel einer Komponentenspezifikation

```
(ExUnit ALU() (OPCODES alu_ops))
```

Pipeline und Datentransferpfad Diese Sektion beschreibt die Netzliste des Prozessors. Die Pipelinebeschreibung (engl. *Pipeline Description*) bietet einen Mechanismus, die in den Pipelinestufen enthaltenen Einheiten zu spezifizieren. Die Datentransferpfadbeschreibung (engl. *Data-Transfer Path Description*) ermöglicht die Spezifikation von zulässigen Datentransfers. Die Informationen in dieser Sektion werden sowohl für den retargierbaren Simulator als auch für die automatische Generierung der Reservierungstabellen, die vom Scheduler benötigt werden, verwendet.

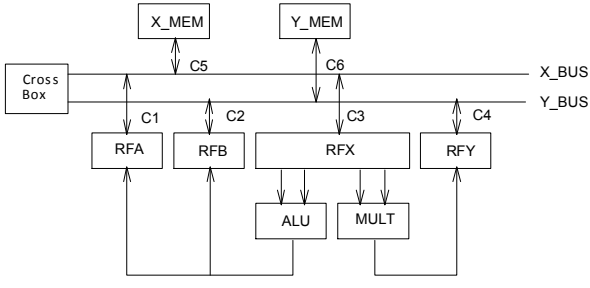


Abbildung 2.14: Eine Beispielarchitektur in EXPRESSION (Quelle [48])

Quelltext 2.17: EXPRESSION: Beispiel einer Spezifikation einer Pipeline und eines Datentransferpfads

```
(PIPELINE FETCH DECODE EX)
    (EX: ALU | Mult | AGU1 | AGU2)
```

```
(DPATHS (TYPE BI)
    (RFA X_bus C1)
    (RFB Y_bus C2)
    (RFX X_bus C3)
    (RFY Y_bus C4)
    (X_mem X_bus C5)
    (Y_mem Y_bus C6)
    (X_bus Y_bus CrossBox)
)
```

Das Beispiel definiert als erstes eine dreistufige Pipeline, wobei die Ausführungsphase *EX* aus einer *arithmetisch-logischen Einheit* (ALU, engl. *Arithmetic Logical Unit*), einem Multiplizierer und zwei Adressgeneratoren besteht.

Mit *DPATHS* werden die Datentransferpfade beschrieben. Der Typ (*TYPE*) gibt an, ob es sich um uni- oder bidirektionale Verbindungen handelt. Jede einzelne Verbindung besteht aus einer Quelle, Senke und einer Komponente, die während des Transfers aktiv ist. Abbildung 2.14 verdeutlicht den Aufbau des Beispiels. *AGU1* und *AGU2* sind in der Grafik nicht eingezeichnet.

Speichersystem Diese Untersektion [50] beschreibt den Typ und die Eigenschaften von verschiedenen Speicherkomponenten, wie Registerbänke, SRAMs, DRAMs, Caches, usw.

Das folgende Codestück zeigt einen 256 KiB großen, zwei-fach assoziativen Level-1-Datencache und einen vier MiB großen Hauptspeicher. Der Datencache besteht aus 1024 Zeilen. Eine Zeile hat 32 Wörter und ein Wort ist 64 Bits breit. Es wird die *Least Recently Used* (LRU)-Verdrängungsstrategie und das Write-Back-Rückschreibeverfahren angewandt.

Quelltext 2.18: EXPRESSION: Beispiel einer Speichersystemspezifikation

```
(STORAGE_SECTION
    (L1
        (TYPE DCACHE)
        (WORDSIZE 64)
        (LINESIZE 32)
        (NUM_LINES 1024)
        (ASSOCIATIVITY 2)
        (REPLACEMENT_POLICY LRU)
        (WRITE_POLICY WRITE_BACK)
        (LEVEL 1)
        (ADDRESS_RANGE 0 32K)
        (NEXT_LEVEL 2)
    )
    (MainMemory
        (TYPE DRAM)
```

```
        (SIZE 4M)
        (WIDTH 8)
        (LEVEL 3)
    )
)
```

2.3.3.7 Language for Instruction Set Architecture (LISA)

Language for Instruction Set Architecture (LISA) [51] zählt zu den gemischten ADLs. Sie wurde ursprünglich an der Universität RWTH Aachen [52] aus einer simulationsorientierten Sichtweise entwickelt und definiert. Die Sprache wurde bereits für produktionsreife Simulatoren [53] verwendet. Ein wichtiger Aspekt der Sprache stellt die Möglichkeit dar, den Kontrollflusspfad direkt zu spezifizieren. Eine explizite Modellierung des Datenflusses und Kontrollflusspfades ist für die taktgenaue Simulation wichtig. Mittlerweile bietet Coware Inc. [54] ein Framework namens Coreware Processor Design basierenden auf der LISA-ADL für die automatische C-Compiler- [55], Assembler- und Simulatorgenerierung an.

Die LISA-Beschreibung setzt sich aus den zwei Deklarationstypen Ressourcen und Operationen zusammen:

Ressourcen Die Ressourcenbeschreibung deckt alle Hardwareressourcen wie Register, Pipelines und Speicher ab.

Das Pipelinemodell definiert alle möglichen Pipelinepfade, die eine Operation durchlaufen kann. Folgendes Beispiel zeigt die Pipelinebeschreibung der VLIW-DLX-Architektur.

Quelltext 2.19: LISA: Beispiel eine Pipelinespezifikation

```
PIPELINE int = {Fetch; Decode; IALU; MEM; WriteBack}
PIPELINE flt = {Fetch; Decode; FADD1; FADD2; FADD3; FADD4; MEM; WriteBack}
PIPELINE mul = {Fetch; Decode; MUL1; MUL2; MUL3; MUL4; MUL5; MUL6; MUL7; MEM;
      WriteBack}
PIPELINE div = {Fetch; Decode; DIV; MEM; WriteBack}
```

Operationen Operationen sind die Basisobjekte von LISA. Sie repräsentieren das Verhalten aus der Sicht des Designers, die Struktur und den Befehlssatz der Architektur. Operationendefinitionen beinhalten die Beschreibung verschiedener Eigenschaften des Systems, wie das Operationsverhalten, Befehlssatzinformationen und das zeitliche Verhalten. Diese werden in verschiedenen Sektionen definiert.

LISA nutzt die Gemeinsamkeiten verschiedener Operationen durch Gruppierung aus. Das folgende Codestück beschreibt das Verhalten in der Dekodierpipelinestufe der beiden Operationen (*ADDI* und *SUBI*). Diese haben beide eine Immediate im Befehlswort enthalten. Das komplette Verhalten einer Operation ergibt sich aus der Kombination der Verhaltensdefinitionen aus sämtlichen Pipelinestufen.

Quelltext 2.20: LISA: Beispiel einer Operationenspezifikation

```
OPERATION i_type IN pipe_int.Decode {
    DECLARE {
        GROUP opcode={ADDI || SUBI}
        GROUP rs1, rd = {fix_register};
    }
    CODING {opcode rs1 rd immediate}
    SYNTAX {opcode rd "," rs1 "," immediate}
    BEHAVIOR { reg_a = rs1; imm = immediate; cond = 0; }
    ACTIVATION {opcode, writeback}
}
```

2.3.3.8 Target Description Language (TDL)

Target Description Language (TDL) [56] wurde an der Universität des Saarlandes entwickelt. Sie zielt auf die Beschreibung von VLIW-Prozessoren. Die Sprache wird von *Postpass Retargetable Optimizer and Analyser* (PROPAN) verwendet. Das PROPAN-Framework wird für die postpass-orientierte Analyse und die Optimierung von Programmen eingesetzt. Die Eingabe besteht aus einem Assemblerprogramm sowie einer dazugehörigen TDL-Beschreibung. Das PROPAN-System analysiert das Programm und erzeugt eine optimierte Assemblerdatei als Ausgabe. Als Optimierungsverfahren wird unter anderem *Integer Linear Programming* (ILP) eingesetzt.

Eine TDL Beschreibung besteht aus vier Sektionen:

Ressourcenspezifikation In der Ressourcenspezifikation wird jede Hardwarekomponente aufgeführt, die die durchzuführende Analyse und Optimierung beeinflussen kann. Dem Anwender steht eine Menge vordefinierter Ressourcentypen zur Verfügung: funktionale Einheiten (FuncUnit), Register und Registermengen (Register), Speicher (Memory) und Caches (Cache). Für jeden dieser Ressourcentypen existieren eine Menge vordefinierter Attribute.

Quelltext 2.21: TDL: Beispiel einer Ressourcenspezifikation

```
Resources Section
...
FuncUnit ALU replication=2;
```

```
Register gpr "r%d" [0:31] size=32, type=signed<32>;
SetProperties gpr[30] usage=SP;
RegisterAlias dreg "d%d" gpr mapping=[2:1], type=float<56,8>;
Memory DM type=data, align=16, access=32;
DefineAttribute Replacement {"LRU","FIFO"} associated to Cache;
Cache InstrCache assoc=2, size=256, linesize=32, type=instr, Replacement = LRU
      ;
```

Quelltext 2.21 zeigt ein Beispiel einer Ressourcenspezifikation. Zuerst wird als Ressourcentyp eine funktionale Einheit deklariert, die durch den eindeutigen Namen *ALU* identifiziert werden kann. Von ihr können genau zwei Instanzen mit identischen Eigenschaften existieren.

Danach wird eine Menge von 32 32 Bit Registern definiert, die zur Speicherung von Zweierkomplementzahlen verwendet werden. Innerhalb der Zielanwendung kann diese Registermenge durch den eindeutigen Namen *gpr* identifiziert werden. Die Assemblerrepräsentation der einzelnen Register lautet *r0,r1,...,r31*. Durch das Schlüsselwort *SetProperties* können Attributen bereits deklarierter Ressourcen eigenständigen Werten zugewiesen werden. In diesem Beispiel wird das Register *r30* als Stack-Pointer deklariert.

Für alle Ressourcen können Aliasbeziehungen deklariert werden. Hierdurch können unterschiedliche Sichten für bereits deklarierte Ressourcen eingeführt werden. Die Attributsetzungen dieser Sichten können sich unterscheiden. Im Beispiel aus Quelltext 2.21 wird spezifiziert, dass jeweils zwei aufeinanderfolgende Integer-Register als ein Double-Register verwendet werden können. Die Assemblerrepräsentation dieser Register lautet *d0,d1,...,d15* und ihr Inhalt wird standardmäßige als Gleitkommazahl mit 56 Bit Mantisse und 8 Bit Exponent interpretiert.

Ein Beispiel für benutzerdefinierte Attribute ist die Deklaration eines Attributes, das die Ersetzungsstrategie von Caches spezifiziert. Benutzerdefinierte Attribute werden durch das Schlüsselwort *DefineAttribute* deklariert. Darauf folgt der Name des Attributes, der zulässige Wertebereich und schließlich die Menge der Ressourcen, zu deren Beschreibung dieses Attribut verwendet werden kann. In 2.21 kann das Attribute *Replacement* die Werte *LRU* oder *FILO* annehmen. Die restlichen in der Cache-Deklaration verwendeten Attribute sind vordefiniert Attribute.

Neben den vordefinierten Ressourcentypen kann der Benutzer mit dem Schlüsselwort *DefineResource* neue Ressourcentypen definieren. Diese können durch benutzerdefinierte Attribute beschrieben werden. Im Beispiel wird zur Beschreibung der Ressource *MyResource* ein Attribut *MyAttrib* eingeführt, dessen Wertebereich die Menge der ganzen Zahlen ist. Daneben besteht die Möglichkeit, vordefiniert Attribute dahingehend zu erweitern, dass sie auch für benutzerdefinierte Ressourcen verwendet werden können.

Befehlssatzspezifikation In der Befehlssatzspezifikation wird von jedem Befehl die Assemblerdarstellung, die Menge an Quell- und Zieloperanden sowie dessen Zeitverhalten beschrieben.

Jeder Befehl kann aus mehreren parallel auszuführenden Operationen bestehen. Jede Operation wird einer Ressource zugeordnet, auf der sie ausgeführt wird. Durch diese Zuordnung wird implizit die Parallelisierbarkeit von Operationen erfasst. In der Abhängigkeitssektion kann diese Parallelität für nicht-orthogonale Befehlssätze noch weiter eingeschränkt werden.

Jede Operationsdeklaration wird durch das Schlüsselwort *DefineOp* eingeleitet. Nichtterminale, die innerhalb der Befehlssatzspezifikation verwendet werden, jedoch keine eigenständigen Operationen darstellen, werden mittels *OpNT* deklariert. Jeder Operation und jedem Nichtterminal werden ein eindeutiger Name und eine Assemblerrepräsentation zugeordnet. Anschließend werden innerhalb von drei Attributgruppen die wesentlichen Eigenschaften der Operationen spezifiziert. Das folgende Beispiel zeigt eine Operationsdeklaration.

Quelltext 2.22: TDL: Beispiel einer Befehlssatzspezifikation

```
DefineOp IAdd "%!(optguard) %s = %s + %s"
    { dst1="$2" in {gpr}, src2="$3" in {gpr}, src3="$4" in {gpr} },
    { ALU(exectime=1, latency=1); },
    {
        extern unsigned<1> gval;
        if (((guarded=true)&&(gval<0>=1))||(guarded=false))
            { dst1:=src2+src3;}
    };

OpNT optguard
    "if %s"
        {src1="$1" in {gpr}, guarded=true},
        {;},
        {unsigned<1> gval; gval:=src1<0>;}
    | "if !%s"
        {src1="$1" in {gpr}, guarded=true},
        {;},
        {unsigned<1> gval; gval:=!src1<0>;}
    | ""
        {guarded=false},
        {;},
        {};
```

In der ersten Attributgruppe wird die Menge der Quell- und Zieloperanden einer Operation spezifiziert. In diesem Beispiel wird mit *dst1* ein Zielregister der Registermenge *gpr* angegeben. `$2` bezieht sich auf den Assemblersyntax und verweist in diesem Beispiel auf den zweiten %-Formatstring.

Für häufig auftretende Operationskomponenten können Nichtterminale mit eigenen Produktionen eingeführt werden. Diese können in der Assemblerrepräsentation durch %!...)) referenziert werden. In Quelltext 2.22 wird auf diese Weise ein optionaler Guard *optguard* spezifiziert. An der Position %!optguard)) kann einer der Zeichenfolgen „if r0)"', ..., "'\lstinlineif r31)", „if !r0", ..., „if !r31" oder aber die leere Zeichenkette auftreten. Im letzten Fall ist die Ausführung der Operation nicht durch einen Guard geschützt, was durch die Attributssetzung guard=false dargestellt wird.

Innerhalb der zweiten Attributgruppe wird die Befehlsausführung durch einen Reservation-Table Mechanismus beschrieben. Jeder Operation wird eine Menge von Alternativressourcen (i.a. funktionalen Einheiten) zugeordnet, auf der sie ausgeführt werden kann.

Die letzte Attributgruppe beschreibt die detaillierte Semantik der definierten Operation. Diese wird durch eine erweiterte Registertransfersprache spezifiziert. Die Semantik einer Operation wird als Änderung des Maschinenzustandes beschrieben, der durch die Ausführung der Operation verursacht wird. In Quelltext 2.22 wird die Summe zweier Zweierkomplementzahlen berechnet, falls kein Guard existiert oder das erste Bit des als Guard verwendeten Registers den Wert 0 bei „f %s)"' bzw. 1 bei "'\lstinlineif !

Constraintspezifikation In der Constraintspezifikation kann eine Menge von Nebenbedingungen spezifiziert werden, die bei der Codetransformation berücksichtigt werden müssen, um Korrektheit zu bewahren. Dies umfasst z.B. Einschränkungen von Parallelverarbeitungskapazitäten oder Abhängigkeiten zwischen Befehlsanordnung oder Registerverteilung.

Bei dem digitalen Signalprozessor, Analog Devices ADSP-2106x, ist z.B. die Parallelausführung einer ALU- und einer Multiplizierer-Operation nur möglich, falls alle Operationen in bestimmten Registern liegen. Andernfalls liegen zwar gültige Operationen vor, diese können aber nicht parallel ausgeführt werden. Dies kann in TDL wie folgt spezifiziert werden:

Quelltext 2.23: TDL: Beispiel einer Constraintspezifikation

```
(op1 in {AluOps} & op2 in {MulOps}):
    (op1 && op2) -> (
        (op1.src1 in {GroupA}) &
        (op1.src2 in {GroupB}) &
        (op2.src1 in {GroupC}) &
        (op2.src2 in {GroupD})
    );
```

Assemblerspezifikation Die Assemblerspezifikation befasst sich mit der Gesamtsyntax, der als Eingabe verwendeten Assemblersprache, wie z.B. Befehlsbegrenzer,

Kommentare, Makros und Direktiven.

2.3.3.9 xADL

xADL [57, 58, 59] ist eine strukturorientierte ADL und wurde von Florian Brandner innerhalb seiner Promotion bis 2009 an der Vienna University of Technology entwickelt. Die Entwicklung der Sprach hat als eine einfache Konfigurationssprache für stark geclusterte und parallele Signalprozessoren begonnen. Mittlerweile ist xADL eine voll ausgereifte ADL, die eine komplette Prozessorimplementierung mit dem Assemblersyntax, der Binärkodierung, der Hardwarestruktur und dem *Binärschnittstelle* (ABI, engl. *Application Binary Interface*) abdecken kann. Die Besonderheit der ADL ist, dass sie nur strukturorientiert ist und Befehlssatz mittels Befehlssatzextraktion generiert wird. Die Befehlssatzextraktion wird dabei durch einfache Regeln durchgeführt und kann von dem Prozessordesigner aktiv kontrolliert werden. Dadurch wird das Problem der redundanten Spezifikation von gemischten Architekturbeschreibungssprachen umgangen. Eine Beschreibung des Prozessors wird durch eine einzelne Spezifikation realisiert, die sowohl eine detaillierte Sicht der Mikroarchitektur als auch eine abstrakte Sicht der Befehlssatzarchitektur abdeckt. Im Gegensatz zu früheren strukturorientierten ADLs findet die Beschreibung auf einer höheren Abstraktionsebene statt, wodurch erst eine effiziente Befehlssatzextrakation ermögliche wird, aber auch die Flexibilität der Beschreibung auf bekannte Elemente beschränkt.

xADL erlaubt die Beschreibung von Prozessoren, wie traditionell gepipelinete CISC-, RISC- und VLIW-Prozessoren. Theoretisch werden sogar superskalare Prozessoren mit out-of-order Ausführung unterstützt, so lange die Befehle innerhalb der Befehlsreihenfolge angestoßen werden. Die Sprach kann für Compiler-Backend-Generierung, Befehlssatzsimulation und Hardwaregenerierung eingesetzt werden. Wie moderne Beschreibungssprachen basiert xADL auf einer Auszeichnungssprache XML [60], wodurch die Sprach auch ihren Namen zu verdanken hat. Abbildung 2.15 zeigt eine Beispielnetzliste von einer einfachen 4-stufigen Prozessorarchitektur, wie sie in xADL modelliert wurde.

Eine strukturelle Beschreibung in xADL besteht aus einer Menge von Komponenten (engl. *components*) verbunden durch Datenleitungen (engl. *Data Links*). Komponenten können entweder Hardware (wie z.B. Register, Caches, Speicher und funktionelle Einheiten) oder Abstraktionen (wie z.B. Immediates oder Konstanten) repräsentieren. Hardwarekomponenten sind Anschlüsse (engl. *Ports*) zugeordnet, an denen Datenleitungen angeschlossen werden können. Immediates und Konstanten können direkt an Datenleitungen gelegt werden können. Komponenten werden über vorher definierte Typen instanziiert, wodurch die Wiederverwendbarkeit und Austauschbarkeit von Komponenten für verschiedene Architekturbeschreibungen gewährleistet wird. Quelltext 2.24 zeigt eine Beispiel einer Typdefinition R_t von einem Registerfile und der Instanziierung dessen. Der Anschluss `Rd_hi` greift nur auf die obere Hälfte der Register zu und Register 0 ist fest auf 0 kodiert.

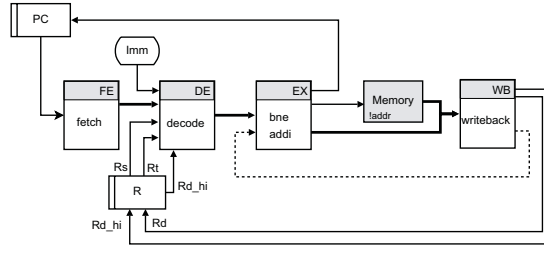


Abbildung 2.15: xADL: Netzliste einer Prozessorarchitektur mit 4 Pipelinestufen Quelle [58]

Quelltext 2.24: Beispiel einer Deklaration eines Registerfiles mit acht 32-Bit Registern.

```
<!-- define register type R_t -->
<RegisterType name="R_t" width="32" repeatcount="8">
    <Port name="Rs" wri teab l e="false" />
    <Port name="Rt" wri teab l e="false" />
    <Port name="Rd" readable="false" />
    <Port name="Rd_hi" offset="16" width="16" />

    <Constant index="0" value="0" />
</RegisterType>

<!-- define a concrete registerfile -->
<Register name="R" type="R_t" category="integer base index" />
```

Abbildung 2.16 zeigt den xADL-Toolflow zur Umsetzung der Retargierbarkeit. Als Frontend wird ein xADL-Prozessormodell geparst und eine sogenannte *web*-Datenstruktur aufgebaut, die die strukturelle Beschreibung des Prozessors in einem Graphen widerspiegelt. Dafür werden die Typen und Instanzen als auch deren Verbindungen aufgelöst und auf Konsistenz überprüft, d.h. Namenskonflikte, passende Bitbreiten, fehlende Definitionen, etc. erkannt. Danach wird die Befehlssatzarchitektur aus der strukturellen Beschreibung extrahiert. Dazu werden zunächst die verschiedenen Module zu Pipelinestufen zugeordnet. Dabei wird funktionellen Einheiten und Immediates als Ganzes einer Pipelinestufe zugeordnet werden. Registerports und Ports zu speicherende Elementen können allerdings in verschiedenen Pipelinestufen verwendet werden. Der Befehlssatz kann dann extrahiert werden, indem zunächst alle Instruktionspfade durch eine Tiefensuche aufgestellt werden. Ein Instruktionspfad repräsentiert einen möglichen Weg, den Instruktionen bei der Abarbeitung nehmen können. Für jeden Instruktionspfad werden dann alle möglichen Kombinationen von Operationen durchgegangen und somit der Befehlssatz extrahiert.

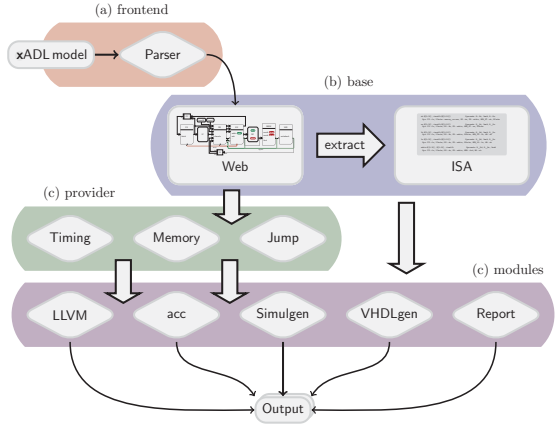


Abbildung 2.16: xADL: Übersicht des xADL-Toolflows
Quelle [59]

Provider stellen optionale Analyseschritte dar, die nicht immer sondern je nach Verwendung der ADL durchgeführt werden. *timing* analysiert den Befehlssatz und extrahiert genaue Timing-Information zu jeder Operation. *memory* analysiert sind Speicherzugriffspatern und Adressmodi von Load/Store-Instruktionen. *jump* analysiert das Sprungverhalten von Instruktionen, d.h. ob, wann und wie eine Instruktion den Befehlszeiger (engl. *Instruction Pointer*) verändert.

Module stellen dann das Backend des xADL-Toolflow dar, das die web Datenstruktur und die extrahierten Zusatzinformationen je nach Verwendungsart in Quellcode, Testfälle, Diagramme, Dokumentationen, usw. überführen. Folgende Generatormodule werden im xADL-Toolflow genauer beschrieben:

Simulgen erzeugt automatisch einen taktgenauen Simulator der Prozessorpipeline. [61, 62]

VHDLgen ist ein Prototyp eines VHDL-Generators, der große Teile einer synthetisierbaren VHLD-Beschreibung eines Prozessormodells erzeugen kann.

LLVM ist ein Compilergenerator für den LLVM-Compiler, der automatisch die Registerfilebeschreibung, ein Befehlssatzmodell, ein Ressourcenmodell fürs Scheduling, Baumstrukturen für die Befehlsauswahl und Glue-Code für die Retargierung generiert.

acc ist ein weiterer Compilergenerator für das proprietäre Compilerbackend acc (siehe Abschnitt 2.4.2.2). [57, 58, 59, 63]

2.3.3.10 ArchC

ArchC ist eine Open-Source-Architekturbeschreibungssprache, die von 2003 bis 2007 an der Universidade Estadual de Campinas in Brasilien entwickelt wurde. ArchC basiert auf der Modellierungs- und Simulationssprache SystemC. SystemC wiederum basiert auf der Programmiersprache C++ und erweitert diese mittels einer Klassenbibliothek, um Hardwaresystem modellieren und simulieren zu können. Mittels Makros wird in C++ ein eigener SystemC-Syntax definiert, der die SystemC-C++-Klassen versteckt. Basierend auf dem SystemC-Syntax-Stil erlaubt ArchC eine einfache Beschreibung von Prozessorarchitekturen als auch Speicherhierarchien. ArchC wurde mit dem Ziel entwickelt neue Architekturen durch die Generierung von Assemblern, Simulatoren, Linkern und Debuggern zu explorieren und verifizieren.

Für ArchC sind die Prozessormodelle für IBM/Motorola PowerPC [64], MIPS-I [65], SPARC-V8 [66], ARM [67], Intel 8051 [68] und der PIC 16F84 [69] Befehlssatzarchitektur frei verfügbar. Des Weiteren wurden Prozessormodelle der Architekturen Motorola 68k/ColdFire [70], Altera Nios [71], OpenCores OR1K [72], Hitachi SH-4, Intel XScale und TMS320C62x [73] erstellt. Zudem wurde ArchC für die Erforschung des 2D-VLIW-Architekturprinzips [74, 75] verwendet.

Eine ArchC-Architekturbeschreibung ist in zwei Bereiche unterteilt:

Befehlssatzarchitektur (AC_ISA) In AC_ISA werden Details über die Befehlssatzarchitektur beschreiben. Dazu zählt das Verhalten eines Befehls, Befehlsformat, -größe und -namen als auch sämtliche Information, die zum Dekodieren eines Befehls notwendig sind.

Architekturressourcen (AC_ARCH) Die AC_ARCH-Beschreibung hingegen beinhaltet eine strukturelle Beschreibung der Speicherelemente, Prozessorpipeline, usw.

Basierend auf beiden Beschreibungen ist ArchC in der Lage einen interpretierten Simulator, einen kompilierenden Simulator und einen Assembler zu generieren.

2.3.4 Systembeschreibung in ADLs

2.3.4.1 Distributed Operation Layer (DOL)

Distributed Operation Layer (DOL) [76, 77, 78] ist ein Framework für das semi-automatische Mapping von Applikationen auf MPSoC-Systeme. DOL wurde im Rahmen des *Scalable Hardware/Software Architecture Platform for Embedded Systems* (SHAPES) EU-Projekts von der *Eidgenössische Technische Hochschule Zürich* (ETH Zürich) entwickelt und wird aktuell im Rahmen des *European Reference Tiled Architecture Experiment* (EURETILE) EU-Projekts weiterentwickelt.

Das DOL-Framework besteht hauptsächlich aus vier Teilen, die in Abbildung 2.17 dar-

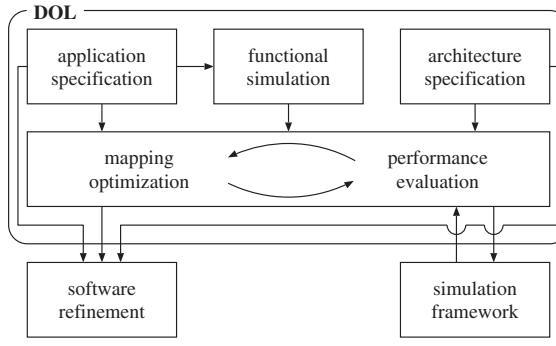


Abbildung 2.17: Übersicht über das Distribued Operation Layer Framework (Quelle [78])

gestellt sind:

Application Specification DOL definiert ein eigenes Programmiermodell für Applikationen, das die Grundlage für das automatisch Mapping der Applikationen auf parallele Hardware bietet. Das DOL-Programmiermodell verwendet dabei das Konzept der Process Network Models.

Architecture Specification DOL definiert die Spezifikation der MPSoC-Zielarchitekturen für das Framework.

Functional Simulation Ein rein funktioneller Simulator kann automatisch aus der Applikationsspezifikation generiert werden. Die Simulation kann dabei zum Debuggen, Testen und zur Extraktion relevanter Mappingparameter verwendet werden.

Mapping Optimization DOL implementiert ein Werkzeug zur Optimierung von Mappings basierend auf einem eng gekoppelten Werkzeug zur Performanzanalyse.

Als Ausgabe erzeugt das Framework ein Mapping, das zusammen mit der Applikation und der Architekturbeschreibung als Eingabe für weitere Optimierung der Anwendung genutzt werden kann.

Die Architekturspezifikation verwendet dabei das XML-Format und ist auf die Beschreibung von regulären MPSoC-Strukturen optimiert. Sie beinhaltet alle relevanten Informationen über die Zielarchitektur, die für das Mapping auf Systemebene notwendig sind. Daher spezifiziert die DOL-ADL nicht sämtliche Details des Systems sondern verwendet ein abstraktes Model.

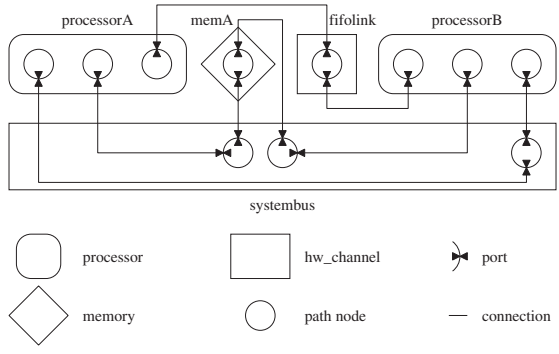


Abbildung 2.18: Beispiel einer Architekturspezifikation in der DOL ADL (Quelle [78])

Die DOL-ADL besteht aus strukturellen, Performanz- und parametrische Daten. Auf struktureller Ebene werden die Plattformressourcen wie Prozessoren, Speicher, Hardwarekanäle und ihre Verbindung, beschrieben, wie in Abbildung 2.18 beispielhaft dargestellt. Als Besonderheit ist zu erwähnen, dass für die Kommunikation bereits fertige Kommunikationspfade und ihrer verwendeten Ressourcen spezifiziert werden anstelle des strukturellen Aufbau des Kommunikationsnetzwerks, der einen höheren Freiheitsgrad in der Kommunikation bedeuten würde. Dieser Ansatz ermöglicht die wirklichkeitsgetreue Beschreibung von *Networks-on-Chips* (NoCs) mit statischem Routing.

Die Performanzdaten annotieren dann die strukturellen Daten durch z.B. den Durchsatz von Bussen, Verzögerung von Kommunikationskanälen, Prozessor- und Busfrequenzen und den Overhead der hardwareabhängigen Softwareschicht (z.B. Gerätetreiber oder Ressourcenverwaltungsmechanismen). Ebenso werden parametrische Daten wie Speichergröße, unterstützte Ressourcenverwaltungsmechanismen (z.B. FIFO, feste Prioritäten, statischen Scheduling oder zeitgesteuerte Architekturen) und Betriebssystemparameter an die Elemente annotiert.

2.3.4.2 Machine Markup Language (MAML)

Machine Markup Language (MAML) [79, 80, 81, 82] ist eine Architekturbeschreibungssprache für die Modellierung, Simulation und Architektur/Compiler-Cogenerierung von domänenspezifischen Prozessorarchitekturen, die für Spezialanwendungen entwickelt wurden. MAML basiert auf XML und erlaubt die Charakterisierung von Ressourcen einer Prozessorarchitektur sowohl auf der strukturellen als auch Verhaltensebene. MAML wurde ursprünglich im Rahmen des BUILDABONG-Projekts

2002 an der Universität Paderborn für das Design von ASIPs entwickelt, bietet heute allerdings die Möglichkeit zu Beschreibung von Multicore- und sogar Manycore-Systemen. Eine MAML-Beschreibung kann auf der eine Seite zur Generierung von schnellen taktgenauen Simulatoren verwendet werden, bietet auf der anderen Seite auch die Möglichkeit zur Generierung von Compilern. Zusätzlich wird ein bibliotheksbasierter Ansatz zur Synthese eines MAML-Prozessors unterstützt. MAML wird insbesondere zur Beschreibung eines grobgranularen Prozessorarrays namens *Weakly Programmable Processor Arrays* (WPPA) [83] verwendet.

Eine MAML-Beschreibung charakterisiert ein Multiprozessorsystem auf zwei Abstraktionsebenen:

1. Auf der *Prozessor-Ebene* (PE-Level, engl. *Processing Element Level*) wird die interne Struktur von verarbeitenden Elementen beschrieben. Diese Beschreibungsebene wurde im Allgemeinen in der Sektion 2.3.3 tiefgreifend behandelt und wird daher gehend nicht näher betrachtet.
2. Auf der Array-Ebene werden systemweite Parameter festgelegt, wie die Topologie des Array sowie die Anzahl und Platzierung von Prozessoren und I/O-Schnittstellen. Diese Ebene wird im Folgenden näher betrachtet.

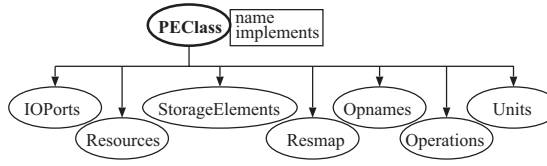


Abbildung 2.19: Struktur des PEClass-Elements
(Quelle [81])

Zur Beschreibung von Mehrprozessorsystemen setzt MAML auf die Beschreibung von Typen von *verarbeitenden Elementen* (PE, engl. *Processing Elements*) auf, sog. PEClass-Elementen. Eine PEClass kann auf der Array-Ebene beliebig häufig instanziiert werden. Zur Beschreibung von heterogenen Systemen können verschiedene PEClass-Elemente beschrieben werden, die sogar Vererbung zur effiziente Beschreibung unterstützt . In Abbildung 2.19 wird die Struktur der PEClass-Elemente gezeigt. Sie beinhaltet die Spezifikation der I/O-Ports (Bitbreite, Richtung, usw.), interne Ressourcen (z.B. interne Ports, Funktionelle Einheiten, Busse, usw.), Speicherelemente (z.B. Daten- oder Kontrollregister, lokale Speicher, Instruktionsspeicher, FIFOs, Registerdateien), Ressourcenmapping (Verbindung von Ports mit internen Elementen) und funktionelle Einheiten (Ressourcenverwendung, Pipelining, usw.).

Ein Mehrprozessorsystem wird innerhalb der Array-Ebene mit dem sog. Processor-Array-Elementen beschrieben. Der Aufbau eines ProcessorArray-Elements ist in Abbildung 2.20 dargestellt. Es beschreibt im Allgemeinen den Aufbau des gesamten Pro-

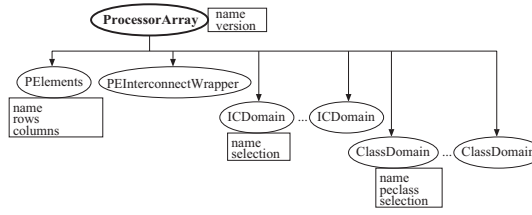


Abbildung 2.20: Struktur des ProcessorArray-Elements
(Quelle [81])

zessorarrays. Die Beschreibung umfasst insbesondere die Verbindungstopologie sowie die Anzahl und Typen von Prozessorkernen.

Im Unterelement PEElements werden sämtliche verarbeitenden Elemente des Prozessorarrays definiert. Die Anzahl der Elemente wird dabei durch ein zweidimensionales Gitter mit einer festen Anzahl von Spalten und Zeilen festgelegt. Die Anzahl der Prozessoren muss nicht mit der Anzahl der Elemente des zweidimensionalen Arrays zusammenhängen. Das Gitter dient nur als Basis, um verschiedenen Arten von Prozessor-, Speicher- oder I/O-Elementen zu platzieren.

Zur kompakten Beschreibung von großen MPSoCs unterstützt MAML die sog. parametrisierbaren Domains (engl. *Parametric Domains*). Diese erlauben die rekursionsfreie Beschreibung einer Menge von verarbeitenden Elementen mit einer homogenen internen Struktur oder einer gemeinsamen regulären Verbindungstopologie. In MAML werden zwei Arten von parametrisierbaren Domains unterschieden: die *Verbindungsdomain* (ICDomain, engl. *Interconnect Domain*) und die *Klassendomain* (ClassDomain). Von jeder Domain können beliebig viele in einem ProcessorArray-Element spezifiziert werden.

Die Verbindungsdomain erlaubt die Beschreibung von regulären Verbindungstopologien auf einer Untermenge verarbeitender Elemente des Grids. Als Verbindungstypologien können zwischen verschiedenen vorgefertigten, parametrierbaren Topologien (wie z.B. Baum, Ring, vermaschtes Netz (engl. *Mesh*), Honigwaben und Torus) ausgewählt werden. Die Elemente der Domain kann über eine Art Polytop auf dem Grids definiert werden. Zunächst wird ähnlich wie bei der Formulierung bei *Lineare Programmierung* (LP, engl. *Linear Programming*) eine Menge von Ungleichen definiert, die ein Polytop auf dem Grid aufspannen. Zusätzlich kann in MAML dieses Polytop noch skaliert und verschoben werden.

Eine Klassendomain erlaubt die Beschreibung von gleichartigen Typen für eine Untermenge verarbeitender Elemente des Grids. Genauso wie bei der Verbindungsdomain wird die Untermenge durch ein Polytop festgelegt. Der Type der Elemente wird durch eine Referenz einer vorher (außerhalb des ProcessorArray) definierten PEClass festge-

legt.

2.3.4.3 ArchC

In Abschnitt 2.3.3.10 wurde die ArchC-Architekturbeschreibungssprache vorgestellt, die auf der SystemC-Modellierungs- und Simulationssprache basiert. In [84] wurde die ArchC-Sprache um die Möglichkeit zur Beschreibung von MPSoCs erweitert. Dabei wurde auf der Beschreibung von einzelnen Prozessoren in ArchC aufgesetzt.

Die Systembeschreibung innerhalb von ArchC wird dabei über eine AC_SYSTEM-Sektion durchgeführt. Diese ist in zwei Bereiche unterteilt. Im ersten Teil werden Instanzen von Plattformkomponenten instanziiert. Dies umfasst Prozessoren, Busse sowie Devices. Bei Prozessoren wird auf ein vorher definierte ArchC-Prozessorbeschreibung verwiesen. Devices werden verwendet um z.B. Speicher zu instanziiert. Busse repäsentieren Verbindungsstrukturen, die durch ihren Typ (z.B. Seriell), Daten-Bitbreite und Adress-Bitbreite beschrieben werden.

Im zweiten Teil werden das System mit Hilfe der instanziierten Plattformkomponenten aufgebaut. Dabei werden die Komponenten durch ein spezieller Statement miteinander verbunden. Zusätzlich kann das Verhalten verschiedener Komponenten festgelegt werden. So kann bei Slave-Teilnehmern eines Busses der Adressbereich festgelegt werden. Bei Prozessoren kann Binärcode der Anwendung festgelegt werden, die auf diesem ausgeführt werden soll. Im zweiten Teil werden das System mit Hilfe der instanziierten Plattformkomponenten aufgebaut. Dabei werden die Komponenten durch ein spezieller Statement miteinander verbunden. Zusätzlich kann das Verhalten verschiedener Komponenten festgelegt werden. So kann bei Slave-Teilnehmern eines Busses der Adressbereich festgelegt werden. Bei Prozessoren kann Binärcode der Anwendung festgelegt werden, die auf diesem ausgeführt werden soll.

Damit man in der AC_SYSTEM-Sektion die Komponenten miteinander verbinden kann, müssen das Interface einer Komponente im Vorfeld beschrieben sein. Das Interface einer Komponenten wird durch Ports festgelegt, wobei die verfügbaren Ports mittels des ac_protocol-Statements beschrieben werden. Ein Protokoll ist dabei durch fünf Parameter festgelegt: (1) der Protokolltyp, (2) die Richtung, d.h. Master oder Slave, (3) der Portname, (4) die Bitbreite der Adressleitung und (5) die Anzahl an Ports diese Typs. Der Portname wird dann zur Referenzierung des Ports beim Verbinden verwendet.

2.3.5 Zusammenfassung

Der Begriff *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*) wird für die Beschreibung von Software- und Hardwarearchitekturen gleichermaßen verwendet. Innerhalb dieser Arbeit wird der Begriff allerdings nur für

Hardware-Architekturbeschreibungssprachen für Prozessoren und Mehrprozessorsystemen verwendet.

Architekturbeschreibungssprachen können nach Inhalt und Ziel klassifiziert werden. Bei der inhaltsbasierten Klassifizierung wird zunächst festgelegt, ob eine ADL einen Prozessor oder ein System beschreibt. Bei prozessorbeschreibenden ADLs wurde zusätzlich zwischen strukturorientierten, verhaltensorientierten oder gemischten ADLs unterschieden. Bei der zielbasierten Klassifizierung wird entschieden, für was eine ADL eingesetzt wird. Hierbei wird zwischen kompilierungsorientierten, simulationsorientierten, syntheseorientierten und validierungsorientierten ADLs unterschieden. Bei der Simulation ist zusätzlich die Abstraktionsebene relevant, ob dieser funktional, taktgenau oder auf RTL-Ebene stattfindet.

Architekturbeschreibungssprachen zur Beschreibung von Prozessoren wurden in der Vergangenheit intensiv erforscht. Dabei waren sie früher stärker auf einzelne Ziele fokussiert als heutige ADLs. Frühere ADLs konnten eher als strukturell oder verhaltensorientiert klassifiziert werden. So waren z.B. die strukturellen ADLs MIMOLA 2.3.3.1 oder UDL/I 2.3.3.2 stark an Hardwarebeschreibungssprache angelehnt, wodurch eine Hardware-Generierung und -Synthese der beschriebenen Prozessorarchitektur ermöglicht wurden. Versuche den Befehlssatz aus der strukturellen Beschreibung z.B. für die Compilergenerierung zu extrahieren waren unter Einschränkungen möglich. Rein verhaltensorientierte ADLs wie z.B. ISDL oder nML sind zwar gut für die Compilergenerierung geeignet, sind allerdings schlecht für die Generierung von Hardware oder taktgenauen Simulatoren verwendbar. Daher hat sich bei späteren ADLs, wie z.B. LISA oder EXPRESSION, ein gemischter Ansatz durchgesetzt, bei dem die Sprache sowohl die Struktur als auch das Verhalten der Architektur abdeckt.

Architekturbeschreibungssprachen zur Beschreibung von Systemen setzen häufig auf der Beschreibung von Prozessoren auf. Dabei wird die Struktur von Systemen ähnlich zu Hardwarebeschreibungssprachen modelliert, wobei die Systembeschreibung in ADLs im Vergleich sehr abstrakt gehalten ist. Bei Prozessoren ist das Verhalten nicht mehr Teil der Systembeschreibung, die häufig auf eine Beschreibung innerhalb der Prozessorbeschreibung der ADL verweist. Aktuelle Systembeschreibungen werden für flexible Simulatoren oder für das Mapping von parallelen Anwendungen auf Multiprozessorsystemen verwendet. Hierbei sind allerdings mit stärkere Verbreitung von MPSoCs weitere Einsatzgebiete zu erwarten. Auf Systemebene bietet sich eine ADL gut für die Entwurfsraumexploration an, bei der insbesondere die Anzahl und Typ von Prozessoren sowie der Aufbau der Speicherhierachie

Im Allgemeinen lässt sich sagen, dass der Inhalt einer ADL stark von der Verwendung dieser abhängt. ADL-Beschreibungen werden häufig für retargierbare Compiler, Assembler, funktionale Simulatoren, taktgenaue Simulatoren und Hardware-Generierung eingesetzt. Dabei ist der Inhalt einer ADLs immer stark auf die verwendeten Werkzeuge optimiert. Dies liegt darin begründet, dass die Hauptkomplexität bei der Entwicklung der Werkzeuge und nicht im Design einer ADL liegt. Bei einem neuen Werkzeug ist es meistens einfacher eine neue maßgeschneiderte ADL zu ent-

wickeln anstatt eine bereits existierende wiederzuverwenden. Da die meisten ADL-Werkzeuge nicht frei verfügbar sind, müssen diese häufig neu entwickelt werden und darin ist auch die schiere Anzahl an existierenden ADLs begründet.

Zusätzlich ist der Inhalt einer ADL noch stark von der Zielarchitektur abhängig. So macht es z.B. für den Compiler einen großen Unterschied, ob man Code für eine CISC-, RISC-, VLIW- oder Clustered-VLIW-Befehlssatzarchitektur erzeugen möchte. Auch Features innerhalb der Befehlssatzarchitektur wie Registerfenster, heterogene Registersätze, Realisierung von Kontrollfluss, Predication oder SIMD-Instruktionen müssen von den verschiedenen ADL-Werkzeugen unterstützt und innerhalb der ADL beschrieben werden können. Somit lässt sich erst nach einer Beschreibung einer Befehlssatzarchitektur und/oder Mikroarchitektur sagen, ob und wie gut diese von den ADL-Werkzeugen unterstützt werden können. Aus diesem Grund werden ADLs auch häufig im Umfeld von applikationsspezifischen Prozessoren verwendet, bei denen der Entwurfs von den Fähigkeiten der ADL und ADL-Werkzeuge vorgegeben wird und man innerhalb dieses Entwurfsraums nach eine effiziente applikationsspezifische Lösung sucht.

2.4 Compiler

2.4.1 Definition

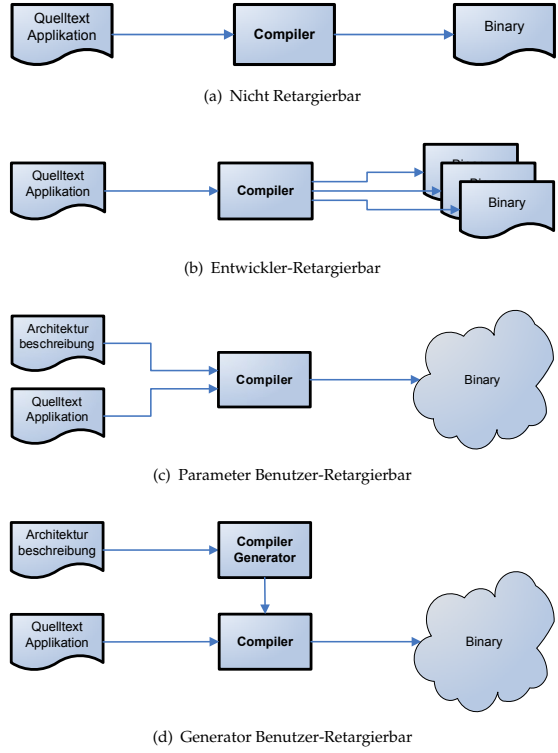
Im Allgemeinen ist ein Compiler ein Computerprogramm, das den Quellcode in einer Programmiersprache in den Quellcode (Quellsprache) einer anderen Programmiersprache (Zielsprache) übersetzt. Falls nicht anders angegeben, wird innerhalb dieser Arbeit unter den Namen ein Programm verstanden, dass Quellcode von einer Höhere Programmiersprache (wie C/C++) in eine Assemblersprache übersetzt.

2.4.2 Retargierbare Compilierung

2.4.2.1 Definition

Ein retargierbarer Compiler ist ein Compiler, der so entworfen wurde, dass er vergleichsweise einfach verändert werden kann, um Code für eine andere Befehlssatzarchitektur zu erzeugen. Wie in Abbildung 2.21 dargestellt, lassen sich retargierbare Compiler in zwei Kategorieren einteilen: entwickler-retargierbare (engl. *Developer-Retargetable*) und benutzer-retargierbare (engl. *User-Retargetable*) Compiler [85].

Entwickler-Retargierbar Bei einem entwickler-retargierbarer Compiler muss ein Entwickler den Quellcode des Compilers anpassen, um eine neue Befehlssatzarchitektur zu unterstützen. Der Quellcode des Compilers ist dabei schon so strukturiert, dass möglichst viel Teile des Compilers zwischen unterschiedlichen Be-



(a) Nicht Retargierbar

(b) Entwickler-Retargierbar

(c) Parameter Benutzer-Retargierbar

(d) Generator Benutzer-Retargierbar

Abbildung 2.21: Die verschiedenen Arten von retargierbaren Compilern

fehlssatzarchitekturen wiederverwendet werden kann. Ein Benutzer des Compilers kann hier nur zwischen den bereits unterstützten Befehlssatzarchitekturen wählen. Zwei bekannte Beispiele dieser Kategorie sind der GCC Compiler [86] und LLVM Compiler [87], die bereits für eine Vielzahl von Architekturen portiert wurden.

Benutzer-Retargierbar Im Gegensatz dazu benötigt ein benutzer-retargierbarer Compiler zur Unterstützung einer neuen ISA keinen Eingriff eines Entwicklers in dessen Quellcode. Dazu wird die Befehlssatzarchitektur in einer *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*) (siehe Ab-

schnitt 2.3) spezifiziert. In der ADL werden für den Compiler alle benötigten Information zur Unterstützung der neuen Befehlssatzarchitektur gespeichert.

Generator Benutzer-Retargierbar Hierbei wird ein Generator verwendet, um den eigentlichen Compiler aus der ADL Beschreibung zu erzeugen. Typischerweise wird zur Generierung des Compilers mittels Metaprogrammierung Teile des Compiler-Quellcodes automatisch erzeugt, der dann auf die Befehlssatzarchitektur abgestimmt ist. Anschließend wird der Quellcode danach kompiliert und somit der Compiler erzeugt. Dies ist z.B. beim Compiler der xADL ADL (siehe Abschnitt 2.4.2.2) der Fall.

Parameter Benutzer-Retargierbar Es gibt allerdings auch Compiler, bei denen die ADL als zusätzlicher Parameter bei jedem Übersetzungsvorgang angegeben ist. Der Vorteil liegt in der schnelleren Retargierung dieser Compiler, weil der Compiler nicht extra generiert werden muss. Beispiele sind hier der VEX (siehe Abschnitt 2.4.2.3) Compiler.

2.4.2.2 xADL

Abbildung 2.22 zeigt die Hauptkomponenten des retargierbaren xADL-Compilers. Als Frontend wird eine modifizierte Version des GCC-Compilers [86] verwendet, das die Zwischendarstellung von GCC in eine XML-Datei exportiert und in eine eigene High-Level Zwischendarstellung überführt. In der Lowering-Phase werden von dieser dann die High-Level-Konstrukte (Arrayreferenzen, Funktionsaufrufe und Variablen) in eine architekturunabhängige Low-Level-Form umgewandelt. Zusätzlich werden Funktionsaufrufe und globale Symbole gemäß dem ABI spezifiziert in der ADL umgeschrieben. Die Befehlsauswahl wurde über eine modifiziert Version lburg [88, 89], einem Tree-Pattern-Matching-Generator, realisiert. Virtuelle Register werden durch physikalische Register mittels einer erweiterten Graph-Coloring-Registerzuteilung [90] ersetzt. Jeweils vor und nach der Registerzuteilung wird ein List-Scheduler zur Umsortierung der Instruktionen verwendet um die Anzahl der Stall-Zyklen zu reduzieren. Zur Behandlung von Struktur- und Datenkonflikten im Scheduler wird der Operationstabellenansatz aus [91, 92] verwendet.

2.4.2.3 VEX

VLIW Example (VEX) [93, 23] ist eine Compiler- und Simulator-Toolchain und wird momentan von den Hewlett-Packard Laboratories entwickelt. VEX unterstützt eine breite Klasse von Clustered-VLIW-Prozessorarchitekturen nach dem NUAL-Prinzip. VEX enthält einen Parameter benutzer-retargierbarer C-Compiler. Dieser ist eine Abwandlung des Lx/ST200-C-Compilers, der wiederum aus dem Multiflow-C-Compiler [94] entstanden ist.

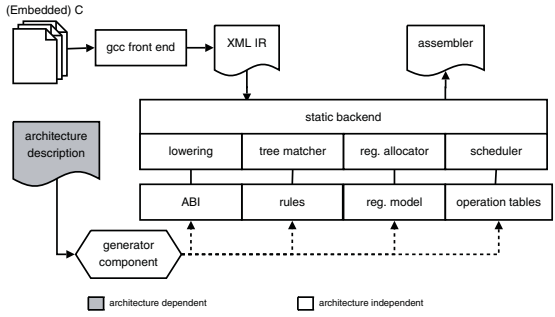


Abbildung 2.22: xADL: Übersicht der Hauptkomponenten des Compiler Backends Quelle [58]

Bei dem VEX-C-Compiler kann die Zielarchitektur mittels einer Datei beim Kompilieren angegeben werden. Diese ADL ist im Vergleich zu anderen ADLs vergleichsweise einfach aufgebaut. Darin kann man die Größe von Maschinenressourcen festlegen (z.B. Anzahl an Multiplizierer, Issue-Slots, Cluster), das Delay von Operationen und die Anzahl an Registern. Die VEX-ISA der angegebenen Parameter in der ADL flexibel gehalten und kann bezüglich neuer Operationen erweitert werden. Allerdings ist der Basisbefehlssatz der ISA fest vorgegeben und es ist nicht möglich die Struktur oder Aufbau der ISA darüber hinaus zu beeinflussen.

2.4.2.4 Trimaran

Die Trimaran-Compilerinfrastruktur [95] ist ein integrierter Forschungscompiler- und Performanzmonitoring-Infrastruktur. Abbildung 2.23 zeigt die drei Komponenten von Trimaran: (1) den OpenIMPACT-Compiler als C-Frontend, (2) den Elcor-Compiler [47] als Backend und (3) den Simu-Simulator zur Performanzevaluierung. Trimaran unterstützt als Zielsprache die ARM-ISA sowie die parametrisierbare HDL-PD-ISA [96], die clustered-VLIW, EPIC und superskalare Architekturen abdeckt. Es wird unter anderem für die Erforschung von Instruction-Level Parallelism, Compiler-Optimierungen, Technologien zur Codegenerierung und retargierbare Kompilierung eingesetzt. Trimaran ist ein benutzer-retargierbare Softwareframework und verwendet zur Retargierung der ISA, des Backends und des Simulators die HMDES-Architekturbeschreibungssprache (siehe Abschnitt 2.3.3.5).

Abbildung 2.24 zeigt den Aufbau des Elcor-Compilers. Im Elcor-Compiler wird jede Funktion nacheinander verarbeitet. Im folgenden werden die relevanten Compiler-Passes beschrieben:

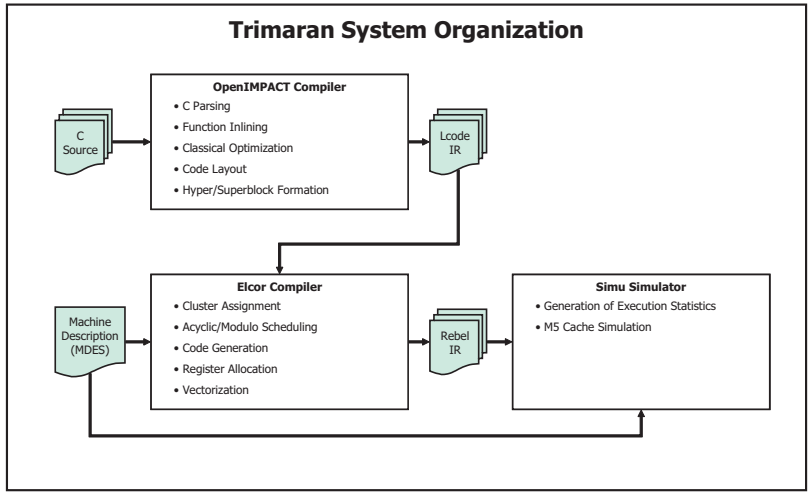


Abbildung 2.23: Trimaran: System Organization
Quelle [95]

Classic optimizations Klassische Optimierungen wie Eliminierung von gemeinsamen Unterausdrücken (engl. *common subexpression elimination*), Propagieren von Konstanten (engl. *constant propagation*), Entfernung von nichtverwendeten Code (engl. *Dead code elimination*), etc. werden durchgeführt.

Initial code generation Operanden werden in ein passendes Format der Zielarchitektur konvertiert.

Clustering (Prepass) Bei Architekturen mit geclusterten Registerfiles werden die Operationen nach Clustern partitioniert und Interclusterkommunikation eingefügt.

Scheduling (Prepass) Der Code wird je Region mit Hilfe des Skalaren- oder Modulo-Schedulers gescheduled. Der Module-Scheduler wird für Softwarepipelining von Schleifen verwendet.

Register allocation Physikalische Register werden für die Operanden festgelegt und Spill-Code wird eingefügt.

Clustering (Postpass) Operationen von neu hinzugefügtem Spill-Code werden Clustern zugewiesen.

Scheduling (Postpass) Der Spill-Code wird gescheduled.

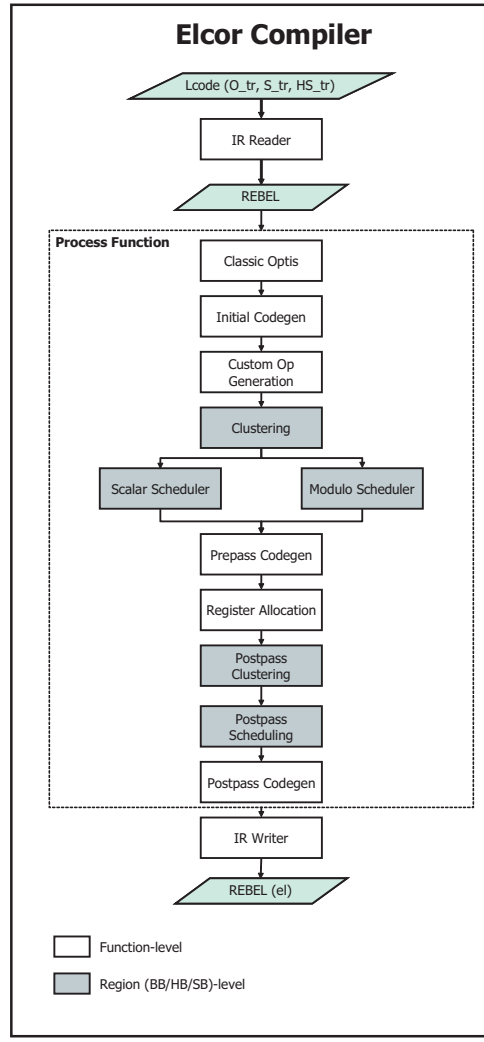


Abbildung 2.24: Trimaran: Elcor Compiler Organisation
Quelle [95]

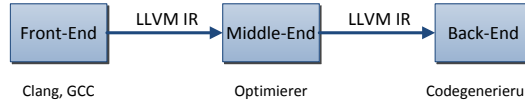


Abbildung 2.25: Struktur eines Compilers der LLVM-Compilerinfrastruktur

Zur Unterstützung von geclusterten Architekturen verwendet der Elcor-Compiler einen entkoppelten Ansatz, bei dem das Clustering vor dem Scheduling durchgeführt wird. Zur Lösung des Phase-Ordering-Problems zwischen dem Clustering/Scheduling und der Registerzuteilung kommt ein Postpass-Clustering und -Scheduling zum Einsatz, dass den neu hinzugefügten Spill-Code der Registerzuteilung sowohl Clustern zuweist als auch einplant.

2.4.3 LLVM

Das *Low-Level Virtual Machine* (LLVM)-Projekt [87] ist eine Kollektion von modularen, wiederverwendbaren Compiler- und Toolchain-Technologien. LLVM hat wenig mit traditionellen virtuellen Maschinen zu tun, wie man aus dem Namen vermuten könnte.

LLVM startete als ein Forschungsprojekt der Universität von Illinois mit dem Ziel, moderne SSA-basierte Compilierungsstrategien zur Verfügung zu stellen, die sowohl statische als auch dynamische Übersetzung von unterschiedlichen Programmiersprachen unterstützen. Seitdem hat sich LLVM zu einem Dachprojekt entwickelt, das aus unterschiedlichen Subprojekten besteht, die sowohl in Produktivsystemen von Firmen wie auch in Forschungsprojekten eingesetzt wird. Im folgenden werden die wichtigsten LLVM-Projekte genannt:

LLVM Core Den Kern von LLVM bilden eine Reihe von Bibliotheken, die moderene quell- und zielsprachenunabhängige Optimierungen zusammen mit Codegenerierung für viele gängige Prozessoren und ISAs bereitstellen. Diese Bibliotheken sind um eine wohldefinierte Zwischendarstellung, der *LLVM-Zwischendarstellung* (LLVM-IR, engl. *LLVM Intermediate Representation*) gebaut.

Clang ist ein LLVM-Frontend für C, C++ und Objective-C. In Kombination mit dem Optimierer und Codegenerator der LLVM-Core-Bibliotheken bildet Clang einen vollwertigen Compiler für die von LLVM unterstützen Prozessoren.

dragonegg kombiniert den LLVM-Optimierer und -Codegenerator mit dem GCC 4.5 Frontend. Dadurch kann LLVM, zusätzlich zu C, C++ und Objective-C, Ada, Fortran und jede weitere von GCC unterstützte Programmiersprache verarbeiten. Zusätzlich werden C-Features, wie z.b. OpenMP, die noch nicht in Clang

integriert sind, unterstützt werden.

libc++ stellt eine standardkonforme and hochperformante Implementierung der C++ Standardbibliothek bereit.

compiler-rt stellt eine Implementierung von low-level Unterstütztungfunktionen für den Codegenerator bereit. Dies Funktionen werden aufgerufen, wenn eine Zielarchitektur bestimmte Operationen nicht unterstützt. Darunter fällt z.b. eine Emulation sämtlicher floating-point Operationen.

polly implementiert eine Reihe von Optimierungen zur Cachelokalität sowie Autoparallelisierung und -vektorisierung unter der Verwendung eines polyedrischen Modells.

2.4.3.1 LLVM-Struktur

Abbildung 2.25 zeigt die Struktur eines Compilers, der mit den Komponenten der LLVM-Compilerinfrastruktur aufgebaut werden kann:

Frontend Als Frontend steht entweder Clang oder das Frontend des GCC-Compilers (dragonegg) zur Verfügung. Das Frontend übersetzt die Eingangssprache in die LLVM-IR, die unabhängig von der Eingangssprache ist. Dadurch ist es möglich, dass unterschiedliche Frontends innerhalb des LLVM-Compilers verwendet werden.

Middleend Das Middleend führt optional plattformunabhängig Optimierungen des Quellcodes innerhalb der LLVM-IR durch. Zu den Optimierungen zählen unter anderem Dead Global Elimination, Constant Propagation, Dead Argument Elimination, Inlining, Reassociation, Loop Optimizations, Memory Promotion, Dead Store Elimination etc.

Backend Das Backend übersetzt die LLVM-IR in den plattformabhängigen Maschinen- oder Assemblercode. Es ist dabei abhängig von der jeweiligen Zielarchitektur. Durch die Verwendung eines anderen Backends, kann Code für eine andere Zielarchitektur generiert werden. LLVM enthält unter anderem Backends für X86, AMD64, Sparc 32/64, ARM, PowerPC und Alpha Befehlssatzarchitekturen.

2.4.3.2 LLVM-Zwischendarstellung

Die *LLVM-Zwischendarstellung* (LLVM-IR, engl. *LLVM Intermediate Representation*) [97] beschreibt den Quellcode durch eine sprachunabhängige, RISC-ähnlichen Befehlssatz in *Static-Single-Assignment-Darstellung* (SSA, engl. *Static Single Assignment Form*) [98], die Typ- und Datenflussinformationen enthält. Die SSA-Darstellung zeichnet sich dadurch aus, dass jede Variable statisch nur einmal zugewiesen wird. Da-

durch werden Datenabhängigkeiten zwischen Befehlen explizit dargestellt, was Compileroptimierungen im Middle- und Backend vorteilhaft ist. Bei der Erzeugung der SSA-Darstellung werden die ursprüngliche Variablen in Versionen gesplittet, wenn dieses mehrfach geschrieben werden. Da so in alternativen Zweigen verschiedene Versionen einer Variablen geschrieben werden, muss nach der Vereinigung des Kontrollflusses (z.B. nach einem if/then/else) das Problem gelöst werden, dass späterer Code nur auf eine Variable zugreifen kann. Gelöst wird das Problem über die Phi-Operation, die die richtige Version einer Variable abhängig vom tatsächlich genommenen Kontrollfluss als Ergebnis zurückgibt.

Die LLVM-IR wurde kann in drei Formen auftreten, die jeweils die gleichen Daten repräsentieren können.

Speicherdarstellung Bei der Repräsentation im Speicher des Compilers werden die Daten mittels Klassen repräsentiert und können schnell durch den Compiler abgefragt und bearbeitet werden.

LLVM-Bitcode Bei der Bitcode-Repräsentation kann die LLVM-IR als Binärdatei auf der Festplatte gespeichert werden. Dabei ist die Repräsentation für ein schnelles Laden und Speichern sowie eine niedrige Codegröße optimiert.

LLVM-Assemblersprache Bei der Repräsentation in einer Assemblersprache kann die LLVM-IR als Textdatei auf der Festplatte gespeichert werden. Die LLVM-Assemblersprache repräsentiert eine menschenlesbare Form der LLVM-IR, die mit Hilfe eines Texteditors sowohl gelesen als auch verändert werden kann.

Obwohl die LLVM-IR designt wurde um unabhängig von der Eingangssprache zu sein, ist eine Repräsentation in der LLVM-IR nicht unabhängig von der Zielarchitektur. Bereits im Frontend muss die Zielarchitektur bei der C/C++-Programmiersprache bekannt sein, da diese von Natur aus abhängig von der Zielarchitektur ist. So reflektiert der fundamentale Datentyp „int“ in C für gewöhnlich die Breite der Allzweckregister der Zielarchitektur. Dies beeinflusst insbesondere vordefinierte Präprozessor-Makros im Frontend, wie z.B. *INT_MAX*. Genauso wird die Bytereihenfolge der Architektur durch ein Präprozessor-Makro definiert und wird benötigt um portablen Code zu erzeugen. Bei der Vorverarbeitung im Präprozessor wird portabler Code dann unwiederbringlich abhängig von der Zielarchitektur.

Zu diesem Zweck kann innerhalb der LLVM-IR des Datenlayout der Zielarchitektur gespeichert werden. Dieses enthält neben Bytereihenfolge die Ausrichtung der Stacks, von Pointer, Integer, Vektor, Floating-Point und zusammengesetzte Datentypen.

2.4.3.3 LLVM-Backend

Das LLVM-Compiler-Backend ist ein Modul im LLVM-Compilerframework. Die Synthese – also die Codeerzeugung – im Backend ist in verschiedene Durchläufe (engl. *Pass*) unterteilt, wie z.B. die Befehlsauswahl (engl. *Instruction Selection*) oder Register-

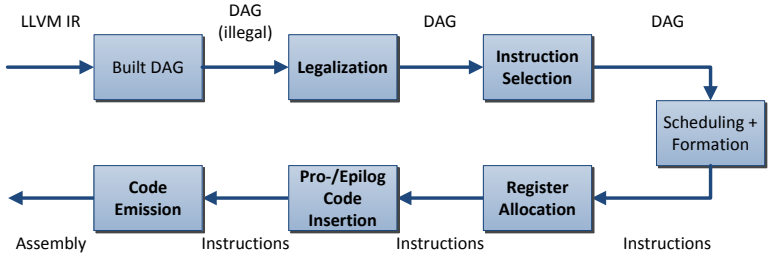


Abbildung 2.26: Synthese im LLVM-Backend

zuteilung (engl. *Register Allocation*). Jeder Durchlauf kann wieder verschiedene Phasen enthalten, der bestimmt Modifikationen am internen Status durchführen. Jede dieser Phasen besteht aus einem architekturunabhängigen Algorithmus, der im Hintergrund den Ablauf bestimmt. Dabei werden verschiedene Callback-Funktionen aufgerufen, die die architekturabhängigen Teile des Durchlaufes beinhalten. Auf diese Weise sind sowohl die Zielarchitektur als auch der Algorithmus leicht austauschbar und können sogar über die Kommandozeile ausgewählt werden. Des weiteren ist es möglich, zusätzliche Durchläufe in die Synthese einzufügen. So wird z.B. im SPARC-Backend ein zusätzlicher Durchlauf für das Füllen der Delay Slots nach einem Verzweigungsbefehl durchgeführt. Abbildung 2.26 zeigt die wichtigsten Schritte der Synthese im LLVM-Backend.

Der Quellcode eines gewöhnlichen LLVM-Backends basiert auf sog. *Target Description* (td) Dateien. Die td-Dateien abstrahieren die Architekturbeschreibung innerhalb der Codeerzeugung mit dem Zweck die Entwicklung im Backend signifikant zu vereinfachen. Die td-Dateien enthalten unter anderem eine Beschreibung des Registersatzes, Befehlssatzes und der Aufrufskonvention (engl. *Calling Convention*) der Zielarchitektur. Während der Kompilierung des LLVM-Compilers parst das sog. TableGen Werkzeug die td-Dateien und erzeugt daraus C++ Quellcodedateien, die dann innerhalb verschiedenen Durchläufe in der Synthese verwendet werden.

Built DAG Der erste Schritt der Synthese besteht aus der Erzeugung eines *gerichteten azyklischen Graphen* (DAG, engl. *Directed Acyclic Graph*), bestehend aus architekturunabhängigen LLVM-Befehlen. Als Eingabe dient der sequentielle Eingangsstrom der LLVM-IR.

Legalization In der Legalisierungsphase (engl. *Legalization*) wird der illegale DAG in einen legalen überführt. Legal bedeutet in diesem Fall, dass keine Befehle und Da-

tentypen, die von der Zielarchitektur nicht unterstützt werden, enthalten sind. Dies geschieht durch Eliminierung der nicht unterstützten Datentypen und Befehle und kann durch verschiedene Arten geschehen, die für jeden Datentyp und Befehl angegeben werden müssen.

Eliminierung von Datentypen Bei der Eliminierung von Datentypen kommen zwei Methoden zum Einsatz:

Promotion ersetzt kleinere Datentypen durch größere. So wird z.B. ein acht-Bit Integer-Datentyp durch einen 32-Bit ersetzt. Dabei muss der DAG bei Integer noch um zusätzliche Signed- und Zero-Extension Befehle ergänzt werden.

Expansion ersetzt größere Datentypen durch kleinere. So kann ein 64-Bit Register durch zwei 32-Bit Register emuliert werden. Eine Addition wird dann z.B. durch zwei Additionen mit Carry-Übertrag ersetzt.

Eliminierung von Befehlen Die Eliminierung von Befehlen ist weitaus schwieriger als die Eliminierung von Datentypen. Dies liegt daran, dass man für bestimmte Befehle eine manuelle Behandlung durchführen muss. Für jeden Befehl stehen drei Behandlungsmethoden zur Auswahl:

Promotion ersetzt Befehle, die auf kleineren Datentypen arbeiten durch die gleichen Befehle, welche auf größeren Datentypen arbeiten. So kann z.B. eine acht-Bit Multiplikation durch eine 32-Bit Multiplikation ersetzt werden, wenn die Architektur erstere nicht kennt, aber trotzdem acht-Bit Register zur Verfügung stellt.

Expansion ersetzt Befehle durch alternative Sequenzen von Befehlen oder Funktionsaufrufe. Dies findet im Hintergrund von LLVM statt. Man hat keinen Einfluss darauf, durch welche Sequenz die Befehle ersetzt werden.

Custom beschreibt, dass der Befehl durch eine spezifische Behandlung legalisiert wird. Es wird für jeden Befehl eine Callback-Funktion aufgerufen, in der der DAG verändert werden kann. Der Befehl an sich muss nicht notwendigerweise eliminiert werden, sondern es kann an dieser Stelle der DAG auf beliebige Art und Weise verändert werden, um die Befehlsauswahl in der nächsten Phase zu unterstützen.

Instruction Selection In der Befehlsauswahlphase (engl. *Instruction Selection*) wird der DAG, bestehend aus zielarchitekturunabhängigen Befehlen, in einen DAG, bestehend aus zielarchitekturabhängige Befehlen, umgewandelt. Dabei werden zuerst alle Möglichkeiten, einzelne oder mehrere Befehle umzuwandeln (manche zielunabhängigen Befehle können mehrere zielabhängigen entsprechen, wie z.B. *Multiply and Accumulate* (MAC)), ermittelt. Danach wird eine Graphpaarung (engl. *Graph Matching*) durchgeführt, um jedem zielunabhängigen Befehl genau einem zielabhängigen Befehl zuordnen zu können.

Für die Befehlsauswahl müssen dem Algorithmus sämtliche Befehle der Zielarchitektur angegebenen werden. Dabei wird das Verhalten des Befehls mit einem Teilgraphen, bestehend aus zielunabhängigen Befehlen, beschrieben. Zusätzlich können Regeln angeben werden, die jeweils einen zielunabhängigen Befehl in mehrere zielabhängige Befehle umwandeln.

Die Modellierung der Informationen im LLVM-Backend erfolgt mittels so genannter *Target Description* (.td)-Dateien. Diese haben einen Regelsatz für die Auswahl der Befehle als Eingabe und generieren daraus automatisch C++ Dateien, die eine Anwendung des Regelsatzes darstellen. Ein manuelles Schreiben der Regeln in C++ ist zwar auch möglich, aber die Target Description Dateien stellen eine Vereinfachung für den Entwickler dar. So besteht ein Vorteil darin, dass automatisch die Kommutativität und Assoziativität in den Teilgraphen erkannt und alle möglichen Kombinationen automatisch erzeugt werden.

Folgendes Beispiel zeigt die Definition einer Addition. Der interne Name des Befehls lautet **ADDrr**. *ops* gibt an, dass die drei Parameter *dst*, *src1* und *src2* vom Typ **Int32Regs** sind und somit GPR-Register darstellen. Danach wird die Assemblersyntax angegeben. In den $[(\ldots)]$ Klammern wird der Teilgraph spezifiziert. Er sagt aus, dass zuerst *src1* und *src2* miteinander addiert werden und das Ergebnis danach in *dst* geschrieben wird.

Quelltext 2.25: Beispiel einer Befehlsbeschreibung im LLVM-Backend

```
def ADDrr : Inst<
    (ops Int32Regs:$dst, Int32Regs:$src1, Int32Regs:$src2),
    "add $dst, $src1, $src2",
    [(set Int32Regs:$dst, (add Int32Regs:$src1, Int32Regs:$src2))]
>;
```

In dem nachfolgenden Beispiel wird eine Regel definiert, die den Umgang mit 32-Bit Immediatewerten in einer Architektur beschreibt, in welcher der Wert nicht mit einem Befehl geladen werden kann. So wird der obere (*HI16*) und untere Teil (*LO16*) des Wertes als Eingabe verwendet. Der *LIS*-Befehl lädt die oberen 16 Bits und schiebt sie 16 Bits nach rechts. Danach wird der verschobene obere Bereich mit dem unteren Bereich verodert.

Quelltext 2.26: Beispiel einer Regel im LLVM-Backend

```
def : Pat<
    (i32 imm:$imm),
    (ORI (LIS (HI16 imm:$imm)), (LO16 imm:$imm))
>;
```

Scheduling + Formation In der Schedulingphase wird den Befehlen eine Reihenfolge zugeordnet. Es wird der DAG aus zielabhängigen Befehlen als Eingabe genommen und eine sequenzielle Liste aus Befehlen generiert. Bei der Schedulingphase können verschiedene Hardwareconstraints berücksichtigt werden. So kann man bei jedem Befehl angeben, von welchen Rechenwerken (engl. *Function Units*) der Befehl ausgeführt werden kann und wie lange seine Ausführung in diesem Fall dauert.

Register Allocation Bei der Registerzuteilung (engl. *Register Allocation*) werden die virtuellen Register auf die physikalischen Register des Prozessors abgebildet. Zu diesem Zweck wird eine Lebenszeitanalyse der virtuellen Register vorgenommen, um herauszufinden, über welche Distanz der Inhalt eines Registers gültig ist.

Bei der Registerzuteilung muss auf Register geachtet werden, die physikalisch den gleichen Platz belegen. So verwenden z.B. das EAX und das AL Register, in der x86 Architektur, gemeinsam die ersten acht Bits. Dies wird in Form von Alias-Registern berücksichtig, die nicht verwendet werden dürfen, so lange das Register gültige Werte besitzt.

Die Phase ergänzt den sequenziellen Befehlsstrom um drei Arten von Befehlen, die alle manuell hinzugefügt werden müssen:

1. Registertransferbefehle
2. Register von dem Stack laden
3. Register auf dem Stack speichern

LLVM bietet mehrere Registerzuteilungsalgorithmen zur Auswahl. Diese können über die Kommandozeile umgestellt werden:

Simple ist eine sehr einfache Implementierung, die keinen Wert in Registern über Befehle hinweg behält. Vor jedem Befehl werden direkt alle Register geladen und nach jedem Befehl direkt zurückgeschrieben.

Local ist eine Verbesserung des Simple-Registerzuteilungsalgorithmus. Er erzeugt, auf Basisblockebene, eine statische Zuordnung von virtuellen auf physikalische Register und versucht, falls möglich, Register wiederzuverwenden.

Linear Scan ist der Standardalgorithmus [99].

Prologue/Epilogue Insertion In der Prologue/Epilogue Einfügungsphase werden hauptsächlich zwei Aufgaben erledigt:

1. Prologue/Epilogue-Code

   Mit Prologue/Epilogue-Code werden Assemblerbefehle zu Beginn und am Ende einer Funktion bezeichnet, die z.B. den Stack-Pointer erhöhen/erniedrigen oder andere architekturtypische Befehle ausführen. In dieser Phase wird der

Code bei jeder Funktion hinzugefügt. Dabei sind Sonderfälle bei der Addition/-Subtraktion des Stack-Pointers zu beachten, wenn z.B. die Zahl nicht in ein Immediate hineinpasst.

2. Frame-Index Eliminierung

   Das Stack-Layout einer Funktion steht erst nach der Registerzuteilung fest. Deswegen ist erst zu diesem Zeitpunkt bekannt, an welcher relativen Adresse im Stack (Frame-Index) eine lokale Variable liegt. In diesem Schritt werden sämtliche Frame-Indices durch konkrete Immediatewerte ersetzt.

Code Emission In der letzten Codeemissionsphase wird der sequentielle Befehlsstrom in eine Assemblerdatei umgewandelt. Dabei kann man den grundlegenden Assemblersyntax global einstellen, z.B. wie Kommentare anfangen oder Register formatiert werden.

3 Kahrisma-Prozessorarchitektur

Dieses Kapitel beschreibt eine Ausprägung der Kahrisma-Prozessorarchitektur, wie sie im Rahmen der Dissertation von Ralf König [100] und des Kahrisma-DFG-Projekts [128] entstanden ist.

Das Kapitel ist wie folgt aufgebaut. Nach einer Motivation in Sektion 3.1 werden in Abschnitt 3.2 die Ziele der Architektur definiert. Zur Umsetzung der Ziele wird zunächst ein Konzept aufgestellt (Abschnitt 3.3) und dann die Realisierung getrennt nach Befehlssatzarchitektur (Abschnitt 3.4) und Mikroarchitektur (Abschnitt 3.5) beschrieben. Die Realisierung wird in Abschnitt 3.6 bezüglich der aufgestellten Ziele charakterisiert. Danach geht Abschnitt 3.7 auf die dadurch resultierenden Anforderungen des Softwareframeworks zur Gewährleistung der Programmierbarkeit ein. Abschnitt 3.8 fasst das Kapitel zusammen.

3.1 Motivation

Wie in Kapitel 1.1 beschrieben, sind die Power Wall und damit einhergehend das Ende des exponentiellen Wachstums der Taktfrequenz sowie die immer ineffizientere Ausnutzung von *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) durch zusätzliche Transistoren bei einem Prozessorkern die treibenden Faktoren zur Verwendung von Mehrkernprozessoren oder *Multi-Prozessor Systems on Chip* (MPSoCs). Insbesondere bei hochperformanten Anwendungen oder zur Verbesserung der Energieeffizienz führt heutzutage kein Weg an der Integration mehrere Prozessoren auf einem Chip mehr vorbei. Allerdings steigt nicht die Performanz jeder Applikation linear mit der Anzahl der Kerne an. Nur wenn eine Anwendung genügend *Parallelität auf Threadebene* (TLP, engl. *Thread-Level Parallelism*) besitzt, kann diese auch mehrere Kerne auslasten. Bestimmte Algorithmen können sogar nur einen Prozessor sinnvoll verwenden, wenn z.B. bei einem Verschlüsselungsalgorithmus jeder Schritt direkt vom vorherigen Schritt abhängt. Allgemein kann man sagen, dass bei einer niedrigen Parallelität auf Threadebene die Performanz stärker von der Performanz eines einzelnen Prozessors anstatt deren Anzahl abhängt.

Die Performanz eines einzelnen Prozessors hängt hingegeben wiederrum von der Taktfrequenz sowie der zeitlichen und räumlichen parallelen Abarbeitung des Befehlsstroms ab. So enthalten heutige hochperformante superskalare *General-Purpose Prozessoren* (GPP) diversitäre, gepipelinte parallele Funktionseinheiten, die durch auf-

wendige Mikroarchitekturmechanismen wie Registerumbenennung, Sprungvorhersagetechniken, spekulativer Ausführung und out-of-order Ausführung möglichst effizient ausgelastet werden sollen. Im eingebetteten Bereich, in dem Chipfläche und Energieverbrauch wichtiger sind als z.B. Abwärtskompatibilität des Befehlssatzes, wird hingegeben die *Very Long Instruction Word* (VLIW) Technik eingesetzt, um sowohl zeitliche und räumliche Parallelität in einem Prozessorkern auszunutzen. Gemeinsam haben diese Techniken, dass die Performanzsteigerung durch zusätzliche Komplexität der Mikroarchitektur nicht linear mit der Anzahl der Transistoren steigt. So hat der Intel-Mitarbeiter Fred Pollack im Jahre 1999 die Faustregel aufgestellt, dass eine um Faktor n erhöhte Transistoranzahl mit einer Performanzsteigerung um $\sqrt{n}$ einhergeht. Insbesondere hängt die Performanzsteigerung durch die Mikroarchitektur auch stark von der Anwendung selbst ab. So ist die zeitliche Parallelität durch die Güte von Sprungvorhersagetechniken limitiert. Bei manchen Anwendungen mit Sprüngen, die von den Eingangsdaten abhängen, kann sogar die theoretische Trefferrate der Sprungvorhersage bei zufälligen Eingangsdaten auf 50% limitiert sein. Die Ausnutzung der räumlichen Parallelität ist ebenfalls applikationsabhängig und hängt mit der Unabhängigkeit der Befehle im Befehlsstrom zusammen, oder auch Parallelität auf Befehlsebenen genannt.

		Instruction-Level Parallelism (ILP)	
		Low	High
Thread-Level Parallelism (TLP)	Low	**Scalar RISC** e.g. Huffman decoder	**VLIW/Superscalar** e.g. Small DCT
	High	**Many scalar RISC** e.g. H.264 frame decode	**Some VLIW/Superscalar** **Many scalar RISC** e.g. Large DCT

Abbildung 3.1: Effizientes MPSoC-Layout abhängig vom TLP/ILP-Charakteristika der Algorithmen

Da sowohl die ausnutzbare Parallelität auf Befehls- als auch auf Threadebene von der jeweiligen Applikation abhängt, ist die Entscheidung, ob man bei einem Multiprozessorsystem die Transistoren eher in die Komplexität einzelner Kerne oder eher in die Anzahl der Kerne investiert, zur Designzeit nicht zu entscheiden. Abbildung 3.1 zeigt die jeweils effiziente Prozessorarchitektur einer Anwendung in Abhängigkeit ihrer TLP/ILP-Charakteristika. Bei niedrigem TLP würde man als Zielplattform ein Einprozessorsystem bevorzugen, weil zusätzliche Kerne oder Prozessoren nicht ausgelastet werden könnten. Bei niedrigem ILP würde man in diesem Fall entweder einen skalaren Prozessoren mit wenigen Funktionseinheiten oder bei hohem ILP einen superskalaren oder VLIW-Prozessor mit viel räumlicher Parallelität bevorzugen. Wenn

allerdings von Anwendungsseite viel TLP zur Verfügung steht, würde man ein Mehrprozessorsystem bevorzugen. Bei niedrigem ILP würde man diese klar aus einfachen Prozessoren aufbauen. Wenn sowohl ausreichend ILP und TLP vorhanden ist, hat man die Wahl welche der Parallelitätsformen man bevorzugt ausnutzen möchte.

Die einzige Möglichkeit dieses Problem zu lösen, besteht darin, die Entscheidung über die Auslegung eines Multiprozessorsystems von der Designzeit in die Laufzeit zu verlagern. Nur dann ist es möglich, ein Mehrprozessorsystem zu entwickeln, dass für eine breite Masse an Anwendung effizient bezüglich Performanz, Energieverbrauch und Flächenbedarf ist.

3.2 Ziele

In diesem Kapitel wird daher eine Prozessorarchitektur vorgestellt, die insbesondere die vorgestellten Herausforderungen zukünftiger Mehrkernprozessoren adressiert. Dabei ist es möglich die Anzahl und Komplexität der Prozessoren im MPSoC durch Rekonfiguration dynamisch anzupassen. Mit dieser Möglichkeit ist es nicht mehr notwendig, einen Tradeoff zur Designzeit einer Mehrkernprozessorarchitektur zwischen Prozessoranzahl und -komplexität einzugehen, sondern diese Entscheidung kann flexibel zur Laufzeit getroffen werden. Bei dem Design und der Entwicklung der Prozessorarchitektur waren folgende Ziele maßgeblich:

Adaptive Ausnutzung von ILP und TLP Klassische Multiprozessorarchitekturen bieten nur zur Designzeit eine festgelegte Menge an Parallelität auf Befehls- und Threadebene an und können somit Anwendung mit genau dieser Menge an TLP und ILP effizient ausführen. Anwendungen mit anderen TLP/ILP-Charakteristika werden allerdings ineffizient auf solchen Architektur ausgeführt. Bei der entwickelten Prozessorarchitektur soll sich die verfügbare Parallelität auf Befehls- und Threadebene so konfigurieren lassen, dass sich die Hardware auf unterschiedliche TLP/ILP-Charakteristika von unterschiedlichen Anwendungen adaptieren kann. Somit soll eine breitere Masse an Anwendungen effizient auf der Architektur ausgeführt werden können.

Partiell Dynamische Rekonfiguration von ILP und TLP Die Konfiguration von ILP und TLP innerhalb der Hardware soll dabei dynamisch und partiell veränderbar sein, d.h. dass zur Laufzeit eine Konfiguration veränderbar (dynamisch rekonfigurierbar) ist und dies getrennt (partiell) für unterschiedliche parallel laufende Anwendungen durchführt werden kann. Eine Anwendung besteht typischerweise aus verschiedenen Algorithmen, die jeweils unterschiedliche ILP/TLP Charakteristika aufweisen können. Durch eine Rekonfiguration während der Laufzeit einer Anwendung kann auf die unterschiedlichen Bedürfnisse dieser eingegangen werden. Insbesondere lässt sich damit der sequentielle Teil einer Anwendung durch ILP auf Befehlsebene parallelisieren, während der parallele Anwendungsteil mehr von vielen Prozessoren profitiert.

Skalierbarkeit Das Konzept der Rekonfiguration von ILP und TLP soll dabei sowohl funktionell als auch bezüglich der physikalischen Realisierung skalierbar sein. Funktionell soll das Konzept unabhängig von der maximalen Anzahl an verfügbaren Prozessorkernen in der Architektur funktionieren. Somit wird eine Architektur mit homogenen Strukturen angestrebt, die allerdings die Vorteile von heterogenen Strukturen bezüglich der Spezialisierung auf unterschiedlichen Anwendungen bietet. Somit werden die Vorteile von Homogenität mit den Vorteilen der Heterogenität mittels Rekonfiguration kombiniert.

Aus Sicht der physikalischen Realisierung soll das Prozessordesign in möglichst wenige gleichartige Module unterteilt werden, die dann im Rahmen der Backend Realisierung des Prozessors identisch ausgelegt werden. Durch die Wiederverwendung gleichartiger Module wird die Designkomplexität insbesondere bei großen Architekturen mit vielen Modulen reduziert. Die ist heutzutage ein übliches Mittel zur Beherrschung der enormen Anzahl an Transistoren und zur Schließung der Lücke zwischen dem exponentiellen Wachstums der Transistoren und der superlinearen Komplexität der Algorithmen zur Verwendung der Transistoren. Eine Unterteilung gleichartige Module hat allerdings den Nachteil, dass nur innerhalb der Module der kritische Pfad während der Synthese optimiert werden kann und sich somit verhältnismäßig lange Signallaufzeiten zwischen den Modulen ergeben. Dadurch muss bereits im Design des Prozessors die Realisierung in Module partitioniert und dabei berücksichtigt werden, welche Funktionalität innerhalb eines Moduls und welche über die Modulgrenzen hinaus realisiert werden muss.

Programmierbarkeit Ein sehr wichtiges Designkriterium ist die Gewährleistung der Programmierbarkeit der Architektur durch eine High-Level Programmiersprache. Bei vielen grobgranular rekonfigurierbaren Architekturen wird dieses Kriterium entweder ganz vernachlässigt oder die Programmierbarkeit ist nur durch Hardwarebeschreibungssprachen gegeben. Dies führt häufig zu Architekturen, die zwar effizient handoptimierte Kernels ausführen können, aber verhältnismäßig aufwändig zu programmieren sind und somit in der Industrie nur eine untergeordnete Rolle spielen.

In diesem Fall steht die Programmierbarkeit von Anfang an im Vordergrund und wurde bei der Entwicklung der Architektur berücksichtigt. Aus diesem Hintergrund war auch klar, dass die Architektur eine sequentielle Befehlsverarbeitung nach den Prinzipien der Harvard- oder Von-Neumann-Architektur gewährleistet und somit auch ein Compiler für die Architektur ermöglicht wird. Insbesondere kann somit die Rekonfiguration als eine Erweiterung heutiger MPSoCs verstanden werden. Hierbei spiegelt sich der erhöhte Freiheitsgrad der Rekonfiguration auch in der Programmierung wieder und erhöht somit die Softwarekomplexität, allerdings wird in dieser Promotionsarbeit gezeigt, dass die Programmierbarkeit der neuen Architektur in gleichem Maße wie bei herkömmlichen MPSoCs gewährleistet werden kann.

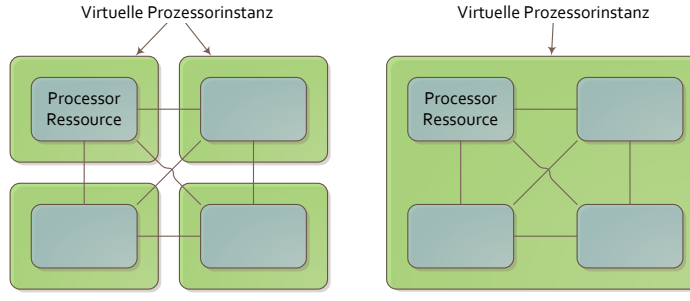
Interruptfähigkeit Die Architektur folgt nicht nur einem sequentiellen Prozessormodell um die Programmierbarkeit zu gewährleisten, sondern das sequentielle Prozessormodell soll auch Interrupts und Exceptions unter der Berücksichtigung der Bedingung von präzisen Interrupts (vgl. Kapitel 2.1.10) unterstützten. Interrupts bzw. Exceptions sind die Grundvoraussetzung um Betriebssysteme auf der Architektur ausführen zu können. Somit soll die Architektur nicht nur für Signalverarbeitung sondern auch für komplexe Steuer- und Kontrollaufgabe einsetzbar sein.

Designzeit-Flexibilität Die Architektur soll nicht eine statische Lösung für ein Architekturdesign darstellen, sondern soll möglichst flexibel zur Designzeit gehalten werden. So soll die Architektur weitestgehend parametrierbar sein um z.B. auf Systemebene die Anzahl, Anordnung und Verschaltung der unterschiedlichen Module festlegen zu können. Auch auf Prozessorebene ist eine Parametrisierung der Mikroarchitektur z.B. bezüglich der Anzahl an Registern wünschenswert. Zusätzlich soll der Befehlssatz sowohl von der Kodierung als auch von der Mikroarchitektur erweiterbar gestaltet werden, so dass später mit vergleichsweise geringem Aufwand neue – ggf. anwendungsspezifische – Befehle in die Prozessormodule integriert werden können. Mit dieser Flexibilität zur Designzeit wird die Erforschung und Validierung von verschiedenen Designparametern und sogar eine automatisch *Entwurfsraumexploration* (DSE, engl. *Design-Space Exploration*) ermöglicht. Diese Flexibilität muss dann allerdings auch von dem jeweiligen Softwareframework unterstützt werden, dass in dieser Arbeit vorgestellt wird.

3.3 Konzept

3.3.1 Konzept auf Systemebene

In diesem Kapitel wird eine neue innovative Prozessorarchitektur vorgestellt, die insbesondere die vorgestellten Probleme von Mehrkernprozessoren adressiert. Mittels einer rekonfigurierbaren Prozessorpipeline kann diese zur Laufzeit virtuelle Prozessorinstanzen erzeugen, die aus einer variablen Anzahl an Ressourcen bestehen. Abhängig von den Anforderungen können die Konfigurationen (die virtuellen Prozessorinstanzen) den Bedürfnissen der Anwendung angepasst werden. So kann z.B. durch Rekonfiguration eine hohe Anzahl an virtuellen Prozessorinstanzen mit wenigen funktionellen Einheiten je Prozessor instanziiert werden. Genauso sind allerdings auch Konfigurationen mit weniger virtuellen Prozessorinstanzen dafür aber mit vielen funktionellen Einheiten je Prozessor möglich. Mit dieser Möglichkeit ist es nicht mehr notwendig, einen Tradeoff zur Designzeit einer Mehrkernprozessorarchitektur zwischen Prozessoranzahl und -komplexität einzugehen, sondern diese Entscheidung kann flexibel zur Laufzeit getroffen werden.



(a) Konfiguration mit 4 virtuellen Prozessorinstanzen (b) Konfiguration mit 1 virtuellen Prozessorinstanz

Abbildung 3.2: Konzept der Kahrisma-Prozessorarchitektur

Abbildung 3.2 visualisiert das neuartige Konzept an einem Architekturbeispiel bestehend aus 4 gleichartigen Prozessorressourcen. Diese Prozessorressourcen können zu unterschiedlichen virtuellen Prozessorinstanzen konfiguriert werden. In Abbildung 3.2(b) ist z.B. nur eine virtuelle Prozessorinstanz konfiguriert, bestehend aus 4 Ressourcen, während die gleichen Ressourcen auch zu 4 virtuellen Prozessorinstanzen, bestehend aus jeweils einer Ressource, konfiguriert werden können. Im Vergleich zu statischen MPSoC Architekturen kann eine Prozessorressource als ein Prozessor angesehen werden, der bei diesem Ansatz mit benachbarten Prozessoren zusammengeschaltet werden kann. Somit ist ein Prozessor bei diesem Konzept nicht mehr fest einer Ressource zugeordnet sondern muss erst konfiguriert werden und kann sich über mehrere Ressourcen erstrecken. Dazu sind sämtliche benachbarte Prozessorressourcen mit einem dedizierten Punkt-zu-Punkt Kommunikationsnetzwerk verbunden, das das Zusammenschalten der Ressourcen ermöglicht und während der Ausführung die Prozessorressourcen synchronisiert.

Abbildung 3.3 zeigt eine Architektur, bestehend aus 16 Prozessorressourcen, mit einer Vielzahl von unterschiedlichen virtuellen Prozessorinstanzen. Eine virtuelle Prozessorinstanz wird von der größte durch das Verbindungsnetzwerk eingeschränkt. Als Bedingungen müssen sämtliche Ressourcen einer virtuelle Prozessorinstanz paarweise verbunden sein, also sie müssen eine Clique bilden. Bei der gezeigten Topologie können somit maximal 4 benachbarte Ressourcen zu einer virtuellen Prozessorinstanz unterschiedlicher Formen zusammengeschaltet werden.

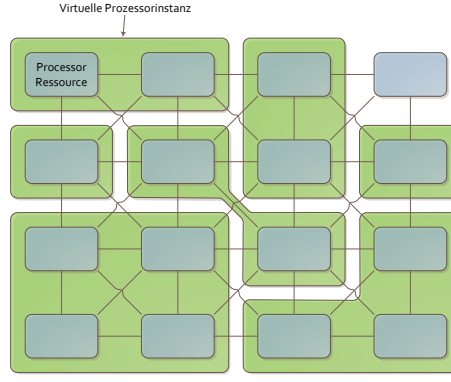


Abbildung 3.3: Prozessorarchitektur mit unterschiedlichen Konfigurationen

3.3.2 Konzept auf Prozessorebene

Das Neuartige an der Prozessorarchitektur ist die Möglichkeit, zur Laufzeit virtuelle Prozessorinstanzen bestehend aus einer unterschiedlichen Anzahl an Ressourcen erzeugen zu können. Je mehr Ressourcen für eine virtuelle Prozessorinstanz verwendet werden, desto höher ist ihre maximale Performanz aber genauso steigt auch ihr Energieverbrauch. Auf der Prozessorebene stand daher das Problem des Entwurfs einer modulbasierten, partitionierten Prozessorpipeline im Vordergrund, die die Erhöhung der parallelen Funktionseinheiten zur Befehlsabarbeitung mittels Rekonfiguration erlaubt. Dabei muss die Prozessorpipeline in verschiedene Module gesplittet werden, die zur Laufzeit dynamisch kombiniert werden können, um eine flexible Anzahl an funktionellen Einheiten einer virtuellen Prozessorinstanz zuordnen zu können. Zur Performanzwahrung müssen dann eng gekoppelte Pipelinestufen innerhalb eines Moduls realisiert werden. Ein Modul muss dann eine bestimmte Anzahl an funktionellen Einheiten besitzen, so dass deren Anzahl durch die Kombination mehrere Module erhöht werden kann. Weiterhin sind funktionelle Einheiten immer eng mit dem Registerspeicher und der Forwarding Logik verknüpft, so dass diese in einer sinnvollen Partitionierung ebenfalls über die Module manuell partitioniert werden müssen.

In der Literatur gibt es zwei dominierende Ansätze, die die räumliche Parallelität in der Hardware und die Parallelität auf Befehlsebene ausnutzt um durch gleichzeitige Abarbeitung unabhängiger Befehle die Performanz der Anwendung zu steigern. Im Folgenden werden diese Möglichkeiten unter dem Aspekt der Partitionierbarkeit in Module und der dynamischen Steigerung der räumliche Parallelität analysiert.

Superskalarität mit dynamischen Scheduling In Kapitel 2.2.7 wurde das Prinzip von superskalaren Prozessoren mit dynamischen Scheduling bzw. out-of-order Ausführung erklärt. Dabei wird zur Laufzeit der Befehlsstrom auf Abhängigkeiten untersucht. Ein Dispatcher teilt die Befehle dann unterschiedlichen Reservation Stations zu, die den funktionellen Einheiten vorgeschaltet sind. Sobald ein Befehl keine Abhängigkeiten mehr besitzt, wird dieser außerhalb der Befehlsreihenfolge bei der jeweiligen funktionellen Einheit zur Ausführung angestoßen. Die funktionellen Einheiten können bei diesem Prinzip relativ einfach in Module partitioniert werden. Allerdings ist hier das Problem die Partitionierung des Registerspeichers. Der Dispatcher muss dann die einzelnen Befehle nicht nur anhand ihres Typs den funktionellen Einheiten zuweisen, sondern muss auch auf die Lokalität der Eingangsdaten achten. Ein Kopieren der Daten zwischen den Modulen ist sowohl wegen der langen Leitungen als auch wegen der Forwarding Logik mit zusätzlichen Verzögerungen verbunden, die sich negativ auf die Performanz auswirken. In der Praxis hat sich gezeigt, dass einen effiziente Verteilung der Operationen auf Module sehr aufwändig ist und somit mit einem hohen zusätzlichen Ressourcenbedarf einhergeht.

VLIW-Konzept In Kapitel 2.2.8 wurde das VLIW-Prinzip vorgestellt, das im Gegensatz zur Superskalarität mit dynamischen Scheduling die Parallelisierung von der Hardware in den Compiler verlagert. Allgemein gilt beim VLIW-Prinzip möglichst viele Berechnungen von der Laufzeit in die Compilezeit zu verlagern und diese Berechnungen in der Befehlssatzarchitektur zu kodieren. Beim VLIW Prinzip ist eine Partitionierung des Registerspeichers vergleichsweise einfach möglich, da man die komplexe Berechnung, wann man Daten zwischen den Registerspeichern kopieren muss, effizient im Compiler vorberechnen kann. Allerdings gestaltet sich eine modulbasierte Partitionierung der funktionellen Einheiten wegen der synchronisierten Abarbeitung schwierig. So müssen z.B. im Falle eines Cache-Misses sämtliche funktionelle Einheiten parallel angehalten werden oder bei einer Exception sämtliche Pipelines der funktionellen Einheiten gleichzeitig geflusht werden. Dies führt zu einem kritischen Pfad über die Modulgrenzen hinaus und somit zwangsläufig zu einer Beschränkung der Performanz.

Bei der Kahrisma-Architektur wurde daher zur Realisierung einer modulbasierten Partitionierung der Prozessorpipeline beide Prinzipien kombiniert, um sowohl eine effiziente Partitionierung der funktionellen Einheiten als auch des Registerspeichers ermöglichen zu können. Dabei wird die VLIW Befehlssatzarchitektur mit einer superskalaren Mikroarchitektur kombiniert. Im Befehlsformat wird dabei festgelegt, welche Befehle parallel ausgeführt werden können, in welchem Modul die jeweiligen Befehle ausgeführt werden und wann Daten zwischen den Modulen kopiert werden müssen. In der Hardware findet hingegen weiterhin eine dynamische Ausführung der Befehle statt, so dass das Anstoßen der Befehle nicht zwischen den Modulgrenzen synchron ist sondern zwischen den Modulen zeitlich versetzt stattfinden kann. Dabei müssen dann Befehle dynamische auf die Ergebnisse von anderen Befehlen warten und am

Ende wird das Zurückschreiben so lange verzögert, bis sämtliche Befehle in einer Instruktion sicher, d.h. ohne Auslösung eines Interrupts, ausgeführt wurden.

Bei diesem Konzept muss eine dynamische Anzahl von Modulen innerhalb einer virtuellen Prozessorinstanz auch direkt innerhalb des Befehlsformates berücksichtigt werden. So ist das Befehlsformat abhängig von der jeweiligen Konfiguration und erhöht somit die Komplexität der Programmierung dieser Architektur. Innerhalb dieser Arbeit wird allerdings gezeigt, dass diese zusätzliche Komplexität effizient durch heutige Compiler Toolchains beherrscht werden kann und somit eine dynamisch Konfiguration von ILP und TLP praktisch – von Seiten der Software und Hardware – realisierbar ist.

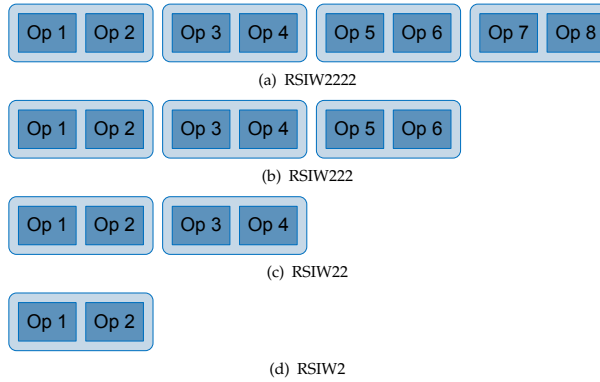
3.4 RSIW-Befehlssatzarchitektur

Wie in der vorherigen Abschnitten beschrieben, wird für die Realisierung von rekonfigurierbaren Prozessorkernen ein neuartiges Prozessorprinzip verwendet, das die VLIW-Befehlssatzarchitektur mit out-of-order Mikroarchitekturkonzepten kombiniert. Die ISA von virtuellen Prozessorinstanzen der Kahrisma-Architektur hat den Namen *Run-Time Scalable Issue-Width Processors* (RSIW). Durch die Rekonfiguration bzw. die dynamische Verschaltung von Modulen kann zur Laufzeit der ILP dynamisch angepasst werden und dies geht auch mit der Rekonfiguration der ISA einher. Somit ist die ISA nicht statisch wie herkömmlichen ISAs sondern ändert sich je Konfigurationen.

In diesem Abschnitt wird nur eine spezifische Ausprägung der Kahrisma Befehlssatzarchitektur zur Förderung der Verständlichkeit diskutiert. Bei der Realisierung wurde die Befehlssatzarchitektur (siehe Kapitel 5) allgemeiner gehalten, so dass wichtige Größen, wie z.B. die Anzahl der Register oder Operanden, flexibel eingestellt werden können.

3.4.1 RSIW-Instruktionsformat

Abbildung 3.4 zeigt die verschiedenen Instruktionsformate der RSIW-ISA. Im Allgemeinen ist die RSIW-ISA mit der ISA eines Clustered-VLIW Prozessors vergleichbar. Allerdings ist die ISA rekonfigurierbar und das Format ändert sich je nach Konfiguration. RSIW2 kodiert dabei zwei parallele Operationen mit einem Registerfile während RSIW2222 acht parallele Operationen mit insgesamt 4 Registerfiles enthält. Die RSIW2 Konfiguration der rekonfigurierbaren ISA entspricht dabei der ISA eines VLIW Prozessors. Die restlichen Konfigurationen mit mehr als einem Registerfile entsprechen dann einer ISA für Clustered-VLIW Prozessoren. Die Anzahl, der in der ISA sichtbaren Registerfiles, ist auch gleichzeitig die Anzahl der verwendeten Cluster. Durch die Position einer Operation im Instruktionswort wird kodiert, in welchem Cluster bzw.



(a) RSIW2222

(b) RSIW222

(c) RSIW22

(d) RSIW2

Abbildung 3.4: Das konfigurationsabhängige Instruktionsformat der RSIW-ISA

Modul die jeweilige Operation ausgeführt wird.

3.4.2 Register

In der ISA gibt es zwei verschiedene Typen von Allzweckregistern und zusätzliche Steuerregister. Die Allzweckregister sind jeweils pro Cluster enthalten während die Steuerregister nur einmal pro virtueller Prozessorinstanz existieren.

Daten-Register Jedes Cluster besitzt 32 32-bit Allzweckregister, die Daten und Adressen aufnehmen können.

Event-Register Jedes Cluster besitzt 8 1-bit Allzweckregister. Die Eventregister werden hauptsächlich für den Kontrollfluss verwendet, können aber auch z.B. für Carry-Bits verwendet werden. Zur Realisierung von bedingten Sprüngen wird das Ergebnis einer Vergleichsoperation in einem Event-Register gespeichert, das dann für bedingte Sprungbefehle verwendet wird.

Steuer-Register Die Steuer-Register sind nur einmal pro virtueller Prozessorinstanz vorhanden. Sie umfassen insbesondere den *Befehlszeiger* (IP, engl. *Instruction Pointer*) sowie Informationen über Exceptions.

Die Allzweckregister werden nach einem Namensschema benannt:
`R<Cluster>_<Nummer>` für Daten- bzw. `E<Cluster>_<Nummer>` für Event-Register. So bezeichnet z.B. `R1_23` das 23ste Datenregister aus Cluster 1. Sämtliche Register mit Nummer 0 haben eine Sonderrolle. Von ihnen wird immer die Zahl 0 gelesen und

beim Schreiben wird das Ergebnis verworfen.

3.4.3 Operationsformat

Eine Operation besteht aus folgenden Elementen:

- Opcode
- Operanden
  - 2 Daten-Quellregister
  - 1 Event-Quellregister
  - 1 Daten-Zielregister
  - 1 Event-Zielregister
- Immediate
  - 1 Daten-Immediate
  - 1 Event-Immediate

3.4.4 Inter-Cluster Kommunikationsmodell

Die ISA verwendet das Extended Results inter-cluster Kommunikationsmodell (siehe Kapitel 2.2.9.1). Dabei kann das Zielregister einer Operation in jedem Cluster liegen, für die Quellregister gilt allerdings die Einschränkung, dass sie nur von dem lokalen Cluster gelesen werden kann.

Zur Optimierung der Mikroarchitektur existieren hierbei noch weitere Ressourcen-Constraints, die in der ISA sichtbar sind und vom Compiler beachtet werden müssen. So ist die Anzahl der Register, die in einer Instruktion auf ein Cluster und Registertyp geschrieben werden können, beschränkt. So können z.B. auf Cluster 0 nur 3 Daten-Register pro Instruktion als Zielregister verwendet werden.

3.4.5 Befehlssatz

In Tabelle 3.1 ist der Befehlssatz bzw. die Operationsliste der Kahrisma-Architektur aufgelistet. Dabei ist die Liste auf die Operationen beschränkt, die für das Softwareframework und insbesondere den Compiler relevant sind. In der Tabelle werden folgende Aspekte einer Operation beschrieben:

Op beschreibt die Opcodenummer der Operation.

Op	Mnemonic	Exception	CtrlSignals	Event In	Event Out	ALU-Pfad	Beschreibung
0	NOP	-		-	-	-	Keine Operation
1	OR	-	0000	-	Zero	Integer	Bitweise Oder
1	AND	-	0001	-	Zero	Integer	Bitweise Und
1	XOR	-	0010	-	Zero	Integer	Bitweise Exklusiv Oder
1	NOT	-	0100	-	Zero	Integer	Bitweise Negation
4	SHL	-		-	Shifted-Out Bit	Integer	Schiebeoperation Links
5	SHR	-	S	-	Shifted-Out Bit	Integer	Schiebeoperation Rechts
6	ADD	-	S, SAT	Carry	Carry	Integer	Addition
7	SUB	-	S, SAT	Carry	Carry	Integer	Subtraktion
8	MUL	-	S, SAT, HR	-	-	Mul	Multiplikation
9	DIV	DIV0	S, RE	-	-	Div	Division
10	LD1	MMU	W, S	-	Data	LS	1-Bit Ladeoperation
10	LD	MMU	W, S	Predication	-	LS	Ladeoperation
11	ST1	MMU, SPEC	W	Data	-	LS	1-Bit Speicheroperation
11	ST	MMU, SPEC	W	Predication	-	LS	Speicheroperation
13	CMP	-	COND	-	Cmp	Integer	Vergleich
14	SEL	-		Flag	-	Integer	Auswahlbefehl (Select)
15	BR	BR	REL, BPHINT	Predication	-	Integer	Sprungbefehl
18	EXC	UNKN, SPEC	EXC	-	-	Integer	Exception (nur intern)
19	INT/ERET	INT	-	-	-	Integer	Software Interrupt, Exception Return
20	MOV	-	S	Data	Data	Integer	Kopierbefehl
21	PACK	-	-	-	-	-	Umsortieren von Bytes aus zwei Quelloperanden.
31	SIM	-	-	-	-	-	Simulator-Befehl

Tabelle 3.1: Operationsliste

Mnemonic ist der Name der Operation, wie er typischerweise im Assembler verwendet wird. Ein Opcode in der Tabelle steht häufig für eine gesamte Gruppe von Befehle und gibt somit nur einen Hinweis auf den Assembler-Namen.

Exception gibt an, welche Exceptions von dem Befehl ausgelöst werden können. Hier wird zwischen *Division durch 0* (DIV0), *Memory Management Unit* (MMU) und *Speculation Failed* (SPEC) Exception unterschieden.

CtrlSignals gibt Kontrollsignale für den Befehl an. Die Kontrollsignals stellen eine Erweiterung des Opcodes dar. Die Kontrollsignals sind aus einzelnen Feldern aufgebaut, die in Tabelle 3.2 aufgelistet sind. Für arithmetische Befehle kann der *Überlauf* (SAT, engl. *Saturation*) und *Vorzeichen* (S, engl. *Signed*) gesteuert werden. Für die Multiplikation kann wahlweise das *Obere Ergebnis* (HR, engl. *High-Result*) zurückgegeben werden. Genauso kann bei der Division optional der *Rest* (RE, engl. *Remainder*) berechnet werden. Für Lade- und Speicheroperationen wird deren *Breite* (W, engl. *Width*) in den Steuersignalen kodiert. Bei Sprungbefehlen muss angegeben werden, ob dieser *relativ* (REL, engl. *Relative*) oder absolut ist und es werden *Hinweise für die Sprungvorhersage* (BPHINT, engl. *Branch Prediction Hint*) angegeben. Bei Vergleichsoperationen wird die *Bedingung* (COND, engl. *Condition*) angegeben.

Event In/Out gibt an, wozu der Event-Quell- und -Zieloperand verwendet wird.

ALU-Pfad beschreibt, durch welche Ausführungspipeline in der Hardware der Befehl abgearbeitet wird.

Beschreibung gibt eine kurze Beschreibung des Befehls.

Beim Design des Befehlssatzes wurde insbesondere darauf geachtet, dass alle notwendigen Befehle für den C Compiler vorhanden sind. Also insbesondere alle Operatoren aus der C Programmiersprache effizient unterstützt werden können.

3.4.6 Optimierung für Parallelität auf Befehlsebene

Das Befehlsformat und der Befehlssatz wurden darauf optimiert, dass möglichst viel Parallelität auf Befehlsebene in der Befehlssatzarchitektur ausgedrückt werden kann und der Compiler darin unterstützt wird. So wird der Vergleich zwischen zwei Zahlen nicht wie bei RISC üblich in einem Flags-Register gespeichert, sondern wird in einem 1-Bit Event-Register abgelegt. Ein Sprungbefehl kann dann abhängig von dem Wert in einem Event-Register ausgeführt werden. Das erlaubt als einfachste ILP-Optimierung, die Entkoppelung zwischen Vergleichs- und Sprungbefehlen und somit das parallele Scheduling der Vergleichsbefehle mit anderen Befehlen.

Zusätzlich unterstützt die ISA *Partial Predication* zur Steigerung des ILPs. Abhängig vom Inhalt eines Event-Registers (also typischerweise einem Vergleich) kann durch den Select-Befehl entweder der Inhalt des ersten oder zweiten Operanden in ein Register kopiert werden. Somit lassen sich kleine Kontrollverzweigungen auflösen, indem

Kürzel	Bits	Name	Beschreibung
SAT	1	Saturation	Bei 1 wird der Befehl mit Saturation gerechnet.
S	1	Signed	Signed (1) oder Unsigned (0)
HR	1	High-Result	Ergebnis der Multiplikation: Untere 32-Bit (0) oder oberen 32-Bit (1)
RE	1	Remain	Ergebnis der Division: Rest (1) oder Quotient (0)
W	2	Width	Breite für den Hauptspeicherzugriff: 1-Bit (00), 8-Bit (01), 16-Bit (10) oder 32-Bit (11)
REL	1	Relative	Realtiver (1) oder Absoluter (0) Sprung
COND	4	Condition	Bedingung für den Vergleich.
EXC	4	Exception	Nummer der Exception
BPHINT	2	Branch Prediction Hint	Hinweis für die Sprungvorhersage, ob es sich um einen bedingten/unbedingten Sprung, Unterfunktionsaufruf oder Return handelt

Tabelle 3.2: Übersicht über die Kontrollsignale der Operationen

das Registerergebnis abhängig von einem vorherigen Vergleich mit Hilfe des Select-Befehls verwendet wird oder nicht. Voraussetzung hierfür ist, dass in einem aufgelösten Basisblock kein Befehl vorhanden ist, der – mit Ausnahme der Zielregister – Änderungen am Prozessorzustand vornimmt. Diese Voraussetzung ist für alle exceptionfreie arithmetisch-logischen Befehle erfüllt, scheitert jedoch bereits bei den weit verbreiteten Load/Store-Befehlen. Um auch Basisblöcke mit Load/Store-Befehlen auflösen zu können, können die meisten Load/Store-Befehle abhängig durch den Inhalt eines Event-Register ausgeführt werden.

Eine weitere Besonderheit der ISA ist die Unterstützung von *Geordneten Parallelen Load/Store-Operationen*. Zum einen ist die Anzahl von Load/Store-Operationen innerhalb einer Instruktion in der ISA nicht beschränkt. Dadurch wird die Flexibilität des Compilers für das Scheduling erhöht und somit das Finden eines kürzeren Schedules begünstigt. Zum anderen werden innerhalb einer Instruktion parallele Load/Store-Operationen geordnet anhand der Slotnummer ausgeführt. D.h. greifen zwei Load/Store-Operationen der gleichen Instruktion auf dieselbe Speicherstelle zu, wird die Linke vor der Rechten ausgeführt. Dies ist ein Vorteil gegenüber anderen VLIW Architekturen, bei denen der Fall häufig undefiniert ist. Der Compiler kann somit abhängige Load/Store-Operationen, die potentiell auf die gleiche Speicherstelle zugreifen können, unter Berücksichtigung der Slotnummer gemeinsam in eine Instruktion einplanen. Dadurch wird weiterhin das Finden eines kürzeren Sche-

dules, insbesondere bei einer großen Anzahl an abhängigen Load/Store-Operationen, begünstigt. In Sektion 3.5.2.8 wird dieses Feature aus Sicht der Mikroarchitektur beschrieben.

3.4.7 Exceptions und Interrupts

Die Architektur unterstützt interne Exceptions und externe Interrupts (siehe Tabelle 3.2(a)). Durch das Auslösen eines Interrupts oder einer Exception springt der Prozessor an eine vorher festgelegte IP, an dem eine *Unterbrechungsbehandlungsroutine* (ISR, engl. *Interrupt-Service Routine*) hinterlegt ist. Zusätzlich wird die auslösende IP-Adresse, die Exception/Interrupt-Identifikationsnummer und ein Fehlercode in einem Kontrollregister gespeichert und das Interrupt-Flag gelöscht. Innerhalb der ISR kann dann die jeweilige Exception identifiziert und behandelt werden. Durch das manuelle Setzen bzw. Löschen des Interrupt-Flags innerhalb einer ISR werden zudem verschachtelte Exceptions/Interrupts ermöglicht. Am Ende einer ISR kann mit einem speziellen Exception-Return-Befehl zum normalen Programmfluss zurückgekehrt werden.

Zusätzlich zu den Interrupts/Exceptions, die für die ISA sichtbar sind, gibt es noch versteckte Exceptions (siehe Tabelle 3.2(b)). Die versteckten Exception sind Mikroarchitekturabhängig und werden nur verwendet, um das aktuelle Instruktionsfenster neu auszuführen. Damit reagiert die Hardware z.B. auf falsch vorhergesagt Sprünge oder zur Auflösung von Deadlocksituation infolge der out-of-order Ausführung.

(a) Sichtbare Exceptions

Name	32-Bit Fehlercode	Parameter	Beschreibung
DIV0	-	-	Division by Zero
MMU	FaultAddr	Type (Code)	Memory Management Unit. Error on Address Translation
UNKN	-	-	Unknown Opcode
INT	IntrNo	-	Software oder Hardware-Interrupt

(b) Versteckte Exceptions

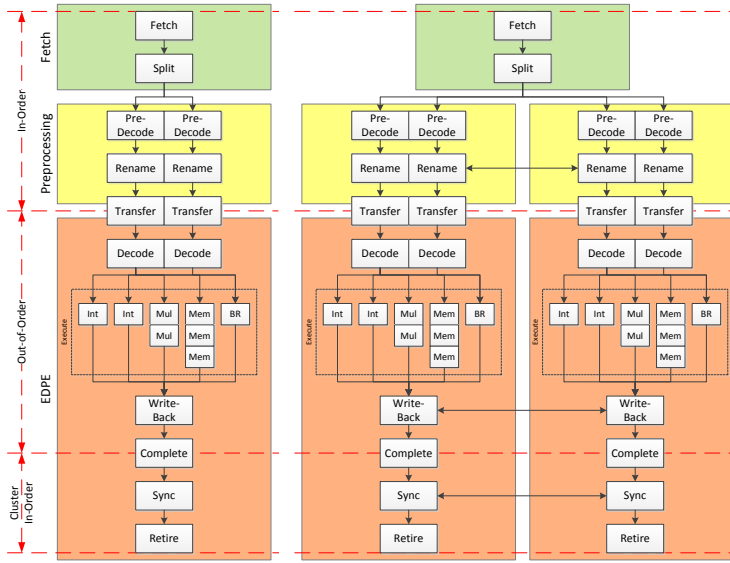
Name	32-Bit Daten	Parameter	Beschreibung
SPEC	-	-	Speculation Failed
BR	Zieladresse	Relativ	Sprung

Tabelle 3.3: Liste der Exception und Interrupts

3.5 Kahrisma-Mikroarchitektur

In diesem Abschnitt wird die Mikroarchitektur der Kahrisma-Architektur vorgestellt, die die Befehlssatzarchitektur aus Abschnitt 3.4 umsetzt. Auch hier wird aus Gründen der Verständlichkeit nur eine spezifische Ausprägung der Kahrisma Mikroarchitektur diskutiert. Viele Parameter, wie z.B. die Anzahl an Operationen pro EDPE, wurden im Design flexibel gehalten.

3.5.1 Pipeline



(a) RSIW2 Pipeline mit 1 Modul (b) RSIW22 Pipeline mit 2 Modulen

Abbildung 3.5: Zwei Konfigurationen der rekonfigurierbaren Prozessor-Pipeline

Abbildung 3.5 zeigt die Pipeline der Kahrisma-Architektur. Im Gegensatz zu statischen Prozessoren hat Kahrisma eine rekonfigurierbare Pipeline, die je nach Anzahl an verwendeten Ressourcen, eine andere Ausprägung aufweist. So ist in Abbildung 3.5(a) die Pipeline-Konfiguration mit nur einem Modul zu sehen, während Abbildung

3.5(b) exemplarisch die Pipeline mit zwei Ressourcen zeigt.

Die Pipeline unterstützt out-of-order Ausführung der Befehle, allerdings werden die Operanden nur in-order zur Ausführung angestoßen. Das in-order Anstoßen findet dabei pro Slot im RSIW-Instruktionswort statt, sodass die einzelnen Slots gegeneinander driften können. Die Realisierung der Pipeline erfolgt durch die Konfiguration und das Zusammenschalten von drei unterschiedlichen Prozessormodulen: Fetch-Module, Preprocessing-Module und *Encapsulated Datapath Elements* (EDPEs). Pro Konfiguration wird ein Fetch-Modul, n Preprocessing-Module und n EDPEs benötigt, wobei n zwischen 1 und 4 liegen kann. Ein Preprocessing-Modul und eine EDPE zusammen werden auch als **Cluster** bezeichnet. Durch die Erhöhung der Anzahl der Cluster kann die Anzahl der Parallel ausführbaren Operationen flexibel erhöht und somit die Superskalarität einer virtuellen Prozessorinstanz gesteigert werden. Pro Cluster können zwei Operationen gleichzeitig verarbeitet werden. Somit ist die Anzahl an paralleler ausführbarer Operationen pro Konfiguration ist immer $2 * n$.

Im Folgenden werden die einzelnen Pipelinestufen aus Abbildung 3.5 beschrieben:

Fetch lädt Cache-Zeilen vom Instruktions-Cache. Eine Cache-Zeile ist so breit gewählt, dass mindestens 8 Operationen, also die maximale Zuweisungsbandbreite, parallel geladen werden.

Split schneidet die einzelnen Operationen aus der geladenen Cache-Zeile aus und weißt die Operation anhand der Position im RSIW-Instruktionswort den jeweiligen Preprocessing-Modul zu.

Predecode führt eine Vordekodierung der Operationen durch. Die Operationen werden vom externen ins interne Operationsformat überführt. Dabei wird unter anderem jede Instruktion mit einer Kennung versehen, die für die gesamte Lebenszeit eindeutig sein muss.

Renaming führt eine Registerumbenennung durch, um Namensabhängigkeiten zwischen Registern zu beseitigen. Das Renaming muss zwischen den verschiedenen Preprocessing-Modulen synchronisiert werden, da im Falle von Inter-Cluster Kommunikation ein Cluster eine Registernummer von einem entfernten Cluster benötigt.

Transfer transferiert die Operationen zu den zugeteilten EDPEs. Ab dieser Stufe beginnt die out-of-order Ausführung, da die Transfer-Stufe FIFOs beinhalten, die ein Driften und Zurückstauen der Operationen ermöglichen.

Decode dekodiert die Operationen, lädt alle Quellregister und weist die Operation einem der Datenpfade zu. Im Decoder werden echte Daten- und Ressourcenkonflikte aufgelöst, indem die Abarbeitung der Operation angehalten wird, solange ein Quelloperand noch nicht vorhanden oder der Datenpfad belegt ist.

Execute führt die Operationen aus und besteht aus verschiedenen diversitären Datenpfaden, die je nach Operationstyp verwendet werden. In Execute sind folgende Datenpfade vorhanden:

Integer (Int) behandelt die meisten einfachen Integer-Operationen und ist daher einmal pro Decoder vorhanden.

Multiplikation (Mul) führt eine gepipelinte Multiplikation aus.

Memory (Mem) führt sämtliche Lade-/Speicheroperationen aus. Die Speicherzugriffe müssen zwischen den Clustern synchronisiert werden, um Speicherdisambiguierung zu verhindern.

Branch (BR) behandelt sämtliche Operationen, die den Befehlszähler beeinflussen. Dies sind insbesondere bedingte und unbedingte Sprungbefehle.

Write-Back schreibt die Ergebnisse der Datenpfade zurück in das spekulative Registerfile und enthält die Forwarding Logik, um Ergebnisse direkt an die Datenpfade weiterzuleiten. Die Fertigstellung eines Befehls wird in einem Reorder Buffer quittiert. Im Fall einer Exceptions, wird dies ebenfalls im Reorder Buffer gespeichert.

Complete überprüft den Reorder Buffer, welche Befehle in der ursprünglichen Befehlsreihenfolge fertig berechnet sind und ob eine Exception ausgelöst wurde. Dabei überprüft Complete jeweils nur Operationen, die im lokalen Cluster ausgeführt wurden.

Sync synchronisiert die fertig berechneten Befehle der verschiedenen Cluster und synchronisiert den Exception-Status. Sobald alle Operationen einer Instruktion auf sämtlichen Clustern berechnet sind, können diese Operationen abgeschlossen werden.

Retire macht die Operationen gültig, wenn keine Exception ausgelöst wurde oder verwirft die Ergebnisse im Falle einer Exception. Dabei werden Ergebnisse aus dem spekulative Registerfile in das permanente Registerfile übernommen.

3.5.2 Mikroarchitekturkonzepte

3.5.2.1 Identifikation von Instruktionen durch den Modulus Cycle

Der *Modulus Cycle* (MC) wird innerhalb der Architektur zur eindeutigen Identifikation von Instruktionen verwendet. Eine eindeutige Identifikation ist notwendig, um sämtliche Operationen nach der out-of-order Ausführung wieder in die richtige Reihenfolge bringen zu können. Zusätzlich wird der MC noch als Vergleich verwendet, welcher MC älter bzw. jünger ist, um z.B. eine optimale Arbitrierung zwischen Ressourcen zu gewährleisten.

Ein MC mit n Bits kann 2^n Zustände kodieren. Damit der MC vergleichbar ist, dürfen gleichzeitig allerdings nur die Hälfte der Zustände verwendet werden, also 2^{n-1}. In der Predecode-Stufe wird der MC den Instruktionen zugewiesen. Hierbei wird durch anhalten der Pipeline bis zu dieser Stufe sichergestellt, dass sich zu jeder Zeit maximal

2^{n-1} Instruktionen gleichzeitig in der Pipeline befinden, um die Vergleichbarkeit und eindeutige Identifizierbarkeit dieser zu gewährleisten. Je nachdem wie viele Operationen eine Instruktion maximal haben kann, wird hierdurch auch die maximale Anzahl an Operationen, die sich in der Pipeline befinden können, beschränkt.

3.5.2.2 Behandlung von Register-Namensabhängigkeiten

Namensabhängigkeiten zwischen Registern werden in der Mikroarchitektur mittels Registerumbenennung behandelt. Dazu existieren in der Architektur zwei Registerfiles: Das Architektur- und Rename-Registerfile. Im Architektur-Registerfile werden sämtlichen sicheren Werte abgelegt während das Rename-Registerfile spekulative Ergebnisse, die noch nicht als sicher markiert sind, beinhaltet. Die Register des Architektur-Registerfiles entsprechen den Registern der Befehlssatzarchitektur. Allerdings werden die Register temporär auf das zusätzliche Rename-Registerfile umbenannt, so dass sich in der Pipeline gleichzeitig immer nur eine Operation befindet, die auf ein Rename-Register schreibt. Dadurch kann es nicht mehr zu Konflikten, ausgelöst durch den gleichen Namen des Registern, kommen.

In der Renaming-Pipelinestufe werden dazu die Register umbenannt. Jedes Zielregister bekommt ein neues, freies Rename-Register zugewiesen und wird dementsprechend umbenannt. Sämtliche Umbenennungen werden in einer Mapping-Tabelle abgelegt. Die Quellregister werden anhand der Mapping-Tabelle umbenannt. Wenn kein Eintrag vorhanden ist, dann verweisen sie weiterhin auf das Architektur-Registerfile. Ein Rename-Register wird nur einmal geschrieben und somit sind keine Namensabhängigkeiten mehr in der Pipeline vorhanden. In der Retire-Stufe werden die Werte aus den Rename- in die Architektur-Register in der Programmreihenfolge kopiert und das Mapping in der Tabelle wieder gelöscht. Somit wird sichergestellt, dass Ausgabeabhängigkeiten korrekt behandelt werden.

3.5.2.3 Behandlung von Register-Datenabhängigkeiten

Durch die Registerumbenennung ist die Behandlung von Datenabhängigkeiten vergleichsweise einfach. Während der Lebenszeit eines Rename-Registers wird dieses nur einmal geschrieben. Somit kann ein Rename-Register um ein Valid-Flag ergänzt werden. Dieses wird zurückgesetzt, sobald ein Rename-Register durch eine Registerumbenennung vergeben wurde. Es wird gesetzt, sobald das Rename-Register geschrieben wird. So Lange das Flag nicht gesetzt ist, werde sämtlichen Operationen, die auf das Register zugreifen möchten, angehalten. Somit wird sichergestellt, dass erst nach der Berechnung der Daten für das Register dieses auch gelesen wird und somit die Datenabhängigkeiten eingehalten werden.

3.5.2.4 Rename-Register Lebenszeit

Die Rename-Register haben eine komplexe Lebenszeit bzw. Lebensphase, die in den folgenden Zuständen zusammengefasst sind:

Frei Das Rename-Register ist nicht vergeben.

Leer Das Rename-Register wurde durch eine Registerumbenennung vergeben, aber es wurde noch kein Wert geschrieben.

Voll Das Rename-Register wurde geschrieben und enthält gültige Werte.

Retire Das Rename-Register wurde zurückgeschrieben, d.h. der Inhalt wurde in das Architektur-Registerfile kopiert. Das Register-Mapping für das Architektur-Register wird gelöscht. Der Rename-Register kann allerdings erst freigegeben bzw. wiederverwendet werden, sobald keine Operation mehr in der Pipeline existiert, die das Register lesen möchte.

3.5.2.5 Inter-Cluster Kommunikation

Wie in Kapitel 3.4.4 beschrieben, verwendet die Befehlssatzarchitektur das Extended Results inter-cluster Kommunikationsmodell. Dabei kann das Zielregister einer Operation in jedem Cluster liegen. Für die Quellregister gilt allerdings die Einschränkung, dass sie nur von dem lokalen Cluster gelesen werden können. Dies muss auch in der Mikroarchitektur umgesetzt werden. Dies fängt in den verteilten Rename-Stufen an. Wenn eine Operation auf ein entferntes Cluster bzw. Registerfile schreibt, dann benötigt es auch ein freies Rename-Register aus dem Zielcluster. Daher werden in einem solchen Fall die Renaming-Stufen zwischen den Clustern synchronisiert, so dass das Rename-Register auch an andere Cluster vergeben werden können. Die Operation wird dann im lokalen Cluster ausgeführt. In der Write-Back-Stufe wird allerdings erkannt, dass sich das Ziel-Register eines Ergebniswertes in einer entfernten EDPE befindet und somit wird das Ergebnis an die zuständige Write-Back-Stufe weitergeleitet, welche das Register dann in das Zielregister speichert.

3.5.2.6 Dynamic Operation Execution

Durch die VLIW-Befehlssatzarchitektur in Kombination mit der superskalaren Mikroarchitektur werden die Operationen einer Instruktion nicht synchron, wie bei klassischen VLIW-Prozessoren üblich, ausgeführt. Stattdessen können die verschiedenen VLIW-Slots gegeneinander driften. Dieses Ausführungsmodell im Kontext von VLIW-Prozessoren nennt sich *Dynamic Operation Execution* (DOE). In Abbildung 3.6 ist diesen Modell gegenüber VLIW-Prozessoren mit statischen Scheduling gegenübergestellt, die das *Atomic Instruction Execution* (AIE) Modell verwenden. Bei AIE muss eine Instruktion komplett abgearbeitet sein, bevor die nächste Instruktion ausgeführt wer-

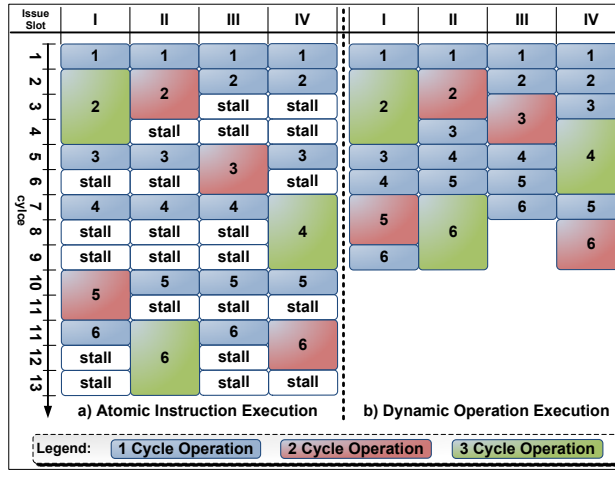
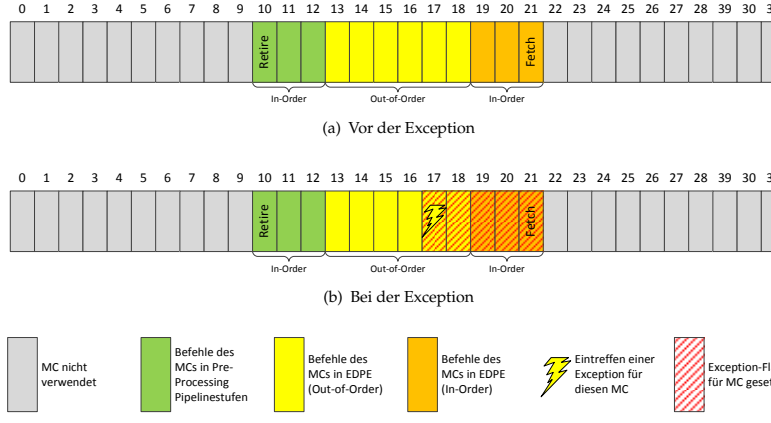


Abbildung 3.6: Vergleich zwischen dem Dynamic Operation Execution und Atomic Instruction Execution VLIW-Ausführungsmodell (Quelle [129])

den kann. Dadurch müssen Stall-Zyklen in die Pipeline eingefügt werden, wenn die Operationen einer Instruktion unterschiedlich lange Ausführungszeiten haben. Bei DOE hingegen können die Issue-Slots gegeneinander driften und somit können Stall-Zyklen, die nicht auf Datenabhängigkeiten basieren, effizient vermieden werden. Zur Vereinfachung wurde in der Abbildung auf die Modellierung von Datenabhängigkeiten verzichtet.

3.5.2.7 Out-of-order Exception Handling

Die Mikroarchitektur unterstützt die Behandlung von Exceptions, wobei hierbei sämtliche Ereignisse, die den Programmfluss beeinflussen, als Exceptions bezeichnet werden, also auch Interrupts und Sprünge. Jede Operation kann bei der Ausführung in der Execution-Stufe eine Exception auslösen. So führt z.B. jeder Sprung zu einer Exception oder jeder Speicherzugriff kann eine Exception auslösen. Da die Ausführung der Operationen außerhalb der Reihenfolge stattfindet, können somit auch die Exceptions außerhalb der Reihenfolge und dezentral verteilt über mehrere EDPEs auftreten und müssen synchronisiert werden. Diese Synchronisation findet an zentraler Stelle in der Fetch-Stufe statt. Dazu wird bei jeder Exception eine Nachricht an die Fetch-Stufe



(a) Vor der Exception

(b) Bei der Exception

Abbildung 3.7: Beispiel einer Exception

geschickt, die entscheidet, ob die Exception ausgeführt wird und dann dafür sorgt, dass die Ergebnisse der spekulativ ausgeführten Instruktionen verworfen werden.

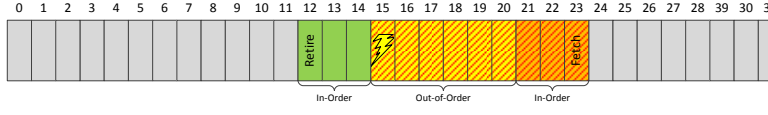
Bei der Behandlung von Exceptions muss die Mikroarchitektur dabei die Bedingungen der präzisen Interrupts erfüllen, wie sie in Kapitel 2.1.10 definiert wurden. Dies ist am Beispiel in Abbildung 3.7 zu sehen. Tritt eine Exception in MC 17 auf, so müssen alle vorherigen Operationen ($MC \in [10, 16]$) fertig ausgeführt werden, alle nachfolgenden Operationen ($MC \in [18, 21]$) verworfen werden und für die Operationen der aktuellen Instruktion muss klar sein, ob sie verworfen oder fertig ausgeführt werden muss. So wirkt sich z.B. eine Sprung-Exception erst auf die folgende Instruktion aus, d.h. die Instruktion mit der Sprung-Operation muss noch fertig ausgeführt werden. Bei anderen Exceptions, wie z.B. eine Speicherschutzverletzung, muss die auslösende Instruktion verworfen werden.

Die Bedingungen der präzisen Interrupts werden sichergestellt, indem jeder Instruktion bzw. jedem MC ein Exception-Flag zugewiesen ist, das entscheidet, ob die Ergebnisse der Operationen verworfen oder übernommen werden sollen. Dieses Exception-Flag muss über alle EDPEs synchronisiert werden. Dazu wird in der Complete-Stufe jeder EDPE lokale pro MC erzeugt und mittels der Sync-Stufe über alle EDPEs und MCs synchronisiert (Oder-Verknüpft). Die Retire-Stufe jeder EDPE verwendet dann das Exception-Flag um ggf. die Ergebnisse von Instruktionen zu verwerfen.

Die Fetch-Stufe bekommt zentral alle Exceptions mitgeteilt und überprüft, ob die Exception behandelt werden muss oder nicht. So schickt z.B. jeder Sprung-Befehl Exception-Informationen an die Fetch-Stufe, die aber nur zu einer wirklichen Exception

(a) Nach der Exception

(b) 2te Exception später im Programmfluss



(c) 2te Exception früher im Programmfluss

Abbildung 3.8: Beispiel von verschachtelte Exceptions

führen, wenn die Sprungvorhersage vorher falsch war, d.h. die falsche Instruktion geladen wurde. Für die Dauer der Überprüfung wird die Operation, die die Exception ausgelöst hat, in der Complete-Stufe blockiert. Nach der Überprüfung, schickt die Fetch-Stufe die Antwort an die verantwortliche EDPE und gibt die Blockierung der Operation in der Complete-Stufe wieder frei. Dabei werden die Exception-Flags für die betroffenen MCs gesetzt.

Zur Steigerung der Performanz ist es wichtig, dass Sprünge (und damit Exceptions im Allgemeinen) so schnell wie möglich abgearbeitet werden. Daher werden Exceptions auch außerhalb der Reihenfolge an die Fetch-Stufe geschickt und nicht erst dann, wenn alle vorherigen Operationen ohne Exception abgearbeitet wurden. Dadurch kann es allerdings zu verschachtelten Exceptions kommen, die durch den Algorithmus korrekt abgearbeitet werden müssen. Hierbei kann es zu zwei Fällen kommen, die in Abbildung 3.8 dargestellt sind. Abbildung 3.8(a) zeigt zunächst den Zustand, nachdem eine Exception ausgelöst wurde und bereits zwei weitere Instruktionen geholt und mit neuen MCs versehen wurden. Im ersten Fall wird nun eine weitere Exception an die Fetch-Stufe gemeldet, die im Programmfluss nach der ersten Exception kommt. Die zweite Exception ist also in dem Bereich der Instruktionen, der verworfen werden soll und wird dabei durch die Fetch-Stufe nicht behandelt. Im zweiten Fall (Abbildung 3.8(c)) wird eine weitere Exception ausgelöst, die allerdings im Programmfluss vor der ersten Exception aufgetreten ist. Damit sind sämtliche Aktionen,

die durch die erste Exception ausgelöst wurden, hinfällig. Dieses Problem wird gelöst, indem der Bereich der Instruktionen, die verworfen werden müssen, vergrößert wird.

3.5.2.8 Behandlung von Speicherzugriffen

Genauso wie bei Registern, können auch bei Speicherzugriffen sämtliche Schreibzugriffe erst abschließend durchgeführt werden, wenn die jeweilige Instruktion sicher, d.h. ohne vorherige Exception, abgearbeitet wurde. Daher werden sämtliche Schreibzugriffe anhand des MCs in einer Store-Queue abgelegt. Sobald der jeweilige MC als fertig abgeschlossen gibt, wird abhängig vom Exception-Flag des MCs der Schreibzugriff durchgeführt oder verworfen.

Verhinderung von Speicherkonflikten Zusätzlich können mehrere Speicherzugriff auf die gleiche identische Hauptspeicheradressen zugreifen, wodurch zwischen den Speicherzugriffen die bekannten Daten-, Ausgabe- und Gegenabhängigkeiten entstehen können, die dann wegen der out-of-order Ausführung wiederum zu Datenkonflikten führen können. Dieses Problem wird in der Literatur als Speicher-Disambiguierung (engl. *Memory Disambiguation*) bezeichnet. Die Ausgabeabhängigkeiten werden durch die Sortierung der Speicherzugriffe in der Store-Queue eingehalten. Bei den Datenabhängigkeiten zwischen einem Load und Store gestaltet sich die Behandlung schwieriger, da zum Zeitpunkt eines Lesezugriffs sich noch ältere Schreibzugriffe in der Pipeline befinden können. Anstatt die Lesezugriffe zu verzögern, bis keine ältere, wartende Schreibzugriffe mehr in der Pipeline sind, werden in der Mikroarchitektur Lesezugriffe spekulativ ausgeführt und in einer Load-Queue einsortiert. Anhand der Load-Queue können dann Datenkonflikte erkannt werden, wenn ein älterer Schreibzugriff die gleiche Adresse wie ein jüngerer Lesezugriff verwendet. Im Falle eines erkannten Datenkonflikts löst der involvierte Schreibzugriff eine „Speculation Failed"-Exception aus und triggert somit die Neuausführung der Leseoperation. Durch den MC in der Load-Queue kann zwischen Daten- und Gegenabhängigkeiten unterschieden werden.

Synchronisation von verteilten Speicher-Datenpfaden In der Mikroarchitektur findet der Zugriff auf den Hauptspeicher im Memory-Datenpfad in der Pipeline statt. Abbildung 3.9 zeigt den Memory-Datenpfad im Detail. Hier tritt wieder das Problem auf, dass die Mikroarchitektur in Module verteilt ist und es deswegen auch einen verteilten Memory-Datenpfad gibt. Hierbei ist eine Synchronisation der Memory-Datenpfade unerlässlich, da sonst die einzelnen EDPE-Module eine inkonsistente Sicht auf den Speicher hätten, der wahrscheinlich zu Konflikten führen würde. Pro Memory-Datenpfad ist eine *Load/Store Queue* (LSQ) für die Speicher-Disambiguierung und ein L1-Cache vorhanden. Zur Wahrung der Kohärenz der beiden Module, wird der Adressraum des Hauptspeichers in verschiedene Partitionen zerteilt und jede LSQ bzw. L1-Cache pro Cluster ist dann für eine Partition zuständig. Die Partition wird

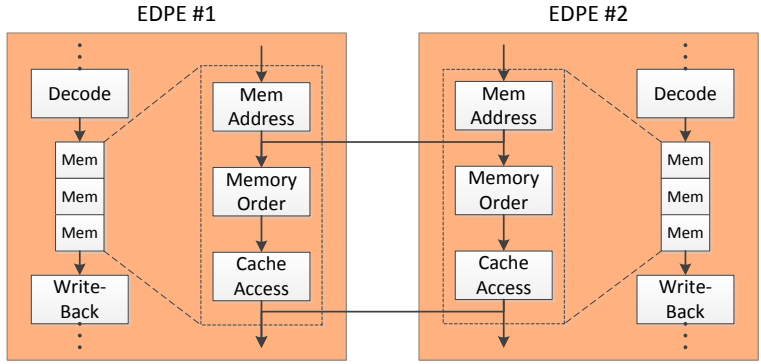


Abbildung 3.9: Die Speicher-Pipeline in der Konfiguration mit zwei EDPEs

über eine Hash-Funktion über die Hauptspeicheradresse bestimmt und der Speicherzugriff wird dann jeweils von der verantwortlichen EDPE bearbeitet. Da die Anzahl an EDPEs pro Konfiguration flexibel ist, muss hierbei dann auch die Hash-Funktion flexibel rekonfigurierbar sein. Der Memory-Datenpfad gliedert sich in folgende Pipeline-Stufen:

Mem Address berechnet die Hauptspeicheradresse und zuständige EDPE anhand der Hash-Funktion. Wenn eine andere EDPE für den Hauptspeicherzugriff zuständig ist, wird dieser dahin transferiert.

Memory Order führt den Zugriff auf die LSQ durch und wird im *Memory Order Buffer* (MOB) durchgeführt.

Cache Access greift auf den L1-Cache zu und leitet bei einem Cache-Miss den Zugriff an das Speicher-Subsystem weiter.

Geordnete parallele Load/Store-Operationen Eine weitere Besonderheit der Mikroarchitektur im Gegensatz zu VLIW-Prozessoren ist die Unterstützung von *Geordneten parallelen Load/Store-Operationen*. Bei statischen VLIW-Prozessoren ist meisten eine Beschränkung der Anzahl an Load/Store-Operationen pro Instruktion geben. Durch diese Optimierung wird die Anzahl an Speicher-Datenpfaden in der Prozessorpipeline reduziert, um Fläche zu sparen. Bei der Kahrisma-Architektur existiert allerdings eine out-of-order Ausführung, so dass man zur Compilezeit im Allgemeinen nicht vorhersagen kann, welche Operationen zur Laufzeit zeitgleich ausgeführt werden. Daher wäre eine Beschränkung der Load/Store-Operation pro Instruktionen nicht vorteilhaft. Stattdessen werden in der Decode-Stufe Ressourcenkonflikte behandelt,

so dass nur so viele Load/Store-Befehle angestoßen werden, wie Speicher-Datenpfade vorhanden sind.

Selbst bei mehreren erlaubten parallelen Load/Store-Operationen innerhalb eines statischen VLIW-Prozessors ist der Zugriff auf die gleiche Speicherstelle bei zwei gleichzeitig ausgeführten Load/Store-Operationen undefiniert. Durch die out-of-order Ausführung innerhalb der Kahrisma-Pipeline müssen die Speicherzugriffe bereits unabhängig davon im Memory Order Buffer wieder in die richtige Reihenfolge gebracht werden. Daher bietet es sich an, ebenfalls die Speicherzugriffe innerhalb einer Instruktion im Memory Order Buffer zu sortieren. Als Sortierungskriterium wurde die Slotnummer der Operation verwendet. Somit werden parallele Speicherzugriffe in einer Instruktion in der Reihenfolge der Slotnummer, also im Instruktionswort von links nach rechts, ausgeführt. Der Compiler kann somit Load/Store-Operationen, die potentiell auf die gleiche Speicherstelle zugreifen können, gemeinsam in eine Instruktion packen. Dadurch wird die Anzahl an benötigten Instruktionen reduziert.

3.5.2.9 Parametrisierbarkeit

Die Mikroarchitektur ist bezüglich verschiedener Parameter zur Designzeit flexibel gehalten. Im Folgenden ist ein Auszug der Parameter aufgeführt:

Anzahl und Layout der Module Die Anzahl der verschiedenen Module, wie EDPE, Preprocessing und Fetch, können festgelegt werden. Typischerweise wird die Anzahl durch die Breite und Höhe eines rechteckigen Feldes, das die Module beinhaltet, angegeben.

Slots pro Cluster Pro Cluster kann die Anzahl der Slots, also die Anzahl parallel ausführbarer Operationen, festgelegt werden.

Anzahl der Register Die Anzahl der Event- und Daten-Register können flexibel festgelegt werden.

Breite des MCs Die Breite des MCs kann in Bits festgelegt werden. Wie in Kapitel 3.5.2.1 beschrieben, wirkt sich die Breite des MCs direkt auf die Größe des Befehlsfensters aus. Jeder Operation im Befehlsfenster muss in verschiedenen Puffern abgelegt werden, wie z.B. im Reorder Buffer, IP History Table und Register Allocation History Table. Dadurch führt ein breiterer MC zu einem höheren Flächenbedarf.

L1-Cache-Größe Die Größe des L1-Caches pro EDPE.

ICC-Kanäle Die Anzahl an Kanäle zur inter-cluster Kommunikation zwischen den EDPEs.

Maximale Anzahl an Schreibzugriffen pro Cluster Pro Cluster bzw. Registerfile kann die maximale Anzahl an Schreibzugriffen innerhalb einer EDPE festgelegt werden. Die maximale Anzahl wirkt sich direkt auf die Anzahl an Register-

Umbenennungen pro Zyklus und die Einträge des Reorder Buffers aus.

Die Parameter wirken sich direkt auf den Flächenbedarf und die Performanz der Prozessorarchitektur aus. Sie dienen zum einem zum Erforschen eines geeigneten Parametersatzes als auch zur Anpassung der Performanz der Prozessorarchitektur an die Bedürfnisse der Zielanwendungen. Einige Parameter wirken sich direkt auf die Befehlssatzarchitektur aus. Somit muss die Flexibilität auch vom Softwareframework, also vom Compiler und Assembler, unterstützt werden.

3.6 Charakterisierung

In Kapitel 3.2 wurden die Ziele für die Architektur festgelegt. In diesem Kapitel wird nun der vorgestellte Ansatz anhand der aufgestellten Zeile charakterisiert.

Adaptive Ausnutzung von ILP und TLP Durch die Rekonfiguration bzw. das Zusammenschalten von Prozessorressourcen zu virtuellen Prozessorinstanzen ist es möglich, eine flexible Anzahl an Ressourcen zu virtuellen Prozessorinstanzen bzw. Threads zuzuweisen. Je nach Menge an Ressourcen kann dadurch die Performanz durch bessere Ausnutzung des verfügbaren ILPs mittels räumlicher Parallelität gesteuert werden. Somit können die gleichen Ressourcen z.B. für viele virtuelle Prozessorinstanzen mit wenig räumlicher Parallelität oder wenigen virtuellen Prozessorinstanzen mit hoher räumlicher Parallelität wiederverwendet werden. Somit kann sich die Architektur dynamisch auf den verfügbaren ILP oder TLP einer Anwendung adaptieren und diesen dynamisch ausnutzen.

Partiell dynamische Rekonfiguration von ILP und TLP Das Zusammenschalten der Prozessorressourcen kann partiell und dynamisch zur Laufzeit geschehen. Somit kann sich die Architektur auf verändernde ILP/TLP Anforderungen innerhalb einer Anwendung anpassen. Der sequentielle Teil einer Anwendung kann z.B. durch mehr räumliche Parallelität beschleunigt werden, während der parallele Teil von mehreren virtuellen Prozessorinstanzen profitiert.

Skalierbarkeit Die zusammenschaltbaren modularen Ressourcen zur dynamischen Ausnutzung von ILP und TLP wurde sowohl funktional als auch bezüglich der physikalischen Realisierung skalierbar gehalten. Funktionell ist das Design durch seine homogenen Strukturen unabhängig von der maximalen Anzahl an verfügbaren Ressourcen. Somit wurde eine Architektur mit homogenen Strukturen vorgestellt, die die Vorteile von Homogenität mit den Vorteilen der Heterogenität mittels Rekonfiguration kombiniert.

Aus Sicht der praktischen Realisierung wurde ein Prozessordesign vorgestellt, das in möglichst gleichartige Module unterteilt werden kann, die dann im Rahmen einer Backend Realisierung des Prozessors identisch realisiert werden können. Die dadurch entstehenden langen Signallaufzeiten zwischen den Modulen wurde mittels eines neuen Mikroarchitekturkonzept kompensiert. Durch die

Kombination von einer superskalaren Prozessorpipeline mit einer VLIW Befehlssatzarchitektur konnte sowohl eine effiziente Partitionierung der funktionellen Einheiten wie auch des Registerspeichers realisiert werden. Durch die Wiederverwendung gleichartiger Module wird die Designkomplexität insbesondere bei großen Architekturen mit vielen Modulen reduziert und somit die praktische Skalierbarkeit sichergestellt.

Interruptfähigkeit Das Zusammenschalten der Ressource wurde mittels einer zusammenschaltbaren, interruptfähigen Prozessorpipeline realisiert. Die Interruptfähigkeit wurde mittels Konzepten aus superskalaren Pipelines mit dynamischem Scheduling sichergestellt. Hierbei stellt unter anderem ein Reorder Buffer innerhalb der Completion Unit die Wiederherstellung der korrekten Reihenfolge der Operationen und die Behandlung von Exceptions während der Abarbeitung sicher. Zur Wahrung der Performanz unterstützt die Pipeline die out-of-order Behandlung von Exceptions. Außerdem werden verschachtelte Exceptions unterstützt und somit die Grundvoraussetzung für die Ausführung von Betriebssystemen auf der Architektur geschaffen.

Designzeit-Flexibilität Die Prozessorarchitektur bietet zwei verschiedene Arten der Designzeit-Flexibilität an. Zum einen ist die Architektur durch verschiedenen Design-Parameter parametrierbar. Hierdurch kann ein Tradeoff zwischen Performanz, Flächen- und Energiebedarf zur Designzeit gefunden und an die Bedürfnisse der Zielanwendungen angepasst werden. Zusätzlich bietet das superskalare Prozessordesign eine einfache Möglichkeit zum Hinzufügen neuer Datenpfade und Operationstypen. Insbesondere können die neuen Datenpfade einen flexiblen Delay und eine flexible Anzahl an Pipeline-Stufen haben. Mögliche Konflikte werden durch die Mechanismen der out-of-order Ausführung ohne zusätzliche Änderungen in der restlichen Pipeline behandelt.

Programmierbarkeit Die neue Flexibilität der Architektur zur Lauf- bzw. Designzeit spiegelt sich allerdings auch im Interface zur Software, der Befehlssatzarchitektur, wieder. Diese ist zum einen auch abhängig von der Laufzeit-Konfiguration und zum anderen muss die Parametrierbarkeit und Erweiterbarkeit auf Operationsebene unterstützt werden. In dieser Arbeit wird gezeigt, dass man heutige Programmierwerkzeuge an die Flexibilität der neuen Prozessorarchitektur anpassen kann und somit auch die Programmierbarkeit und Praxistauglichkeit des gesamten Ansatzes zu demonstrieren. Hierzu wird zunächst in der nächsten Abschnitt die Anforderung seitens der Architektur an die Programmierbarkeit definiert.

3.7 Anforderungen an die Programmierbarkeit

In diesem Kapitel wurde ein neuartiger Ansatz einer skalierbaren, interruptfähigen, flexiblen Architektur vorgestellt, die zur Laufzeit dynamisch ILP und TLP abhängig

von der Anwendung ausnutzen kann. Allerdings benötigt die Architektur zur Gewährung ihrer Programmierbarkeit in einer hohen Programmiersprache, wie z.B. C, ein flexibles Softwareframework, das zum einen die rekonfigurierbare RSIW Befehlssatzarchitektur als auch die Auswirkungen der Designzeit-Flexibilität auf die Befehlssatzarchitektur unterstützt. Nur wenn diese Kriterien eines Softwareframeworks erfüllt sind, lassen sich auch Applikationen entwickeln, die auf der Architektur effizient ausgeführt werden können. Zur Realisierung der Programmierbarkeit werden daher seitens der Hardware folgende Anforderungen an das Softwareframework gestellt:

Unterstützung der rekonfigurierbaren RSIW Befehlssatzarchitektur Ein Softwareframework muss in der Lage sein, Binärcode für die Prozessorarchitektur aus einer Hochsprache erzeugen zu können. Dazu bedarf es eines RSIW-Compilers, -Assemblers und -Linkers, die zusammen sämtliche Konfigurationen der rekonfigurierbaren Befehlssatzarchitektur unterstützten. Die rekonfigurierbare RSIW Befehlssatzarchitektur zeichnet sich dadurch aus, dass sich je Konfiguration ebenfalls die Befehlssatzarchitektur ändert. Somit müssen sämtliche Werkzeuge nicht nur eine Befehlssatzarchitektur sondern eine Menge an Befehlssatzarchitekturen unterstützten. Dabei reicht die Bandbreite von RISC-, über VLIW- zu Clustered-VLIW-Befehlssatzarchitekturen. Zusätzlich muss der Compiler die Kahrisma spezifischen Besonderheiten, die insbesondere durch die neuartige Kombination einer superskalaren Prozessorpipeline mit dynamische Scheduling mit einer VLIW-artigen Befehlssatzarchitektur ergeben haben, unterstützten.

Unterstützung der dynamischen Änderung der Konfiguration zur Laufzeit einer Anwendung Neben der Unterstützung der RSIW Konfigurationen im Compiler, Assembler und Linker wird ebenfalls ein neuartiges Programmiermodell für die Entwicklung von Anwendungen benötigt, das es erlaubt, die Konfiguration einer Anwendung zur Laufzeit, entweder automatisch oder durch den Entwickler gesteuert, zu wechseln. Erst dadurch ist es möglich, den Ressourcenbedarf individuell an variierende Bedürfnisse einer Anwendung zur Laufzeit anzupassen.

Parametrierbarkeit der Befehlssatzarchitektur Neben der Flexibilität zur Laufzeit der Architektur, die sich in den verschiedenen RSIW Konfigurationen in der Befehlssatzarchitektur widerspiegeln, muss auch die Parametrierbarkeit der Architektur unterstützt werden. Erst damit wird es möglich, zur Designzeit die Komplexität der einzelnen Ressourcen der Architektur an die Bedürfnisse der Zielanwendungen anzupassen.

Erweiterbarkeit des Befehlssatzes Durch das superskalare Design bietet die Architektur eine vergleichsweise einfache Möglichkeit zur Befehlssatzerweiterung an, da neue Operationen bezüglich der Ausführungsdauer nicht kompliziert in die Pipeline integriert werden müssen. Stattdessen übernimmt das superskalare Design das dynamische Scheduling und bietet somit eine einfache Erweiterbarkeit der diversitären Datenpfade an. Eine einfache Erweiterbarkeit des Befehlssat-

zes zur Optimierung der Architektur auf eine oder eine Menge von Anwendungen muss allerdings ebenfalls vom Softwareframework unterstützt werden. Nur wenn der Aufwand zur Unterstützung neuer Operationen im Compiler, Assembler und Linker einfach möglich ist, kann diese Feature auch effizient ausgenutzt werden.

3.8 Zusammenfassung

Heutige statische Multiprozessorsysteme haben eine feste Anzahl an Prozessoren und Funktionseinheiten pro Prozessor. Somit können sie nur Anwendungen mit einer vorher festgelegten *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) und *Parallelität auf Threadebene* (TLP, engl. *Thread-Level Parallelism*) effizient ausführen. Anwendungen mit anderen ILP/TLP-Charakteristiken werden folglich ineffizient auf statischen Multiprozessorsystemen ausgeführt. Daher wurde in diesem Kapitel die Kahrisma-Architektur vorgestellt, die die Beschränkung von heutigen Multiprozessorsystemen aufhebt und eine dynamische Ausnutzung von ILP und TLP erlaubt. Somit ist die Architektur nicht mehr auf eine ILP/TLP-Charakteristik optimiert, sondern kann sich zur Laufzeit auf unterschiedliche Parallelitätsanforderungen der Anwendungen anpassen, zur Steigerung der Performanz oder Reduktion der Ressourcen bzw. des Energieverbrauchs.

Es wurde das Konzept, die Befehlssatzarchitektur und die Mikroarchitekur der Kahrisma-Architektur vorgestellt. Diese setzten die aufgestellten Ziele bezüglich (1) der Wiederverwendbarkeit von Ressourcen zur adaptiven Ausnutzung von ILP und TLP, (2) die partiell dynamische Rekonfiguration von ILP und TLP zur Laufzeit einer Anwendung, (3) die funktionale und praktische Skalierbarkeit des Konzeptes, (4) die Interruptfähigkeit zur Unterstützung von komplexen Steuer- und Betriebssystemfunktionen und (4) die Designzeit-Flexibilität um.

Allerdings konnten die Ziele nur mit einer neuen Mikroarchitektur umgesetzt werden, die superskalare Techniken mit dynamischen Scheduling mit einer rekonfigurierbaren Befehlssatzarchitektur verknüpft. Dadurch ist die dynamische Ausnutzung von ILP und TLP nicht mehr transparent zu der darauf laufenden Software sondern muss zur Gewährleistung der Programmierbarkeit dieser Architektur von einem Softwareframework bestehend aus Compiler, Assembler und Linker unterstützt werden. Zusätzlich muss das Softwareframework die Auswirkungen der angestrebten Designzeit-Flexibilität auf die Programmierung unterstützten, damit die Flexibilität auch sinnvoll verwendet werden kann.

In dieser Arbeit wird gezeigt, dass mit einem neuen Programmiermodell für Prozessoren mit verschiedenen Befehlssatzarchitektur und einem Softwareframework, das dieses Programmiermodell umsetzt, die Programmierung der Kahrisma-Architektur unter dynamische Ausnutzung von ILP und TLP möglich und effizient eingesetzt werden kann.

4 Konzeption des Softwareframeworks

4.1 Ziele

Das Hauptziel ist die Entwicklung einer Hardware- und Software-Architektur, die eine dynamische Ausnutzung von TLP und ILP erlaubt. Dafür wurden in Kapitel 3.3 die Konzepte zur Laufzeitadaption der Hardware basierend auf einer rekonfigurierbaren Befehlssatzarchitektur vorgestellt. Um auch die Programmierbarkeit der Prozessorarchitektur gewährleisten zu können, wird dementsprechend ein Softwareframework benötigt, das die gleiche Flexibilität wie die Hardware zur Laufzeit als auch zur Designzeit aufweist. Nur dann können die Vorteile der Hardware gegenüber statischer MPSoC-Architekturen auch gewinnbringend eingesetzt werden. Dementsprechend hat das Softwareframework das Ziel, die Programmierbarkeit zur dynamischen Ausnutzung von TLP und ILP zu zeigen.

Dazu werden zunächst im nächsten Abschnitt 4.2 die Anforderung des Softwareframeworks beschrieben. Basierend auf den Anforderungen werden danach in Abschnitt 4.3 die Konzepte des mixed-ISA Softwareframework vorgestellt. Das daraus resultierend Softwareframework wird in Abschnitt 4.4 vorgestellt. Daraus folgt in Abschnitt 4.5 die Beschreibung der neuartigen mixed-ISA Programmiermodelle, die durch das Framework realisiert werden können. In Abschnitt 4.6 wird dann auf die individuellen Anforderungen der Komponenten des Frameworks eingegangen, die über den Stand der Technik hinausgehen. Abschnitt 4.7 fasst das Kapitel zusammen.

4.2 Anforderungen

Um die Programmierbarkeit der Kahrisma-Architektur sicherzustellen, wurden folgende Anforderungen an das Softwareframework festgelegt:

Statische mixed-ISA Programmierbarkeit Das Ziel des Softwareframeworks, die Programmierbarkeit der Prozessorarchitektur zur dynamischen Ausnutzung von TLP und ILP sicherzustellen, ist nur durch die Unterstützung der rekonfigurierbaren RSIW Befehlssatzarchitektur möglich. Im Vergleich zu einer statischen, nicht-rekonfigurierbaren Befehlssatzarchitektur muss dabei das gesamte Framework sämtliche Ausprägungen der rekonfigurierbaren Befehlssatzarchitektur unterstützten. Eine spezifische Ausprägung der rekonfigurierbaren Be-

fehlssatzarchitektur kann dabei wie eine statische Befehlssatzarchitektur verstanden werden. Somit muss das Framework parallel mehrere Befehlssatzarchitekturen, d.h. sämtliche Konfigurationen der rekonfigurierbaren Befehlssatzarchitektur, unterstützten. Die parallele Unterstützung von mehreren ISAs wird in dieser Arbeit allgemein als „**mixed-ISA**" bezeichnet. Bei der statischen mixed-ISA Programmierbarkeit kann man die ISA nur beim Start einer Anwendung wählen und diese bleibt dann für die gesamte Laufzeit unverändert.

Dynamische mixed-ISA Programmierbarkeit Neben der statischen Programmierbarkeit der rekonfigurierbaren Befehlssatzarchitektur, soll auch die dynamische Programmierbarkeit unterstützt werden. D.h. die Programmierung von Anwendungen, die zur Laufzeit die Befehlssatzarchitektur und somit die Konfiguration wechseln können. Dadurch kann auf unterschiedlichen ILP/TLP-Charakteristika innerhalb der Anwendung reagieren werden. Damit wird eine effiziente Ausnutzung der Ressourcen über die Zeitspanne einer Anwendung ermöglicht. Dabei ist ebenfalls entscheidend, wie der Endbenutzer die dynamische Programmierbarkeit kontrollieren kann. Nur wenn dieser die dynamische Rekonfigurierbaren innerhalb einer Anwendung mit möglichst einfachen Mitteln steuern (d.h. programmieren) kann, ist eine Reduktion der Ressourcen bzw. Energie oder Steigerung der Performance mit diesem Mittel praktikabel.

Automatische mixed-ISA Programmierbarkeit Eine weitere Steigerung der dynamischen Programmierbarkeit ist die automatische Programmierbarkeit. Hierbei soll die Entscheidung, welche Konfiguration d.h. ISA wann innerhalb der Anwendungen eingesetzt werden soll, vom Endbenutzer in das Softwareframework verlagert werden. Innerhalb des Frameworks soll automatisch eine effiziente Partitionierung der Anwendung bezüglich der Befehlssatzarchitektur unter Vorgabe von Optimierungskriterien vorgenommen werden.

Retargierbarkeit bezüglich der Designzeit-Flexibilität Ein weiteres Ziel, das in Kapitel 3.2 vorgestellt wurde, ist die Designzeit-Flexibilität der Hardware. Hierbei soll zum einen die Erweiterbarkeit der Hardware bezüglich neuer Operationen bzw. Datenpfaden als auch die Parametrisierung anhand von verschiedenen Designparametern sichergestellt werden. Damit diese Flexibilität der Hardware auch in der Software erfolgreich umgesetzt werden kann, ist die Retargierbarkeit des Softwareframeworks bezüglich der Designparameter und neuen Operationen unerlässlich. Nur in diesem Fall kann ein effizienter Parametersatz als auch eine sinnvolle Erweiterung der Operationen sowohl in der Mikroarchitektur als auch im Befehlssatz sinnvoll umgesetzt werden.

4.3 Konzepte

Die Anforderungen aus dem vorherigen Abschnitt können nur mit neuartigen Konzepten innerhalb des gesamten Softwareframeworks erreicht werden, die über den

aktuellen Stand der Technik hinausgehen und in ihrer Gesamtheit neuartig sind. In den folgenden Unterabschnitten werden die neuartigen Konzepte des Softwareframeworks vorgestellt:

4.3.1 Retargierbarkeit durch eine mixed-ISA Architekturbeschreibungssprache

Wie in den Grundlagen in Kapitel 2.3 dargestellt, werden Architekturbeschreibungssprachen schon seit Jahrzehnten als Grundlage für die Realisierung von retargierbaren Softwareframeworks verwendet. Häufig wird hierfür die Technik der Metaprogrammierung eingesetzt, um von einem statischen Framework zu einem retargierbaren Framework zu gelangen. Dabei werden automatisiert Teile des Quellcodes des Frameworks unter Verwendung der ADL generiert. Allerdings gibt es auch Frameworks, die eine automatisch Retargierbarkeit durch die Erweiterung der Algorithmen innerhalb des Frameworks direkt unterstützten. Der Vorteil ist dabei, dass man die Komponenten des Frameworks nicht nach einer Änderung der Architekturbeschreibung neu übersetzen muss und somit eine zügigere Retargierung vorgenommen werden kann.

Das Konzept der Retargierbarkeit des Softwareframeworks mittels einer ADL kann in diesem Fall zur Erfüllung der statischen mixed-ISA Programmierbarkeit als auch der Retargierbarkeit der Designzeit-Flexibilität verwendet werden. Bei der statischen Programmierbarkeit muss jeweils für eine Konfiguration eine ADL-Beschreibung vorhanden sein. Somit kann durch Wechseln der Beschreibung innerhalb eines benutzerretargierbaren Compilers jeweils Code für eine andere Konfiguration generiert werden. Dabei wird die Architekturbeschreibungssprache zur Konfigurationsbeschreibung genutzt. Die Designzeit-Flexibilität ist dabei implizit enthalten, da man alle Designzeitparameter ebenfalls in den einzelnen Konfigurationsbeschreibungen verwenden kann.

Allerdings ist das Problem hierbei, dass die dynamische mixed-ISA Programmierbarkeit eine tiefgreifende Unterstützung der verschiedenen Befehlsastarchitekturen im Softwareframework verlangt. Nur wenn der Compiler innerhalb der Codegenerierung die jeweilige Konfiguration bzw. Befehlssatzarchitektur umschalten kann, wird eine dynamische Programmierung der Architektur ermöglicht. Daher ist das Konzept des mixed-ISA Architekturbeschreibungssprache entstanden. Anstatt nur eine Befehlssatzarchitektur innerhalb einer ADL-Beschreibung zu spezifizieren, werden bei einer mixed-ISA Architekturbeschreibungssprache mehrere Befehlssatzarchitekturen auf einmal beschrieben. Im Kontext von Kahrisma heißt das Übertragen, dass eine ADL-Beschreibung sämtlicher Konfigurationen der rekonfigurierbaren Befehlssatzarchitektur enthält. Alle enthaltenen Befehlssatzarchitekturbeschreibungen innerhalb einer ADL-Beschreibung müssen dann auch parallel in den verschiedenen Werkzeugen unterstützt werden. Zusätzlich müssen die Werkzeuge das Umschalten der Ziel-ISA beherrschen.

4.3.2 Erweiterung der Programmiersprache zur mixed-ISA Annotation

Um die dynamische mixed-ISA Programmierbarkeit realisieren zu können, muss die Eingangssprache so erweitert werden, dass der Benutzter die Konfiguration bzw. Befehlssatzarchitektur für Teile der Anwendung festlegen kann. Hierfür ist eine Erweiterung der Eingangsprogrammiersprache vorgesehen. In diesem Fall wurde die C/C++-Programmiersprache so erweitertet, dass man die Befehlssatzarchitektur für Teile einer Anwendung spezifizieren kann. Als Granularitätsebene wurden Funktionen innerhalb der Programmiersprache ausgewählt, so dass man pro Funktion die jeweilige Ziel-ISA festlegen kann. Es hat sich herausgestellt, dass eine Reduktion der Granularitätsebene auf Subfunktionsebene nur mit erheblichem Zusatzaufwand möglich wäre und sich dies auch nachteilig auf die Abwärtskompatibilität der Programmiersprache ausgewirkt hätte. Eine Festlegung der Granularitätsebene auf Funktionen ist hierbei nicht als eine Beschränkung der Allgemeinheit zu betrachten, da der Endanwender bei diesem Konzept durch das Teilen einer Funktion auch auf Subfunktionsebene die Befehlssatzarchitektur festlegen kann.

Der Compiler muss dann die Erweiterung der Programmiersprache zum einen unterstützen und zum anderen in der Lage sein, während der Kompilierung auf Funktionsebene die Ziel-Befehlssatzarchitektur zu wechseln, da selbst innerhalb einer Übersetzungseinheit verschiedene Befehlssatzarchitekturen auftreten können. Zusätzlich muss bei Funktionsaufrufen darauf geachtet werden, welche Befehlssatzarchitektur die Zielfunktion hat. Weicht die ISA der Zielfunktion von der aufrufenden Funktion ab, so muss zusätzlicher Umschaltcode zur Änderung bzw. Rekonfiguration der ISA generiert werden, so dass vor dem Unterfunktionsaufruf die korrekte ISA auf der Prozessorarchitektur hergestellt wird. Genauso muss nach der Rückkehr vom Unterprogrammaufruf die alte ISA wiederhergestellt werden. Beim Wechsel der ISA muss insbesondere darauf geachtet werden, dass die Registerwerte vor dem ISA-Wechsel gesichert und danach wiederhergestellt werden.

Eine genaue Beschreibung der mixed-ISA C/C++- als auch Compilererweiterungen kann dem Realisierungskapitel 6 entnommen werden.

4.3.3 Profile-Guided mixed-ISA Partitionierung

Zur Realisierung der automatischen mixed-ISA Programmierbarkeit ist eine profileguided Partitionierung der Anwendung nach Befehlssatzarchitekturen vorgesehen. Dabei wird zunächst ein Profiling der Anwendung für jede Befehlssatzarchitektur durchgeführt, um die Performanz jeder Funktion einer Anwendung in Abhängigkeit der Befehlssatzarchitektur zu bestimmen. Danach wird basierend auf diesen Daten eine Partitionierung der Anwendung durchgeführt, um ein gutes Performanz/Energie/Ressourcen-Tradeoff zu berechnen. Dabei ist insbesondere der Rekonfigurationsoverhead zu berücksichtigen, so dass die ISA nur gewechselt werden soll, wenn der

Overhead durch eine effizientere Ausführung gerechtfertigt ist. Nach der Partitionierung kann am Ende die Anwendung für die berechnete ISA-Partitionierung kompiliert und ausgeführt werden.

4.3.4 Komponenten des Softwareframeworks

Zur Realisierung der oben genannten Konzepte muss ein Softwareframework entwickelt werden, das die flexible Kompilierung von Anwendungen für die rekonfigurierbare Befehlssatzarchitektur der Kahrisma-Architektur unterstützt. Dieses Softwareframework muss als Kernkomponente einen retargierbaren Compiler enthalten, der anhand einer mixed-ISA Architekturbeschreibungssprache die unterschiedlichen Konfigurationen der RSIW-Befehlssatzarchitektur unterstützt. Wie in gängigen C-Compilern üblich, generiert der Compiler nicht direkt Binärcode für die Zielarchitektur sondern als Zwischenschritt Assembler-Code. Hierbei ist insbesondere hervorzuheben, dass ebenfalls der Assembler verschiedene Befehlssatzarchitekturen unterstützten muss und sich sogar in einer Assembler-Datei unterschiedliche ISAs befinden können. Somit muss in der Assembler-Datei die jeweilige ISA mitgeteilt werden. Zur Übersetzung der Assembler-Datei in eine Objektdatei wird folglich ein mixed-ISA Assembler benötigt, der ebenfalls sämtliche RSIW-ISAs unterstützt. Die Objektdateien werden dann von einem Linker zusammengesetzt. Dabei muss das Format der Objekte unabhängig von einer spezifischen Konfiguration sein, um eine einheitliche Eingangsformat für den Linker bereitzustellen.

Neben dem Compiler, Assembler und Linker ist der Simulator eine weitere wichtige Komponente des Frameworks. Nur wenn der erzeugt Code auch simuliert bzw. evaluiert werden kann, kann zum einen die Funktionsfähigkeit des Frameworks als auch die Funktionsfähigkeit der Anwendung überprüft werden. Daneben ist das Ziel des Simulators möglichst performant zu sein aber trotzdem eine Abschätzung der Zyklenzahl der Kahrisma-Architektur vornehmen zu können. Dann kann der Simulator sowohl für die funktionale Evaluation als auch zur Messung der Performanz herangezogen werden. Zusätzlich muss der Simulator noch Profiling-Informationen während der Ausführung aufzeichnen, die dann für die automatische mixed-ISA Partitionierung verwendet werden können.

Zur Realisierung der automatischen mixed-ISA Partitionierung wurde noch ein ISA-Partitionierer benötigt, der anhand von Profiling-Informationen der verschiedener ISA-Implementierungen eine effiziente Partitionierung bezüglich der Optimierungskriterien (Zielfunktion und Nebenbedingungen) berechnet. Diese Partitionierung wird dann dem Compiler zur erneuten Übersetzung übergeben.

4.4 Aufbau des Softwareframeworks

In Abbildung 4.1 wird ein Überblick über das ADL-basierte, mixed-ISA Softwareframework gegeben, das die Konzepte aus dem vorherigen Abschnitt umsetzt. Das Framework verwendet eine ADL-Beschreibung sowie C/C++-Quellcode als Eingabe. Als Basis für den Compiler des Softwareframeworks wurde der LLVM-Compiler 2.4.3 verwendet. Das C/C++-Frontend der LLVM-Compiler-Infrastruktur übersetzt zunächst den Quellcode in eine *LLVM-Zwischendarstellung* (LLVM-IR, engl. *LLVM Intermediate Representation*). Die LLVM-IR beschreibt den Quellcode durch einen sprachunabhängigen, virtuellen, RISC-ähnlichen Befehlssatz in *Static-Single-Assignment-Darstellung* (SSA, engl. *Static Single Assignment Form*), die Typ- und Datenfluss-Informationen enthält. Die SSA-Darstellung ist sehr vorteilhaft für Compileroptimierungen, die im Middle- und Backend durchgeführt werden.

Das LLVM-Frontend und die LLVM-IR sind nicht vollständig unabhängig von der Zielarchitektur, da C/C++ an sich eine architekturabhängige Programmiersprache ist. So spiegelt der fundamentale Datentyp „int“ typischerweise die Bitbreite der Allzweckregister der Zielarchitektur wieder. Dies beeinflusst insbesondere vordefinierte Präprozessor-Makros. Genauso ist die Byte-Reihenfolge durch Präprozessor-Makros definiert und wird verwendet, um plattformunabhängigen Quellcode zu schreiben. Während der Vorverarbeitung im Compiler-Frontend werden die Makros aufgelöst und portabler Code wird unwiederbringlich plattformabhängig. Daher benötigt das C/C++-Frontend plattformspezifische Informationen aus der ADL. Dies umfasst unter anderem die Größe und Ausrichtung der fundamentalen Datentypen, die Byte-Reihenfolge und das Gleitkommaformat.

Das Middleend führt optional architekturunabhängige Optimierungen auf der LLVM-Zwischendarstellung durch, wie z.B. Entfernung von nicht genutztem Code, das Propagieren von Konstanten oder Schleifenausrollen. Es ist daher unabhängig von der Architekturbeschreibung innerhalb der ADL. Danach wird die LLVM-IR als Eingang für das Backend verwendet, in dem die Zwischendarstellung in Maschinencode der Zielarchitektur übersetzt wird. Das gesamten Backend ist hochgradig abhängig von der Befehlssatzarchitektur der Zielarchitektur und wurde daher mit Hilfe der ADL retargierbar gehalten. Dabei wurde die Technik der Metaprogrammierung verwendet, so dass aus der ADL-Beschreibung automatisiert Teile des Quellcodes des Backends generiert werden. Als Ergebnis generiert das Backend abschließend Assemblerdateien für die Zielarchitektur, wobei die final erzeugte Syntax ebenfalls in der ADL spezifiziert ist.

Die Binärwerkzeuge (engl. *Binary Utilities*) bestehen aus einem Assembler und Linker, der die Assemblerdateien in eine ausführbare Datei für die rekonfigurierbare Architektur übersetzt. In einem ersten Schritt wird jede Assemblerdatei einzeln in eine Objektdatei übersetzt. Danach werden sämtliche Objektdateien zu einer finalen ausführbaren Dateien gelinkt.

Die ausführbare Datei kann dann wahlweise vom Core- oder System-Simulator si-

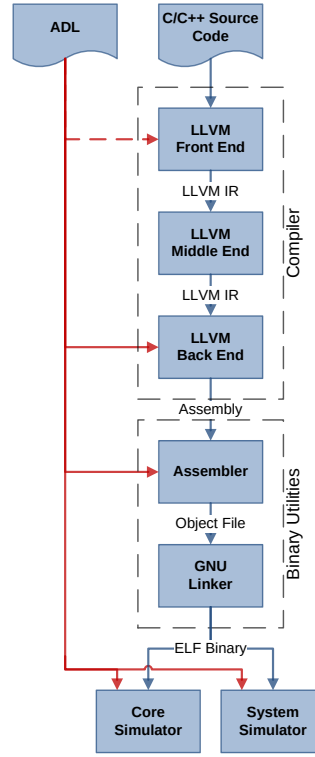


Abbildung 4.1: Überblick über das ADL-basierte Softwareframework

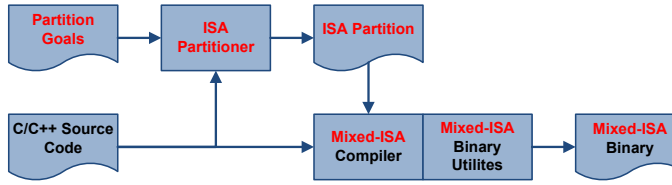
muliert werden. Der Core-Simulator ist ein verhaltensorientierter *Befehlssatzsimulator* (ISS, engl. *Instruction Set Simulator*), der die rekonfigurierbare Befehlssatzarchitektur einer virtuellen Prozessorinstanz nach der Beschreibung innerhalb der ADL simulieren kann. Obwohl der Simulator nur den Befehlssatz und nicht den Aufbau einer virtuellen Prozessorinstanz simuliert, ist er in der Lage, die Performanz einer solchen Instanz zu approximieren. Dadurch wird ein guter Tradeoff zwischen Performanz und Genauigkeit erzielt und der Simulator kann sowohl effizient für die funktionale Evaluation des Frameworks als auch zur Performanzabschätzung des kompilierten Codes abhängig von der Konfiguration verwendet werden.

Neben dem Core-Simulator, der nur eine virtuelle Prozessorinstanz simulieren kann, existiert noch ein System-Simulator, der die gesamte Kahrisma-Architektur mit mehreren virtuellen Prozessorinstanzen (Kernen) simuliert. Hierbei werden mehrere Core-Simulatoren zusammengeschaltet und sowohl das Kommunikationsnetzwerk als auch die Ressourcenverbrauch der Konfiguration für die gesamte Architektur simuliert. Der Simulator kann dann für die Entwicklung paralleler Anwendungen verwendet werden.

4.5 Mixed-ISA Programmiermodelle

(a) Statische mixed-ISA Programmiermodell

(b) Dynamische mixed-ISA Programmiermodell



(c) Automatische mixed-ISA Programmiermodell

Abbildung 4.2: Kahrisma mixed-ISA Programmiermodelle

Um die neue Flexibilität des Softwareframeworks zur dynamischen Ausnutzung von ILP und TLP auch durch den Programmierer ausnutzen zu können, benötigt der Programmierer eine Möglichkeit entscheiden zu können, zu welchem Zeitpunkt innerhalb der Anwendung welche Konfiguration bzw. Befehlssatzarchitektur verwendet wird. Nur dann kann der Programmierer durch eine geeignete Auswahl der Befehlssatzarchitektur einen effizienten Tradeoff zwischen Ressourcen-, Energieverbrauch und Performanz erzielen.

Zu diesem Zweck bietet das Softwareframework drei Programmiermodelle an, die in Abbildung 4.2 dargestellt sind. Die drei Programmiermodelle werden im Folgenden erklärt:

Statische mixed-ISA Programmiermodell In Abbildung 4.2(a) wird zunächst das statische Programmiermodell gezeigt. Hierbei wird C/C++-Quellcode mittels eines Compilers und den Binärwerkzeugen in eine ausführbare Datei übersetzt. Der Compiler übersetzt die Anwendung automatisch für sämtliche Befehlssatzarchitekturen. Beim Start einer Anwendung kann dann die jeweilige Befehlssatzarchitektur ausgewählt werden, die dann während der Ausführung immer konstant bleibt.

Dynamische mixed-ISA Programmiermodell In diesem Programmiermodell kann der Programmierer über die Eingangssprache kontrollieren, welche Befehlssatzarchitektur in welchen Teilen der Anwendung verwendet wird. Dazu kann der Entwickler die Befehlssatzarchitektur pro Funktion im C/C++-Quellcode annotieren. Der Compiler verwendet die vom Programmierer vorgegebenen Partitionierung der Anwendung nach Befehlssatzarchitekturen und übersetzt jede Funktion nach den Vorgaben des Programmierers. Bei einem Wechsel zwischen Befehlssatzarchitekturen fügt der Compiler automatisch zusätzlichen Umschaltcode zum Wechsel der ISA bzw. zur Rekonfiguration ein.

Automatisches mixed-ISA Programmiermodell Bei dem automatischen mixed-ISA Programmiermodell wird die Entscheidung, welche Funktion welche Befehlssatzarchitektur verwenden soll, nicht vom Programmierer direkt getroffen sondern von einem neuem Werkzeug, dem ISA-Partitionierer, getroffen. Der Programmierer legt nur die Optimierungskriterien fest und der ISA-Partitionierer berechnet anhand von Profiling-Informationen automatisch eine effiziente ISA-Partitionierung. Die Informationen über die ISA-Partition werden innerhalb eines zweiten Kompilierdurchlaufs zur Generierung der automatisch partitionierten mixed-SA Anwendung verwendet.

4.6 Anforderungen der Komponenten

In diesem Kapitel wird genauer auf die einzelnen Komponenten des Softwareframeworks sowie ihre individuellen Anforderungen und Herausforderungen eingegangen, die innerhalb dieser Arbeit umgesetzt und gelöst werden mussten. Das LLVM-Frontend und -Middleend wird dabei aus der LLVM-Compiler-Infrastruktur verwendet und wird daher nicht näher in diesem Abschnitt erklärt.

4.6.1 Mixed-ISA Architekturbeschreibungssprache

Wie bereits in den Konzepten des Frameworks in Abschnitt 4.3 beschrieben, wird die ADL benötigt um zum einen die rekonfigurierbare Befehlssatzarchitektur im Framework zu unterstützt und zum anderen um eine Erweiterbarkeit der Prozessorarchitektur zu gewährleisten. Für die rekonfigurierbare Befehlssatzarchitektur unterstützt die ADL insbesondere die gleichzeitige Beschreibung von verschiedenen Befehlssatzarchitekturen, die jeweils einer Konfiguration der rekonfigurierbaren Befehlssatzarchitektur entspricht. Die ADL wird dann von den verschiedenen Komponenten des Frameworks zur Realisierung der Retargierbarkeit derer verwendet. Je nach Komponenten bestehen deswegen unterschiedliche Anforderungen an den Inhalt und der beschriebenen Daten innerhalb der ADL. Diese Anforderungen werden nun, aufgeschlüsselt nach Komponente, dargelegt:

C/C++-Frontend Das LLVM-Frontend benötigt nur wenige Informationen über Eigenschaften der Zielarchitektur, die die C/C++-Eingangssprache beeinflussen. Dies sind insbesondere die Größe und Ausrichtung der fundamentalen Datentypen, die Byte-Reihenfolge und das Gleitkommaformat.

Compiler Backend Das LLVM-Backend führt die Codegenerierung durch. Dabei wird Schritt für Schritt die LLVM-IR in Assembler-Code der Zielarchitektur überführt. Dazu benötigt das Backend insbesondere Informationen über die Befehlssatzarchitektur, das *Binärschnittstelle* (ABI, engl. *Application Binary Interface*) sowie den Assembler-Syntax. Dazu muss die ADL eine Beschreibung sämtlicher Befehle der Prozessorarchitektur, der Register und der Calling Convention enthalten. Pro Befehl muss eine Verhaltensbeschreibung, implizit verwendete Register und Assembler-Syntax spezifiziert sein. Für VLIW-Befehlssatzarchitekturen müssen zusätzlich der Aufbau einer Instruktion sowie eine Beschreibung, welche Befehle in eine Instruktion zusammengefasst werden dürfen, vorhanden sein. Für Clustered-VLIW-Prozessoren werden zusätzlich eine Aufteilung der Cluster sowie eine Beschreibung der Inter-Cluster-Kommunikation benötigt.

Assembler Der Assembler benötigt von der ADL ebenfalls eine Liste sämtlicher Befehle. Neben der Assembler-Syntax ist hier allerdings der genaue Aufbau eines Befehls notwendig, so dass der Assembler die Binärdarstellung eines jeden Befehls erzeugen kann. Für VLIW-Befehlssatzarchitekturen wird weiterhin der Aufbau der Instruktion, bestehend aus mehreren Befehlen, benötigt.

Core-Simulator Der Core-Simulator simuliert die Befehlssatzarchitektur einer virtuellen Prozessorinstanz. Dazu benötigt dieser das Instruktions- und Befehlsformat der ADL, um zunächst die einzelnen Befehle aus der ausführbaren Datei dekodieren zu können. Daneben muss der interne Zustand des Prozessors, also sämtliche Register, modelliert sein. Jede Instruktion führt eine Modifikation des internen Zustands des Prozessors durch und kommuniziert optional mit der Außenwelt des Prozessors. Diese Modifikation wird anhand der Verhaltensbeschreibung sämtlicher Befehle einer Instruktion in der ADL festgelegt. Zusätz-

lich werden Performanzinformationen über Befehle benötigt, um den zeitlichen Verlauf der Befehlsabarbeitung approximieren zu können.

System-Simulator Der System-Simulator simuliert die Kahrisma-Architektur auf Systemebene. Dabei setzt er auf einzelne Core-Simulatoren auf. Für die Simulation auf Systemebene ist daher zusätzlich eine abstrakte, strukturelle Beschreibung des Gesamtsystems nötigt. Als besonderes Feature muss die ADL auf Systemebene die Rekonfiguration beschreiben können.

Anhand der verschiedenen Anforderungen, die sich von den einzelnen Komponenten des Frameworks ergeben, hat sich eine Zweiteilung der Architekturbeschreibungssprache in einer Core- und System-ADL als sinnvoll erwiesen. Die beiden ADLs, die jeweils eine unterschiedliche Abstraktionsebene beschreiben, werden im Folgenden näher beschrieben. Mit dieser Aufteilung wird auch die Herausforderung der Beschreibung einer rekonfigurierbaren Prozessorarchitektur angegangen. Eine genaue Spezifikation der ADL befindet sich in Kapitel 5.

4.6.1.1 Core-ADL

Die Core-ADL erlaubt die Beschreibung von verschiedenen virtuellen Prozessorinstanzen der Kahrisma-Architektur. Sie enthält somit eine formale Spezifikation des Verhaltens jeder möglichen Kahrisma-Konfiguration sowie zusätzlich Informationen für den Compiler, die zum Erzeugen von Code für die unterschiedlichen Konfigurationen nötig ist. Dies umfasst neben allgemeinen globalen Parameter insbesondere die Beschreibung des Instruktions- und Befehlsformats, der Register, des Application Binary Interfaces inklusive Calling Conventions sowie das Verhalten und die Kodierung sämtlicher Befehle.

4.6.1.2 System-ADL

Die System-ADL bietet eine strukturelle Beschreibung des Gesamtsystems der Kahrisma-Architektur auf einer hohen Abstraktionsebene. Dabei werden die einzelnen Module der Prozessorarchitektur nach den Grundprinzipien von Hardwarebeschreibungssprachen in Form von Modulen, Instanzen und Verbindungen untereinander modelliert. Das exakte Verhalten innerhalb eines Moduls ist allerdings nicht in der ADL-Beschreibung sondern als Bibliothek im System-Simulator enthalten. Die ADL verweist dazu lediglich auf die Elemente der Bibliothek im System-Simulator.

Zur Unterstützung der Rekonfiguration können zum einen Module als rekonfigurierbar gekennzeichnet und zum anderen Konfigurationen beschrieben werden. Eine Konfiguration ist durch eine Untermenge an Modulen und Verbindungen gekennzeichnet, die mit den als rekonfigurierbar gekennzeichneten Modulen abgeglichen werden können. Dadurch werden implizit sämtlichen möglichen Instanzen von Konfiguration spezifiziert. Weiterhin ist jede Konfiguration einer Befehlssatzarchitektur

aus der Core-ADL zugewiesen. Auf diese Weise wird eine Verbindung zwischen der Struktur und benötigten Ressourcen einer Konfiguration und ihrem Verhalten hergestellt.

4.6.2 Mixed-ISA Compiler

Der mixed-ISA Compiler übersetzt ein C/C++-Programm in Assembler-Dateien, die dann von den mixed-ISA Binärwerkzeugen in eine finale, ausführbare Datei übersetzt werden. Aktuelle Compiler lassen sich in ein Frontend, Middleend und Backend unterteilen. Dies hat den Vorteil, dass sie durch das Austauschen des Frontends unabhängig von der Eingangssprache sind und durch Austauschen den Backends unabhängig von der Zielarchitektur. Zur Realisierung der oben genannten Konzepte und der Programmiermodelle heben sich die Komponenten des Compilers wie folgt von klassischen Compilern für statischen, nicht-rekonfigurierbare Architekturen ab.

4.6.2.1 C/C++-Frontend

Das C/C++-Frontend übersetzt zunächst die C/C++-Eingangssprache in eine Zwischendarstellung. Dabei ist das Frontend teilweise von der Zielarchitektur abhängig und muss durch die Beschreibung innerhalb der ADL angepasst werden. Das Frontend muss die Erweiterung der C/C++-Programmiersprache zur Annotation der Befehlssatzarchitektur innerhalb einer Anwendung zur Realisierung des dynamische mixed-ISA Programmiermodells unterstützen. Die damit realisierte ISA-Partitionierung muss dann in der Zwischendarstellung gespeichert werden, um sie dem Backend bekannt zu geben.

4.6.2.2 Zwischendarstellung

Die Zwischendarstellung ist eine compiler-interne Repräsentation der Anwendung und wird sowohl zwischen dem Front- und Middleend als auch zwischen Middle- und Backend verwendet. Die Zwischendarstellung im LLVM-Compiler (LLVM-IR) hat drei Repräsentationen, die jeweils erweitert werden müssen, um die ISA-Partitionierung des dynamischen mixed-ISA Programmiermodells speichern zu können. Die drei Repräsentationen der LLVM-IR umfassen das Binärdateiformat, Textdateiformat und In-Speicher-Format. Das Binärdateiformat ist besonders platzsparend und kann effizient vom Compiler gelesen und geschrieben werden. Das Textdateiformat ist menschenlesbar aber dafür aufwändiger vom Compiler zu lesen und schreiben. Das In-Speicher-Format umfasst Klassen und Strukturen, die den Inhalt der LLVM-IR im Speicher während des Übersetzens enthalten.

4.6.2.3 Middleend

Das Middleend benötigt keine Erweiterung. Durch die Erweiterung des In-Speicher-Formats der Zwischendarstellung bleiben sämtliche mixed-ISA Informationen während der Optimierungen im Middleend enthalten.

4.6.2.4 Backend

Das Backend übersetzt die Zwischendarstellung zu Assembler-Dateien der Zielarchitektur. Das ist die Komponente des Compilers, die hochgradig von der Zielarchitektur abhängig ist und bedarf deswegen auch am meisten Anpassung zur Realisierung der mixed-ISA Programmierbarkeit der Kahrisma-Architektur. Das Backend muss sowohl die rekonfigurierbare Befehlssatzarchitektur von Kahrisma als auch die definierten unterschiedlichen Programmiermodelle unterstützen. Daher ist es auch stark von der Core-ADL abhängig, die sämtliche Konfigurationen spezifiziert.

Zur Unterstützung der rekonfigurierbaren Befehlssatzarchitektur und der Designzeit-Flexibilität ist das Backend von der Core-ADL abhängig. Dazu werden mittels Metaprogrammierung Teile des Quellcodes des Backends generiert und somit das Backend an die flexiblen Konfigurationen angepasst. Durch die Partitionierung des Prozessors in einzelne Module, entsprechen die meisten Konfigurationen einem Clustered-VLIW-Prozessor. Im Vergleich zu nicht geclusterten VLIW-Prozessoren muss hierbei der Compiler zusätzlich noch sämtliche Befehle und Register zu den Clustern zuordnen.

Bereits für die Unterstützung des statischen mixed-ISA Programmiermodells muss das Backend nicht nur Code für eine Befehlssatzarchitektur, sondern in einem Durchlauf bereits Code für sämtliche unterstützte Befehlssatzarchitekturen gleichzeitig erzeugen. Dabei kann der Compiler nicht einfach für jede Befehlssatzarchitektur separat ausgeführt werden, weil bei jedem Durchlauf die globalen Variablen neu erzeugt werden würden, diese aber nur einmal vorhanden sein dürfen. Daher bedarf es neuer Techniken im Compiler-Backend, die ein Umschalten der Befehlssatzarchitektur zur Laufzeit des Compilers auf Funktionsebene erlauben.

Zur Unterstützung des dynamischen und automatischen mixed-ISA Programmiermodells muss das Backend zusätzlich zum statischen Modell automatisch Code zur Rekonfiguration der Befehlssatzarchitektur generieren. Eine Rekonfiguration kann das Backend vergleichsweise einfach durch Hinzufügen eines speziellen Befehls auslösen. Allerdings müssen nach dem Rekonfigurationsbefehl sämtliche Befehle aus einer anderen Befehlssatzarchitektur generiert werden. Somit muss das Backend für diesen Fall sogar das Umschalten der Codegenerierung auf eine andere Befehlssatzarchitektur innerhalb einer Funktion auf Basisblockebene unterstützen.

4.6.3 Mixed-ISA Binärwerkzeuge

Die Binärwerkzeuge für Kahrisma umfassen einen Assembler und Linker. Der Assembler generiert Objektdateien, die vom Linker zusammen in eine ausführbare Datei gelinkt werden. Sowohl die Objektdateien als auch die ausführbare Datei werden dabei im *Executable and Linkable Format* (ELF)-Format [101, 102] gespeichert. Sowohl der Assembler, Linker und das ELF-Format werden durch die mixed-ISA Konzepte beeinflusst.

4.6.3.1 Mixed-ISA Assembler

Der mixed-ISA Assembler ist genauso wie das Backend stark von der ADL-Beschreibung abhängig. Er übersetzt die Befehle, die in der ADL spezifiziert sind, in deren Binärformat. Dabei muss der Assembler das Umschalten der Befehlssatzarchitektur in der Assembler-Eingangsdatei unterstützen. Innerhalb eines Compilerdurchlaufs wird eine Assembler-Datei erzeugt, die unterschiedliche Befehlssatzarchitekturen beinhaltet. Innerhalb einer Assembler-Datei kann die Befehlssatzarchitektur durch einen speziellen Pseudo-Assemblerbefehl umgeschaltet werden, der die Befehlssatzarchitektur für die Assemblierung der folgenden Befehle festlegt.

4.6.3.2 ISA-unabhängiges ELF-Format

Obwohl das ELF-Format ein Standard für Objektdateien repräsentiert, bleibt das ELF-Format trotzdem von der Befehlssatzarchitektur abhängig. So werden beim Linken Symbole zwischen den Objektdateien aufgelöst und die Symbole auf konkrete Speicheradresse abgebildet. Durch das Festlegen eines Symbols auf eine konkrete Speicheradresse müssen häufig auch die Befehle im Binärformat modifiziert werden. Da die Befehlskodierung pro Befehlssatzarchitektur unterschiedlich ist, werden auch für jede ISA individuelle Modifikationsregeln benötigt. Für das Konzept der Beschreibung mittels der ADL wurde daher ein ISA-unabhängiges ELF-Format definiert, das sämtliche Ausprägungen der Befehlskodierung der ADL zur Modifikation unterstützt.

4.6.3.3 Mixed-ISA Binärformat

Zur Unterstützung von mehreren ISAs innerhalb einer ausführbaren Binärdatei wird basierend auf den Binärwerkzeugen ein mixed-ISA Binärformat definiert. Dieses erlaubt, dass mehrere ISA-Implementierungen einer Anwendung in einer Datei kodiert werden können. Beim Starten der Anwendung kann ausgewählt werden, mit welcher ISA diese ausgeführt werden soll. Dazu zeigt das Startsymbol nicht auf ausführen Code sondern auf eine Sprungtabelle, die für jede ISA eine andere Einsprungadresse in

die Anwendung bietet. Somit ist der Start- und Initialisierungscode der Anwendung für jede ISA vorhanden und kann abhängig von dieser ausgeführt werden. Der Start- und Initialisierungscode ruft dann die main-Funktion in der korrekten ISA auf.

4.6.3.4 Linker

Durch die Definition des ISA-unabhängigen ELF-Formats ist der Linker ebenfalls unabhängig von der ISA. Außer der Unterstützung des ISA-unabhängigen ELF-Formats bedarf es keine weitere Anpassung des Linkers.

4.6.4 Mixed-ISA Simulator

Der mixed-ISA Simulator wird aus folgenden Gründen benötigt:

Funktionale Evaluation Bei der funktionalen Evaluation geht es darum, eine Anwendung oder das Softwareframework auf Korrektheit zu überprüfen. Das Softwareframework kann dadurch auf Korrektheit evaluiert werden, in dem eine korrekte Anwendung kompiliert und simuliert sowie das Ergebnis mit dem erwarteten Ergebnis verglichen wird. Das erwartete Ergebnis kann z.B. durch das Ausführen der Anwendung auf einem anderen Prozessor erzeugt werden. Nur wenn der Compiler, Assembler, Linker und Simulator die Anwendung korrekt übersetzen und abarbeiten, entspricht das Endergebnis danach auch den Erwartungen. Insbesondere im Compiler-Backend können sehr schnell Fehler entstehen, die durch die Simulation dann erkannt werden.

Abschätzung der Performanz Neben der funktionalen Evaluation für das Softwareframework bietet der Simulator noch eine Abschätzung der Performanz an. Damit können die Taktzyklen, die zur Ausführung einer Anwendung auf der Prozessorarchitektur benötigt werden, abgeschätzt werden. Eine Abschätzung bedeutet hier, dass keine taktgenaue Simulation durchgeführt sondern die Performanz mit einem gewissen zeitlichen Fehler approximiert wird. Dadurch wird ein Tradeoff zwischen Performanz und Genauigkeit erzielt, der für die Evaluation der Architektur und des Softwareframeworks ausreichend ist.

Validierung der Hardware Zusätzlich ist das Ziel des Simulators noch die Validierung der Hardware mittels der Erzeugungen von Trace-Dateien. Dazu wird die Änderung des internen Zustandes für jede Instruktion protokolliert. Die Hardware kann dann während der RTL-Simulation mit dem Trace-File des Befehlssatzsimulators verglichen werden. Somit können Fehler in der wesentlich komplexeren Hardware erkannt und instruktionsgenau lokalisiert werden. Dadurch wird die Fehlersuche in der Hardware erheblich vereinfacht.

Erzeugen von Profiling-Informationen Neben Trace-Dateien können während der Simulation auch Profiling-Information erzeugt werden. Diese Informationen ge-

ben Aufschluss darüber, an welcher Stelle eine Anwendung wie viel Zeit verbraucht hat. Neben der Nützlichkeit für den Programmierer sind die Profiling-Informationen die Voraussetzung, um eine automatische ISA-Partitionierung durchzuführen.

Für die Simulation stehen zwei Simulatoren zur Verfügungen. Einen Core- und einen System-Simulator:

4.6.4.1 Core-Simulator

Der Core-Simulator simuliert den RSIW-Befehlssatz einer virtuellen Prozessorinstanz bzw. Konfiguration der Kahrisma-Architektur. Er ist von der Core-ADL abhängig, die die verschiedenen ISAs der rekonfigurierbaren RSIW-ISA beschreibt. Der Simulator unterstützt das ISA-unabhängige ELF-Format zum Laden der Anwendung in den Speicher sowie das mixed-ISA Binärformat. Beim Starten der Simulation kann die ISA ausgewählt werden, mit der die Anwendung ausgeführt werden soll. Zusätzlich bietet der Simulator als besonderes Feature die Simulation von mixed-ISA Anwendungen an. Dies erlaubt das Wechseln der Befehlssatzarchitektur während der Simulation mittels spezieller Befehle, die eine Rekonfiguration auslösen. Daher muss der Core-Simulator zur Laufzeit sämtliche ISAs der rekonfigurierbaren RSIW-ISA unterstützten und die aktive ISA wechseln können.

4.6.4.2 System-Simulator

Der System-Simulator simuliert die Kahrisma-Prozessorarchitektur auf Systemebene. Insbesondere wird, im Vergleich zu System-Simulatoren nicht-rekonfigurierbarer MP-SoCs, hierbei die Rekonfigurierbarkeit der Kahrisma-Architektur ebenfalls simuliert. Dazu wird die Struktur der Architektur in der System-ADL mittels Komponenten modelliert. Einzelne Komponenten können als rekonfigurierbar gekennzeichnet werden. Zusätzlich befindet sich in der System-ADL eine Spezifikation möglicher Konfigurationen, die je nach Konfiguration eine bestimmte Anzahl und Typ an rekonfigurierbaren Komponenten sowie eine bestimmte Konnektivität dieser benötigt. Durch die Verwendung der System-ADL kann ebenfalls der System-Simulator die Designzeit-Flexibilität der Kahrisma-Architektur auf Systemebene umsetzen.

Während der Simulation wird zwischen statischen und rekonfigurierbaren Komponenten der Architektur unterschieden. Statische Komponenten werden von Anfang an mittels Modulen simuliert, während für rekonfigurierbare Komponenten nur deren Verwendung für eine Konfiguration gespeichert wird. Beim Start sind zunächst sämtliche rekonfigurierbare Komponenten ungenutzt. Sobald eine Konfiguration geladen wird, werden zunächst passende ungenutzten rekonfigurierbare Komponenten für die Konfiguration gesucht. Falls vorhanden, werden die Komponenten als konfiguriert markiert und ein Simulationsmodul für die neue Konfiguration im Simulator

instanziiert. Das Simulationsmodul einer Konfiguration entspricht dabei einer Instanz des Core-Simulators, die je nach Konfiguration eine unterschiedliche ISA ausführt. Auf diese Art und Weise kann eine effiziente Simulation der rekonfigurierbaren Architektur auf Systemebene sichergestellt werden.

4.6.5 ISA-Partitionierer

Zur Umsetzung des automatischen mixed-ISA Programmiermodells für die Kahrisma-Architektur bedarf es eines Werkzeugs, das für eine Anwendung eine effiziente ISA-Partitionierung bestimmt. Zur Bestimmung dieser ISA wird zunächst ein Profiling der Anwendung für jede relevante Befehlssatzarchitektur mittels des Core-Simulators durchgeführt und die Daten dem ISA-Partitionierer als Eingabe gegeben. Dieser berechnet unter Einbeziehung des Rekonfigurationsoverheads eine Partitionierung der Anwendung anhand der Befehlssatzarchitektur und unter Berücksichtigung der Zielparameter für die Partitionierung. Als Ergebnis wird für jede Funktion die Befehlssatzarchitektur bestimmt und dem Compiler zur finalen Übersetzung der Anwendung übergeben.

4.7 Zusammenfassung

In diesem Kapitel wurde das Konzept des Kahrisma-Softwareframeworks beschrieben. Das Softwareframework wird benötigt, um die dynamische Ausnutzung von TLP und ILP durch die Hardware auch dem Programmierer zugänglich zu machen. Nur durch das Softwareframework wird die Programmierbarkeit gewährleistet und eine dynamische Ausnutzung von TLP und ILP kann praktisch durchgeführt werden.

Als Anforderungen an das Softwareframework wurde zunächst die statische, dynamische und automatische Programmierbarkeit der Prozessorarchitektur definiert. Während sich die statische Programmierbarkeit auf Anwendungen bezieht, die sich während der Laufzeit nicht an die unterschiedlichen ILP/TLP-Charakteristika anpassen können, wird bei der dynamischen und automatischen mixed-ISA Programmierbarkeit eine Rekonfiguration während der Laufzeit einer Anwendung ermöglicht. Die unterschiedlichen Anforderungen der Programmierbarkeit spiegeln sich äquivalent in den drei vorgestellten Programmiermodellen wieder. Damit kann ein Programmierer unter Auswahl eines geeigneten Programmiermodells mixed-ISA Anwendung in C/C++ für die Kahrisma-Architektur entwickeln und für diese kompilieren.

Die Programmiermodelle werden durch das Softwareframework umgesetzt, das auf einer Architekturbeschreibungssprache basiert. Diese ADL wird zum einen benötigt, um die rekonfigurierbare Befehlssatzarchitektur im Framework zu unterstützt und zum anderen um eine Erweiterbarkeit der Prozessorarchitektur zu gewährleisten. Für die rekonfigurierbare Befehlssatzarchitektur unterstützt die ADL insbesondere die

gleichzeitige Beschreibung von verschiedenen Befehlssatzarchitekturen, die dann jeweils eine Konfiguration der rekonfigurierbaren Befehlssatzarchitektur entsprechen. Die ADL wird dann von den verschiedenen Komponenten des Frameworks zur Realisierung der Retargierbarkeit deren verwendet.

Die Komponenten des Softwareframeworks bestehen aus einem Compiler, Assembler, Linker, Core- und System-Simulator sowie einem ISA-Partitionierer. Der Compiler, Assembler und Linker unterstützen die Entwicklung von mixed-ISA Anwendungen, die mehrere Befehlssatzarchitekturen gleichzeitig unterstützten können. Durch die Wahl der Befehlssatzarchitektur für die Ausführung wird auch gleichzeitig die Konfiguration festgelegt und es kann ein Tradeoff zwischen Performanz, Energieverbrauch und Ressourcenverbrauch dynamisch zur Laufzeit getroffen werden.

Der Core- und System-Simulator erlaubt die Simulation einzelner virtueller Prozessorinstanzen aber auch der gesamten Kahrisma-Architektur. Er dient der funktionalen Evaluation, der Performanzabschätzung, der Validierung der Hardware sowie dem Erzeugen von Profiling-Informationen. Der ISA-Partitionierer erlaubt die automatische Partitionierung eine Anwendung in Teile mit unterschiedlicher Befehlssatzarchitektur. Dabei basiert die Entscheidung auf Profiling-Informationen, die durch den Simulator generiert werden.

5 Mixed-ISA Architekturbeschreibungssprache

In diesem Kapitel wird die mixed-ISA *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*) des Softwareframeworks vorgestellt. Die ADL bildet das Kernstück des Softwareframeworks. Sie wird benötigt, um zum einen die rekonfigurierbare Befehlssatzarchitektur im Framework zu unterstützt und zum anderen, um eine Erweiterbarkeit der Prozessorarchitektur zu gewährleisten. Für die rekonfigurierbare Befehlssatzarchitektur unterstützt die ADL insbesondere die Beschreibung von verschiedenen Befehlssatzarchitekturen gleichzeitig, die dann jeweils einer Konfiguration der rekonfigurierbaren Befehlssatzarchitektur entsprechen. Die ADL wird dann zur Realisierung der Retargierbarkeit der verschiedenen Komponenten des Frameworks verwendet.

Nachfolgend wird zunächst die Einteilung der ADL in Abschnitt 5.1 vorgestellt. In Abschnitt 5.2 wird auf die Datenbeschreibungssprache eingegangen, danach wird die Core-ADL in Abschnitt 5.3 beschrieben und als letztes wird die System-ADL in Abschnitt 5.4 vorgestellt. Abschließend werden die entwickelten ADLs in Abschnitt 5.5 charakterisiert und in Abschnitt 5.6 zusammengefasst.

5.1 Aufbau

Anhand der verschiedenen Anforderungen, die sich aus den einzelnen Komponenten des Frameworks ergeben, hat sich eine Zweiteilung der Architekturbeschreibungssprache in eine Core- und System-ADL als sinnvoll erwiesen. Die beiden ADLs, die jeweils eine unterschiedliche Abstraktionsebene beschreiben, werden im Folgenden näher beschrieben. Mit dieser Aufteilung wird auch das Problem der Beschreibung einer rekonfigurierbaren Prozessorarchitektur angegangen.

Abbildung 5.1 zeigt die Eingruppierung der ADL im Gajski-Kuhn-Diagramm (auch als Y-Diagramm bekannt). Die System-ADL befindet sich auf der obersten Abstraktionsebene und beschreibt den Aufbau der Prozessorarchitektur in der strukturellen Domäne. In dieser Ebene wird festgelegt, aus welchen Modulen bzw. Komponenten eine Prozessorarchitektur besteht und wie diese verbunden sind. Zusätzlich werden die Konfigurationen innerhalb der strukturellen Ebene durch die Ressourcen definiert, die zum Instanziieren dieser Konfiguration notwendig sind. Dadurch ermöglicht die

System-ADL die Beschreibung von rekonfigurierbaren Prozessorarchitekturen wie die Kahrisma-Architektur.

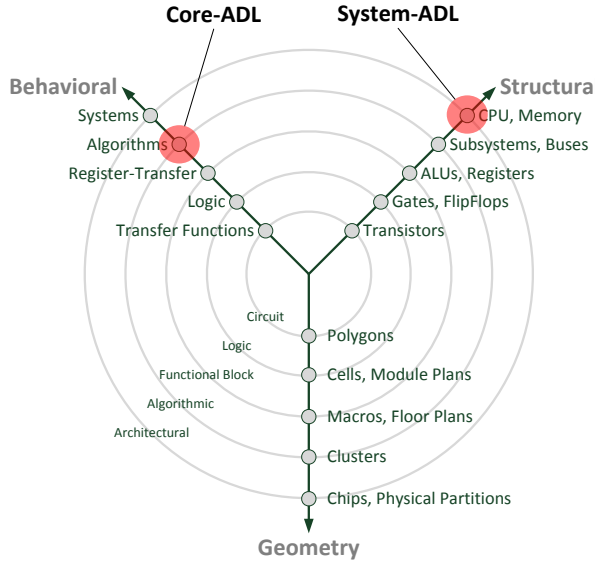


Abbildung 5.1: Eingruppierung der ADL im Y-Diagramm

Eine Konfiguration in der Kahrisma-Architektur zeichnet sich durch eine eigene Befehlssatzarchitektur aus. Das Verhalten einer Konfiguration der System-ADL wird dann in der Core-ADL durch die Spezifikation der Befehlssatzarchitektur festgelegt. Die Core-ADL lässt sich somit in der Verhaltens-Domäne eingruppieren. Da nicht das Verhalten des Systems sondern einer Komponente bzw. Konfiguration beschrieben wird, ist die Core-ADL eine Abstraktionsebene niedriger als die System-ADL angesiedelt. Durch die gemischte Beschreibung von Konfigurationen auf der strukturellen und verhaltensorientierten Domäne auf unterschiedlichen Abstraktionsebenen wird eine einheitliche Beschreibung von rekonfigurierbaren Prozessorarchitekturen wie der Kahrisma-Prozessorarchitektur im Simulationsframework ermöglicht.

Beide ADLs bauen auf einer allgemeinen Datenbeschreibungssprache auf, die strukturelle hierarchische Daten in einem Textdokument beschreiben kann. Die Datenbeschreibungssprache ist ähnliche zu XML [60] oder JSON [103] erlaubt aber zusätzlich die Verwendung von Variablen und konstanten mathematischen Ausdrücken. Zusätzlich können Schleifen und Bedingungen mittels „if“- und „for“-Konstrukten erzeugt

werden. Dies erlaubt die kompakte Beschreibung von regulären MPSoC-Konstrukten innerhalb der System-ADL. Z.B. kann in zentralen Variablen die Höhe und Breite eines MPSoC-Prozessorfelds abgelegt werden. Dieses Prozessorfeld kann dann mittels „for“-Schleifen erzeugt werden. Nach der Variablenpropagierung, Evaluierung von Formeln und der Interpretation von Bedingungen und Schleifen kann die Datenbeschreibungssprache nach XML oder JSON konvertiert und somit flexibel weiterverwendet werden.

5.2 Datenbeschreibungssprache

In diesem Abschnitt wird die Datenbeschreibungssprache vorgestellt. Mit den Datenbeschreibungssprache kann ein Datenbaum beschrieben werden, der als Grundlage für die Core-ADL und System-ADL verwendet wird. Die Datenbeschreibungssprache wurde erstmalig in meiner Diplomarbeit [119] beschrieben und wurde während der Dissertation substantiell weiterentwickelt.

5.2.1 Datenbaum

Die Datenbeschreibungssprache wird verwendet, um einen Datenbaum zu erzeugen. Dieser Datenbaum beinhaltet skalare Datentypen als Blätter, die zum Beispiel eine Zeichenkette oder Zahl beinhaltet könnten. Die Knoten des Baumes werden durch assoziative Container realisiert. Jeder Knoten kann eine beliebige Anzahl von Nachfolgern haben, die je nach Containertyp durch einen Integerwert oder einer Zeichenkette referenziert werden. Es folgt eine Liste aus sämtlichen Datentypen, aus denen der Baum bestehen kann.

- Skalare Datentypen
  - Undef
  - Bool
  - Integer
  - Float
  - String
- Assoziative Container
  - Vektor
  - Objekt

5.2.1.1 Skalare Datentypen

Skalare Datentypen sind einfache Datentypen, die einen Wert – ein Skalar – enthalten. Dies können Undefiniert (Undef), Boolean (Bool), Zeichenketten (String), Ganzzahlen (Integer) oder Gleitkommazahlen (Float) sein.

Der einfachste skalare Datentyp ist **Undef**. Er repräsentiert ein undefiniertes Blatt des Baumes. Der Datentyp kann durch das Schlüsselwort `undef` erzeugt werden.

5.2.1.2 Assoziative Container

Um mehr als nur einzelne Werte im Ergebnisbaum abbilden zu können, kann dieser auch aus assoziativen Containern bestehen. Ein Container beinhaltet beliebig viele Werte. Jeder Wert kann ein beliebiger Datentyp sein, also auch ein Container. Als assoziative Container stehen Vektoren und Objekte zur Verfügung, die sich in ihrer Implementierung und Adressierung unterscheiden. Werte in einem Vektor werden durch einen Integer-Datentypen adressiert während in Objekten Strings zur Adressierung verwendet werden. Eine Mischung zwischen Integer- und String-Schlüsseln innerhalb eines Containers ist nicht möglich.

Vektor Ein Vektor ist ein assoziativer Container, der einem Integer-Schlüssel einen Wert zuweist. Der Vektor fängt mit Index 0 an und ist immer so groß wie der größte verwendete Schlüssel. Wenn man z.B. in einen leeren Vektor an der Stelle 10 einen Wert schreibt, besitzt der Vektor danach 11 Elemente. Davon sind die ersten 9 Elemente undefiniert. Die Elemente eines Vektors sind somit immer anhand der Schlüsselwerte sortiert.

Objekt Ein Objekt ist ein assoziativer Container, der einem String-Schlüssel einen Wert zuweist. Im Gegensatz zum Vektor ist ein Objekt immer so groß, wie die Anzahl der Elemente, die es enthält. Prinzipiell ist die Reihenfolge der Elemente in einem Objekt irrelevant. Die Elemente im Objekt sind anhand der Einfügungsreihenfolge sortiert. Ein Objekt wird durch eine Hashtabelle realisiert, deren Elemente durch eine verkettete Liste verknüpft sind, um die Einfügungsreihenfolge zu speichern.

5.2.2 Lexikalische Struktur

5.2.2.1 Kommentare

Es gibt, in Anlehnung an C/C++, zwei verschiedene Arten von Kommentaren. Einzeilige Kommentare, die die komplette Zeile ab `//` ignorieren, und mehrzeilige Kommentare, die sämtlichen Inhalt zwischen `/*` und `*/` ignorieren.

5.2.2.2 Bezeichner

Bezeichner (engl. *Identifier*) fangen mit einem Buchstaben oder Unterstrich (_) an. Ab dem zweiten Zeichen sind zusätzlich noch Dezimalziffern erlaubt. Ein Bezeichner kann entweder eine Konstante oder Funktion identifizieren.

5.2.2.3 Buchstabensymbole

Buchstabensymbole (engl. *Literals*) sind die kleinstmöglichen Ausdrücke für Zahlen und Zeichenketten.

5.2.2.4 Trennzeichen

Trennzeichen (engl. *Seperators*) sind verschiedene Arten von Klammern sowie Komma und Semikolon.

5.2.2.5 Operatoren

Es werden die üblichen logischen, mathematischen und Vergleichsoperatoren unterstützt. Die Syntax orientiert sich dabei an der Programmiersprache C/C++. Zusätzlich werden auch Komma (`,`), Semikolon (`;`) und verschiedene Arten von Klammern als Operatoren angesehen.

5.2.2.6 Füllzeichen

Leerzeichen, Zeilenende und Kommentare können an beliebiger Stelle zwischen Bezeichnern, Buchstabensymbolen und Operatoren eingefügt werden.

5.2.3 Syntax

5.2.3.1 Zahlen

Zahlen beginnen immer mit einer Dezimalziffer, unabhängig davon in welchem Zahlensystem sie kodiert werden. Prinzipiell gibt es zwei verschiedene Zahlentypen: Ganzzahlen (Integer) und Gleitkommazahlen. Integerwerte können im Dual-, Hexadezimal- und Dezimalsystem kodiert werden. Eine Zahl im Dualsystem muss mit dem Präfix `0b` und eine Zahl im Hexadezimalsystem mit `0x` anfangen. Eine Gleitkommazahl ergibt sich, wenn ein Punkt in der Zahl vorkommt und ist nur im Dezimalsystem erlaubt.

Vor der Zahl kann ein Vorzeichen stehen. Das Vorzeichen wird nicht zu der eigentlichen Zahl dazu gezählt, sondern als Operator behandelt, der auf die Zahl angewandt wird. Aus diesem Grund können zwischen Vorzeichen und Zahl Füllzeichen vorkommen.

Quelltext 5.1: Beispiele von Zahlen

```
100
0xabcd
0b001100
10.4
```

5.2.3.2 Zeichenketten

Zeichenketten werden in dieser Sprache auf die nachfolgenden Arten modelliert:

Zeichenkette in einfachen Anführungszeichen Bei Zeichenketten in einfachen Anführungszeichen werden sämtliche Zeichen als Zeichenkette interpretiert. Die Darstellung des einfachen Anführungszeichens ist nicht möglich, da dieses als Terminatorzeichen interpretiert werden würde. Eine Zeichenkette, die über das Zeilenende hinausgeht, ist nicht erlaubt.

Quelltext 5.2: Beispiel einer Zeichenkette in einfachen Anführungszeichen

```
'Hallo Welt'
```

Zeichenkette mit doppelten Anführungszeichen Bei Zeichenketten in doppelten Anführungszeichen ist der Backslash (\) ein Maskierungszeichen (engl. *Escape Character*), das es ermöglicht, Sonderzeichen zu kodieren. Tabelle 5.1 zeigt alle erlaubten Escape-Sequenzen.

Quelltext 5.3: Beispiel einer Zeichenkette in doppelten Anführungszeichen

```
"Hallo\tWelt\n"
```

Quote-Operator Der Quote-Operator wurde von Perl übernommen. Er fängt mit dem Buchstaben `q` an, gefolgt von einem beliebigen Startzeichen. Die Zeichenkette befindet sich zwischen dem Start- und Terminatorzeichen und darf über das Zeilenende

Zeichen	Beschreibung (Englisch)	Beschreibung (Deutsch)
\'	Single quote	Einfache Anführungszeichen
\"	Double quote	Doppelte Anführungszeichen
\\	Backslash	Umgekehrter Schrägstrich
\0	Null	Nullzeichen
\a	Bell	Tonsignal
\b	Backspace	Rückschritt
\f	Form Feed	Seitenvorschub
\n	Line Feed	Zeilenvorschub
\r	Carriage Return	Wagenrücklauf
\t	Horizontal Tab	Horizontaler Tabulator
\v	Vertical Tag	Vertikaler Tabulator
\xnn oder \Xnn	Hex Code	Hexadezimaler Code

Tabelle 5.1: Escape-Sequenzen bei Zeichenketten in doppelten Anführungszeichen

hinausgehen. Das Terminatorzeichen ergibt sich direkt aus dem Startzeichen. Bei einer sich öffnenden Klammer als Startzeichen ({, [, (oder <) ist das Terminatorzeichen automatisch die passende schließende Klammer (},],) oder >). Andernfalls ist das Terminatorzeichen mit dem Startzeichen identisch.

Im ersten Fall wird zusätzlich eine Klammerzählung vorgenommen. Das Terminatorzeichen wird nur als Ende der Zeichenkette erkannt, wenn die Summe der Terminatorzeichen genau so groß wie die Summe der Startzeichen ist. So ist z.B. bei der Zeichenkette q{{}} das Startzeichen { und das Terminatorzeichen }. Beim letzten Zeichen endet die Zeichenkette, da die Summe der beiden Zeichen { und } identisch ist.

Der Quote-Operator hat den großen Vorteil, dass man damit fremden Source-Code fast direkt in die ADL übernehmen kann, ohne auf Spezialzeichen achten zu müssen.

Quelltext 5.4: Beispiel von Zeichenkette im Quote-Operator

```
q{
    void test() {
        printf("Hallo Welt\n");
    }
}
```

Die gleiche Zeichenkette sieht ohne den Quote-Operator wie folgt aus:
`'\n\tvoid test() {\n\t\tprintf("Hallo Welt\n");\n\t'`

Automatisches Zusammenfügen von Zeichenketten Zeichenketten, die ohne Operator direkt hintereinander vorkommen, werden zu einer Zeichenkette zusammenge-

fügt. Somit ist es möglich, eine Zeichenkette über mehrere Zeilen hinweg zu beschreiben. Die verwendete Modellierungsart ist dabei unwichtig.

Zum Beispiel ist

Quelltext 5.5: Beispiel einer Zeichenkette mit automatischer Zusammenfügung

```
'Hallo'
"\tWelt\n"
```

identisch mit `"Hallo\twelt\n"`.

5.2.3.3 Konstanten

Tabelle 5.2 zeigt alle Konstanten der Datenbeschreibungssprache.

Name	Datentyp	Beschreibung
undef	Undef	Undefinierter Wert
true	Boolean	Wahr
false	Boolean	Falsch

Tabelle 5.2: Konstanten

5.2.3.4 Operatoren

Die Sprache bietet die meisten Operatoren, die auch aus C/C++ bekannt sind. Tabelle 5.3 beinhaltet eine komplette Liste der Operatoren, ihre Priorität und Assoziativität.

Arithmetische, logische, bitweise und Vergleichsoperatoren Es werden sämtliche arithmetische Operatoren (`+`, `-`, `*`, `/`, `%`), logische Operatoren (`&&`, `||`, `!`), bitweise Operatoren (`&`, `|`, `~`, `^`, `<<`, `>>`) und Vergleichsoperatoren (`<`, `<=`, `>`, `>=`, `!=`, `==`) aus C unterstützt und können beim Parsen evaluiert werden. Tabelle 5.4 zeigt die Evaluation der Operatoren in der Datenbeschreibungssprache.

Vektoroperator („,“) Der Vektoroperator wird durch das Kommazeichen repräsentiert und ist eine Möglichkeit, Vektoren mit zwei oder mehr Elementen zu erzeugen. Zum Beispiel repräsentiert `10, undef, 40` einen Vektor mit drei Elementen: Zwei Integer-Werten und einem undefinierten Zustand. Optional kann man auch die Vektor-Funktion verwenden, die in dem Beispiel mit `v(10, undef, 40)` das gleiche Ergebnis erzeugen würde. Die Vektor-Funktion wird benötigt, um leere oder einelementige Vektoren zu erzeugen, da diese sich nicht mit dem Kommaoperator ausdrücken lassen.

Operator	Beschreibung	Priorität	Assoziativität
()	Gruppierungsoperator		–
[]	Elementzugriff		–
{}	Blockoperator		–
	Leere Operator	1	Links nach Rechts
!	Logische Negation	2	Recht nach links
~	Bitweise Komplement	2	Recht nach links
+	Unäres Plus	2	Recht nach links
-	Unäres Minus	2	Recht nach links
*	Multiplikation	3	Links nach rechts
/	Division	3	Links nach rechts
%	Modulus	3	Links nach rechts
+	Addition	4	Links nach rechts
-	Division	4	Links nach rechts
<<	Bitweise Linksverschiebung	5	Links nach rechts
>>	Bitweise Rechtsverschiebung	5	Links nach rechts
<	Kleiner Vergleich	6	Links nach rechts
<=	Kleiner gleich Vergleich	6	Links nach rechts
>	Größer Vergleich	6	Links nach rechts
>=	Größer gleich Vergleich	6	Links nach rechts
==	Gleich Vergleich	7	Links nach rechts
!=	Ungleich Vergleich	7	Links nach rechts
&	Bitweises UND	8	Links nach rechts
^	Bitweises exklusive ODER	9	Links nach rechts
|	Bitweises ODER	10	Links nach rechts
&&	Logisches UND	11	Links nach rechts
||	Logisches ODER	12	Links nach rechts
,	Vektoroperator	13	Links nach rechts
=	Zuweisungoperator	14	Rechts nach links
+=	Addition und Zuweisung	14	Rechts nach links
-=	Subtraktion und Zuweisung	14	Rechts nach links
*=	Multiplikation und Zuweisung	14	Rechts nach links
/=	Division und Zuweisung	14	Rechts nach links
%=	Modulus und Zuweisung	14	Rechts nach links
&=	Bitweises UND und Zuweisung	14	Rechts nach links
^=	Bitweises Exklusiv-ODER und Zuweisung	14	Rechts nach links
|=	Bitweises ODER und Zuweisung	14	Rechts nach links
<<=	Bitweise Linksverschiebung und Zuweisung	14	Rechts nach links
>>=	Bitweise Rechtsverschiebung und Zuweisung	14	Rechts nach links
;	Anweisungsoperator	15	Links nach rechts

Tabelle 5.3: Operatoren

Zugriffsoperator („[]“) Der Zugriffsoperator wird durch zwei eckige Klammern ausgedrückt und dient dem Zugriff auf assoziative Container. Man kann sowohl lesend als auch schreibend auf ein Containerelement zugreifen. Je nach Datentyp, der sich zwischen den eckigen Klammern befindet, hat der Operator unterschiedliche Bedeutungen:

String Wenn sich ein String zwischen den Klammern befindet, z.B. `['Text']`, ist ein Objektzugriff gemeint. Falls der Datentyp, auf den der Operator angewandt wird, kein Objekt ist, wird der aktuelle Inhalt verworfen und ein leeres Objekt erzeugt.

Integer Wenn sich ein Integerwert zwischen den Klammern befindet, z.B. `[10]`, ist ein Vektorzugriff gemeint. Falls der Datentyp, auf den der Operator angewandt wird, kein Vektor ist, wird der aktuelle Inhalt verworfen und ein leerer Vektor erzeugt.

Ohne Inhalt Wenn sich nichts zwischen den Klammern befindet, z.B. `[]`, ist ein Vektorzugriff auf ein Element gemeint, das an den Vektor angehängt wird. Falls der Datentyp, auf den der Operator angewandt wird, kein Vektor ist, wird der aktuelle Inhalt verworfen und ein leerer Vektor erzeugt.

Undef Wenn sich ein undefinierter Wert zwischen den Klammern befindet, z.B. `[undef]`, wird kein Zugriff vorgenommen, sondern der Datentyp direkt zurückgegeben, auf den der Pseudozugriff angewandt wurde.

Vektor Wenn sich ein Vektor zwischen den Klammern befindet, werden die Zugriffe, die durch die Vektorelemente repräsentiert werden, nacheinander auf den Datentyp angewandt. So greift z.B `['Text', 10]` zuerst auf den Objektwert an der Stelle `'Text'` zu. Danach wird ein Vektorzugriff an der Stelle `10` vorgenommen. Eine alternative Schreibweise für diesen Fall wäre `['Text'][10]`.

Blockoperator („{}“) Der Blockoperator wird verwendet, um einen neuen Block (siehe 5.2.3.5) aus Anweisungen zu erzeugen, der später als Knoten in den Ergebnisbaum eingefügt wird. Meistens wird er dazu verwendet, um ein Objekt oder einen Vektor zu erzeugen.

Anweisungsoperator („;“) Der Anweisungsoperator dient dazu, verschiedene Anweisungen in einem Block (siehe 5.2.3.5) zu trennen. Alle Anweisungen werden immer der Reihenfolge nach abgearbeitet.

5.2.3.5 Block

Ein Block ist an einen Datenknoten gebunden. Gewöhnlich besteht ein Block aus verschiedenen, durch den Anweisungsoperator getrennte, Anweisungen. Dabei werden alle Zugriffsoperatoren, die in den Anweisungen vorkommen, auf dem Datenknoten des Blockes ausgeführt. Ein Block kann allerdings auch nur aus einem skalaren Da-

tentyp bestehen. In diesem Fall wird der Datentyp dem Datenknoten zugewiesen.

Die ADL beginnt immer mit dem Wurzelblock, dessen Datenknoten die Wurzel des Ergebnisbaumes darstellen. Alle anderen Blöcke werden durch den Blockoperator eingeleitet und müssen z.B. einem Element aus dem Überblock zugewiesen werden.

So wird z.B. mit `{ [] = 10; [] = 20; }` ein Vektorknoten mit den zwei Integerelementen `10` und `20` erzeugt.

`{ ['a'] = 10; ['b'] = ['a'] + 5; ['a'] = 20; }` erzeugt ein Objekt bestehend aus zwei Elementen. Zuerst wird `'a'` auf den Wert 10 gesetzt. Danach wird `'b'` auf 15 gesetzt und `'a'` mit 20 überschrieben.

Bei `{ ['a'] = 10; [5+5] = 5; }` wird zuerst ein Objekt mit dem Element `'a'` erzeugt. Die darauf folgende Anweisung ist allerdings ein Vektorzugriff, der das Objekt wieder zerstört und einen Vektor mit 11 Elementen erzeugt, der an der 10ten Stelle mit einer `5` gefüllt ist.

5.2.3.6 Variablen

In einer Variablen kann ein Datenknoten und somit ein Baum gespeichert werden. Variablen werden über das Dollarzeichen (`$`) identifiziert gefolgt von einem Identifier. Variablen können jederzeit gelesen oder geschrieben werden. Ihre Gültigkeit ist auf einen Block beschränkt.

Quelltext 5.6: Beispiel einer Variable

```
$Var = 100;
['Test'] = $Var;
```

Variablen sind insbesondere in Kombination mit Kontrollstrukturen und include-Funktionen sinnvoll.

5.2.3.7 Kontrollstrukturen

Kontrollstrukturen (engl. *control flow*) können den Ablauf der Evaluierung steuern. Es werden If-Verzweigungen und For-Schleifen unterstützt:

If-Verzweigung Eine If-Verzweigung wird durch das „if“-Schlüsselwort eingeleitet gefolgt von einer Bedingung und Anweisungen, die nur ausgeführt werden, wenn die Bedingung als Wahr evaluiert wurde. Optional können der initialen If-Anweisung noch beliebig viele „elif“-Anweisung angehängt werden, deren Anweisungen ausgeführt werden, wenn die jeweilige Bedingungen wahr ist und sämtliche vorherige Bedingungen falsch waren. Optional kann am Ende noch ein „else“-Anweisung folgen, die ausgeführt wird, wenn keine der vorherigen

Bedingungen als wahr evaluiert wurden. Das „if-elif-else“-Konstrukt muss mit einem Semikolon abgeschlossen werden.

Quelltext 5.7: Beispiel einer If-Verzweigung

```
if ($Var == 10) {
    ['xy'] = 10;
} elif ($Var == 20) {
    ['yx'] = 10;
} else {
    ['yy'] = 10;
};
```

For-Schleife Die For-Schleife wird mit dem „for“-Schlüsselwort eingeleitet. Der Syntax der For-Schleife ist dem Syntax der Programmiersprache C nachempfunden. Der einzige Unterschied besteht darin, dass die For-Schleife am Ende mit einem Semikolon abgeschlossen werden muss.

Quelltext 5.8: Beispiel einer for-Schleife

```
for ($i=0; $i<10; $i += 1) {
    [$i] = $i*2;
};
```

5.2.3.8 Funktionen

Es wird eine Reihe von vordefinierten Funktionen unterstützt. Eine Funktion besteht aus einem Bezeichner, gefolgt von einer in runden Klammern geschriebenen Parameterliste, die durch Kommas getrennt ist.

Im Folgenden werden die existierenden Funktionen beschrieben:

v(...) oder vector(...) Die vector-Funktion dient der Erzeugung von Vektoren. Leere bzw. einelementige Vektoren können nur mit dieser Funktion erzeugt werden, da der Kommaoperator hierfür nicht geeignet ist. Es empfiehlt sich generell die vector-Funktion zum Erzeugen zu verwenden, da es zum einen die Lesbarkeit erhöht und zum anderen jederzeit ein Element entfernt werden kann, ohne auf die beschriebenen Sonderfälle zu stoßen.

set(...) Die set-Funktion erzeugt ein Objekt. Die Übergabeparameter werden als Schlüssel des Objektes verwendet, wobei der Wert immer auf true gesetzt wird. Folglich sind nur String-Werte als Übergabeparameter sinnvoll. Undefinierte Werte werden ignoriert. Bei Integer-Werten wird anstelle eines Objektes ein Vektor erzeugt.

Zum Beispiel ist der Ausdruck:

Quelltext 5.9: Beispiel der set-Funktion

```
set('Green', 'Blue')
```

gleichbedeutend mit:

```
{
    ['Green'] = true;
    ['Blue'] = true;
}
```

keys(Objekt) Die keys-Funktion wandelt ein Objekt in einen Vektor um, wobei alle Schlüssel des Objektes im Vektor gespeichert werden. Die Reihenfolge im Vektor entspricht der Sortierung des Objektes.

Zum Beispiel erzeugt der Ausdruck

Quelltext 5.10: Beispiel der keys-Funktion

```
keys({
    ['Green'] = 4;
    ['Blue'] = 10;
})
```

einen Vektor mit den zwei Elementen `'Green'` und `'Blue'`.

values(Objekt) Die values-Funktion wandelt ein Objekt in einen Vektor um, wobei alle Werte des Objektes im Vektor gespeichert werden. Die Reihenfolge im Vektor entspricht der Sortierung des Objektes.

Zum Beispiel erzeugt der Ausdruck

Quelltext 5.11: Beispiel der values-Funktion

```
values({
    ['Green'] = 4;
    ['Blue'] = 10;
})
```

einen Vektor mit den zwei Elementen `4` und `10`.

sort(Vektor) Die sort-Funktion sortiert einen Vektor anhand seiner Werte und gibt den sortierten Vektor zurück.

include(String /* Filename */ [, Objekt /* Variables */]) Die include-Funktion fügt eine Datei an der aufrufenden Stelle ein. Dazu wird der Dateiname als Parameter übergeben. Da die Behandlung der Funktion erst in der semantischen Analyse stattfindet, müssen sowohl die aufrufende Datei als auch die einzufügende Datei syntaktisch und semantisch korrekt sein. Als optionaler Parameter können die Variablen, die initial bei der Evaluierung gesetzt sein sollen, als Objekt übergeben werden.

r(String [, Objekt]) oder replacevars(String [, Objekt]) Die r- oder replacevars-Funktion ersetzt Variablen innerhalb von einer Zeichenkette. In der Zeichenkette fängt eine Variable mit einem Dollarzeichen (`$`) gefolgt vom Variablennamen an. Optional kann der Variablennamen in geschweiften Klammern geschrieben werden, um den Variablennamen eindeutig von einer folgenden Zeichenkette zu trennen. Als optionaler Parameter können die Variablen, die zum Ersetzen verwendet werden sollen, als Objekt angegeben werden.

5.2.3.9 Syntax in Erweiterter Backus-Naur-Form

Es folgt eine Beschreibung des Syntax der ADL in EBNF Notation gemäß ISO/IEC 14977 : 1996(E) [104] Standard.

Block	::= Statements | RValue | Empty;
Statements	::= { Statement , ';' }-;
Statement	::= Assignment | Selection | Iteration;
Assignment	::= { LValue , AssignOperator }- , RValue;
Selection	::= 'if' , '(' , RValue , ')' , RValue , { 'elif' , '(' , RValue , ')' , RValue } , ['else' , RValue]
Iteration	::= 'for' , '(' , RValue , ';' , RValue , ';' , RValue , ')' , RValue
LValue	::= Access | Variable , [Access];
Access	::= { '[' , [RValue] , ']' }-;
RValue	::= Scalar | LValue | Variable | Function | '{' , Block , '}' | Vector | Operation;
Scalar	::= Undef | Boolean | Integer | String | Float;
Operation	::= RValue , BinaryOperator , RValue | UnaryOperator , RValue | '(' , RValue , ')';

Vector ::= RValue { ',' RValue }-;

Variable ::= '$' , Identifier

Function ::= Identifier , '(' , [RValue] , ')'

AssignOperator ::= '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '<<=' | '>>=' | '|=' | '&=' | '^=';

BinaryOperator ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '&&' | '||' | '<<' | '>>' | '>' | '>=' | '<' | '<=' | '!=' | '==';

UnaryOperator ::= '+' | '-' | '!' | '~';

Undef ::= 'undef';

Boolean ::= 'true' | 'false';

Integer ::= { Digit }- | '0x' , { HexDigit }- | '0b' , { BinDigit }-;

String ::= ? Single Quoted String ? | ? Double Quoted String ? | ? Quote Operator ?;

Float ::= { Digit }- , '.' , { Digit }-;

Identifier ::= Letter , { '_' | Digit | Letter };

BinDigit ::= '0' | '1';

Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

HexDigit ::= Digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

Letter ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';

5.3 Core-ADL

Die Datenbeschreibungssprache aus Abschnitt 5.2 wird als Grundlage für die Core-ADL verwendet. Sie beschreibt einen Baum bestehend aus skalaren Datentypen (String, Boolean, Integer und Gleitkommazahlen) als Blätter und Container-Datentypen (Vektor und Objekte) als innere Knoten.

Basierend auf dem Datenbaum der Datenbeschreibungssprache wird die Struktur der Core-ADL-Beschreibung festgelegt. Abbildung 5.2 zeigt die Struktur der ADL-Beschreibung. Im Kontext der ADL wird eine Beschreibung einer Befehlssatzarchitektur auch **Target** genannt. Eine ADL-Beschreibung kann mehr als ein Target enthalten. Dazu wurde eine ADL-Beschreibung in sieben Hauptsektionen unterteilt, wobei sechs davon für alle Targets gemeinsam gelten (*Global*, *Nodes*, *RegisterFile*, *FieldFormat*, *Ope-*

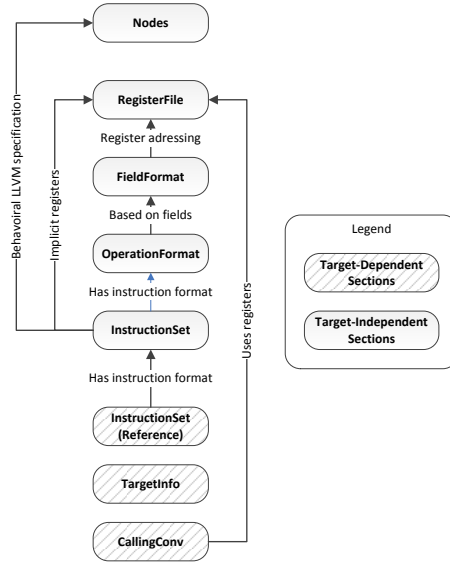


Abbildung 5.2: Struktur der Core-ADL

rationFormat und *OperationSet*). Zusätzlich enthält die *Target*-Sektion eine Liste aller Befehlssatzarchitekturen, wobei jede Target-Sektion drei Untersektionen (*OperationSet*, *TargetInfo* und *CallingConvention*) hat. Im Folgenden wird ein kurzer Überblick über die Sektionen gegeben:

Global
: enthält allgemeine globale Informationen, wie z.B. der Name der Architektur.

Nodes
: enthält eine Liste von plattformunabhängigen Knoten, die für die Verhaltensbeschreibung einer Operation verwendet werden.

RegisterFile
: beschreibt sämtliche Register des Prozessors. Dabei wird zwischen physikalischen und logischen Registern unterschieden.

OperationFormat
: beschreibt mittels Feldern verschiedene Formate einer Operation. Jedes Feld hat einen Typ, wobei zwischen fundamentalen Datentypen (siehe Abschnitt

5.3.10.1) und spezifizierten Datentypen (siehe Abschnitt 5.3.4) unterschieden wird.

FieldFormat
beschreibt die spezifizierten Datentypen für Operationsformate aus der OperationFormat-Sektion.

OperationSet
enthält einer Liste von Befehlssätzen. Ein Befehlssatz aus dieser Sektion wird von einem Target referenziert. Durch den Verweis von mehreren Targets auf denselben Befehlssatz wird eine redundante Befehlssatzbeschreibung vermieden. Jede Instruktion hat ein Format, das vorher in der OperationFormat-Sektion beschrieben wurden.

Targets
enthält einer Liste von Targets, wobei jedes Target folgende Untersektionen enthält:

OperationSet
referenziert ein vorher definierten Befehlssatz.

TargetInfo
enthält allgemeine Informationen über das Target.

CallingConvention
beschreibt die Aufrufkonvention für Funktionen.

Abbildung 5.3 zeigt den globalen Aufbau der Core-ADL.

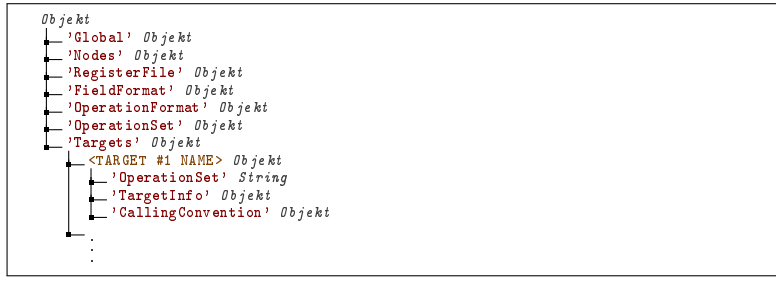


Abbildung 5.3: Aufbau der Core-ADL

5.3.1 Sektion „Global“

Die Sektion beinhaltet globale Einstellungen, die hauptsächlich für den Compiler relevant sind.

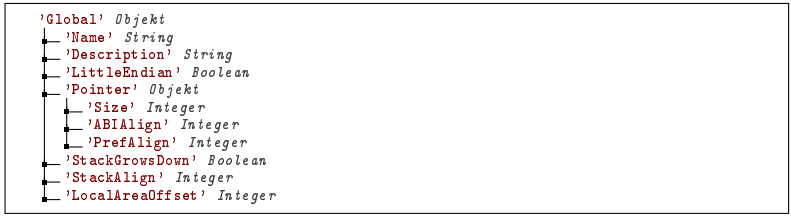


Abbildung 5.4: Core-ADL: Aufbau der Global-Sektion

Name
ist eine alphanumerische Zeichenkette (ohne Leerzeichen), die den Namen der Architektur (z.B. X86, ARM) beschreibt.

Description
ist eine einzeilige Beschreibung für die Architektur. Sie taucht später beim Compiler in der Liste der Kommandozeilenparameter auf.

LittleEndian
legt die Byte-Reihenfolge der Architektur fest. Bei true ist sie Litte-Endian, ansonsten Big-Endian.

Pointer
beinhaltet die Eigenschaften eines Zeigers.

- **Size**
  legt die Breite eines Zeigers in Bits fest.
- **ABIAlign**
  legt die Speicherausrichtung eines Zeigers fest, wie sie in der *Binärschnittstelle* (ABI, engl. *Application Binary Interface*) festgelegt ist.
- **PrefAlign**
  legt die bevorzugte Speicherausrichtung eines Zeigers fest.

StackGrowsDown
spezifiziert die Richtung, in die der Stack wächst. Bei true wächst der Stack nach unten, ansonsten nach oben.

StackAlign
legt die Speicherausrichtung des Stacks in Bits fest.

LocalAreaOffset
: legt den Offset fest, bei dem der Stack in einer Funktion anfängt.

5.3.2 Sektion „Nodes“

Die **Nodes**-Sektion enthält eine Liste sämtlicher architekturunabhängigen Befehlen des Compiler-Backends. Durch diese Befehle wird in der OperationSet-Sektion das Verhalten einer Operation für den Compiler modelliert. Während der Befehlsauswahl im Compiler-Backend wird dann ein *gerichteter azyklischer Graph* (DAG, engl. *Directed Acyclic Graph*) bestehend aus diesen architekturunabhängigen Befehlen in einen DAG bestehend aus architekturabhängigen Operationen überführt.

In der Nodes-Sektion werden nun sämtliche bekannte, architekturunabhängige Operationen spezifiziert. Zusätzlich können den Operationen optional noch Parameter zugewiesen werden, die insbesondere bei der Generierung der Regeln für die Befehlsauswahl verwendet werden.

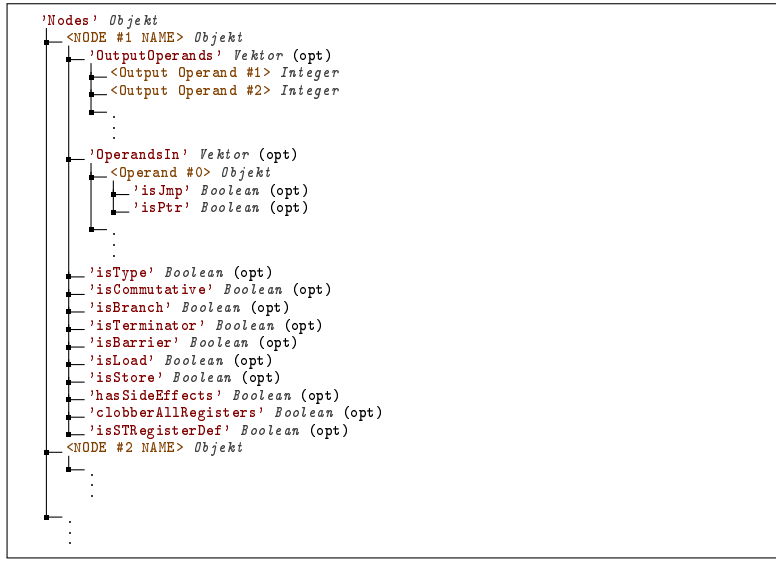


Abbildung 5.5: Core-ADL: Aufbau der Nodes-Sektion

Abbildung 5.5 zeigt den Aufbau der Nodes-Sektion. Im Folgenden werden die einzelnen Felder erklärt:

OutputOperands
Enthält einen Vektor bestehend aus Integern. Darin werden sämtliche Ausgabeoperanden des plattformunabhängigen Befehls festgelegt. Alle nicht aufgezählten Operanden sind Eingangsoperanden.

OperandsIn
Enthält einen Vektor, der Information über Eingangsoperanden enthält. Jedes Element des Vektors repräsentiert einen Operanden, wobei die Schlüsselnummer die Operandennummer ist. Der Wert ist ein Objekt und kann folgende Elemente enthalten:

isJmp Falls vorhanden und der Wert true ist, kann der Operand ein Sprungziel aufnehmen.

isPtr Falls vorhanden und der Wert true ist, kann der Operand eine Speicheradresse aufnehmen.

isType
Falls vorhanden und der Wert true ist, ist der plattformunabhängige Befehl eine Typumwandlung (engl. *typecast*).

isCommutative
Falls vorhanden und der Wert true ist, ist der Befehl kommutativ und es kann das Kommutativgesetz angewandt werden.

isBranch
Falls vorhanden und der Wert true ist, ist der Befehl ein bedingter oder unbedingter Sprung.

isTerminator
Falls vorhanden und der Wert true ist, werden nach dem Befehl keine weiteren Befehle mehr ausgeführt. Dies ist bei einem unbedingten Sprung oder Return-Befehl der Fall.

isReturn
Falls vorhanden und der Wert true ist, ist der Befehl ein Return-Befehl.

isBarrier
Falls vorhanden und der Wert true ist, ist der Befehl eine Barriere und die Reihenfolge darf beim Scheduling nicht verändert werden.

isCall
Falls vorhanden und der Wert true ist, ist der Befehl ein Unterfunktionsaufruf.

isLoad
Falls vorhanden und der Wert true ist, ist der Befehl ein Ladebefehl.

isStore
Falls vorhanden und der Wert true ist, ist der Befehl ein Speicherbefehl.

hasSideEffects
: Falls vorhanden und der Wert true ist, hat der Befehl Seiteneffekte, die für den Compiler nicht modelliert also unbekannt sind. Er kann dabei für viele Compileroptimierungen nicht berücksichtigt werden.

clobbersAllRegisters
: Falls vorhanden und Wert true ist, löscht der Befehl den Inhalt von sämtlichen Registern.

isSTRegisterDef
: Falls vorhanden und Wert true ist, ändert der Befehl die Konfiguration des Prozessors.

5.3.3 Sektion „RegisterFile“

Die **RegisterFile**-Sektion beschreibt alle physikalische Registerspeicher, logischen Register und semantische Informationen. Physikalische Registerspeicher werden durch einen Namen identifiziert und ihre Größe wird durch die Anzahl der Elemente und deren Bitbreite festgelegt. Zusätzlich kann jedem physikalischen Registerspeicher optional ein Cluster zugewiesen werden.

Die logischen Register werden auf den physikalischen Registerspeicher abgebildet. Dazu referenzieren sie ein Element und eine Bitposition in einem physikalischen Registerspeicher. Sie haben ebenfalls einen Namen zur Identifizierung in der ADL, einen Assemblernamen und ein festes Format, in dem Daten im Register gespeichert werden.

Durch semantische Informationen wird für den Compiler festgelegt, welches der Instruction-, Stack- und Frame-Pointer ist. Für den Stack- und Frame-Pointer können mehr als ein Register angegeben werden. Dadurch können mehr als ein Register als Stack-Pointer identifiziert werden und bekommen dann im Compiler immer den gleichen Wert zugewiesen. Dadurch kann leicht sichergestellt werden, dass der Stack-Pointer in jedem Cluster für den Compiler jederzeit verfügbar ist.

Durch die Aufteilung in physikalische Registerspeicher und logische Register ist es möglich, überlappende Register zu spezifizieren. Dies sind Register, die in der ISA einen unterschiedlichen Namen haben aber auf die gleiche Speicherstelle zugreifen. So teilen sich z.B. in der x86-Befehlssatzarchitektur die MMX-Register und die Floating-Point-Register den gleichen Speicher. Durch den Registernamen wird die Interpretation des Registerspeichers verändert.

Physical
: beinhaltet eine Objektstruktur, bestehend aus physikalischen Registerspeicher. Die Schlüssel sind die Namen und dienen der Referenzierung. Jeder Wert besteht aus folgenden Elementen:

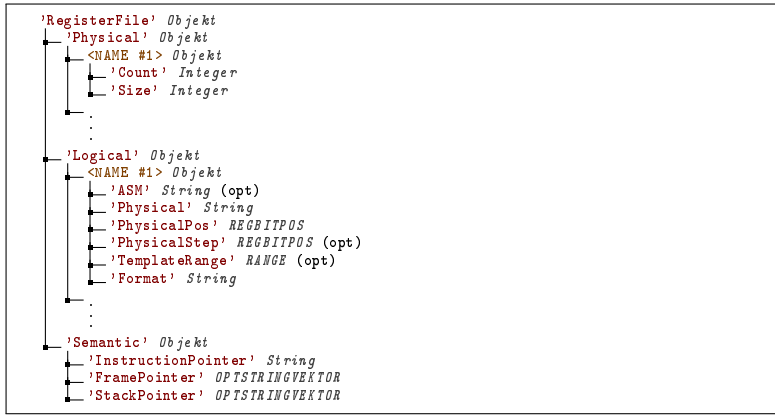


Abbildung 5.6: Core-ADL: Aufbau der RegisterFile-Sektion

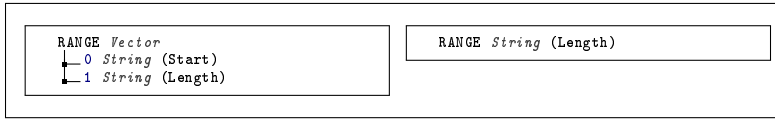


Abbildung 5.7: Core-ADL: Aufbau des RANGE-Typs

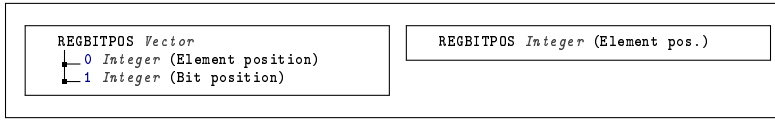


Abbildung 5.8: Core-ADL: Aufbau des REGBITPOS-Typs

Count
ist die Anzahl der Elemente, die der Registerspeicher enthält.

Size
ist die Größe eines einzelnen Elementes in Bits. Die Gesamtgröße lässt sich durch die Multiplikation aus Count mit Size berechnen.

Cluster
gibt optional die Clusternummer des Registerspeichers an.

Logical
beinhaltet ein Objekt, bestehend aus logischen Registern. Die Schlüssel sind die Namen und dienen der Referenzierung. Jeder Wert besteht aus folgenden Elementen:

ASM
gibt den Assemblernamen des Registers an. Falls der Wert nicht vorhanden ist, wird der Name des logischen Registers verwendet.

Physical
referenziert einen physikalischen Registerspeicher, auf den das logische Register verweist.

PhysicalPos
beschreibt die Position, an welcher das logische Register im physikalischen Registerspeicher liegt. PhysicalPos ist vom Typ REGBITPOS. Als Vektor werden zwei Zahlen angegeben, wobei die erste die Elementenposition des physikalischen Registers angibt und die zweite die Bitposition im Element. Wenn man letzteres weglässt – also PhysicalPos nur ein Integer ist – wird die Bitposition als Null angenommen.

Format
definiert das Format des Registers. Siehe Abschnitt 5.3.10.1 für eine Liste von erlaubten Formaten.

TemplateRange
ist optional und stellt eine einfache Möglichkeit dar, mehrere Register auf einmal zu definieren. Es wird ein Zahlenbereich (RANGE) angegeben. Dieser besteht aus einem Vektor mit zwei Zahlen. Die erste Zahl ist die Startnummer und die zweite Zahl die Anzahl der Register. Zum Beispiel steht `(16, 16)` für die Zahlen von 16 bis 31. Ähnlich wie bei PhysicalPos kann auch hier auf eine Zahl verzichtet werden. In diesem Fall wird die Startnummer 0 angenommen.

Für jede Zahl, die in TemplateRange angegeben wurde, wird jetzt die logische Registerbeschreibung wiederholt. Dabei wird für den Namen und den Assemblernamen eine printf-Formatierung vorgenommen, wie sie aus der Programmiersprache C bekannt ist. Somit wird z.B. `%d` durch die Zahl aus der TemplateRange ersetzt. Die PhysicalPos wird bei jedem Schritt um PhysicalStep erhöht.

PhysicalStep
ist optional und nur sinnvoll, wenn auch TemplateRange angegeben ist. Es ist vom Typ REGBITPOS (siehe PhysicalPos). Wenn es nicht angegeben ist, wird der Wert `(1,0)` also Elementenposition Eins und Bitposition Null angenommen.

Semantic
beinhaltet semantische Informationen zu Registern:

InstructionPointer
gibt den Namen des logischen Registers des Befehlszeigers an.

FramePointer
gibt einen oder mehrere Namen der logischen Register der Frame-Pointer an. Bei geclusterten Architekturen ist es sinnvoll, für jedes Cluster einen eigenen Frame-Pointer zu erstellen.

StackPointer
gibt einen oder mehrere Namen der logischen Register der Stack-Pointer an. Bei geclusterten Architekturen ist es sinnvoll, für jedes Cluster einen eigenen Stack-Pointer zu erstellen.

5.3.3.1 Beispiel

Die RegisterFile-Sektion besteht aus den drei Untersektionen Physical, Logical und Semantic. Dadurch ergibt sich folgende Struktur:

Quelltext 5.12: Struktur der RegisterFile-Sektion

```
['RegisterFile'] = {
    ['Physical'] = {
        ...
    };
    ['Logical'] = {
        ...
    };
    ['Semantic'] = {
        ...
    };
};
```

In Physical wird der Speicherbereich für die *Allzweckregister* (GPR, engl. *General Purpose Register*) aus Cluster 0 definiert. Dieser besteht aus 32 Elementen zu je 32 Bits:

Bei den logischen Registern werden als erstes ebenfalls 32 32 Bit Integer-Register definiert:

Quelltext 5.14: Beispiel der Beschreibung von logischen 32 Bit Allzweckregistern

```
        ['R0_%02d'] = {
            ['ASM'] = "r0.%d";
            ['Physical'] = 'GPR0';
            ['TemplateRange'] = (0, 32);
```

Quelltext 5.13: Beispiel eines physikalischen Registerspeichers

```
['Physical'] = {
    ['GPR0'] = {
        ['Cluster'] = 0;
        ['Count'] = 32;
        ['Size'] = 32;
    };
};
```

```
        ['Format'] = 'Int<32>';
    };
```

Diese werden durch die Zeichenketten *R*0_00, *R*0_01, ···, *R*0_31 identifiziert. Die Registernamen, die später im Assembler verwendet werden, lauten *r*0.0, *r*0.1, ···, *r*0.31. Diese Mehrfachdefinition wird durch das TemplateRange-Attribut eingeleitet. `%z02d` wird in diesem Fall durch 00, 01, ···, 31 und `%d` durch 0, 1, ···, 31 ersetzt.

Es folgt die Definition von alternativen Namen für bestimmte Spezialregister:

Quelltext 5.15: Beispiel der Beschreibung von logischen Spezialregistern

```
    ['SP'] = {
        ['Physical'] = 'GPR0';
        ['PhysicalPos'] = 32-1;
        ['Format'] = 'Int<32>';
    };
    ['FP'] = {
        ['Physical'] = 'GPR0';
        ['PhysicalPos'] = 32-2;
        ['Format'] = 'Int<32>';
    };
```

PhysicalPos definiert an dieser Stelle das Element im physikalischen Speicher, an dem das Register angelegt werden soll. Die hier definierten Register SP und FP sind in der Größe und Position identisch mit den Registern R31 und R30. Sie repräsentieren einen alternativen Namen der Register, um die Lesbarkeit zu erhöhen, sind aber ansonsten identisch.

Für den Compiler müssen noch zusätzliche Informationen über die Bedeutung der Register bereitgestellt werden. Dazu werden in der Semantic-Untersektion die logischen Register des Instruction-, Frame- und Stack-Pointers angegeben:

Quelltext 5.16: Beispiel von semantischen Registerinformationen

```
    ['Semantic'] = {
        ['InstructionPointer'] = 'IP';
        ['FramePointer'] = 'FP';
        ['StackPointer'] = 'SP';
};
```

5.3.4 Sektion „FieldFormat“

Die FieldFormat-Sektion beschreibt den Aufbau von Feldern, die später von der OperationFormat-Sektion verwendet werden. Für jede Binärkodierung von einem Feld kann spezifiziert werden, ob das Feld im Befehlswort ein Register, Immediate oder eine benutzerdefinierte Semantik referenziert. Alle nicht verwendeten Kodierungen sind im Feld nicht erlaubt.

Jedes Feld hat ein Zielformat. Das Zielformat beschreibt das Format der Daten, die durch das Feld repräsentiert/referenziert werden. So entspricht z.B. das Zielformat von Feldern mit Registern dem Datenformat der Register, z.B. 32 Bit Integer. Pro Feld kann nur genau ein Zielformat spezifiziert werden, so dass sämtliche Elemente des Feldes das gleiche Zielformat haben müssen. Dies gilt insbesondere für Register. So kann man z.B. keine 32 Bit Register und 16 Bit Register innerhalb eines Feldes kodieren. Immediates übernehmen automatisch das Zielformat.

TargetFormat
: beschreibt das Zielformat der Felder als fundamentalen Datentyp (siehe Abschnitt 5.3.10.1).

Assembler
: beschreibt einen Vektor von Zeichenketten, der den Assemblersyntax für die jeweilige Kodierung enthält.

Code
: beschreibt einen Vektor von Zeichenketten, die für die jeweilige Kodierung Simulationscode enthält. Das Feld im Simulationscode der Instruktionen wird dann entsprechend der Zeichenkette ersetzt.

Register
: beschreibt einen Vektor von Zeichenketten, die für die jeweilige Kodierung Registernamen enthält.

LLVM
: beschreibt einen Vektor von DAGs, der LLVM-Pattern für den LLVM-Compiler enthält. Das Feld im Pattern der Instruktionen wird dann entsprechend dem Pattern des Feldes ersetzt.

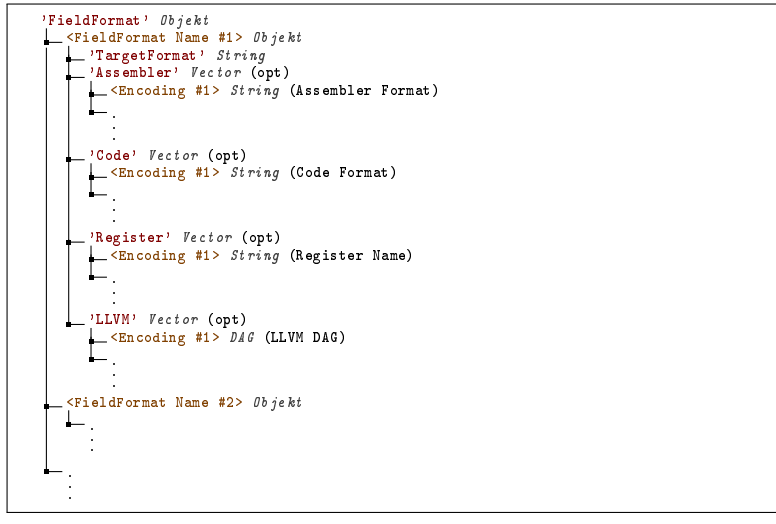


Abbildung 5.9: Core-ADL: Aufbau der FieldFormat-Sektion

5.3.4.1 Beispiel

Der folgende Code definiert die beiden Feldformate StdRegsIn und CondCode:

Quelltext 5.17: Beispiel der FieldFormat-Sektion

```
['FieldFormat'] = {
    ['StdRegsIn'] = {
        ['TargetFormat'] = 'Int<32>';

        ['Assembler'] = {
            [0] = "0";
        };

        ['Register'] = {
            [1] = ('R%02d', 1, 32-1);
        };
    };

    ['CondCode'] = {
        ['Assembler'] = {
            [0] = 'N';      // N        Never
            [1] = 'E';      // E        Equal
```

```
            [2] = 'LE';      // LE        Less or Equal
            [3] = 'L';       // L         Less
            // ...
        };
        ['Code'] = {
            [0] = 'false';                          // N    Never
            [1] = 'flag_ze';                        // E    Equal
            [2] = '(flag_ze || flag_ov)';           // LE   Less or Equal
            [3] = 'flag_ov';                        // L    Less
            // ...
        };

        ['LLVM'] = {
            // Never is not used by the compiler
            [1] = '!SETEQ';
            [2] = '!SETLE';
            [3] = '!SETLT';
            // ...
        };
    };
```

Das StdRegsIn-Feldformat definiert das Feld für Quelloperanden. Das Zielformat ist 32 Bit Integer. Bei der 0-Kodierung ist das Feld fest auf das Immediate 0 kodiert. Bei jeder anderen Kodierung repräsentiert es die Register $R01, \cdots, R31$.

Das CondCode-Feldformat definiert das Feldformat für Vergleichsbedingungen. Hierbei wird für jede Kodierung separat der Assembler-Syntax, der Simulationscode und ein LLVM-Pattern angegeben.

5.3.5 Sektion „OperationFormat“

Die OperationFormat-Sektion beschreibt das Operationsformat, d.h. den Aufbau bzw. Kodierung einer Operation im Instruktionswort. Jeder Operation ist ein Operationsformat zugewiesen.

Abbildung 5.10 zeigt die Struktur der OperationFormat-Sektion. Jedes Operationsformat wird durch einen eindeutigen Namen identifiziert und kann mehrere Operationsformate (Subformate) gleichzeitig definieren. Daher gibt es zwei Alternativen bei der ADL-Beschreibung. Bei der ersten (links in der Abbildung) wird nur ein Subformat spezifiziert während bei der zweiten Alternative mehrere Subformate spezifiziert werden können (rechts in der Abbildung). In diesem Fall verfügt jedes Subformat über einen eindeutigen Namen (SUBNAME). Durch die Spezifikation mehrere Subformate können mehrere Kodiervarianten für eine Operation auf einmal beschrieben und somit die Anzahl an Operationen in der OperationSet-Sektion reduziert werden. So kann z.B. ein Operationsformat zwei Kodierungen enthalten wobei bei einer ein Quelloperand ein Register und bei der anderen ein Immediate ist. Für die Funktion

einer Operation ist es unerheblich, woher die Eingangsdaten kommen bzw. wie sie kodiert sind. Wenn eine Addition z.B. das Operationsformat mit zwei Kodierungen verwendet, wird automatisch die Register- und Immediate-Variante erzeugt.

Ein Operationsformat kann von einem vorher definierten Operationsformat sämtliche Werte und Felder erben. Dazu hat jedes Operationsformat ein optionales Extends-Attribut, das den Feldnamen des Basisfeldes als Zeichenkette enthält.

Abbildung 5.11 zeigt den Aufbau eines Operationssubformats. Ein Subformat besteht aus verschiedenen Feldern, die durch ihren Namen, Typ, Größe und Position angegeben werden.

Size ist optional und gibt die Gesamtgröße des Formats in Bits an. Die Gesamtgröße muss Bytealigned, also durch acht teilbar, sein. Falls nicht angegeben, wird die Gesamtgröße aus der Feldliste automatisch berechnet.

AddSize
ist eine alternative Variante, um die Größe eines Formats anzugeben und ist nur in Verbindung mit Vererbung sinnvoll. Bei AddSize wird die Größe des Basisformats um den angegebenen Wert erhöht.

Fields
ist ein Vektor. Jedes Element besteht aus einem Objekt, das für ein Feld im Format steht. Im Objekt werden die Attribute des Feldes abgelegt. Wenn das Feld abgeleitet ist, dann werden die Felder zu den abgeleiteten Feldern hinzugefügt.

Name
gibt den Namen des Feldes an.

Pos ist optional und gibt die Bitposition des Feldes an, bei der das Feld startet. Falls nicht gesetzt, wird die Endposition des vorherigen Feldes übernommen.

Format
gibt das Format des Feldes an. Man kann entweder einen fundamentalen Datentypen angeben (siehe Abschnitt 5.3.10.1) oder ein Feldformat, das in der Sektion FieldFormat definiert wurde. Durch den Typ wird auch implizit die Größe des Feldes definiert.

TargetFormat
ist optional und überschreibt das Zielformat des Feldes. Dies ist insbesondere für Immediates sinnvoll, die z.B. auf 32 Bit erweitert werden sollen.

DecodeName
ist optional und wird für Trace-Dateien im Simulator verwendet. Falls gesetzt wird die Kodierung des Feldes unter dem Namen ins Trace geschrieben, d.h. bei einer registerindirekten Adressierung wird die Nummer des Registers geschrieben.

Operatoren	Datentypen	Anmerkung
+, -, *, /, %	Integer *op* Integer = Integer	
+, -, *, /, %	{ Float, Integer } *op* Float = Float	
+, -, *, /, %	Float *op* { Float, Integer } = Float	
+, -	*op* Integer = Integer	
+, -	*op* Float = Float	
&, |, ~, ^, <<, >>	Integer *op* Integer = Integer	
&&, ||, !	{ * } *op* { * } = Boolean	Beide Operanden werden vorher nach Bool konvertiert
<, <=, >, >=, !=, ==	{ * } *op* { * } = Boolean	
+	String + { * } = String	String Konkatenation
+	{ * } + String = String	String Konkatenation
+	Objekt + Objekt = Objekt	Elemente von beiden Objekten werden zusammengeführt. Elemente vom rechten Objekt überschreiben Elemente vom linken.
+	Vektor + Vektor = Vektor	Vektoren werden konkateniert

Tabelle 5.4: Evaluation von Operatoren

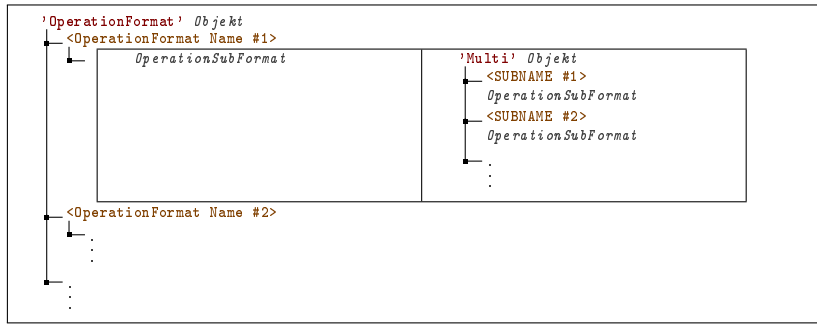


Abbildung 5.10: Core-ADL: Aufbau der OperationFormat-Sektion

```
OperationSubFormat Objekt
 ├─ 'Extends' String (opt)
 ├─ 'Size' Integer (opt)
 ├─ 'AddSize' Integer (opt)
 ├─ 'Fields' Vector (opt)
 │   ├─ <Field #1> OperationFormatField
 │   ├─ <Field #2> OperationFormatField
 │   ├─ .
 │   .
 │   .
 └─ 'ChangeFields' Objekt (opt)
     ├─ <Field Name #1> OperationFormatField
     ├─ <Field Name #2> OperationFormatField
     ├─ .
     .
     .
```

Abbildung 5.11: Core-ADL: Aufbau der OperationSubFormat-Sektion

```
OperationFormatField Objekt
 ├─ 'Name' String (opt)
 ├─ 'Pos' Integer (opt)
 ├─ 'Format' String (opt)
 ├─ 'TargetFormat' String (opt)
 ├─ 'DecodeName' String (opt)
 ├─ 'DataName' String (opt)
 ├─ 'Used' Boolean (opt)
 ├─ 'Value' OPTSTRINGVEKTOR (opt)
 ├─ 'DefaultValue' String (opt)
 ├─ 'Encoding' Integer (opt)
 ├─ 'DefaultEncoding' Integer (opt)
 ├─ 'Type' String (opt)
 └─ 'Direction' String (opt)
```

Abbildung 5.12: Core-ADL: Aufbau der OperationFormatField-Sektion

DataName
ist optional und wird für Trace-Dateien im Simulator verwendet. Falls gesetzt wird der Wert des Feldes unter dem Namen ins Trace geschrieben, d.h. bei einer registerindirekten Adressierung wird der Inhalt des Registers geschrieben.

Used
ist optional (Default auf true) und gibt an, ob das Feld benutzt wird. Ein benutztes Feld muss bei der Beschreibung einer Operation ebenfalls verwendet werden.

Value
ist optional und gibt an, welche Werte ein Feld annehmen kann. Dabei können mehrere Werte wahlweise in einem Vektor angegeben werden. Mit der Zeichenkette werden jeweils die Werte eines Feldes angegeben. Die Zeichenkette kann ein Immediate (Zahl), Register (beginnend mit %) oder der Assembler-Syntax eines Feldes sein.

Encoding
ist optional und kann den Wert eines Feldes durch dessen Kodierung festlegen.

DefaultValue
ist optional und gibt den Wert eines Feldes an, falls dieses nicht verwendet wird, d.h. Used gleich false ist.

DefaultEncoding
ist optional und gibt die Kodierung eines Feldes an, falls dieses nicht verwendet wird, d.h. Used gleich false ist.

Direction
ist optional und gibt die Datenrichtung des Feldes an. Die Werte 'In', 'Out' und 'InOut' sind möglich. Die beiden letzteren sind nur für Feldformate mit Registeradressierung sinnvoll.

Type
ist optional und gibt den Typ des Feldes an. Es stehen folgende Typen zur Verfügung:

None (Standard)
Das Feld hat keinen speziellen Typ.

Ptr
Das Feld kann den Inhalt eines Pointers aufnehmen.

Jmp
Das Feld kann den Inhalt eines Sprungziels aufnehmen.

NextBit
Das Feld gibt an, ob die Operation parallel zur nächsten ausgeführt

wird.

ChangeFields
ist ein Objekt und ermöglicht im Zusammenhang mit Ableiten das Verändern vorher definierte Felder. Dazu werden sämtliche Attribute für ein angegebenes Feld (identifiziert durch dessen Namen) überschrieben.

5.3.5.1 Vererbung mit mehreren Subformaten

Eine Besonderheit des Feldformats ist Vererbung in Kombination mit mehreren Subformaten. Dabei wird jedes Subformat des Basisformats (Oberformats) mit jedem Subformat des abgeleiteten Formats (Unterformats) kombiniert. Dadurch wird die Anzahl der Subformate des Ober- und Unterformats multipliziert. Es gilt also folgende Formel:

$$\#Subformate = \#Subformat\_Unterformat * \#Subformate\_Oberformat$$

Dadurch lässt sich mittels Vererbung schnell eine große Anzahl an Subformaten definieren.

5.3.5.2 Beispiel

Als Beispiel werden die drei Befehlsformate Root, OpOutInIn und OpCCImm definiert:

Quelltext 5.18: Beispiel der OperationFormat-Sektion

```
['OperationFormat'] = {
    ['Root'] = {
        ['Size'] = 64;

        ['Fields'] = {
            [] = {
                ['Type'] = 'Int<8>';
                ['Name'] = 'Opcode';
            };
        };
    };

    ['OpOutInIn'] = {
        ['Extends'] = 'Root';

        ['Multi'] = {
            ['_reg'] = {
```

```
                ['Fields'] = table(
                    ('Name',      'Type',     'Direction', 'Value'),

                    ('Dst',       'StdRegs', 'Out',       undef  ),
                    ('Src1',      'StdRegs', 'In',        undef  ),
                    ('Src2Type', 'Int<1>',  'In',        '0'    ),
                    ('Src2',      'StdRegs', 'In',        undef  )
                );
            };

            ['_imm'] = {
                ['Fields'] = table(
                    ('Name',      'Type',     'Direction', 'Value'),

                    ('Dst',       'StdRegs', 'Out',       undef  ),
                    ('Src1',      'StdRegs', 'In',        undef  ),
                    ('Src2Type', 'Int<1>',  'In',        '1'    ),
                    ('Src2',      'Int<16>', 'In',        undef  )
                );
            };
        };

    };

    ['OpCCImm'] = {
        ['Extends'] = 'Root';

        ['Fields'] = {
            [] = {
                ['Name'] = 'CC';
                ['Type'] = 'CondCode';
            };
            [] = {
                ['Name'] = 'Dst';
                ['Type'] = 'Int<32>';
                ['Direction'] = 'In';
            };
        };
    };
};
```

Root beschreibt das generelle Format. Es besteht aus einem acht Bit Opcode-Feld. Die Gesamtgröße ist auf 64 Bits festgelegt. Die beiden anderen Formate, OpOutInIn und OpCCImm, erben durch die Extends-Anweisung alle Werte, also die Gesamtgröße und das erste Opcode-Feld, von Root.

Das OpOutInIn-Format ist für Operationen im Dreiadressformat gedacht, die ein Ziel- und zwei Quelloperanden haben. Es besteht neben dem Opcode aus vier weiteren Feldern: ein Zieloperanden (Dst), zwei Quelloperanden (Src1, Src2) und der Typ des

zweiten Quelloperanden (Src2Type). Dementsprechend ist auch die Datenrichtung (Direction) gesetzt. Dst und Src1 verwenden immer eine registerindirekte Adressierung und beziehen sich auf das StdRegs-Feldformat aus der FieldFormat-Sektion. Es gibt zwei Subformate, um zwei Kodierungsalternativen zu definieren. Beim ersten Subformat ist Src2Type fest auf 0 kodiert und Src2 ist vom Typ StdRegs. Beim zweiten Subformat ist Src2Type auf 1 kodiert und Src2 ist ein Immediate. Das OpOutInInFormat ist für Operationen im Dreiadressformat gedacht, die ein Ziel- und zwei Quelloperanden haben. Es besteht neben dem Opcode aus vier weiteren Feldern: ein Zieloperanden (Dst), zwei Quelloperanden (Src1, Src2) und der Typ des zweiten Quelloperanden (Src2Type). Dementsprechend ist auch die Datenrichtung (Direction) gesetzt. Dst und Src1 verwenden immer eine registerindirekte Adressierung und beziehen sich auf das StdRegs-Feldformat aus der FieldFormat-Sektion. Es gibt zwei Subformate, um zwei Kodierungsalternativen zu definieren. Beim ersten Subformat ist Src2Type fest auf 0 kodiert und Src2 ist vom Typ StdRegs. Beim zweiten Subformat ist Src2Type auf 1 kodiert und Src2 ist ein Immediate.

OpCCImm ist ein Befehlsformat, das zusätzlich zwei Felder enthält. Es wird für bedingte Sprünge verwendet. Hierfür existiert ein CondCode-Feld, dessen Inhalt die Bedingung enthält, ob der Sprung genommen wird oder nicht. Zusätzlich ist noch ein 32 Bit Immediate-Wert für die Zieladresse des Sprungs angegeben.

5.3.6 Sektion „OperationSet“

In der OperationSet-Sektion können verschiedene Sätze von Operationen definiert werden. Dies entspricht einer Erweiterung im Gegensatz zu einer ADL mit einem einzigen Befehlssatz. Abbildung 5.13 zeigt den generellen Aufbau der Sektion. Jeder Operationssatz wird durch einen eindeutigen Namen repräsentiert und enthält eine Liste von Operation.

Der Aufbau einer Operation ist in Abbildung 5.14 dargestellt. Jede Operation ist an ein Operationsformat gekoppelt und enthält ergänzende Informationen zu den Feldern des Befehlsformates, dem Assemblersyntax, Simulationscode und eine semantische Beschreibung für den Compiler.

Format
: verknüpft den Befehl mit einem Operationsformat aus der OperationFormat-Sektion.

Field
: ist optional und kann verschiedene Änderungen der Felder des Operationsformats enthalten. Es besteht aus einem Objekt, dessen Schlüssel jeweils ein Feldname des Formates ist. Mit dem Field-Eintrag können sämtliche Eigenschaften eines Feldes des Operationsformats genauso wie mit ChangeField geändert werden.

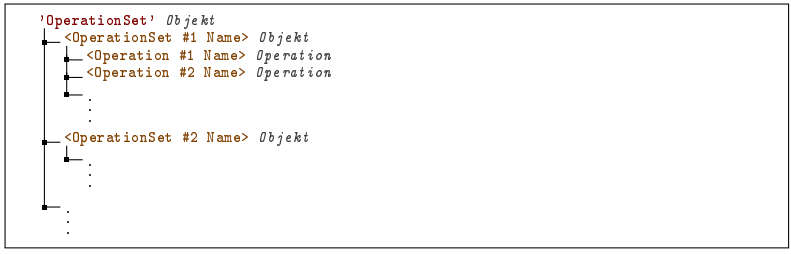


Abbildung 5.13: Core-ADL: Aufbau der OperationSet-Sektion

Type
ist optional und legt einen speziellen Operationstyp fest, der insbesondere für den Simulator verwendet wird.

NOP legt fest, dass die Operation ein NOP ist.

MOV legt fest, dass die Operation ein Registertransferbefehl ist.

Cost
ist optional und legt die Kosten für die Operation fest. Anhand der Kosten entscheidet der Compiler, welche Operation zu bevorzugen ist. Per Default hat jede Operation die Kosten 1.

Delay
ist optional und legt die Abarbeitungsdauer der Operation fest. Der Delay wird dann im Simulator für die Approximation der Zyklen verwendet.

ReadRegisters
ist optional und erlaubt die Spezifikation von implizit gelesenen Registern. Diese werden in einem Vektor aus Registernamen angegeben. Eine Call-Operation liest z.B. implizit den Befehlszeiger, um ihn auf den Stack zu schreiben.

WriteRegisters
ist optional und erlaubt die Spezifikation von implizit geschriebenen Registern. Diese werden in einem Vektor aus Registernamen angegeben. Ein Sprungbefehl schreibt z.B. den Befehlszeiger, ohne dass das Register explizit im Operationsformat kodiert ist.

ASMName
ist eine Zeichenkette und enthält den Assembler-Operationsnamen.

ASMParam
ist eine Zeichenkette und enthält die Parameter, die hinter der Assembleroperation angehängt werden. Die Parameter sind meistens durch Komma getrennt, aber die Syntax ist weitgehend beliebig. In der Zeichenkette werden einzelne

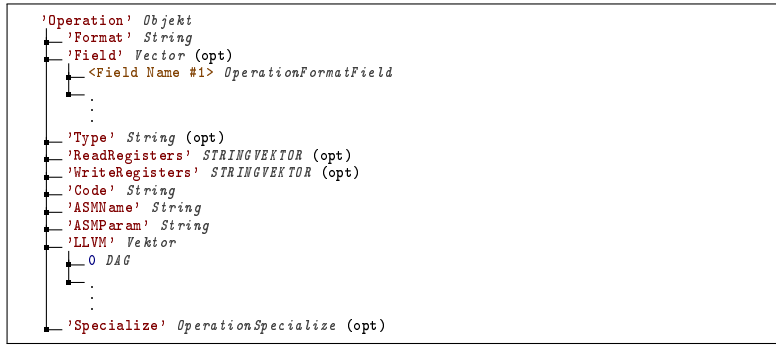


Abbildung 5.14: Core-ADL: Aufbau der Operation-Sektion

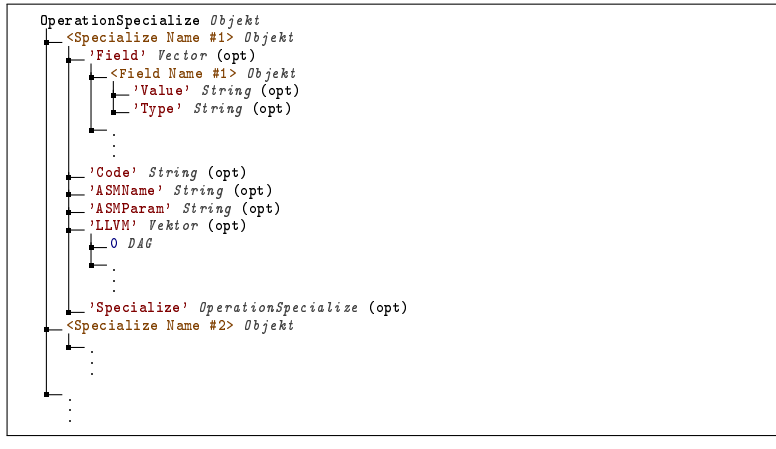


Abbildung 5.15: Core-ADL: Aufbau der OperationSpecialize-Sektion

Felder mit einem Dollarzeichen, gefolgt von ihrem Feldnamen, eingebettet. Je nach Typ des Feldes kann im Assembler ein Register (beginnend mit %), Immediate (als Zahl) oder eine andere Zeichenkette sein, die im Feldformat angegeben werden.

Code
ist eine Zeichenkette und enthält den Simulationsquellcode. Der Quellcode folgt dem C/C++-Syntax und wird auch mittels Metaprogrammierung verwendet, um Quellcode vom Simulator zu erzeugen. Bei Feldern, die eine Codeangabe in der FieldFormat-Sektion besitzen, kann der Feldname mit einem Dollarzeichen als Präfix verwendet werden. In diesem Fall wird der Feldname entsprechend der Codeangabe ersetzt.

LLVM
beschreibt die Semantik der Operation aus Sicht des Compilers. Dazu wird ein oder mehrere LLVM-Pattern für die Befehlsauswahl angegeben.

Specialize
bietet die Möglichkeit, Spezialisierungen der Operation anzugeben. Eine Spezialisierung wird durch eine Zeichenkette identifiziert. Bei einer Spezialisierung kann man einen spezialisierten Assembler-Syntax, Simulationscode oder LLVM-Pattern festlegen, wenn ein oder mehrere Felder der Operationen auf einen festen Wert gesetzt werden. So haben z.B. manche Architekturen eine Additionsoperation, die zusätzlich ein Carry-Bit als dritten Quelloperanden verwenden kann. Durch eine Spezialisierung dieser Additionsoperation, der den Carry-Operanden auf 0 setzt, kann der gleiche Befehl auch als gewöhnliche Addition (ohne Carry-Eingang) verwendet werden.

Field
gibt an, welche Felder mit welchen Werten spezialisiert werden sollen. Im Gegensatz zu der Field-Beschreibung außerhalb der Specialize-Sektion, können hier nur zwei Eigenschaften des Feldes angegeben werden.

Value
legt den Wert bei der Spezialisierung fest. Wenn das Feld vorher bereits eine Menge von Werten hatte, dann muss der angegebene Wert in den vorherigen Werten enthalten sein.

Type
legt den Typ des Feldes fest. Dies ist mit der Typ-Eigenschaft des Operationsformats identisch.

ASMName
gibt eine Spezialisierung des Assembler-Namens an.

ASMParam
gibt eine Spezialisierung des Assembler-Parameters an.

Code
gibt eine Spezialisierung des Simulationscodes an.

LLVM
gibt eine Spezialisierung der LLVM-Pattern an.

Specialize
gibt eine Unterspezialisierung an. Jede Spezialisierung kann durch eine weitere Spezialisierung eingeschränkt werden.

5.3.6.1 Simulationsquellcode

Der Simulationsquellcode ist ein Fragment eines C/C++-Quellcodes. Dies liegt darin begründet, dass das Simulationscodefragment in einer Funktion eingebettet und direkt dem Simulator innerhalb seines Quellcodes mittels Metaprogrammierung bereitgestellt wird. Innerhalb des Simulationscodes sind spezielle Datentypen vorhanden, die eine bitgenaue Berechnung von arithmetischen und logischen Operationen über den Sprachumfang von C/C++ hinaus ermöglichen.

Mittels Template-Klassen und Operatorüberladung wurde C/C++ um bitgenaue Datentypen erweitert, um die Funktionalität einer Operation innerhalb des Simulationscodes ähnlich wie in Hardwarebeschreibungssprachen modellieren zu können. Als Integer-Datentyp steht die Template-Klasse `Integer<BITWIDTH>` bereit, bei der man die Bitbreite in spitzen Klammern als Template-Parameter angeben kann.

Sämtliche Felder der Operation stehen im Simulationscode als lokale Variablen bereit. Diese haben den Datentyp, wie er im Zielformat des Feldes festgelegt ist. Je nach Richtung des Feldes (In, Out oder InOut) wird die Variable am Anfang der Funktion initialisiert bzw. der Wert am Ende ins Registerfile übernommen. Eine Besonderheit nehmen Felder ein, die im Feldformat bereits Simulationscode angegeben haben. Bei diesen Feldern ist der Wert des Feldes nicht als Variable gegeben. Stattdessen muss der Feldname mit einem vorangestellten Dollarzeichen angegeben werden und wird dann im Simulationscode ersetzt.

In Tabelle 5.5 sind alle unterstütze Funktionen im Simulationscode aufgelistet, wobei die `Integer<BITWIDTH>`-Template-Klasse durch `I<W>` abgekürzt wurde.

5.3.6.2 LLVM-Pattern

Abbildung 5.16 zeigt den Aufbau eines LLVM-Patterns. Ein LLVM-Pattern besteht aus einer verschachtelten Anzahl von plattformunabhängigen Nodes. Jede Node wird durch einen Vektor beschrieben, wobei das erste Vektorelement den Node-Typ als Zeichenkette enthält und die restlichen Vektorelemente die Parameter festlegen. Jeder Node-Typ muss vorher in der Nodes-Sektion spezifiziert werden. Als Parameter stehen eine Vielzahl von Möglichkeiten zur Verfügung:

Ergebnis	Funktion	Op.	Beschreibung
I<W>	ADD(I<W>, I<W>)	+	Addition
I<W>	SUB(I<W>, I<W>)	−	Subtraktion
I<W>	NEG(I<W>)		Bitweise Negation
I<W>	AND(I<W>, I<W>)	&	Bitweise Und-Verknüpfung
I<W>	OR(I<W>, I<W>)	|	Bitweise Oder-Verknüpfung
I<W>	XOR(I<W>, I<W>)		Bitweise Exklusiv-Oder-Verknüpfung
I<W>	ANDL(I<W>, I<W>)	&&	Logische Und-Verknüpfung
I<W>	ORL(I<W>, I<W>)	||	Logische Oder-Verknüpfung
I<W1>	NOT(I<W1>)	!	Logische Negation
I<W>	MUL(I<W>, I<W>)		Untere Hälfte der Multiplikation
I<W>	MULHS(I<W>, I<W>)		Obere Hälfte einer signed Multiplikation
I<W>	MULHU(I<W>, I<W>)		Obere Hälfte einer unsigned Multiplikation
I<2*W>	MUL2S(I<W>, I<W>)		Signed Multiplikation
I<2*W>	MUL2U(I<W>, I<W>)		Unsigned Multiplikation
I<W>	DIVS(I<W>, I<W>)		Signed Division
I<W>	DIVU(I<W>, I<W>)		Unsigned Division
I<W>	REMS(I<W>, I<W>)		Rest einer signed Division
I<W>	REMU(I<W>, I<W>)		Rest einer unsigned Division
I<W1>	SHRU(I<W1>, I<W2>)		Unsigned Schiebeoperation nach Rechts
I<W1>	SHRS(I<W1>, I<W2>)		Signed Schiebeoperation nach Rechts
I<W1>	SHL(I<W1>, I<W2>)		Schiebeoperation nach Links
I<W1>	ROTR(I<W1>, I<W2>)		Bitrotation nach Rechts
I<W1>	ROTL(I<W1>, I<W2>)		Bitrotation nach Links
I<W>	CONST<W>(<int>)		Erzeugt eine Konstante mit W-Bits
I<1>	SUB_CY(I<W>, I<W>)		Carry-Bit der Subtraktion
I<1>	SUB_ZE(I<W>, I<W>)		Zero-Bit der Subtraktion
I<1>	SUB_OV(I<W>, I<W>)		Overflow-Bit der Subtraktion
I<1>	SUB_NE(I<W>, I<W>)		Negate-Bit der Subtraktion
I<1>	ADD_CY(I<W>, I<W>)		Carry-Bit der Addition
I<1>	CMP_LES(I<W>, I<W>)		Kleiner-Gleich Vergleich Signed
I<1>	CMP_LEU(I<W>, I<W>)		Kleiner-Gleich Vergleich Unsigned
I<1>	CMP_LTS(I<W>, I<W>)		Kleiner Vergleich Signed
I<1>	CMP_LTU(I<W>, I<W>)		Kleiner Vergleich Unsigned
I<1>	CMP_GES(I<W>, I<W>)		Größer-Gleich Vergleich Signed
I<1>	CMP_GEU(I<W>, I<W>)		Größer-Gleich Vergleich Unsigned
I<1>	CMP_GTS(I<W>, I<W>)		Größer Vergleich Signed
I<1>	CMP_GTU(I<W>, I<W>)		Größer Vergleich Unsigned
I<1>	CMP_EQ(I<W>, I<W>)	==	Äquivalenz Vergleich
I<1>	CMP_NE(I<W>, I<W>)	!=	Anti-Äquivalenz Vergleich
I<W2>	SEXT<W2>(I<W>)		Signed Integer erweitern
I<W2>	ZEXT<W2>(I<W>)		Unsigned Integer erweitern
I<W2>	TRUNC<W2>(I<W>)		Integer abschneiden
I<W>	ADDS_SAT(I<W>, I<W>)		Signed Addition mit Saturation
I<W>	ADDU_SAT(I<W>, I<W>)		Unsigned Addition mit Saturation
I<W>	SUBS_SAT(I<W>, I<W>)		Signed Subtraktion mit Saturation
I<W>	SUBU_SAT(I<W>, I<W>)		Unsigned Subtraktion mit Saturation
I<W/2>	LO(I<W>)		Unter Hälfte
I<W/2>	HI(I<W>)		Obere Hälfte
I<1>	BIT(I<W>, pos)		Gibt das Bit an Position pos zurück
I<W2>	BITS<W2>(I<W>, pos, size)		Gibt die size Bits an Position pos zurück

Tabelle 5.5: Core-ADL: Unterstützte Funktionen im Simulationscode

Abbildung 5.16: Core-ADL: Aufbau der DAG-Sektion

DAG (Vektor)
Jeder Parameter kann wiederum einen DAG enthalten. Dadurch können verschachtelte Nodes definiert werden. Ein DAG ist immer ein Vektor.

Zahl (String '#<format>:number')
Eine Zeichenkette, die mit einer Raute beginnt, wird als Zahl interpretiert. Dabei muss vor der Zahl das Format nach Tabelle 5.6 festgelegt werden.

Register (String '%<register>')
Eine Zeichenkette, die mit einem Prozentzeichen beginnt, entspricht einem Register.

LLVM-Text (String '!<llvm>')
Eine Zeichenkette, die mit einem Ausrufezeichen beginnt, wird direkt an LLVM durchgereicht.

Feld (String '<feldname>')
Eine Zeichenkette mit einem Feldnamen verweist auf das korrespondierende Feld.

5.3.6.3 Beispiel

Folgender Codeausschnitt zeigt die Beschreibung einer ADD-Operation (Addition) mit optionalem Carry-Eingang und -Ausgang.

Quelltext 5.19: Beispiel einer Operationsdefinition

```
['ADD'] = {
    ['Format'] = 'EDPE_DE_DDE';
    ['Field']['Opcode']['Value'] = 3;
    ['Field']['CtrlSignals']['Value'] = 0b00000;

    ['Code'] = q{
        DDst = DSrc1 + DSrc2 + ZEXT<32>(ESrc);
        EDst = ADD_CY(DSrc1, DSrc2);
    };

    ['ASMName'] = 'ADD';
```

```
    ['ASMParam'] = "$DDst, $EDst, $DSrc1, $DSrc2, $ESrc";

    ['Specialize']= {
        ['__'] = {
            ['Field']['EDst']['Value'] = '%-';
            ['Field']['ESrc']['Value'] = '0';

            ['ASMParam'] = "$DDst, $DSrc1, $DSrc2";

            ['LLVM'] = {
                [] = ('set', 'DDst', ('add', 'DSrc1', 'DSrc2'));
            };
        };

        ['_c'] = {
            ['Field']['EDst']['Value'] = '%e0.1';
            ['Field']['ESrc']['Value'] = '0';

            ['LLVM'] = {
                [] = ('set', 'DDst', ('addc', 'DSrc1', 'DSrc2'));
            };
        };

        ['_e'] = {
            ['Field']['EDst']['Value'] = '%-';
            ['Field']['ESrc']['Value'] = '%e0.1';

            ['LLVM'] = {
                [] = ('set', 'DDst', ('adde', 'DSrc1', 'DSrc2'));
            };
        };
    };
};
```

Die ADD-Operation verwendet das EDPE_DE_DDE-Format, das zwei Daten-Quelloperanden, ein Event-Quelloperand, ein Daten-Zieloperand und ein Event-Zieloperand hat. Die Event-Register sind 1 Bit breit und werden für das Carry-Bit verwendet. In den folgenden Field-Anweisungen wird das Opcode- und CtrlSignals-Feld auf feste Werte gesetzt, um die Kodierung der Operation festzulegen. Der Simulationscode addiert die drei Quelloperanden und schreibt das Ergebnis in den Daten-Zieloperanden. Der Event-Quelloperand muss mittels einer Zero-Extension auf die richtige Bitbreite erweitert werden. Parallel dazu wird das Carry-Bit berechnet und in den Event-Zieloperanden geschrieben. Der Assembler-Syntax der allgemeinen ADD-Operation ist durch die fünf Operanden durch Kommas getrennt angegeben.

Für die allgemeine ADD-Operation werden im folgenden Specialize-Block drei Spezialisierungen angegeben. Diese sind insbesondere für den Compiler wichtig, weil dieser mit einer ADD-Operation mit fünf Operanden nichts anfangen kann:

'__' Die erste Spezialisierung definiert die ADD-Operation ohne die Verwendung des Carry-Eingangs und -Ausgangs. Dazu wird der Event-Quelloperand auf den Wert 0 festgelegt und das Ergebnis im Event-Zieloperand wird verworfen. Das EDPE_DE_DDE-Format hat dabei mehrere Subformate. Manche dieser Subformate erlauben es nicht, dass der Event-Quelloperand fest auf in Immediate mit Wert 0 kodiert werden. Diese werden durch die Spezialisierung automatisch herausgefiltert. Innerhalb der Spezialisierung ist ein weiterer Assembler-Syntax angegeben. Der Compiler verwendet diese Spezialisierung für die Realisierung einer gewöhnlichen ADD-Operation.

'_c' Die Spezialisierung ist nur für den Compiler gedacht und beschreibt eine ADD-Operation, die ein Carry-Bit schreibt. Das Carry-Bit wird fest in das erste Event-Register geschrieben.

'_e' Die Spezialisierung ist nur für den Compiler gedacht und beschreibt eine ADD-Operation, die ein Carry-Bit verwendet. Das Carry-Bit wird fest vom ersten Event-Register gelesen.

5.3.7 Sektion „Targets“

In der Targets-Sektion werden sämtliche Targets spezifiziert, die die ADL-Beschreibung umfasst. Dadurch unterscheidet sich die ADL maßgeblich von bisherigen ADLs, die immer nur eine Befehlssatzarchitektur beschreiben. Abbildung 5.17 zeigt den Aufbau der Targets-Sektion.

Abbildung 5.17: Core-ADL: Aufbau der Targets-Sektion

OperationSet
ist ein String, der den Befehlssatz des Targets festlegt. Dazu referenziert er einen vorher definierten Befehlssatz aus der OperationSet-Sektion (siehe Abschnitt 5.3.6).

TargetInfo
enthält allgemeine Informationen über das Target. Insbesondere wird hier der Aufbau einer Instruktion bestehend aus mehreren Operationen festgelegt. Der genaue Aufbau ist in Abschnitt 5.3.8 beschrieben.

CallingConvention
beschreibt die Aufrufkonvention für Funktionen. Der genaue Aufbau ist in Abschnitt 5.3.9 beschrieben.

5.3.8 Sektion „Targets-TargetInfo“

Die TargetInfo-Sektion ist für jedes Target vorhanden, das innerhalb der ADL beschrieben ist. Abbildung 5.18 zeigt den Aufbau der TargetInfo-Sektion.

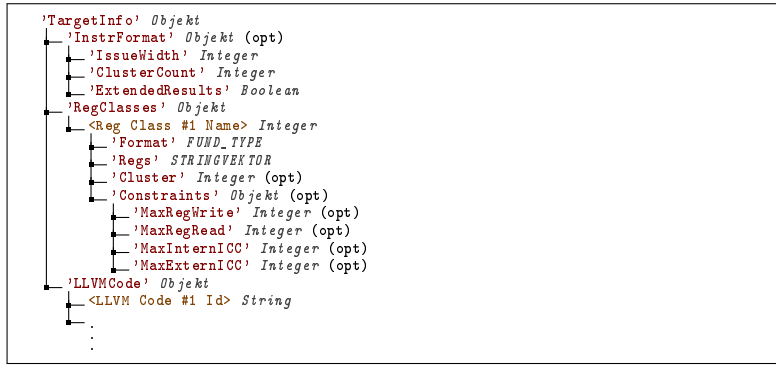


Abbildung 5.18: Core-ADL: Aufbau der TargetInfo-Sektion

InstrFormat
ist ein Objekt und enthält Informationen über den Aufbau des Instruktionswortes.

IssueWidth
gibt die Anzahl an Operationen pro Instruktion an. Bei 1 werden RISC-Prozessoren beschrieben und bei einer höhere Anzahl VLIW-Prozessoren.

ClusterCount
gibt die Anzahl an Register-Cluster pro Instruktion an. Mit einer Zahl größer als 1 können Clustered-VLIW-Prozessoren beschrieben werden. IssueWidth muss ein Vielfaches von der Anzahl der Cluster sein.

ExtendedResults
erlaubt die Beschreibung des *inter-cluster Kommunikationsmodells* (ICC-Modells, engl. *Inter-Cluster Communication Model*) von Clustered-VLIW-Prozessoren (siehe Abschnitt 2.2.9.1). Wenn der Parameter true ist, verwendet der Prozessor das Extended-Results- ansonsten das Extended-Operands-ICC-Modell.

RegClasses
gibt die Registerklassen für den Compiler an. Jede Registerklasse hat einen Namen und besteht mindestens aus einem Format und einer Registerliste.

Format
gibt das Format der Registerklasse als fundamentale Datentypen (siehe Abschnitt 5.3.10.1) an.

Regs
enthält einen Vektor mit Registernamen, der alle Register der Klasse enthält.

Cluster
kann die Registerklasse optional einem Cluster zuweisen.

Constraints
legt Nebenbedingungen für das Instruktionsformat für eine Registerklasse fest.

MaxRegWrite
gibt die maximale Anzahl an Register an, die pro Instruktion in der Registerklasse geschrieben werden können.

MaxRegRead
gibt die maximale Anzahl an Register an, die pro Instruktion in der Registerklasse gelesen werden können.

MaxInternICC
beschränkt die Anzahl an ICC-Moves, die von dieser Registerklasse ausgehen.

MaxExternICC
beschränkt die Anzahl an ICC-Moves, die auf diese Registerklasse ankommen.

LLVMCode
gibt verschiedene Codefragmente für die Generierung des Compiler-Quellcodes an, die noch nicht automatisiert von der ADL erzeugt werden.

ASMCode
gibt verschiedene Codefragmente für die Generierung des Assemblers an. Darunter ist auch der Startup-Code für C-Programme, der vor der main-Funktion aufgerufen wird.

5.3.9 Sektion „Targets-CallingConvention“

Die CallingConvention-Sektion beschreibt sämtliche Aufrufkonventionen zur Übergabe von Funktionsparametern an Funktionen. Es muss mindestens eine CallingConvention („Default“) beschrieben werden, die per Default verwendet wird. C/C++ erlaubt die Steuerung der CallingConvention einer Funktion als Funktionsattribut, wodurch auch anderen Konventionen verwendet werden können. Abbildung 5.19 zeigt den Aufbau der CallingConvention-Sektion.

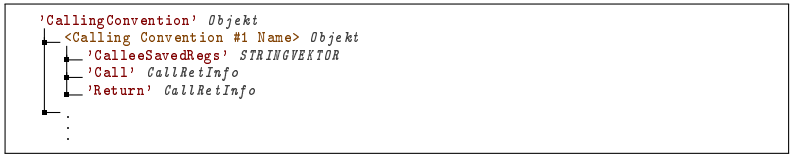


Abbildung 5.19: Core-ADL: Aufbau der CallingConvention-Sektion

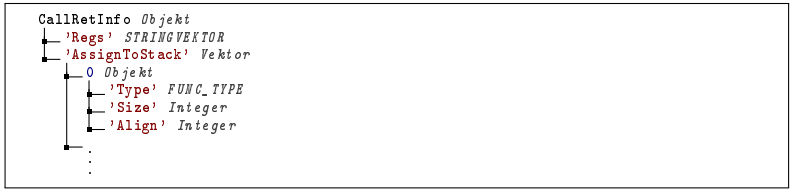


Abbildung 5.20: Core-ADL: Aufbau der CallRetInfo-Sektion

CalleeSavedRegs
gibt einen Vektor mit Registernamen an. Sämtliche Register in der Liste dürfen von der aufgerufenen Funktion nicht verändert werden. Falls eine Funktion die Register verwendet, muss sie die Register im Prologe- und Epiloge-Code auf dem Stack zwischenspeichern.

Call beschreibt die Funktionsparameterübergabe zu einer aufgerufenen Funktion:

Regs
gibt einen Vektor mit Registernamen an. Funktionsparameter werden nacheinander zuerst den Registern zugewiesen, bis keine Register mehr vorhanden sind.

AssignToStack
gibt einen Vektor mit Regeln, wie unterschiedliche Datentypen auf dem Stack abgelegt werden.

Type
gibt den Datentyp als fundamentalen Datentyp an.

Size
gibt die Größe in Bytes an, die auf dem Stack verwendet werden.

Align
gibt das Alignment auf dem Stack an, das eingehalten werden muss.

Return
beschreibt die Übergabe des Return-Wertes von einer Funktion nach dem gleichen Prinzip wie die Funktionsparameter.

5.3.10 Datentypen

In der ADL werden verschiedene Datentypen und -strukturen verwendet, die von mehreren Sektionen verwendet werden. Diese Typen werden im Folgenden vorgestellt:

5.3.10.1 FUND_TYPE (Fundamentale Datentypen)

Tabelle 5.6 zeigt eine Liste von sämtlichen fundamentalen Datentypen an, die in der ADL enthalten sind. Fundamentale Datentypen werden in der ADL als Zeichenkette spezifiziert. Fundamentale Datentypen können entweder Integer oder Floating-Point sein und haben eine festgelegt Bitbreite, die in spitzen Klammern angegeben ist.

Format	Breite	Kodierung
Bit	1	Integer mit einem Bit
Int<*>	beliebig	Integer
FP<*>	beliebig	Floating-Point

Tabelle 5.6: Core-ADL: Fundamentale Datentypen in der ADL

5.3.10.2 STRINGVEKTOR

Abbildung 5.21 zeigt den Aufbau des STRINGVEKTOR-Datentyps an. Der Datentyp beschreibt einen Vektor bestehend aus Strings.

Abbildung 5.21: Core-ADL: Aufbau des STRINGVEKTOR-Typs

5.3.10.3 OPTSTRINGVEKTOR

Abbildung 5.22 zeigt den Aufbau des OPTSTRINGVEKTOR-Datentyps an. Der Datentyp beschreibt einen Vektor bestehend aus Strings. Optional kann anstelle des Vektors auch nur ein einzelner String stehen. Dies wäre dann mit einem einelementigen Vektor vergleichbar.

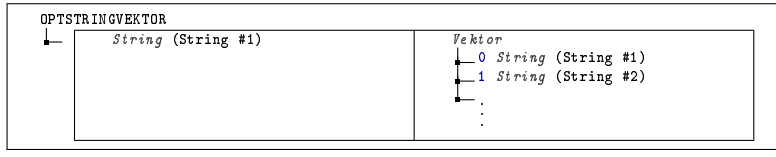


Abbildung 5.22: Core-ADL: Aufbau des OPTSTRINGVEKTOR-Typs

5.4 System-ADL

Die System-ADL erlaubt die simulationsorientierte systemweite Beschreibung von rekonfigurierbaren Prozessorarchitekturen. Sie wird im Kontext dieser Dissertation zur Beschreibung der Kahrisma-Architektur eingesetzt. Dabei ist die ADL sehr abstrakt gehalten und beschreibt ein System lediglich auf der Systemebene. Dies bedeutet, dass ein Prozessorkern oder EDPE (im Kontext von Kahrisma) als ein einzelnes Modul beschrieben ist, wobei keine detaillierteren Informationen einer tieferen Abstraktionsebene in der System-ADL enthalten sind. Auf diese Art und Weise wird eine effiziente Beschreibung ermöglicht, die unnötige Details der Architektur verbirgt und eine Entwurfsraumexploration auf Systemebene erlaubt. Die systemweite Simulation wird durch einen bibliotheksbasierten Ansatz ermöglicht, bei dem der Simulator detaillierte Implementierung der Module der System-ADL enthält, die durch die Beschreibung innerhalb der System-ADL zu einem Gesamtsystem vor der Simulation verbunden werden.

Die Datenbeschreibungssprache aus Abschnitt 5.2 wird als Grundlage für die System-ADL verwendet. Sie beschreibt einen Baum bestehend aus skalaren Datentypen (String, Boolean, Integer und Gleitkommazahlen) als Blätter und Container-Datentypen (Vektor und Objekte) als innere Knoten.

Basierend auf dem Datenbaum der Datenbeschreibungssprache wird die Struktur der System-ADL-Beschreibung festgelegt. Die System-ADL erlaubt eine strukturelle Beschreibung einer Multi-Prozessorarchitektur auf Systemebene. Dabei verwendet die ADL Konzepte zur strukturellen Beschreibung wie sie aus *Hardwarebeschreibungssprachen* (HDLs, engl. *Hardware Description Languages*), wie SystemVerilog oder VHDL, bekannt sind. Die System-ADL setzt sich aus Modulen, Instanzen und Verbindungen

zwischen Instanzen zusammen. Zusätzliche bietet die ADL die Möglichkeit, rekonfigurierbare Prozessorarchitekturen zu beschreiben, bei denen die Funktion einer Prozessors erst durch die Konfiguration verschiedener Module entsteht. Dies wurde am Beispiel der Kahrisma-Architektur umgesetzt.

Abbildung 5.23 zeigt den Aufbau der System-ADL. In der Global-Sektion werden zunächst globale Einstellungen festgelegt. In der Modules-Sektion werden sämtliche Module definiert, die referenziert werden können. Ein Modul entspricht dann dem bekannten Konzept aus HDLs. In der Toplevel-Sektion können vorher definierte Module instanziiert und mit anderen Modulen verbunden werden. In der Configuration-Sektion werden dann Konfigurationen modelliert.

Abbildung 5.23: Aufbau der System-ADL

5.4.1 Sektion „Global“

Die Sektion beinhaltet globale Einstellungen.

Abbildung 5.24: System-ADL: Aufbau der Global-Sektion

Frequency
gibt die Default-Frequenz für sämtliche Module an. Die Frequenz kann von jedem Modul separat überschrieben werden.

BootConfiguration
gibt die Konfiguration an, die beim Booten geladen werden soll.

5.4.2 Sektion „Modules“

Die Modules-Sektion beschreibt sämtliche Module der System-ADL.

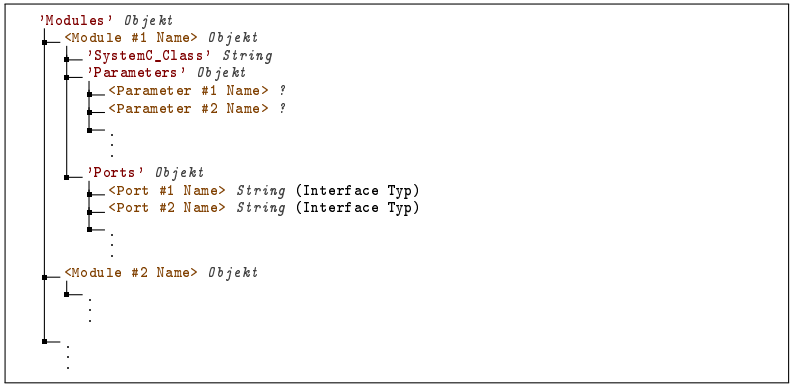


Abbildung 5.25: System-ADL: Aufbau der Modules-Sektion

SystemC_Class
enthält den Namen der SystemC-Klasse, die das Modul simulieren kann.

Parameters
gibt eine Liste von Simulationsparametern an, die an die SystemC-Klasse übergeben werden. Der Inhalt jedes Parameters ist flexibel.

Ports
beschreibt sämtliche Ports des Moduls. Jedem Portnamen ist ein Interface-Typ zugeordnet. Wenn zwei Ports verbunden werden, dann muss das Interface übereinstimmen.

5.4.3 Sektion „Toplevel“

In der Toplevel-Sektion können Module instanziiert und miteinander verbunden werden. Verbindungen zwischen Modulen können an zwei Stellen innerhalb der Toplevel-Sektion modelliert werden. Entweder direkt in der Beschreibung einer Modul-Instanz oder außerhalb einer Instanz. In beiden Fällen existiert dabei jeweils eine Connect-Sektion.

Instances
enthält alle Modul-Instanzen.

Module
gibt den Modulnamen der Instanz an.

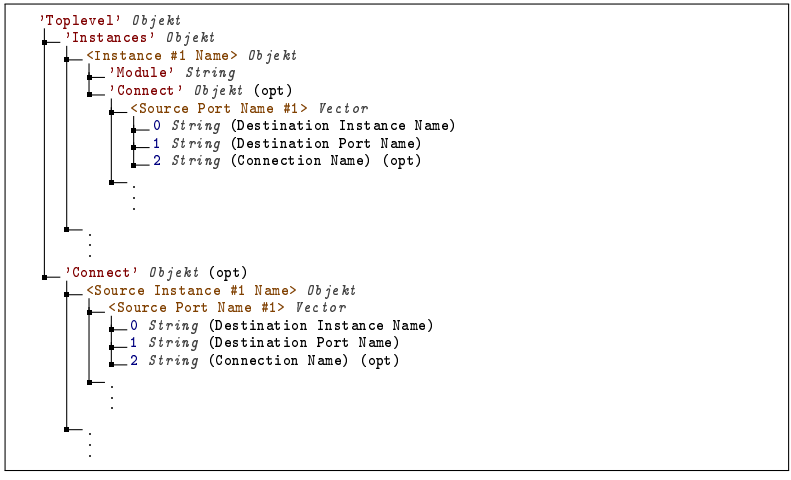


Abbildung 5.26: System-ADL: Aufbau der Toplevel-Instances-Sektion

Connect
: ist optional und erlaubt ein oder mehrere Ports (Source Port) der Instanz mit dem Port (Destination Port) einer anderen Instanz (Destination Instanz) zu verbinden. Optional kann der Verbindung zur Identifikation ein Name (Connection Name) angegeben werden.

Connect
: enthält alle externen Verbindungen zwischen Instanzen. Genau wie in der Connect-Sektion innerhalb einer Instanz wird ein Port einer Quellinstanz mit dem Port einer Zielinstanz verbunden.

5.4.4 Sektion „Configurations“

In der Configurations-Sektion können Konfigurationen definiert werden, die z.B. eine virtuelle Prozessorinstanz der Kahrisma-Architektur repräsentieren. Für das Laden einer Konfiguration wird eine vorgegebene Anzahl an Ressourcen benötigt. Wenn eine Konfiguration instanziiert bzw. geladen wurde, verhält sie sich wie ein Prozessor, der vorher in der Core-ADL spezifiziert wurde. Die benötigten Ressourcen werden mittels einer Schablone (Template) definiert. Die Schablone ist dabei unabhängig von einer konkreten Modulinstanz. Vielmehr werden sämtliche benötigten Module aus Toplevel zusammen mit deren Verbindungen untereinander spezifiziert. Die Ressourcen-

Spezifikation innerhalb der Configurations-Sektion ist dabei ähnlich zu der Spezifikation innerhalb der Toplevel-Sektion, mit dem Unterschied, dass kein vollständiges System sondern ein Subsystem, das für eine Konfiguration erforderlich ist, spezifiziert ist.

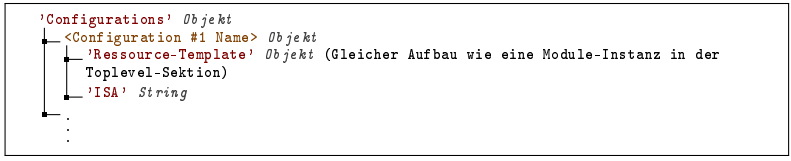


Abbildung 5.27: System-ADL: Aufbau der Toplevel-Instances-Sektion

Ressource-Template
enthält den Aufbau des Ressourcen-Templates. Es hat den gleichen Aufbau wie die Toplevel-Sektion und beinhaltet eine Instances- und eine optionale Connect-Sektion. Die beschriebenen Instanzen und Verbindungen des Templates werden dann auf die Instanzen von Toplevel abgebildet, um sämtliche mögliche Konfigurationsmöglichkeiten zu bekommen. Dabei wird eine Template-Instanz nur anhand des Typen, also dem Modulnamen, verglichen. Genauso muss bei einer Verbindung nur das Interface der Verbindung übereinstimmen. Der Portname z.B. wird ignoriert.

ISA legt fest, welche Funktion die Konfiguration hat, sobald sie konfiguriert bzw. instanziiert ist. Der Parameter referenziert ein Target aus der Core-ADL (siehe Abschnitt 5.17).

5.5 Charakterisierung

In diesem Kapitel wurden zwei Architekturbeschreibungssprachen vorgestellt, die gemeinsam rekonfigurierbare Prozessorarchitekturen wie die Kahrisma-Architektur beschreiben können. Die Abstraktionsebenen der Beschreibung wurden für eine bestmögliche Generierung des Softwareframeworks gewählt. Im Folgenden werden die Features der ADLs charakterisiert.

Trennung zwischen Core- und System-ADL Die Architekturbeschreibungssprache für das Softwareframework wurde in zwei Hauptkomponenten geteilt: die Core- und System-ADL. Dabei beschreibt die Core-ADL das Verhalten einer Konfiguration bzw. einer virtuellen Prozessorinstanz und die System-ADL beschreibt die Struktur der gesamten MPSoC-Architektur. Beide ADLs befinden sich in unterschiedlichen Domänen (Verhalten vs. Struktur) und Abstraktionsebenen

(Algorithmisch vs. System). Abbildung 5.1 zeigt die Eingruppierung im Y-Diagramm.

Mixed-ISA Core-ADL Die Core-ADL erlaubt die Spezifikation einer rekonfigurierbaren Befehlssatzarchitektur. Dies beinhaltet die Beschreibung sämtlicher Befehlssatzarchitekturen bzw. Konfigurationen, wie eine Konfiguration gewechselt wird und wie sich der interne Zustand des Prozessors bei einem Wechsel verändert. Eine Befehlssatzarchitektur wird innerhalb der ADL in verschiedenen Sektionen beschrieben, wobei zwischen ISA-abhängige und ISA-unabhängige Sektionen unterschiedenen wird. Letztere erlauben es, einen Großteil einer Beschreibung zwischen mehreren Befehlssatzarchitekturen wiederzuverwenden.

Beschreibung von RISC- bis Clustered-VLIW-Befehlssatzarchitekturen Die Core-ADL unterstützt die Beschreibung von RISC-, VLIW- und Clustered-VLIW-Befehlssatzarchitekturen, wie sie von den einzelnen Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur der Kahrisma-Architektur benötigt werden. Dazu können in der ADL verschiedene Registerspeicher, der Aufbau einer Instruktion sowie das Inter-Cluster-Kommunikationsmodell beschrieben werden.

Unterstützung rekonfigurierbaren Architekturen Die Kombination aus Core- und System-ADL erlaubt die Beschreibung von rekonfigurierbaren Architekturen wie der Kahrisma-Architektur. In der Core-ADL wird das Verhalten sämtlicher möglicher Konfigurationen modelliert. In der System-ADL wird dann die Struktur des Gesamtsystems sowie der Ressourcenverbrauch der Konfiguration im Gesamtsystem beschrieben.

Kompakte, nicht-redundante Beschreibung Beide ADLs bauen auf einer allgemeinen Datenbeschreibungssprache auf, die strukturelle hierarchische Daten in einem Textdokument beschreiben kann. Die Datenbeschreibungssprache erlaubt zusätzlich die Verwendung von Variablen und konstanten mathematischen Ausdrücken. Zusätzlich können Schleifen und Bedingungen mittels „if“- und „for“-Konstrukten erzeugt werden. Dadurch wird zum einen eine Beschreibung von regulären Strukturen insbesondere innerhalb der System-ADL ermöglicht. Zum anderen kann die ADL-Beschreibung parametrierbar gestaltet werden, so dass man z.B. die Anzahl der Register zentral in der Core-ADL oder die Anzahl an Prozessoren zentral in der System-ADL festgelegt.

Innerhalb der Core-ADL wurden zudem auf höherer Sprachebene viele Konstrukte realisiert, die eine kompakte Beschreibung von verschiedenen Konfigurationen einer rekonfigurierbaren Befehlssatzarchitektur ermöglicht. Im Einzelnen wurden folgende Methodiken in der Core-ADL umgesetzt:

Kompakte Registerbeschreibung Bei den logischen Registern kann man reguläre Registernamen automatisch mit einem Eintrag erzeugen lassen, z.B. $r0$ bis $r31$.

Vererbbare Operationsformate Die Operationsformate können von anderen Operationsformaten ergeben und diese bei der Vererbung verändern. Dadurch können redundante Felder an einer Stelle festgelegt werden.

Operationsformate mit mehreren Subformaten Ein Operationsformat kann aus mehreren Subformaten bestehen. Dadurch kann z.B. ein Operationsformat mehrere Kodiervarianten enthalten. Dadurch braucht man in der Operationsbeschreibung nur eine Beschreibung für gleichartige Operationen.

Automatische Umformung und Spezialisierung Bei der semantischen Beschreibung kann man für Nodes in LLVM-Pattern Umformregeln angeben (z.B. das rechte/linke Neutralelement oder ob der Knoten kommutativ ist). Diese Regeln werden dann für die automatische Spezialisierung und Umformung verwendet, um eine Vielzahl von Verwendungsmöglichkeiten einer Operation zu extrahieren.

Parametrierbare Beschreibung Durch die Verwendung einer Datenbeschreibungssprache mit Variablen und Kontrollflussstrukturen kann eine ADL-Beschreibung parametrierbar gehalten werden. Somit kann man z.B. bei der System-ADL die Anzahl der Ressourcen zentral festlegen (z.B. durch Höhe und Breite des Arrays) oder in der Core-ADL kann die Anzahl an Registern flexibel gehalten werden. Mit diesem Konzept wird eine Entwurfsraumexploration mittels der ADL unterstützt.

Compiler-, Simulator- und Assemblergenerierung Die Core-ADL enthält eine verhaltensorientierte Beschreibung einer Konfiguration bzw. einer virtuellen Prozessorinstanz. Diese Beschreibung ist dafür optimiert um einen Compiler, Core-Simulator und Assembler zu generieren.

Die System-ADL enthält eine strukturorientierte Beschreibung des Gesamtsystems und kann für einen System-Simulator verwendet werden.

5.6 Zusammenfassung

Die ADL bildet die Grundlage für das Softwareframework zur Unterstützung von rekonfigurierbaren Prozessorarchitekturen. Dazu lässt sich die ADL in drei Ebenen einteilen, die jeweils durch einzigartige Features zur Unterstützung rekonfigurierbarer Prozessorarchitekturen sowie zur Umsetzung der Ziele aus Kapitel 4 beitragen. Zunächst erlaubt die System-ADL die Beschreibung von rekonfigurierbaren Prozessorarchitekturen innerhalb der strukturellen Domäne auf höchster Abstraktionsebene. Hier werden sämtliche rekonfigurierbaren Module festgelegt, die zu virtuellen Prozessorinstanzen zusammengeschaltet und konfiguriert werden können. Dabei werden sämtliche Ressourcen, die zum Erzeugen einen Konfiguration benötigt werden, festgelegt.

Die Core-ADL befindet sich dann in der Verhaltens-Domäne. Hier wird das Verhalten der einzelnen Konfiguration der Prozessorarchitektur festgelegt. Das Verhalten zeichnet sich durch die Befehlssatzarchitektur der möglichen virtuellen Prozessorinstanzen aus. Dazu bietet die ADL die Möglichkeit, mehrere Befehlssatzarchitekturen gleichzeitig zu spezifizieren, wodurch der Begriff „Mixed-ISA“ geprägt wird. Durch die Verwendung von zentralen Sektionen, die zwischen den verschiedenen ISAs gemeinsam verwendet werden, kann sowohl der Spezifikationsaufwand einer neuen Konfiguration reduziert werden als auch das Verhalten im Falle einer Rekonfiguration modelliert werden.

Beide ADLs, die Core-ADL und die System-ADL, bauen auf einer allgemeinen Datenbeschreibungssprache auf. Diese Datenbeschreibungssprache ist vergleichbar mit XML und JSON, bietet allerdings die Möglichkeit der Evaluation konstanter mathematischer Ausdrücke, der Verwendung von „if“- und „for“-Kontrollflussanweisungen sowie den Einsatz von Variablen. Dadurch wird eine kompakte Beschreibung von regulären Strukturen insbesondere innerhalb der System-ADL ermöglicht. Innerhalb der Core-ADL kann die Methodik eingesetzt werden, um z.B. kompakt Registerlisten zu erzeugen. Zusätzlich kann durch die Verwendung von Variablen und Kontrollflussstrukturen eine ADL-Beschreibung parametrierbar gehalten werden. Somit kann z.B. bei der System-ADL die Anzahl der Ressourcen zentral festlegen (z.B. durch Höhe und Breite des Arrays) oder in der Core-ADL kann die Anzahl an Registern flexibel gehalten werden. Mit diesem Konzept wird eine Entwurfsraumexploration mittels der ADL unterstützt.

Durch die Trennung einer Beschreibung in Core- und System-ADL auf unterschiedlichen Domänen und Abstraktionsebenen, der mixed-ISA Unterstützung innerhalb der Core-ADL, der Unterstützung von RISC bis Clustered-VLIW-Befehlssatzarchitekturen, der Unterstützung von rekonfigurierbaren Architekturen innerhalb einer kompakten, parametrierbaren, nicht-redundanten Beschreibung bietet die entwickelte ADL die beste Voraussetzung für das Softwareframework zur Unterstützung der rekonfigurierbaren Befehlssatzarchitektur auf Compiler-, Simulator- und Assemblerebene sowie der Entwurfsraumexploration der Kahrisma-Architektur auf System- und Prozessorebene.

6 Realisierung des mixed-ISA Softwareframeworks

In diesem Kapitel wird die Realisierung des mixed-ISA Softwareframeworks vorgestellt. Dazu wird in Abschnitt 6.1 zunächst ein Überblick über das Softwareframework und dessen Komponenten gegeben, die anschließend im Detail behandelt werden. Abschnitt 6.2 beschreibt zunächst die mixed-ISA Erweiterung der C/C++-Programmiersprache als Eingangssprache für das Framework. In Abschnitt 6.3 wird das CoreGen-Werkzeug vorgestellt, das die Benutzer-Retargierbarkeit der nachfolgenden Komponenten durch die ADL aus Kapitel 5 ermöglicht. Danach werden der mixed-ISA LLVM-Compiler (Abschnitt 6.4), die mixed-ISA Binärwerkzeuge (Abschnitt 6.6), der Core-Simulator (Abschnitt 6.7), der System-Simulator (Abschnitt 6.8) und der ISA-Partitionierer (Abschnitt 6.9) vorgestellt. Abschnitt 6.10 fasst das Kapitel zusammen.

6.1 Überblick

Abbildung 6.1 zeigt einen detaillierten Überblick über das Softwareframework zur Unterstützung von Prozessoren mit variablen Befehlssatzarchitekturen. Die Eigenschaft der Unterstützung von variablen Befehlssatzarchitekturen wird bei dem Softwareframework und bei den Werkzeugen als „mixed-ISA“ bezeichnet. Dabei erlaubt das Softwareframework die Entwicklung von mixed-ISA Applikationen mittels dem statischen, dynamischen oder automatischen mixed-ISA Programmiermodell, die im Konzeptionskapitel in Abschnitt 4.2 vorgestellt wurden. Dadurch wird unter anderem die rekonfigurierbare Befehlssatzarchitektur der Kahrisma-Architektur unterstützt. Weiterhin ist das Softwareframework durch die mixed-ISA Architekturbeschreibungssprache (siehe Abschnitt 5) weitgehend benutzer-retargierbar gehalten, wodurch erst die verschiedenen Befehlssatzarchitekturen unterstützt werden konnten aber auch eine Entwurfsraumexploration ermöglicht wird. Dies erlaubt für die Kahrisma-Architektur die Unterstützung der Designzeit-Flexibilität.

Der Kern des Softwareframeworks bildet der LLVM-basierte Compiler. Als Frontend wurde Clang ausgewählt, das zunächst die C/C++-Eingangssprache in die *LLVM-Zwischendarstellung* (LLVM-IR, engl. *LLVM Intermediate Representation*) übersetzt. Diese wird dann im Middleend durch plattformunabhängige Optimierungsalgorithmen optimiert und im Backend in Assembler-Dateien übersetzt. Der Assembler assembliert die Dateien in Objektdateien. Eine C/C++-Anwendung besteht für gewöhnlich,

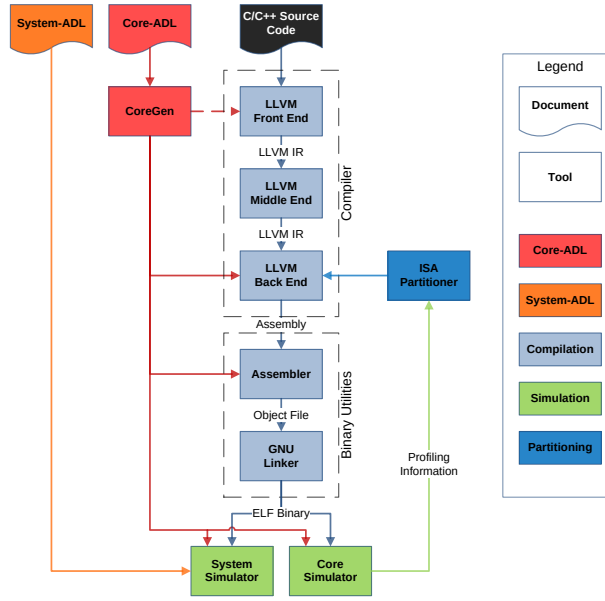


Abbildung 6.1: Überblick über das mixed-ISA Softwareframework

wie in C/C++-Compilern üblich, aus mehreren Übersetzungseinheiten. Der Kompiliervorgang vom Frontend bis zum Assembler wird für jede Übersetzungseinheit separat durchlaufen. Sämtliche daraus resultierende Objektdateien werden dann vom Linker in eine ausführbare Datei zusammengesetzt.

Der LLVM-basierte Compiler ist mit Hilfe der Core-ADL benutzer-retargierbar gehalten. Als Technik kommt hierbei ein Generator, genannt CoreGen, zum Einsatz, der mittels Metaprogrammierung Teile vom Quellcode der unterschiedlichen Werkzeuge des Compilers und der Binärwerkzeuge generiert. Daher muss der Compiler jedes Mal neu übersetzen werden, wenn sich die Core-ADL ändert. Als Alleinstellungsmerkmal kann das gesamte Softwareframework mit mehreren Befehlssatzarchitektur umgehen. Dies fängt bei deren Beschreibung in der Core-ADL an. Der Compiler und die Binärwerkzeuge können, basierend auf der Core-ADL, mehrere Befehlssatzarchitekturen gleichzeitig verarbeiten und können eine ausführbare Datei mit unterschiedlichen Befehlssatzarchitekturen erzeugen. Anschließend kann eine Simulation mit wechselnden Befehlssatzarchitekturen durchgeführt werden. Dieses Allein-

stellungsmerkmal wird innerhalb dieser Arbeit allgemein als „mixed-ISA“ bezeichnet und durchzieht das gesamte Softwareframework.

Der Compiler und das Framework unterstützen drei unterschiedliche Programmiermodelle für mixed-ISA Applikationen. Bei der einfachsten Form, der **statischen mixed-ISA Programmierbarkeit**, kann der Compiler Applikation für mehrere ISAs übersetzen. Die Menge an Ziel-ISAs kann beim Compiler per Kommandozeilenparameter angegeben werden. Die resultierende ausführbare Datei der Applikation enthält dabei Maschinencode für alle übersetzten ISAs. Vor dem Ausführen oder der Simulation einer Anwendung kann dann ausgewählt werden, mit welche ISA bzw. Konfiguration die Applikation ausgeführt werden soll. Ein Wechsel der ISA zur Laufzeit der Applikation ist bei diesem Programmiermodell nicht möglich.

Beim zweiten Programmiermodell, der **dynamischen mixed-ISA Programmierbarkeit**, können Anwendungen entwickelt werden, die während der Laufzeit die Befehlssatzarchitektur ändern. Zu diesem Zweck wurde die C/C++-Programmiersprache um mixed-ISA Sprachkonstrukte erweitert, die es erlauben, pro Funktion die Befehlssatzarchitektur festzulegen. Der Compiler erkennt automatisch Funktionsaufrufe mit unterschiedliche Befehlssatzarchitekturen und fügt in diesem Fall automatisch Umschaltcode zur Rekonfiguration und damit zur Änderungen der Befehlssatzarchitektur ein.

Das dritte Programmiermodell ist die **automatische mixed-ISA Programmierbarkeit**. Hierbei muss der Benutzer nicht manuell festlegen, welche Funktion für welche Befehlssatzarchitektur übersetzt werden soll. Stattdessen wird automatisch eine effiziente ISA pro Funktion unter Berücksichtigung des Rekonfigurationsoverheads ausgewählt. Ein weiteres Werkzeug, der ISA-Partitionierer, wählt dabei für jede Funktion eine effiziente ISA unter Berücksichtigung der Zielfunktion und Rahmenbedingungen basierend auf Profiling-Informationen aus. Dadurch kann z.B. als Zielfunktion der Ressourcenverbrauch reduziert werden unter der Rahmenbedingung, dass sie die Performanz maximal um 10% verschlechtern darf.

Zur Evaluierungen und zum Testen übersetzter mixed-ISA Applikationen stehen zwei Simulatoren zur Verfügung. Der Core-Simulator ist auf die Simulation einer Kahrisma-Konfiguration optimiert. Beim Start der Simulation kann man die Konfiguration und damit die ISA festlegen. Der Simulator ist ebenfalls mit Hilfe der Core-ADL benutzer-retargierbar gehalten und unterstützt sämtliche ISAs, die in der ADL spezifiziert sind. Der Simulator kann neben der funktionalen Simulation noch eine Approximation der Performanz der virtuellen Prozessorinstanzen durchführen. Dabei wird das zeitliche Verhalten der speziellen Pipeline der Kahrisma-Prozessorarchitektur sowie der Speicherhierarchie approximiert.

Zusätzlich zum Core-Simulator gibt es noch einen System-Simulator, der die Simulation mehrerer virtueller Prozessorinstanzen gleichzeitig unterstützt. Der System-Simulator ist mit Hilfe der System-ADL ebenfalls benutzer-retargierbar gehalten. Die System-ADL stellt die Struktur der Prozessorarchitektur auf Systemebene bereit, so dass verschiedene Ausprägungen der Prozessorarchitektur auf Systemebenen explo-

riert werden können. Sowohl der System-Simulator als auch die System-ADL unterstützen dabei das Konzept der Rekonfiguration von virtuellen Prozessorinstanzen.

6.2 Mixed-ISA Erweiterung der C/C++-Programmiersprache

Zur Realisierung des dynamischen mixed-ISA Programmiermodells für Architekturen, die verschiedene Befehlssatzarchitekturen unterstützen, wurde für das Softwareframework die C/C++-Programmiersprache erweitert. Die Erweiterung der Programmiersprache ist unter meiner Anleitung innerhalb der Masterarbeit von Michael Schöffler entstanden [123].

6.2.1 Syntax

Mit der Erweiterung der C/C++-Programmiersprache lässt sich pro Funktion festlegen, für welche ISA bzw. Konfiguration diese kompiliert werden soll. Dazu wurde die C/C++-Programmiersprache um Funktionsattribute zur Steuerung der ISA erweitert. Der GCC- und Clang-Compiler bieten bereits ein allgemeines Konzept für Funktionsattribute, die nicht im C/C++-Standard enthalten sind, an. Diese werden mit dem Schlüsselwort `__attribute__` eingeleitet und wurden als Grundlage für die Erweiterung verwendet.

Im Folgenden wird der erweiterte C/C++-Syntax der mixed-ISA Funktionsattribute in EBNF Notation gemäß ISO/IEC 14977 : 1996(E) [104] Standard beschrieben:

FuncAttribute ::= '`__attribute__`', '((', AttributeISA , '))';

AttributeISA ::= '`target_isa`', '(', AttributeISAList , ')';

AttributeISAList ::= AttributeShould | AttributeMust | AttributeStay | AttributeAuto

AttributeShould ::= '`SHOULD`', ISAList;

AttributeMust ::= '`MUST`', ISAList;

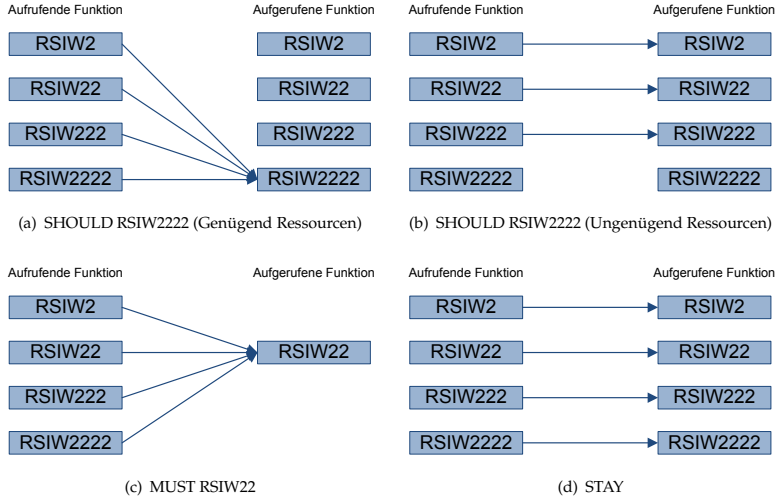
AttributeStay ::= '`STAY`';

AttributeAuto ::= '`AUTO`';

ISAList ::= ISAName, { ',', ISAName }-;

6.2.2 Funktionsmodi

Wie dem EBNF Syntax zu entnehmen ist, werden vier Modi zur Steuerung der Befehlssatzarchitektur einer Funktion unterstützt. Abhängig vom Modus der ISA der aufrufenden Funktion und der zur Laufzeit verfügbaren Ressourcen wird entschieden, in welcher Befehlssatzarchitektur eine Funktion ausgeführt wird. Bei einem Wechsel der Befehlssatzarchitektur fügt der Compiler automatisch Umschaltcode zum Wechsel der ISA ein.



(a) SHOULD RSIW2222 (Genügend Ressourcen)

(b) SHOULD RSIW2222 (Ungenügend Ressourcen)

(c) MUST RSIW22

(d) STAY

Abbildung 6.2: Visualisierung der unterschiedlichen ISA-Modi zur Steuerung der Befehlssatzarchitektur in C/C++

Abbildung 6.2 visualisiert das Verhalten des Compilers und den Ablauf zur Laufzeit abhängig von der ISA der aufrufenden Funktion. Im Folgenden werden die vier Modi erklärt:

SHOULD Mit dem SHOULD-Modus wird ausgedrückt, dass eine Funktion mit der angegebenen Befehlssatzarchitektur ausgeführt werden **soll**, aber nicht muss. Für den Compiler bedeutet dies, dass er die Funktion für sämtliche Befehlssatzarchitekturen kompiliert; d.h., dass Code zum Ausführen in sämtlichen Befehlssatzarchitektur vorhanden ist. Bei einem Aufruf der Funktion von einer anderen Funktion mit unterschiedlicher Befehlssatzarchitektur wird zunächst überprüft, ob genügend Ressourcen zur Verfügung stehen. Falls genügend Ressour-

cen zur Verfügung stehen wird die Befehlssatzarchitektur gewechselt. Andernfalls bleibt die Befehlssatzarchitektur identisch zur der Befehlssatzarchitektur der aufrufenden Funktion.

Zum Beispiel gibt das Attribute

```
__attribute__((target_isa(SHOULD, RSIW2222)))
```

an, dass die Funktion mit der ISA RSIW2222 ausgeführt werden soll. Abbildung 6.2(b) zeigt das Laufzeitverhalten für eine solche Funktion im Fall von genügend Ressourcen.

Wenn ungenügend Ressourcen zur Verfügung stehen, gibt es theoretisch mehrere Möglichkeiten, wie mit dieser Situation umgegangen wird. Naheliegend wäre z.B. in diesem Fall eine Konfiguration zu wählen, die möglichst die zur Verfügung stehenden Ressourcen ausnutzt. In diesem Beispiel könnte man z.B. auf RSIW222 wechseln, wenn RSIW2222 nicht verfügbar ist. Der Nachteil dieses Verhaltens wäre allerdings, dass man zur Laufzeit viele Überprüfungen durchführen müssten um herauszufinden, welche jetzt die nächste effiziente ISA ist. Zur Minimierung des Rekonfigurationsoverheads verhält sich daher SHOULD identisch zu STAY, wenn nicht genügend Ressourcen zur Verfügung stehen. Somit muss zur Laufzeit nur eine Überprüfung durchgeführt werden und falls diese fehlschlägt wird die ISA beibehalten.

MUST Mit dem MUST-Modus wird ausgedrückt, dass eine Funktion mit der angegebenen Befehlssatzarchitektur ausgeführt werden **muss**. Der Compiler übersetzt dann eine Funktion nur für die angegebenen ISA (Abbildung 6.2(a)). Im Falle von ungenügenden Ressourcen kann zur Laufzeit nicht auf eine andere ISA gewechselt werden und es wird eine Exception ausgelöst.

Zum Beispiel gibt das Attribute

```
__attribute__((target_isa(MUST, RSIW2222)))
```

an, dass die Funktion mit der ISA RSIW2222 ausgeführt werden muss. Abbildung 6.2(a) zeigt das Laufzeitverhalten für eine solche Funktion.

STAY Mit dem STAY-Modus wird ausgedrückt, dass die ISA für die Funktion nicht gewechselt werden soll; d.h. die Funktion wird in der gleichen ISA ausgeführt, wie die aufrufende Funktion. Die Funktion wird daher für jede ISA kompiliert, da erst zur Laufzeit bekannt ist, in welcher ISA die Funktion ausgeführt wird. Beim Aufruf der Funktion entsteht kein Rekonfigurationsoverhead, weil die ISA nie gewechselt wird.

Zum Beispiel gibt das Attribute

```
__attribute__((target_isa(STAY)))
```

an, dass die Funktion mit der gleichen ISA wie die aufrufende Funktion ausgeführt wird. Abbildung 6.2(d) zeigt das Laufzeitverhalten für eine solche Funktion.

AUTO gibt an, dass der Compiler selbst entscheiden soll, welcher Modus verwendet werden soll. Dabei verwendet der Compiler profile-guided Kompilierungstechniken zur Auswahl einer effizienten ISA.

```
__attribute__((target_isa(AUTO)))
```

6.2.3 Standardeinstellung

Per Standardeinstellung sind alle Funktionen im STAY-Modus. Wenn in einer Applikation keine Attribute angegeben sind hängt es von der ISA der main-Funktion ab, mit welcher ISA die gesamte Applikation ausgeführt wird.

Wenn im Compiler profile-guided Optimierung aktiviert ist, sind per Default alle Funktionen im AUTO-Modus. Dann wird automatisch für sämtliche Funktionen der Applikation eine effiziente ISA-Partitionierung berechnet.

6.2.4 Funktionspointer

Eine Sonderstellung nehmen Funktionspointer ein. Bei Funktionspointern ist erst zur Laufzeit bekannt, welche Funktion als nächstes ausgeführt wird. Um das Verhalten, beschrieben durch die Funktionsmodi, ebenfalls für Funktionspointer nachbilden zu können, müsste man viele Laufzeitüberprüfungen einbauen. Dies würde zu einem großen Overhead führen, selbst wenn die ISA bei einem Funktionspointer nicht gewechselt wird.

Aus diesem Grund wurde innerhalb dieser Arbeit das Prinzip implementiert, dass bei einem Funktionspointer die ISA nicht gewechselt wird, sondern ein Funktionspointer immer im STAY-Modus ausgeführt wird. Dies impliziert, dass eine Funktion, die durch einen Funktionspointer aufgerufen wird, immer sämtliche ISAs implementieren muss. In der Erweiterung der C/C++-Programmiersprache wird dies durch die Bedingung realisiert, dass man von Funktionen mit MUST-Modi keine Funktionspointer erzeugen darf.

6.2.5 Externe Funktionsdeklaration

Bei C/C++-Compilern werden Applikationen in sog. Übersetzungseinheiten übersetzt. Eine Übersetzungseinheit ist ein Durchlauf eines Compilers, der für gewöhnlich eine Implementierungsdatei in eine Objektdatei übersetzt. Mehrere Implementie-

rungsdateien werden dann im Linker zusammengeführt. Damit eine Übersetzungseinheit auf Funktionen, die von anderen Übersetzungseinheiten implementiert werden, zugreifen kann, werden für gewöhnlich in sog. Header-Dateien die Schnittstelle der Funktionen mit einer externen Deklaration bekannt gegeben.

Bei dem mixed-ISA Konzept fügt der Compiler Code zum Umschalten einer ISA innerhalb der aufrufenden Funktion ein. Daher müssen beim Übersetzen einer Funktion sämtliche Funktionsmodi der aufgerufenen Funktion bekannt sein, selbst wenn diese sich in einer anderen Implementierungsdatei befindet. Aus diesem Grund muss ebenfalls bei externen Funktionsdeklarationen der Funktionsmodi angegeben werden und mit dem Funktionsmodi der Implementierung übereinstimmen.

6.2.6 Fazit

Durch die vorgestellte Erweiterung der C/C++-Programmiersprache um Attribute zur ISA-Beschreibung werden sämtliche Programmiermodelle umgesetzt, die im Konzeptionskapitel unter 4.5 vorgestellt wurden:

Statische mixed-ISA Programmiermodell Das statische mixed-ISA Programmiermodell wird dadurch realisiert, dass per Default alle Funktionsattribute auf STAY sind. Die Main-Funktion wird für alle ISAs übersetzt und man kann beim Starten der Anwendung auswählen in welcher ISA die Main-Funktion ausgeführt werden soll. Bei jedem Funktionsaufruf bleibt die ISA gleich, da alle Funktionen auf STAY gesetzt sind. Dadurch bleibt die ISA während der gesamten Laufzeit der Anwendung gleich.

Dynamische mixed-ISA Programmiermodell Das dynamische mixed-ISA Programmiermodell wird durch die Erweiterung der C/C++-Programmiersprache um Attribute zur ISA-Beschreibung realisiert. Dazu wurden in diesem Kapitel der SHOULD-, STAY- und MUST-Modus vorgestellt.

Automatische mixed-ISA Programmiermodell Das automatische mixed-ISA Programmiermodell wird anhand des AUTO-Modus realisiert. Durch Setzen des Default-Modus auf AUTO kann man ohne Änderung einer Anwendung diese automatisch für eine effiziente ISA-Partitionierung optimieren lassen. Darüber hinaus ist eine Mischung des dynamischen mit dem automatischen Modus möglich.

6.3 CoreGen-Werkzeug

Das *CoreGen*-Werkzeug (engl. *Core Generator*) parst eine ADL-Beschreibung und kompiliert diese in verschiedene Ausgabeformate, die dann zur Erzeugung der retargierbaren Werkzeuge, also den LLVM-Compiler, Assembler und Simulator, verwendet

werden. Dabei kommt die Methodik der Metaprogrammierung zum Einsatz, da unter anderem C++-Quellcode der Werkzeuge generiert wird. Daraus folgt, dass die retargierbaren mixed-ISA Werkzeuge teilweise neu gebaut werden müssen, damit Änderungen an der ADL-Beschreibung wirksam werden.

Eine erste Version der Werkzeugs wurde in meiner Diplomarbeit [119] realisiert. Dieses wurde dann innerhalb dieser Arbeit stark weiterentwickelt.

Ein allgemeiner Überblick über das *CoreGen*-Werkzeug ist in Abbildung 6.3 zu sehen. Das CoreGen-Werkzeug in verschiedene Phase untergliedert:

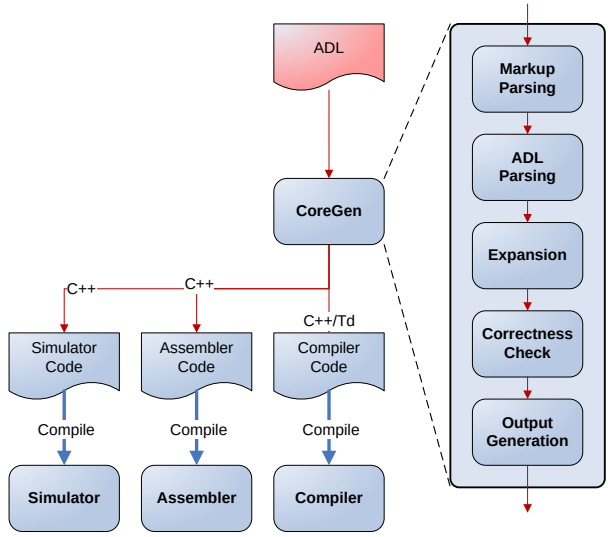


Abbildung 6.3: Aufbau und Verwendung des CoreGen-Werkzeugs (Veröffentlicht [131])

Markup Parsing Als erstes wird die Datenbeschreibungssprache (siehe Abschnitt 5.2) geparst, interpretiert und in eine baumartige Datenstruktur umgewandelt. Bei der Interpretation werden konstante mathematische Ausdrücke ausgerechnet, Variablen aufgelöst und Funktionen und Kontrollstrukturen (if, for) ausgeführt.

ADL Parsing analysiert den Baum und überführt dessen Inhalt in interne Datenstrukturen, die ähnlich den Sektionen der Core-ADL aufgebaut sind. Dabei wird sowohl eine syntaktische als auch semantische Überprüfung der Daten im Baum

vorgenommen. Bei der semantischen Überprüfung werden insbesondere die Referenzen zwischen den Sektionen, wie sie in Abbildung angegeben sind, überprüft. Z.B. müssen alle verwendeten Register vorher definiert worden sein, die LLVM-Pattern dürfen nur in der Nodes-Sektion definierte Knoten verwenden und kein Feld im Befehlsformat darf über die Bitbreite des Formats hinausgehen.

Expansion ist der komplexeste Pass des CoreGen-Werkzeugs. Er leitet automatisch Information über die Zielarchitektur aus deren kompakten Beschreibung innerhalb der ADL ab.

Operation Splitting (Mehrere Befehlskodierungen) Eine Aufgabe von Expansion ist das Teilen von Operationen. Um eine kompakte Repräsentation von Operation innerhalb der ADL zu ermöglichen, können mehrere Befehlskodierungen innerhalb eines Operationsformats definiert werden (siehe Abschnitt 5.3.5). Für spätere Schritte innerhalb des ADL-Compilers ist es wichtig, dass eine Operation genau eine Kodierung hat. So werden Operationen mit mehreren Befehlskodierungen in mehrere Operationen mit jeweils einer Befehlskodierung aufgeteilt.

Operation Splitting (Besondere Felder) Ein zweiter Grund für das Aufteilen von Operationen ist die Verwendung von Feldern, die Simulationsquellcode und/oder LLVM-Fragmente enthalten. Im Simulationsquellcode innerhalb einer Operation wird dann ein Platzhalter für das Feld durch das Codefragment innerhalb des Feldes ersetzt. Für den Simulator muss dieser Platzhalter ersetzt werden, bevor für die Operation die finale Simulationsfunktion generiert werden kann. Somit wird eine Operation, die ein oder mehrere solche Felder enthält, anhand dieses Feldes geteilt. Dabei wird das Feld dann jeweils auf einen festen Wert gesetzt und die Ersetzung im Simulationsquellcode durchgeführt. Auf gleiche Weise wird bei LLVM-Fragmenten verfahren, die einen Knoten im LLVM-Pattern ersetzen.

LLVM Pattern Rewriting Bei dieser Aufgabe werden die LLVM-Pattern für die Befehlsauswahl im Compiler umgeschrieben. Durch die Ausnutzung von mathematischen Gesetzen kann aus einem gegebenen LLVM-Pattern semantisch äquivalente LLVM-Pattern generiert werden. Dazu sind im ADL-Compiler verschiedene Termersetzungsregeln enthalten, die auf die Menge der LLVM-Pattern einer Operation angewandt werden. Dadurch neu erstellte LLVM-Pattern werden zu der Menge der LLVM-Pattern einer Operation hinzugefügt, sofern sie noch nicht in der Menge vorhanden waren. Die Termersetzungsregeln werden so langen auf die LLVM-Pattern angewandt, bis keine neue LLVM-Pattern mehr erzeugt werden.

Als Termersetzungsregeln stehen das Kommutativgesetz sowie die automatische Spezialisierung von Eingangsoperanden zur Verfügung. Bei der automatischen Spezialisierung werden Eingangsoperanden auf konstante oder identische Werte gesetzt. So ist es möglich einen binären Knoten

innerhalb eines LLVM-Pattern durch Setzen des linken bzw. rechten Operanden auf das linke bzw. rechte neutrale Element des Knotens zu eliminieren. Dies spielt z.B. eine wichtige Rolle bei der Spezifikation von Load/-Store-Operationen innerhalb der ADL. Eine typische Load-Operation lädt ein Datum von einer Speicheradresse, die durch eine Addition von einem Register mit einem Immediate-Wert berechnet wird. Die Befehlsauswahl würde eine solche Operation nur auswählen, wenn ein Add-Knoten direkt einem Load-Knoten folgt. Durch das Setzen des Immediate-Werts auf 0 kann die Addition eliminiert werden und es können innerhalb der Befehlsauswahl Load-Knoten ohne folgendem Add-Knoten erkannt werden.

Operation Splitting (Spezialisierung) Bei der Spezialisierung werden bestimmte Operanden einer Operation auf feste Werte gesetzt. Durch die Spezialisierung kann ein spezielles Verhalten für den Compiler oder ein spezieller Syntax für den Assembler ausgedrückt werden. So kann man z.B. ein bedingten Sprung in einen unbedingten Sprung umgewandelt werden, indem die Bedingungen auf einen konstanten Wert gesetzt wird, so dass der Sprung immer genommen wird. Für den Assembler kann durch Spezialisierung ein alternativer Syntax festlegt werden, der die Lesbarkeit erhöht.

Die ADL unterstützt die Beschreibung von manuellen Spezialisierungen (siehe Sektion 5.3.6). Zusätzlich erzeugt der ADL-Compiler automatische Spezialisierungen mittels Termersetzungsregeln. Für jede Spezialisierung wird im Assembler und Compiler eine separate Operation benötigt. So werden für jede Spezialisierung neue Operationen ohne Spezialisierungen erzeugt, bei denen in der Kodierung die spezialisierten Felder auf feste Werte kodiert werden. Dadurch kann der Compiler und Assembler mit einer flachen Operationsliste arbeiten und nur der ADL-Compiler muss das Prinzip der Spezialisierung unterstützen.

Correctness Check In diesem Pass werden verschiedene komplizierte semantische Tests auf den Daten der ADL durchgeführt. Dies wird ergänzend zu den semantischen Überprüfungen während dem ADL Parsing durchgeführt, die sich hauptsächlich auf die Überprüfung von Referenzen beschränkt haben.

Unique Encoding Check Bei diesem Test wird die Eindeutigkeit der Kodierung für jeden Befehlssatz innerhalb der ADL überprüft, d.h. zwei Operationen dürfen nicht die gleiche Kodierung innerhalb des Instruktionsworts haben. Die Überprüfung ist keine triviale Aufgabe, da jede Operation aus unterschiedlichen Feldern bestehen kann, die sich anhand ihres Typs, Kodierung, Position und Länge unterscheiden können. So können zwei Felder nur dadurch eindeutig sein, dass sie unterschiedliche Kodierungen erlauben. Der Überprüfungsalgorithmus vergleicht alle Felder von zwei Operationen von links nach rechts. Sämtliche überlappenden Felder werden dabei individuell verglichen. Wenn die Position und Länge zwei-

er Felder übereinstimmen, können die Felder anhand ihres Inhaltes direkt verglichen werden und es ist möglich, disjunkte Kodierungen zu identifizieren. Ansonsten wird jedes Bit zweier Felder einzeln verglichen, wobei es keine Möglichkeit gibt, Felder anhand ihrer erlaubten Kodierungen zu unterscheiden.

Instruction Selector Completness Bei dem zweiten Test wird die Vollständigkeit der Befehlsauswahl für den Compiler überprüft. Eine vollständige Befehlsauswahl wird dann angenommen, wenn alle elementaren architekturunabhängigen LLVM-internen Befehle durch plattformspezifische Operationen abgebildet werden können. Ansonsten kann die Befehlsauswahl für bestimmte Programme fehlschlagen. Sämtliche nicht-elementare Knoten werden im Backend des Compilers durch semantisch äquivalente Codestücke innerhalb des Legalization-Passes ersetzt.

Output Generation Der letzte Pass des CoreGen-Werkzeugs generiert die Quellcodedateien vom Compiler, Simulator und Assembler. Dazu werden aus den Informationen, die in den internen Datenstrukturen gespeichert sind, C++- und Td-Dateien des Compilers sowie C++-Dateien des Assemblers und Simulators erzeugt. Die dynamisch generierten Quelldateien werden dann zusammen mit den statischen Quelldateien verwendet, um den Compiler, Assembler und Simulator zu erzeugen. Nähere Information zu den erzeugten Dateien befinden sich in den Abschnitten der jeweiligen Werkzeuge in diesem Kapitel.

6.4 Mixed-ISA LLVM-Compiler

In diesem Abschnitt wird der mixed-ISA LLVM-Compiler des Softwareframeworks beschrieben. Dieser baut auf der LLVM-Compiler-Infrastruktur auf. Der Abschnitt ist nach den einzelnen Komponenten des Compilers unterteilt. Abschnitt 6.4.1 beschreibt die Erweiterung der LLVM-Zwischendarstellung als Schnittstelle zwischen den Komponenten. Danach folgt das Frontend (Abschnitt 6.4.2), das Middle-End (Abschnitt 6.4.3) und das Backend (Abschnitt 6.5).

Eine erste Version des LLVM-Backends wurde in meiner Diplomarbeit [119] realisiert. Das Backend und die anderen Komponenten des Compilers wurden dann innerhalb dieser Arbeit sowie unter meiner Anleitung in den Abschlussarbeiten von Patrick Rieder [121, 118] und Michael Schöffler [123] weiterentwickelt.

6.4.1 Mixed-ISA Erweiterung der LLVM-Zwischendarstellung

Wie in Kapitel 2.4.3.1 beschrieben, besteht der Kern des LLVM-Projekts aus Bibliotheken, die für die Erzeugung eines Compilers verwendet werden. Ein wichtiges Element dieser ist die LLVM-Zwischendarstellung (siehe Abschnitt 2.4.3.2). Zur Unter-

Quelltext 6.1: Beispiel der mixed-ISA Erweiterung der LLVM-Assemblersprache der Zwischendarstellung

```
target datalayout = "E-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
      f32:32:32-f64:64:64-n32"
target triple = "kahrisma--"

define i32 @func_should(i32 %a, i32 %b) targetisa("SHOULD,RSIW2222") nounwind
      {
entry:
  %add = add nsw i32 %b, %a
  ret i32 %add
}

declare void @func_auto(...) targetisa("AUTO")
```

stützung der mixed-ISA Erweiterung der C/C++-Programmiersprache ist es unabdingbar, ebenfalls die LLVM-Zwischendarstellung zu erweitern, so dass die mixed-ISA Einstellungen der Eingangssprache in der LLVM-IR repräsentiert werden können. Diese werden dann vom Frontend des Compilers zum Backend durchgereicht und stehen dort zur Verfügung.

Die LLVM-IR kann in drei Formen auftreten, die homomorph zueinander sind, also jeweils die gleichen Daten repräsentieren können: Die LLVM-Assemblersprache, der LLVM-Bitcode und die Speicherdarstellung. Die Erweiterung jeder der drei Repräsentationen der LLVM-IR wird in den folgenden Abschnitten vorstellt:

Speicherdarstellung Die Speicherdarstellung besteht aus verschiedenen Klassen und Strukturen, die die Information der Zwischendarstellung innerhalb von LLVM speichern und zur Verfügung stellen. Hierbei wurde die Funktions-Klasse, die sämtliche Attribute eine Funktion speichert, um eine Zeichenkette erweitert, die alle mixed-ISA Einstellungen der Funktion enthält.

LLVM-Bitcode Der LLVM-Bitcode bietet eine speicherplatzeffiziente Möglichkeit zum Speicherung der LLVM-Zwischendarstellung im Binärformat auf der Festplatte an. Hierbei wurde der Bitcode-Reader bzw. -Writer so erweitert, dass er pro Funktion die zusätzliche mixed-ISA Zeichenkette einliest bzw. schreibt.

LLVM-Assemblersprache Die LLVM-Assemblersprache bietet eine menschenlesbare Repräsentation der LLVM-IR in Textformat an. Hier wurde die Assemblersprache um ein „targetisa“-Funktionsattribut erweitert, das die mixed-ISA Einstellungen als Zeichenkette enthält. Zu diesem Zweck musste der Assembler-Parser als auch der Assembler-Writer im LLVM-Core erweitert werden. Quelltext 6.1 zeigt ein Beispiel der erweiterten LLVM-Assemblersprache.

6.4.2 LLVM-Clang-Frontend

Das Clang-Frontend musste nur vergleichsweise gering für die Realisierung des mixed-ISA Softwareframeworks angepasst werden. Das Frontend wurde um folgende Eigenschaften erweitert:

6.4.2.1 Mixed-ISA Erweiterung der C/C++-Programmiersprache

Zur Unterstützung der mixed-ISA Erweiterung musste zunächst der C/C++-Parser im Clang-Frontend erweitert werden. Dieser muss den vollen Sprachumfang der in Abschnitt 6.2.1 definierten Erweiterungen unterstützen. Beim Parsen im Clang-Frontend wird zunächst aus der Eingangssprache ein *abstrakter Syntaxbaum* (AST, engl. *Abstract Syntax Tree*) aufgebaut. Dieser wurde um mixed-ISA Funktionsattribute erweitert und der Parser so angepasst, dass er bei mixed-ISA Funktionsattribute diese korrespondierend im AST setzt. Durch die Verwendung des allgemeinen `__attribute__`-Syntaxes konnten viele vorhanden Konstrukte und Funktionen im Frontend zu diesem Zweck wiederverwendet werden.

Der AST wird im folgenden Schritt im Frontend in die LLVM-Zwischendarstellung überführt, deren mixed-ISA Erweiterung in Abschnitt 6.4.1 beschrieben ist. Dabei werden die mixed-ISA Funktionsattribute des ASTs in mixed-ISA Funktionsattribute der Zwischendarstellung überführt. Die Zwischendarstellung kann dann vom LLVM-Middleend optimiert werden.

6.4.2.2 Unterstützung der Zielarchitektur

Zur Unterstützung der neuen Zielarchitektur, die in der ADL spezifiziert ist, muss diese Zielarchitektur auch dem Frontend bekannt sein. Dies liegt daran, dass die C/C++-Eingangssprache genauso wie die LLVM-Zwischendarstellung nicht zielarchitekturunabhängig ist (siehe Kapitel 2.4.3.2). Daher wurde eine neue Zielarchitektur im Frontend (und im gesamten LLVM-Compiler) integriert, die alle notwendigen Parameter der Zielarchitektur im Frontend bereitstellt. Die Parameter wurden vorher aus der Core-ADL mittels des CoreGen-Werkzeugs extrahiert und mittels Metaprogrammierung in eine Quellcodedatei, die im Frontend verwendet wird, übersetzt. Darin werden die initialen Definitionen für den Präprozessor festgelegt (z.B. `__KAHRISMA__` für die Kahrisma-Architektur), die Bytereihenfolge der Architektur, die Größe und Speicherausrichtung (engl. *Alignment*) sämtlicher Datentypen sowie das Gleitkommaformat.

Einige dieser Informationen über die Zielarchitektur sind auch für das Middleend relevant und werden daher innerhalb der LLVM-Zwischendarstellung gespeichert. Dazu zählen die Bytereihenfolge sowie die Größe und Ausrichtung der Datentypen.

6.4.3 LLVM-Middleend

Das Middleend führt plattformunabhängige Optimierungen auf der LLVM-Zwischendarstellung durch. Die Optimierungen sind alle optional und durch die Angabe der Optimierungsstufe als Kommandozeilenparameter wird eine festgelegte Untermenge an Optimierungen aktiviert. Zusätzlich kann man über die Kommandozeile sämtliche Optimierungen gezielt aktivieren oder deaktivieren.

Durch die Integration der mixed-ISA Erweiterung innerhalb der LLVM-Zwischendarstellung werden diese Erweiterungen auch automatisch durch das Middleend durchgereicht. Eine explizite Unterstützung der mixed-ISA Erweiterung innerhalb der Optimierungen war nicht notwendig, da nur wenige Optimierungen auf Funktionsebene durchgeführt werden. Einer dieser Optimierungsschritte ist das Function Inlining. Hierbei wird ein Funktionsaufruf innerhalb einer Funktion entfernt und durch den Quellcode der aufgerufenen Funktion ersetzt. Dabei bleiben die mixed-ISA Einstellungen der geinlineten Funktion unberücksichtigt und es ist immer der STAY-Modus aktiv. Dies stellt für den Programmierer allerdings keine große Einschränkung dar, weil er jederzeit durch das noinline-Funktionsattribut Inlining selektiv deaktivieren kann.

6.4.4 LLVM-Backend

Das Backend führt die Codegenerierung durch. Dabei wird die plattformunabhängige LLVM-Zwischendarstellung in Assemblerdateien übersetzt. Für die Realisierung eines mixed-ISA-fähigen Compilers wurde ein komplett neues Backend für den LLVM-Compiler umgesetzt. Da die Realisierung des Backends eine großen Teil dieser Arbeit ausmacht, wird es im nächsten Überabschnitt 6.5 beschrieben.

6.5 Mixed-ISA LLVM-Backend

Das mixed-ISA LLVM-Backend ist Teil des LLVM-Compilers und ist verantwortlich für die Übersetzung der LLVM-Zwischendarstellung in Assemblerdateien der Zielarchitektur. Das Backend ist die Kernkomponente des Softwareframeworks und hat die meisten Änderungen benötigt, um die mixed-ISA Codegenerierung für Prozessoren mit variablen Befehlssatzarchitekturen zu ermöglichen. Dazu wurde innerhalb dieser Arbeit LLVM um ein neues mixed-ISA Backend erweitert. Dieses Backend zeichnet sich im Vergleich zu gewöhnlichen LLVM-Backends durch neuartige Features aus, die in den folgenden Abschnitten vorgestellt werden. Dadurch wurde letztendlich ein Backend geschaffen, das die neuartigen mixed-ISA Programmiermodelle umsetzen kann.

Die Beschreibung des LLVM-Backends ist anhand der umgesetzten Features im Ba-

ckend organisiert. Zunächst wird in Abschnitt 6.5.1 die Benutzer-Retargierbarkeit durch eine ADL-basierte Backend-Generierung thematisiert. Danach wird die Unterstützung der rekonfigurierbaren Kahrisma-ISA durch Clustered-VLIW-Erweiterungen in Abschnitt 6.5.2 beschrieben. Darauf Aufbauend wird die Unterstützung des statischen (Abschnitt 6.5.3), dynamischen (Abschnitt 6.5.4) und automatischen (Abschnitt 6.5.5) mixed-ISA Programmiermodells behandelt.

6.5.1 ADL-basierte Backend-Generierung

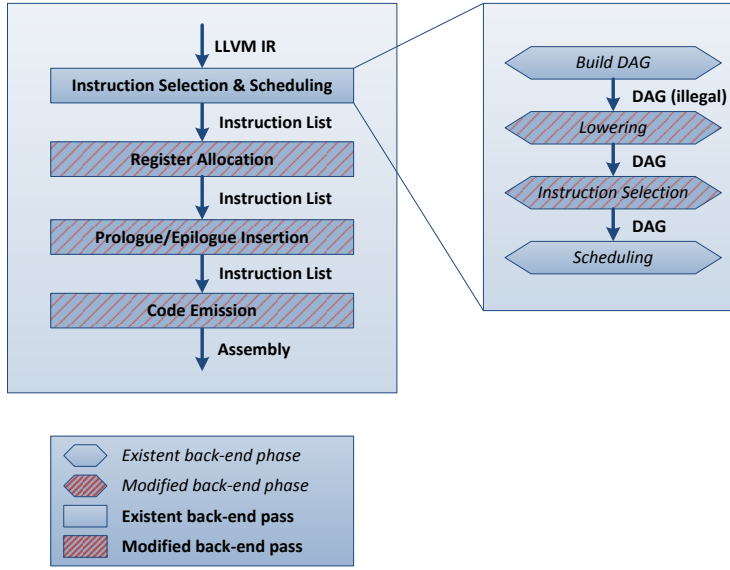


Abbildung 6.4: Realisierung der Benutzer-Retargierbarkeit in einem typischen LLVM-RISC-Backend

Für die Realisierung des mixed-ISA Softwareframeworks wurde ein neues Backend für den LLVM-Compiler entwickelt. Dazu muss das Backend durch eine ADL-Beschreibung benutzer-retargierbar sein, um letztendlich sämtliche Konfiguration der rekonfigurierbaren RSIW-Befehlssatzarchitektur unterstützen zu können. In diesem Abschnitt wird der Fokus auf die Realisierung der Benutzer-Retargierbarkeit des Ba-

ckends gelegt:

Die Grundlage für das mixed-ISA Backend hat ein typisches RISC-Backend gebildet, wie es z.B. für SPARC- oder MIPS-Prozessoren vorhanden ist. Zur Realisierung der Benutzer-Retargierbarkeit wurde das Mittel der Metaprogrammierung eingesetzt, so dass Teile des Quellcodes des Backends aus der ADL-Beschreibung mit Hilfe des CoreGen-Werkzeugs generiert werden. Der Aufbau eines einfachen RISC-Backends ist in Abbildung 6.4 ersichtlich. Sämtliche Durchläufe und Phasen, die durch die Quellcodegenerierung beeinflusst werden, sind rot markiert. Im Folgenden sind sämtliche Durchläufe und Phasen sowie ihre Änderungen zur Realisierung der Benutzer-Retargierbarkeit beschrieben.

6.5.1.1 Built DAG

In einem ersten Schritt wird die LLVM-Zwischendarstellung in einen *gerichteten azyklischen Graphen* (DAG, engl. *Directed Acyclic Graph*) umgewandelt. Dieser DAG enthält zunächst sämtliche Datentypen und Operationen, wie sie in der Zwischendarstellung vorkommen, unabhängig davon, ob sie von der Zielarchitektur unterstützt werden. Daher wird der DAG zu diesem Zeitpunkt noch als illegal bezeichnet.

6.5.1.2 Lowering

Im *Lowering* werden sämtliche illegale Datentypen und Operationen aus dem DAG eliminiert. Dazu kann der Entwickler verschiedene Ersetzungsstrategien für jeden nicht unterstützten Datentypen oder Operation festlegen. So werden z.B. nicht unterstützte kleinere Datentypen auf größere Datentypen abgebildet (wie z.B. ein 8 Bit Integer auf ein 32 Bit Integer) oder größeren Datentypen durch mehrere kleinere Repräsentiert (z.B. ein 64 Bit Integer wird durch zwei 32 Bit ersetzt). Dabei werden vorzeichenbehaftete/vorzeichenlose Datentyperweiterungen (engl. *signed/zero extensions*) je nach Bedarf in den DAG eingefügt, um die Semantik nicht zu verändern. Nicht unterstützte Operationen können durch eine Sequenz von alternativen Operationen, auf Operationen mit größeren Datentypen abgebildet oder durch benutzerdefinierten Code ersetzt werden.

Zur Realisierung der Benutzer-Retargierbarkeit innerhalb des Backends werden vom CoreGen-Werkzeug automatisch sämtlichen unterstützten Datentypen und Operationen aus der ADL extrahiert. Für nicht unterstützte Datentypen berechnet das CoreGen-Werkzeug die optimale Ersetzungsstrategie. Das CoreGen-Werkzeug erzeugt aus diesen Informationen automatisch Initialisierungscode für die Lowering-Klasse. Beim Erzeugen des Durchlaufs, der durch die Lowering-Klasse repräsentiert wird, werden dann im Initialisierungscode durch Funktionsaufrufe die ausgewählten Ersetzungsstrategien konfiguriert.

Zusätzlich werden im Lowering-Durchlauf noch die Aufrufkonventionen am Anfang

und Ende einer Funktion sowie vor und nach einem Funktionsaufruf im DAG umgesetzt. Die Aufrufkonventionen werden in den td-Dateien vom LLVM-Backend spezifiziert. Basierend auf dieser Spezifikation werden für jeden Parameter und Rückgabewert einer Funktion zunächst dessen Übergabeort berechnet. Dies kann entweder eine Speicherstelle im Stack oder ein Register sein. Abhängig vom Speicherort werden dann Operationen zum Zugriff auf die Speicherstelle oder das Register in den DAG vor und nach einem Funktionsaufruf eingefügt. Das CoreGen-Werkzeug erzeugt automatisch aus der ADL-Beschreibung die korrespondierenden td-Dateien zur Spezifikation der Aufrufkonventionen.

6.5.1.3 Instruction Selection (dt. *Befehlsauswahl*)

Nach dem *Lowering* enthält der DAG nur noch Datentypen und Operationen, die von der Zielarchitektur unterstützt und innerhalb der *Befehlsauswahl* behandelt werden können. Die *Befehlsauswahl* wandelt jetzt den DAG basierend aus zielarchitekturunabhängigen Operationen in einen DAG aus zielarchitekturabhängigen Operationen um. Für jede Operation der Zielarchitektur existiert ein Codemuster (engl. *Pattern*), das angibt, welche zielarchitekturunabhängigen Operationen die Operation abdeckt. Durch einen Abgleich der Codemuster (engl. *Pattern Matching*) mit dem DAG werden dann innerhalb der *Befehlsauswahl* sämtliche zielarchitekturunabhängige Operationen durch zielarchitekturabhängige Operationen ersetzt.

Die Codemuster können in LLVM innerhalb der td-Dateien spezifiziert werden. Ein Codemuster setzt sich aus einem DAG mit Knotentypen sowie Nebenbedingungen für die Knotentypen zusammen. So ist es z.B. möglich, dass ein Parameter einer Operation als Knotentyp ein Immediate hat und für das Immediate nur ein benutzerdefinierter Wertebereich erlaubt ist. Für ein Codemuster gibt es zwei Typen von Ersetzungsregeln:

1. Beim ersten Typ wird ein Codemuster durch eine einzelne Operation der Zielarchitektur ersetzt. Hierfür werden die Codemuster innerhalb der td-Dateien direkt zusammen mit dem Befehlssatz spezifiziert. Sie können von einfachen Befehlen wie einer Addition bis hin zu verketteten Operationen, wie z.B. einer *Multiply and Accumulate* (MAC) Operation, reichen.
2. Beim zweiten Typ wird ein Codemuster durch einen DAG ersetzt, der zum einen mehr als einen Knoten und zum anderen sowohl plattformunabhängige als auch plattformspezifische Operationen umfassen kann. Die plattformunabhängigen Knoten müssen dann durch wiederholtes Anwenden der Ersetzungsregeln am Ende der *Befehlsauswahl* komplett durch plattformspezifische ersetzt worden sein. Dieser Typ kommt zum Einsatz, wenn ein plattformunabhängiger Operation durch mehrere plattformspezifische Operationen realisiert werden soll. So gibt es in vielen Architekturen z.B. keine einzelne Operation um ein 32 Bit Immediate in ein Register zu laden. Stattdessen muss hierbei durch eine Typ-2-Regel zunächst das Immediate in mehrere Operationen expandiert

werden, die zuerst die unteren 16 und danach die oberen 16 Bit in das Register laden.

Zur Realisierung der Benutzer-Retargierbarkeit der *Befehlsauswahl* muss das CoreGen-Werkzeug sämtliche Codemuster und Ersetzungsregeln innerhalb der td-Dateien generieren. Dabei generiert das CoreGen-Werkzeug für jede Operation der Zielarchitektur einen Eintrag in den td-Dateien. Für eine Operation können mehrere Codemuster vorhanden sein, die durch Anwendung mathematischer Gesetze im CoreGen-Werkzeug erzeugt wurden. Da in der td-Beschreibung die Parameter einer Operation einen festen Typen, wie z.B. Register oder Immediate, ausweisen müssen, werden für gewöhnlich für jede Operation innerhalb der ADL eine große Anzahl an Operationen, jeweils mit Codemustern, generiert. Zusätzlich führt jede Spezialisierung einer Operation ebenfalls zum Erzeugen einer neuen Operation. Ein LLVM-internes Werkzeug erzeugt dann aus den Codemustern in den td-Dateien Quellcode für die *Befehlsauswahl*.

6.5.1.4 Register Allocation (dt. *Registerzuteilung*)

Bei der *Registerzuteilung* wird die unbeschränkte Anzahl an virtuellen Registern in SSA-Darstellung auf eine limitierte Anzahl an physikalischen Registern abgebildet. Wenn nicht alle virtuellen Register in die physikalischen passen, müssen zwischenzeitlich virtuelle Register auf Speicherstellen im Stack gespeichert werden, die dann nach Bedarf in physikalische Register geladen werden. Dafür werden während der *Registerzuteilung* Load/Store-Operationen zum Kopieren von physikalischen in und aus dem Stack eingefügt. Durch Nebenbedingungen bei der *Registerzuteilung* (z.B. können bestimmte Operanden schon vorher auf physikalische Register festgelegt sein) kann es ebenfalls möglich sein, dass Transferoperationen zwischen physikalischen Registern innerhalb der *Registerzuteilung* eingefügt werden.

Für die *Registerzuteilung* muss das CoreGen-Werkzeug folgende Quellcodeteile des Backends generieren:

1. Sämtliche verfügbaren physikalischen Register müssen innerhalb der td-Dateien spezifiziert werden. Dabei ist es wichtig, überlappende Register zu markieren, damit die *Registerzuteilung* immer nur eines der überlappenden Register zur gleichen Zeit verwendet.
2. Die Register müssen in Registerklassen eingeteilt sein. In einer Registerklasse dürfen nur Register des gleichen Datentyps enthalten sein und es muss die Menge an allokierbaren Registern für die *Registerzuteilung* festgelegt werden. Das sind für gewöhnlich sämtliche Register einer Registerklasse mit Ausnahme des Instruction-, Stack- und Frame-Pointers. Der Frame-Pointer wird innerhalb einer Funktion nur dann benötigt, wenn die Stackgröße dieser zur Compilezeit unbekannt ist. Ansonsten steht er ebenfalls für die *Registerzuteilung* zur Verfügung.

3. Bei der Spezifikation der Operation muss die Registerklasse für die Operanden angegeben sowie die implizit gelesen und geschrieben Register festgelegt werden. Durch die Registerklasse wird der *Registerzuteilung* bekannt gegeben, welche Register sie für einen Operanden verwenden kann. Die implizit gelesen und geschrieben Register beschränken die verwendbaren Register innerhalb der Registerzuteilung.
4. Zusätzlich muss CoreGen drei C++-Funktion zum Hinzufügen von Lade-, Speicher- und Transfer-Operationen für die *Registerzuteilung* generieren. Die Operationen müssen dabei mit Hilfe der nativen LLVM-API erzeugt werden. Die abstrahierte Spezifikation innerhalb der td-Dateien kann hier nicht verwendet werden, da die *Registerzuteilung* nach der *Befehlsauswahl* stattfindet.

6.5.1.5 Prologue/Epilogue Insertion

Der *Prologue/Epilogue Insertion* Pass generiert die Operationen am Anfang und Ende einer Funktion und finalisiert anschließend die Operationen, die auf den Stack zugreifen. Dies erfolgt in verschiedenen Schritten:

1. Nach der *Registerzuteilung* ist bekannt, welche Register eine Funktion verwendet. Je nach Aufrufkonvention darf eine Funktion bestimmte Register nicht verändern. Für diese Register werden Lade- bzw. Speicher-Operationen am Ende bzw. Anfang einer Funktion hinzugefügt und somit der Inhalt der physikalischen Register erhalten. Zu diesem Zweck verwendet der Durchlauf wieder die korrespondierenden C++-Funktionen der *Registerzuteilung*.
2. Danach ist das Stack-Layout vollständig und die endgültige Stackgröße der Funktion bekannt. Der Pro-/Epiloge Code wird durch Operationen erweitert, die den Stack-Pointer um die Größte des Stacks erhöhen bzw. erniedrigen. Hierfür muss CoreGen weitere C++-Funktionen erzeugen, die Operationen zum Erhöhen bzw. Erniedrigen des Stacks in die Funktion hinzufügen.
3. Nachdem das Stack-Layout vollständig und bekannt ist wie der Stack-Pointer modifiziert wurde, können die Load/Store-Operationen zum Zugriff auf den Stack finalisiert werden. Dabei wird ein Immediate der Operation durch die Stackposition ersetzt.

6.5.1.6 Code Emission

Bei der *Code Emission* wird die Operationsliste in Assembler-Quellcode übersetzt. Dafür enthält jede Operation und jedes Register innerhalb der td-Beschreibung eine Assemblerzeichenketten. Innerhalb der Zeichenkette sind die Operanden als Platzhalter angegeben, die durch den Registername oder Immediate-Wert bei der Generierung der Ausgabe ersetzt werden. Das CoreGen-Werkzeug erzeugt bei der Generierung

der td-Dateien pro Operation die Assemblerzeichenketten aus der ADL-Beschreibung. Ein Sonderfall stellen hierbei Immediate-Feldern bzw. Operanden dar, bei denen für jede Kodierung eine spezielle Assemblersyntax angegeben ist (siehe ADL-Kapitel Abschnitt 5.3.4). Hier müssen spezielle C++-Funktionen, eingebettet in die td-Dateien, generiert werden.

6.5.2 Unterstützung von Clustered-VLIW-Konfigurationen

Nach der allgemeinen Umsetzung eines benutzer-retargierbaren Backends, muss dieses jetzt noch um die Möglichkeit der Codegenerierung für die verschiedenen Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur erweitert werden. Dazu wurde das Backend um die Möglichkeit der Codegenerierung von VLIW- und Clustered-VLIW-Prozessoren erweitert.

Abbildung 6.5 gibt einen generellen Überblick über sämtliche Passes des Backends, das Kahrisma-spezifische RSIW-Prozessoren unterstützten kann. In diesem Abschnitt wird jetzt genauer auf die Durchläufe eingegangen, die für die Codegenerierung von Clustered-VLIW-Prozessoren zusätzlich zu einem gewöhnlichen RISC-Compiler-Backend in LLVM benötigt werden. Dazu muss das Backend zwei zusätzliche Aufgaben übernehmen:

1. Da VLIW-artige RSIW-Instruktionen aus mehreren Operationen bestehen, müssen die sequentiellen Operationen innerhalb der Codegenerierung in RSIW-Instruktionen gepackt werden. Dies wird typischerweise im Scheduling Pass erledigt. Daher wurde das Backend um einen *Parallelizing Scheduling* Pass erweitert.
2. Da manche Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur mehrere Registersätze (Cluster) hat, müssen Operationen und Register zu den Clustern zugewiesen werden. Durch die Zuweisung von Operationen zu Clustern wird entschieden, auf welchem Clustern eine Operation ausgeführt wird. Durch die Zuweisung von Registern zu Clustern wird entschieden, auf welchem Clustern ein Registerwert gespeichert wird und wie dieser zwischen den Clustern kopiert wird. Diese Aufgaben werden vom neuen *Clustering* Pass übernommen.

6.5.2.1 Repräsentation von RSIW-Instruktionen

RSIW-Instruktionen bestehen aus mehreren parallelen Operationen wobei jede Operation durch ihren Slot (ihrer Position innerhalb der Instruktion) genau zu einem Cluster zugeordnet ist. LLVM 2.9 unterstützt von sich aus keine direkte Repräsentation von Instruktionen bestehend aus mehreren paralleler Operationen. Daher musste zuerst ein Weg für die Repräsentation von paralleler Operationen sowie deren Clustern innerhalb LLVM gefunden werden. Innerhalb des Assemblers werden parallele Operationen durch den || Token gekennzeichnet und das Cluster ist implizit durch die

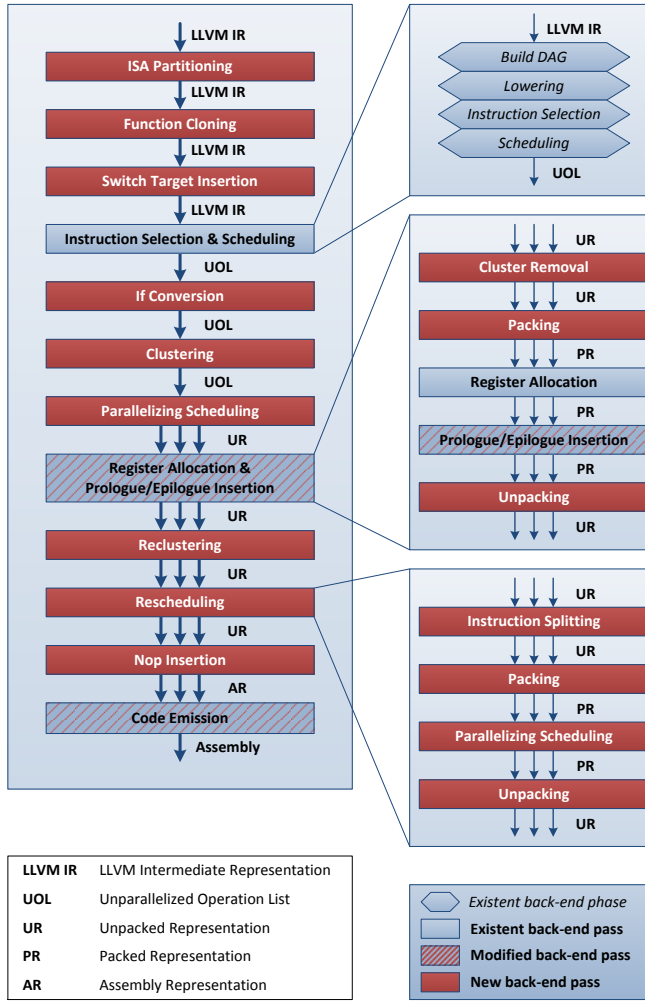


Abbildung 6.5: Überblick über die Passes des LLVM-Backends (Veröffentlicht [133])

Position (Slot) innerhalb der RSIW-Instruktion gegeben. Unbenutzte Slots müssen mit NOP-Operationen aufgefüllt werden.

(a) Assembly-Repräsentation

(b) Unpacked-Repräsentation

(c) Packed-Repräsentation

Abbildung 6.6: Die drei verschiedenen Repräsentationen von parallelen Instruktionen

Im LLVM-Backend (wie in Abbildung 6.5 dargestellt) werden drei verschiedenen Repräsentationen für Clustered-VLIW-Instruktionen verwendet.

Assembly-Repräsentation In der *Assembly-Repräsentation* (AR, engl. *Assembly Representation*) werden parallele Operationen mit Hilfe von speziellen NEXT-Operationen eingebettet in die Operationsliste gekennzeichnet. Eine NEXT-Operation markiert das Ende einer RSIW-Instruktion. Die Repräsentation wurde vom IA64-Backend übernommen, das wegen Inaktivität in LLVM 2.6 entfernt wurde. Unbenutzte Slots werden mit NOP-Operationen aufgefüllt, da das Cluster eine Operation implizit durch dessen Position in der RSIW-Instruktion gegeben ist. Dadurch ist die Repräsentation ideal für die Assemblerausgabe geeignet, die das gleiche Prinzip verwendet.

Abbildung 6.6(a) zeigt ein Beispiel der Assembly-Repräsentation für eine Clustered-VLIW-Befehlssatzarchitektur mit zwei Clustern mit je zwei Operationen. Pro Instruktion sind immer genau vier Operationen vorhanden, die durch die spezielle NEXT-Operationen getrennt sind. Durch die Position ist die Add-Operation dem ersten Cluster zugeordnet und die Sub- bzw. Load-Operation dem zweiten.

Unpacked-Repräsentation In der *Unpacked-Repräsentation* (UR, engl. *Unpacked Representation*) werden parallele Operationen ebenfalls mit Hilfe von NEXT-Operationen gekennzeichnet, allerdings wird das Cluster nicht mehr implizit durch die Position kodiert. Stattdessen wird die Clusternummer direkt mit jeder Operation gespeichert. Um die Modifikationen an der LLVM-Core-Bibliothek möglichst gering zu halten, wird die Clusternummer innerhalb der Opcodenummer kodiert. Der Vorteil der

Unpacked-Repräsentation gegenüber der Assembly-Repräsentation liegt in der einfacheren Handhabbarkeit, weil bei einer Modifikation nicht auf eine korrekte Anzahl an NOPs geachtet werden muss. Dies Repräsentation wird auch am häufigsten im Backend eingesetzt.

Abbildung 6.6(b) zeigt ein Beispiel der Unpacked-Repräsentation. Pro Instruktion sind nur die verwendeten Operationen vorhanden, wobei die Clusternummer im Opcode kodiert ist. Instruktionen sind wiederum durch NEXT-Operationen getrennt.

Packed-Repräsentation Innerhalb der *Packed-Repräsentation* (PR, engl. *Packed Representation*) werden parallele Operationen innerhalb einer einzelnen PACK-Operation kodiert. Dabei werden die Operationen komplett in die Operandenliste der PACK-Operation gepackt. Der Vorteil der Packed-Repräsentation liegt darin, dass LLVM-Durchläufe keine Kenntnis der Parallelität der Operationen haben müssen. Jeder Instruktion ist in einer einzelnen PACK-Operation versteckt, wodurch eine sequentieller Interpretation der Operationen korrekt ist. Die Repräsentation erschwert zwar die Modifikation einzelnen Operationen innerhalb einer Instruktion, dafür erlaubt sie allerdings die Wiederverwendung von LLVM-Durchläufe wie der Registerzuteilung.

Abbildung 6.6(c) zeigt ein Beispiel der Packed-Repräsentation. Jede Operation repräsentiert eine komplette Instruktion. Parallele Operationen werden in eine spezielle PACK-Operation kodiert. Instruktionen mit einer einzelnen Operationen können so stehen bleiben.

6.5.2.2 If Conversion

Zur Ausnutzung von Partial Predication wird vor dem *Clustering* eine *If Conversion* durchgeführt. Der Vorteil von Partial Predication besteht darin, dass Sprünge reduziert werden können. Dies ist insbesondere bei VLIW-Architekturen wichtig, weil diese mehr Parallelität auf Instruktionsebene ausnutzen können. Die *If Conversion* basiert auf den If-Conversion-Klassen des LLVM-Compilers. Die If-Conversion-Klassen des LLVM-Compilers wurden bisher allerdings nur nach der *Registerzuteilung* zur Unterstützung von Full-Predication-Architekturen wie z.B. der ARM-Befehlssatzarchitektur verwendet. Daher wurde für die Kahrisma-Architektur die If-Conversion-Klasse zur Unterstützung von Partial Predication erweitert. Diese Erweiterung umfasst die Unterstützung von virtuellen Registern und der korrekte Umgang mit PHI-Operationen (siehe Abschnitt 2.4.3.2) während der *If Conversion*.

Bei der *If Conversion* werden If-Konstrukte bestehend aus mehreren Basisblocken zu einen Basisblock zusammengefasst. Dadurch werden Operationen, die vorher wegen eines Sprungs nur bedingt ausgeführt wurden, immer ausgeführt und man muss mittels Partial Predication sicherstellen, dass das Ergebnis der Operationen nur abhängig der Sprungentscheidung verwendet wird oder nicht:

Register Bei Registerwerten kann man in der Kahrisma-Architektur mittels der Se-

lect-Operation eines von zwei Registern auswählen, die abhängig von einem Entscheidungsregister (Sprungregister) ausgewählt werden. Bei der *If Conversion* werden PHI-Operationen, deren Operand von einem eliminierten Basisblock abhängig war, durch eine Select-Operation ersetzt.

Speicherzugriffe und andere Seiteneffekte Wenn eine Operation Speicherzugriffe durchführt oder andere Seiteneffekte hat, dann muss die Operation bedingt ausgeführt werden, d.h. sie bekommt ein zusätzliches Register als Operanden, der angibt, ob die Operation ausgeführt wird oder nicht.

Bei der Realisierung im LLVM-Compiler wird der gesamte CFG nach Kandidaten für die *If Conversion* durchsucht. Eine *If Conversion* kann durchgeführt werden, wenn

1. Der Kontrollfuß sämtlicher Basisblöcke eines If-Konstrukts analysiert werden kann. Da der Pass sich nach der *Befehlsauswahl* befindet, besteht eine If-Anweisung aus zielarchitekturabhängigen Sprungbefehlen. Die Semantik dieser Sprungbefehle muss extrahiert und die Sprungbefehle müssen später umgeschrieben werden können.
2. Das If-Konstrukt unterstützt wird. Es werden Einfache-, Dreieck- und Raute-If-Konstrukte unterstützt.
3. Die betroffenen Basisblöcke müssen bedingt ausgeführt werden können. Dies ist gegeben, wenn
   a) für sämtliche geschriebenen virtuellen Registern, die von einer PHI-Operation verwendet werden, eine passenden Select-Operation in der ISA vorhanden ist;
   b) bei Operationen mit Seiteneffekten (Speicherzugriff, Modifikation des IPs, etc.) der Seiteneffekt bedingt ausgeführt werden kann. So hat die Kahrisma-Architektur zur Optimierung der *If Conversion* z.B. bedingte Load/Store-Operationen, die abhängig von einer Bedingung in einem Eventregister den Speicherzugriff durchführen können. Hierbei muss allerdings nur Seiteneffekt bedingt ausgeführt werden. Register, die von der Operation geschrieben werden, können immer geschrieben werden. Dies erleichtert die Mikroarchitektur, da bedingtes Schreiben im Zusammenhang mit der Registerumbenennung nur mit vergleichsweise hohem Hardwareaufwand realisierbar ist.

Zur Realisierung der *If Conversion* muss das CoreGen-Werkzeug einige Funktionen für die Analyse und zum Umschreiben von architekturspezifischen Operationen aus der Core-ADL generieren. Dies umfasst:

1. Analyse der Sprungbefehle eines Basisblocks
2. Entfernen eines Sprungbefehls in einem Basisblock
3. Erzeugen eines Sprungbefehls in einem Basisblock

4. Negieren eines Sprungbefehls
5. Überprüfung ob eine Select-Operation für einen gegebenen Datentyp unterstützt wird
6. Erzeugen von Select-Operationen
7. Überprüfung ob der Seiteneffekt einer unbedingten Operation bedingt werden kann
8. Eine unbedingte Operation mit Seiteneffekt durch bedingte ersetzen

6.5.2.3 Clustering

In der Literatur sind drei unterschiedliche Lösungen für die Integration des *Clusterings* in das Backend bekannt. *Clustering* kann vor dem Scheduling, innerhalb des Schedulings oder nach dem Scheduling durchgeführt werden. In dieser Arbeit wurde das *Clustering* vor dem Scheduling platziert. Die Umsetzung führte zu einem Problem im Backend-Design von LLVM. Die Befehlszuweisung und das Scheduling sind jeweils innerhalb eines Durchlaufes in LLVM implementiert. Innerhalb dieses Durchlaufes werden die unterschiedlichen Phasen für jeden Basisblock durchgeführt. D.h. die *Befehlsauswahl* und das Scheduling werden gemeinsam für jeden Basisblock ausgeführt. Das *Clustering* arbeitet hingegeben auf kompletten Funktionen anstelle von Basisblöcken. Daher muss bereits die *Befehlsauswahl* sämtlicher Basisblöcke abgeschlossen sein bevor das *Clustering* durchgeführt werden kann. Im LLVM-Backend-Design ist es nicht vorgesehen, dass zuerst alle Basisblöcke die *Befehlsauswahl* und erst danach das Scheduling durchlaufen.

Um den Clustering-Durchlauf vor dem Scheduling ohne große Modifikationen von LLVM realisieren zu können, wird stattdessen das interne Scheduling ignoriert. Nach dem Scheduling wird das *Clustering* durchgeführt ohne das Schedule zu berücksichtigen. Nach dem *Clustering* findet ein eigenes Scheduling statt. Innerhalb dieser Lösung arbeiten beide Durchläufe, das *Clustering* und *Scheduling*, auf der Operationsliste anstelle eines DAGs. Innerhalb der Durchläufe muss stattdessen jeweils einen DAG aus der Operationsliste extrahieren wird.

Der *Clustering* Pass arbeitet auf einem DAG, der durch eine Abhängigkeitsanalyse auf der Operationsliste gebildet wird. Innerhalb dieses Passes werden alle Operationen (Knoten) und Register (Kanten) Clustern zugewiesen. Der Pass ist in zwei Phasen realisiert:

Operation Cluster Assignment Zuerst weißt das Operation Assignment jeder Operation ein Cluster zu. Bei der Initialisierung werden zunächst alle Operationen einem Cluster zugewiesen, die auf einem bestimmten Cluster ausgeführt werden müssen. Dies ist z.B. häufig bei Transferoperationen der Fall, die ein physikalisches Register als Operanden haben. Diese Operationen müssen auf das physikalische Register zugreifen können. Wenn das physikalische Register nur

vom lokalen Cluster zugegriffen werden kann, muss auch die Operation zwingend in diesem Cluster ausgeführt werden. Dies ist abhängig vom Inter-Cluster-Kommunikationsmodell und der Zugriffsart auf das Register. Nach der Initialisierung wird der BUG-Algorithmus [105] zur Zuweisung der Operationen auf Cluster verwendet.

Register Cluster Assignment In der zweiten Phase wird jedes virtuelle Register zu einem Cluster zugewiesen und bei Bedarf Inter-Cluster-Kommunikation eingefügt. Die Phase ist dabei so flexibel gehalten, dass sie unterschiedliche Inter-Cluster-Kommunikationsmodelle, je nach Beschreibung innerhalb der ADL, unterstützt. Durch die SSA-Darstellung in LLVM wird jedes virtuelle Register genau einmal geschrieben bzw. definiert (Def) aber kann mehrere Male gelesen bzw. verwendet (Use) werden. Zur Wahrung der Flexibilität und zur Reduktion der Komplexität wurde das Register Cluster Assignment in zwei Unterphasen realisiert. In der ersten Unterphase wird zunächst Entschieden, in welches Cluster ein Register bei dessen Definition geschrieben werden soll. In der zweiten Phase wird dann entschieden, wie das Register in die Cluster, die es verwenden, kopiert wird.

Register Def Cluster Bei der ersten Unterphase wird zunächst entschieden, in welches Cluster ein Register bei dessen Definition gespeichert werden soll. Dies ist nur bei ICC-Modellen möglich, die eine Auswahl des Clusters bei den Zieloperanden erlauben. Ansonsten ist das Zielcluster bei der Definition schon fest auf das lokale Cluster der Operation festgelegt. Wenn eine Auswahl besteht, wird das lokale Cluster als Zielcluster gewählt, falls es im lokalen Cluster auch verwendet wird. Ansonsten wird das Cluster als Zielcluster ausgewählt, das das Register als nächstes verwendet. Am Ende der Unterphase ist für jedes virtuelle Register ein Cluster zugewiesen.

Register Use Cluster Bei der zweiten Unterphase wird sichergestellt, dass jedes virtuelle Register bei dessen Verwendung auch gelesen werden kann. Bei ICC-Modellen, die entfernt lesen können (wie z.B. das Extended Operands), muss hier nichts gemacht werden. Bei ICC-Modellen, die nur ein lokales Lesen der Quelloperanden erlauben, muss zusätzlich Inter-Cluster-Kommunikation eingefügt werden, falls das Cluster des virtuellen Registers nicht dem Cluster der lesenden Operation entspricht. In diesem Fall wird für jedes unterschiedliche Cluster, in dem das virtuelle Register verwendet wird, eine COPY-Operation eingefügt. Die COPY-Operation ist dem Cluster des virtuellen Registers zugewiesen und kopiert dieses in das Zielcluster. Dazu wird im Zielcluster ein neues virtuelles Register angelegt, das dann von den lokalen Operationen zum Lesen des Wertes verwendet wird.

Mit der Zweiteilung in zwei Unterphase können im Backend effizient mehrere ICC-Modelle unterstützt werden. Innerhalb der Arbeit wurde dies mit der Unterstützung des Extended-Results- und Extended-Operands-ICC-Modells ge-

zeigt.

6.5.2.4 Parallelizing Scheduling (dt. *Parallelisierende Scheduling*)

Nach dem *Clustering* wird das *parallelisierende Scheduling* durchgeführt, das sowohl die Reihenfolge der Operationen als auch das Bündeln der Operationen in parallele RSIW-Instruktion bestimmt. Beim *parallelisierenden Scheduling* mussten insbesondere die Kahrisma-spezifischen Nebenbedingungen (z.B. die maximale Anzahl an Lesezugriffe pro Cluster und Instruktion) und Scheduling-Konditionen (z.B. Scheduling von abhängigen Load/Store-Operationen in einer Instruktion, siehe Abschnitt 3.5.2.8) eingehalten werden.

Für die Realisierung des *parallelisierenden Schedulings* wurden die LLVM-internen Klassen fürs Scheduling wiederverwendet. In LLVM existiert bereits ein allgemeines Konzept zum Durchführen des Schedulings sowohl auf der DAG- als auch auf der Operationslisten-Repräsentation. Scheduling auf dem DAG wird vor der *Registerzuteilung* verwendet und dabei wird der DAG in eine Operationsliste konvertiert. Das Scheduling auf der Operationsliste wird typischerweise zum *Rescheduling* nach der *Registerzuteilung* verwendet.

Wie in Abschnitt 6.5.2 beschrieben, wird das LLVM-interne Scheduling basierend auf dem DAG ignoriert. Stattdessen wird ein zusätzliches *parallelisierendes Scheduling* nach dem LLVM-internen Scheduling aber vor der *Registerzuteilung* auf der Operationsliste durchgeführt. Daher werden für das *parallelisierende Scheduling* die LLVM-internen Scheduling-Klassen, die eigentlich zum *Rescheduling* nach der *Registerzuteilung* konzipiert sind, vor der *Registerzuteilung* verwendet. Das Scheduling wurde um folgende Features erweitert, um ein *parallelisierenden Scheduling* zu realisieren:

1. Die Abhängigkeitsanalyse am Anfang des Schedulings wurde so erweitern, dass sie mit virtuellen Registern umgehen kann. Da die Klassen bisher nur nach der *Registerzuteilung* verwendet wurden und dort keine virtuellen Register auftreten, konnten sie nicht mit virtuellen Registern umgehen.
2. Die Abhängigkeitsanalyse wurde so erweitert, dass sie Kahrisma-spezifische Abhängigkeiten erkennen und modellieren kann. Diese Abhängigkeiten sind in Tabelle 6.1 aufgelistet.
3. Die Parallelisierung der Operation im Scheduling wurde mit Hilfe eines sog. Hazard Recognizers durchgeführt, der im Folgenden erklärt wird.

Ein Backend, das die generischen LLVM-Klassen zum Scheduling oder Rescheduling verwendet, kann einen architekturspezifischen **Hazard Recognizer** den Scheduling-Klassen übergeben. Der Hazard Recognizer ist eine Klasse, die typischerweise dazu verwendet wird, um architekturspezifische Pipelinekonflikte dem Scheduler anzuzeigen, z.B. wenn es nicht vorteilhaft oder nicht erlaubt ist eine Operation nach einer anderen einzuplanen. LLVM verwendet einen List-Scheduler, der eine Warteschlan-

ge von Operationen enthält, die als nächstes eingeplant werden können. Für sämtliche Operationen der Warteschlange wird nach ihrer Priorität der Hazard Recognizer aufgerufen. Dieser gibt für jede Operation entweder NoHazard, Hazard oder NoopHazard zurück. Die erste Operation mit NoHazard wird vom Scheduler als nächstes ausgegeben. Im List-Scheduler wird die Zeit in Zyklen modelliert. Wenn keine NoHazard-Operation in der Warteschlange gefunden wird oder die Warteschlange leer ist, wird ein neuer Zyklus gestartet. Dabei wird eine NOP-Operation ausgegeben, wenn vorher ein NoopHazard aufgetreten ist. Mit diesem Konzept können sowohl Scheduler für Prozessoren ohne Konflikterkennung (Ausgabe von NOPs), mit Konflikterkennung und Scheduler für superskalare Prozessoren realisiert werden. Bei letzterem ist das Konzept der Zyklen wichtig, da hier mehrere Operationen pro Zyklus eingeplant werden.

Für das parallelisierende Scheduling konnten die List-Scheduling-Klassen von LLVM mit einer latenzbasierten Warteschlange zusammen mit einem Kahrisma-spezifischen Hazard Recognizer wiederverwendet werden. Dabei wurde der Hazard Recognizer so entwickelt, das er folgende Aufgaben übernimmt:

1. Das parallelisierende Scheduling muss, im Gegensatz zum normalen Scheduling, Instruktionen bestehend aus mehreren Operationen und nicht nur einzelne Operationen ausgeben. Der LLVM-Compiler kennt allerdings nur Operationen in seiner Operationsliste und hat keine Repräsentation für Instruktionen. Somit erzeugt auch der LLVM-Scheduler als Ergebnis eine Operationsliste. Zur Erweiterung des Schedulers auf Instruktionen, besteht nun die Grundidee, weiterhin eine Operationsliste auszugeben, diese aber um NEXT-Operationen zu erweitern, die ein Ende einer Instruktion kennzeichnen (vgl. Unpacked-Repräsentation in Abschnitt 6.5.2.1). Zur Ausgabe der NEXT-Operationen wurden NoopHazards im Hazard Recognizer verwendet. Wenn eine Operation nicht in der aktuellen Instruktion bzw. im aktuellen Zyklus eingeplant werden kann, wird ein NoopHazard zurückgeliefert. Nach jedem Zyklus im Scheduler wird dann eine Funktion aufgerufen, die für das Hinzufügen einer NOP-Operation zuständig ist. Anstelle einer NOP-Operation wird im Kahrisma-Backend eine NEXT-Operation ausgegeben und somit das Ende einer RSIW-Instruktion gekennzeichnet.
2. Im Scheduler müssen die Kahrisma-spezifischen Nebenbedingungen für die verwendeten Ressourcen innerhalb einer Instruktion im Hazard Recognizer überprüft werden. Im Hazard Recognizer wird daher eine Ressourcenliste gepflegt, die die Anzahl an freien Ressourcen der aktuellen Instruktion enthält. Eine Operation darf nur in die aktuelle Instruktion eingeplant werden, wenn genügend freie Ressourcen vorhanden sind. Ansonsten wird für die Operation ein NoopHazard ausgegeben. Folgende Ressourcen können überprüft und flexibel in der ADL angegeben werden:
    a) Maximale Anzahl an Operationen pro Cluster
    b) Maximale Anzahl an Lese- und Schreibzugriffe auf Register pro Registersatz und Cluster

c) Maximale Anzahl an Inter-Cluster-Kommunikation pro Registersatz und Cluster

3. Es müssen die Scheduling-Konditionen durch die Abhängigkeiten zwischen den Operationen eingehalten werden, damit die Semantik des parallelen Schedules dem des sequentiellen entspricht. Im folgenden Abschnitt wird genauer auf die Scheduling-Konditionen eingegangen.

Abhängigkeit	**Scheduling-Kondition**
Register-Datenabhängigkeit Register-Ausgabeabhängigkeit CALL- zu Speicher-Operation ABI-Register-Abhängigkeit von CALL	$I_{o1} < I_{o2}$
Register-Gegenabhängigkeit ABI-Register-Abhängigkeit zu CALL Speicher- zu CALL-Operation	$I_{o1} \leq I_{o2}$
Speicher-Abhängigkeit	$I_{o1} < I_{o2} \vee (I_{o1} = I_{o2} \wedge S_{o1} < S_{o2})$

Tabelle 6.1: Scheduling von Abhängigkeiten der Kahrisma-Architektur

Zur Überprüfung der Abhängigkeiten wurden drei verschiedene Scheduling-Konditionen zur Beschreibung der Kahrisma-spezifischen Abhängigkeiten eingeführt. Tabelle 6.1 zeigt die Modellierung von drei Scheduling-Konditionen beruhend auf verschiedenen Abhängigkeiten zwischen zwei Operationen. Dabei hängt Operation $o2$ von $o1$ ab, I repräsentiert die Instruktionsnummer der Operation und S die Slotnummer der Operation. Im Folgenden werden die unterschiedlichen Abhängigkeiten und die daraus resultierenden Scheduling-Konditionen erklärt:

Register-Daten- und -Ausgabeabhängigkeit Für echte Datenabhängigkeiten und Ausgabeabhängigkeiten zwischen Registern muss die Operation $o2$ in einer RSIW-Instruktion eingeplant werden, die der RSIW-Instruktion von Operation $o1$ folgt. So muss die Instruktionsnummer I_{o2} größer als I_{o1} sein. Diese Bedingung wird innerhalb der LLVM-Scheduling-Klassen eingehalten, indem die Latenz der Abhängigkeit im Abhängigkeitsgraphen auf 1 gesetzt wird.

Register-Gegenabhängigkeit Operationen, die Gegenabhängigkeiten zwischen Registern haben, können im Vergleich zu echten Datenabhängigkeiten zusätzlich in der gleichen RSIW-Instruktion eingeplant werden. Daher muss die Instruktionsnummer I_{o2} größer oder gleich I_{o1} sein. Diese Bedingung wird innerhalb der LLVM-Scheduling-Klassen eingehalten, indem die Latenz der Abhängigkeit im Abhängigkeitsgraphen auf 0 gesetzt wird.

Speicher-Abhängigkeit Zugriffe auf den Hauptspeicher innerhalb einer VLIW-Instruktion werden anhand der Position innerhalb der Instruktion von links nach rechts abgearbeitet bzw. priorisiert (siehe Abschnitt 3.5.2.8). Daher können auch Operationen mit Abhängigkeiten durch Speicherzugriffe innerhalb einer RSIW-

Instruktion ($I_{o1} = I_{o2}$) eingeplant werden, wenn die Slotnummer S_{o1} der Operation $o1$ kleiner ist als S_{o2}. Um diese Bedingung einzuhalten, werden im Hazard Recognizer abhängigen Operationen innerhalb der aktuellen RSIW-Instruktion überprüft und ein Hazard zurückgeliefert, wenn die Clusternummer C_{o2} der Operation $o2$ kleiner ist als die Clusternummer C_{o1} von $o1$. Die Überprüfung auf die Clusternummer ist hierbei identisch zu der Überprüfung der Slotnummer, da innerhalb einer RSIW-Instruktion sämtliche Operationen anhand ihrer Clusternummer sortiert sind.

CALL-Operation Eine Besonderheit bei der Behandlung der Abhängigkeiten nimmt die CALL-Operation ein. Diese repräsentiert einen Unterfunktionsaufruf. Die aufgerufene Unterfunktion ist im Compiler eine Black-Box und kann beliebige Änderungen am Speicher durchführen sowie Register gemäß dem *Binärschnittstelle* (ABI, engl. *Application Binary Interface*) lesen und schreiben. Alle Änderungen der CALL-Instruktion wirken sich erst nach Ausführung der RSIW-Instruktion, die die CALL-Operation beinhaltet, aus. Daraus lassen sich besondere Abhängigkeiten für die CALL-Operation ableiten:

Da die aufgerufene Unterfunktion beliebige Speicherzugriffe durchführen kann, bestehen Speicherabhängigkeiten zu bzw. nach Speicher-Operationen davor bzw. danach. Speicheroperationen davor dürfen sogar in der gleichen Instruktion wie die CALL-Operation eingeplant werden, da die Unterfunktion erst nach der VLIW-Instruktion ausgeführt wird. Speicheroperationen danach müssen in der folgenden Instruktion oder später eingeplant werden.

Neben Speicherzugriffen liest und schreibt die aufgerufene Unterfunktion bestimmte Register, die im ABI festgelegt sind. Da die ABI-Register nicht während der CALL-Operation gelesen und geschrieben werden sondern erst nach der RSIW-Instruktion, gelten hier ebenfalls spezielle Regeln für die Abhängigkeiten. Sämtliche Operationen vor dem Unterfunktionsaufruf mit ABI-Register-Abhängigkeiten können sogar noch innerhalb der Instruktion mit der CALL-Operation ausgeführt werden. Dies gilt für alle Abhängigkeitstypen zwischen Registern, also insbesondere auch für die echten Datenabhängigkeiten. Sämtliche Operationen nach der CALL-Operation mit ABI-Register-Abhängigkeiten müssen nach der Instruktion der CALL-Operation eingeplant werden. Diese Regeln gelten nur für Abhängigkeiten zu ABI-Registerzugriffen. Die CALL-Operation kannst selbst (also nicht die Unterfunktion) parallel dazu Register lesen und schreiben. Für Abhängigkeiten von/zu diesen Registern gelten die gewöhnlichen Regeln für Register-Abhängigkeiten.

6.5.2.5 Register Allocation and Prologue/Epilogue Insertion

Nach dem Scheduling wird die *Registerzuteilung* durchgeführt. Die native *Registerzuteilung* des LLVM-Compilers arbeitet auf einzelnen Operationen. Die *Registerzuteilung* für RSIW muss allerdings auf kompletten RSIW-Instruktionen arbeiten. Sämtli-

che Operationen innerhalb einer RSIW-Instruktion werden parallel ausgeführt. Deswegen ergibt sich für die Registerzugriffe die Regel, dass sämtliche Quellregister aller Operationen (unabhängig von der Reihenfolge innerhalb der Instruktion) gelesen bevor sämtliche Ergebnisse in Zielregister geschrieben werden. Um dies mit der *Registerzuteilung* des LLVM-Compilers realisieren zu können, wird die Packed-Repräsentation verwendet. Dabei wird eine komplette Instruktion in eine Operation gepackt und die *Registerzuteilung* des LLVM-Compilers arbeitet somit implizit auf ganzen Instruktionen. Der Packing-Pass innerhalb der Abbildung 6.5 transformiert vor der *Registerzuteilung* die Unpacked-Repräsentation in die Packed-Repräsentation.

Ein Nachteil der Packed-Repräsentation ist, dass die LLVM-Klassen nicht mehr die einzelnen Operationen innerhalb der Instruktion erkennen können und somit ihre Semantik für diese unbekannt ist. Allerdings muss die *Registerzuteilung* PHI- und COPY-Operationen innerhalb der Operationsliste identifizieren können. Deswegen gibt es eine Sonderbehandlung für diese Operationen:

1. Beide Operationstypen dürfen nicht in eine PACK-Operation gepackt werden. Dies wird zum einen durch ein Sonderregelung im Scheduler und in der Packed-Repräsentation erreicht. Im Scheduler wird zunächst mit dem Hazard Recognizer sichergestellt, dass jede PHI- und COPY-Operation nur einzeln in eine komplette Instruktion eingeplant werden können. Die Packed-Repräsentation wurde so erweitert, dass diese nur Instruktionen mit mehr als eine Operation in eine PACK-Operation packt, andernfalls wird die ursprüngliche Operation beibehalten. Dabei wird in der Packed-Repräsentation weiterhin jede Instruktion mit einer Operation repräsentiert, bloß Instruktion mit einer Operation können in den LLVM-Klassen erkannt und behandelt werden.
2. Die Clusterinformation muss von den PHI- und COPY-Operationen entfernt werden, weil die Clusternummer im Opcode kodiert ist. Die Clusternummer ist sowieso für die *Registerzuteilung* der COPY- und PHI-Operation irrelevant und für den neu eingefügten Spill Code wird im *Reclustering* eine geeignetes Cluster gesucht.

Die Packed-Repräsentation, die für die Registerzuteilung verwendet wurde, wird bis nach der *Prologue/Epilogue Insertion* beibehalten. Während der *Registerzuteilung* werden Lade-, Speicher- und Transferoperationen eingefügt, um Register vom/zum Stack zu laden/schreiben oder Werte zwischen Registern zu verschieben. Diese zusätzlichen Operationen werden als neue Operation in die Operationsliste ohne Clusterzuteilung eingefügt. Durch die Packed-Repräsentation haben diese zunächst eine Instruktion für sich alleine. Genauso wird in der *Prologue/Epilogue Insertion* am Anfang und Ende einer Funktion neue Operationen eingefügt, die ebenfalls zunächst unparallelisiert und ohne Clusterzuteilung sind. Anschließend wird dann die Unpacked-Repräsentation wieder hergestellt.

6.5.2.6 Reclustering

Während der *Registerzuteilung* und *Prologue/Epilogue Insertion* werden zusätzliche Operationen in die Operationsliste eingefügt. Während dem Einfügen wurde noch nicht entschiedene, in welche Cluster und mit welchen anderen Operationen innerhalb einer RSIW-Instruktion diese parallel ausgeführt werden können. Stattdessen wurden diese Operationen ohne Clusterzuteilung und jeweils einzeln in RSIW-Instruktionen eingefügt. Wegen der fehlenden Clusterzuteilung ist die Instruktionsliste nach der *Registerzuteilung* zunächst illegal. Zu diesem Zweck wird im *Reclustering* für die neuen Operationen ein geeignetes Cluster gesucht.

Das *Reclustering* führt genauso wie das *Clustering* eine Clusterzuteilung durch. Im Gegensatz zum *Clustering* müssen jetzt allerdings nur Operationen und keine Register Clustern zugewiesen werden. Die Register sind durch die *Registerzuteilung* schon final an physikalische Register gebunden und die eingefügten Operationen verwenden i.d.R. keine neuen Register. Ebenso ist das Operation Cluster Assignment wesentlich einfacher realisiert als beim *Clustering*. Während beim *Clustering* der BUG-Algorithmus zur Optimierung angewandt wurde, ist beim *Reclustering* der Freiheitsgrad einer Operation meistens durch die physikalischen Register in der Operandenliste so eingeschränkt, dass diese nur auf einem Cluster platziert werden darf. Falls eine Operation auf mehreren Clustern platziert werden kann, wird das Cluster verwendet, durch das die benötigt ICC minimiert.

6.5.2.7 Rescheduling

Nach der *Registerzuteilung* und dem *Prologue/Epilogue Insertion* ist das Schedule sehr ineffizient. Dies liegt an den neuen Operationen, die nicht parallelisiert eingefügt wurden. Um ein effizientere Schedule zu erhalten, das den verfügbaren ILP effizient ausnutzt, wurde ein erneutes Scheduling in das Backend integriert. Dazu wurde der Scheduling-Pass vor der *Registerzuteilung* als Basis genommen und erweitert. Der Hauptunterschied besteht darin, dass Instruktionen mit bereits parallelen Operationen in der Operationsliste vorhanden sind. Das muss insbesondere bei der Abhängigkeitsanalyse am Anfang des Schedulings beachtet werden, das nur auf einzelnen Operationen arbeitet und keine parallelen Instruktionen kennt. Durch die Verwendung der Packed-Repräsentation war es möglich, die Abhängigkeitsanalyse im Scheduling ohne Modifikationen beizubehalten. Dadurch werden bereits parallelisierte Operationen innerhalb einer RSIW-Instruktion als PACK-Operation repräsentiert und sind dann im Scheduler als untrennbare, atomare Scheduling-Einheiten vorhanden. Der Hazard Recognizer wurde so erweitert, dass er PACK-Operationen nicht weiter mit anderen Operationen parallelisiert. Zur Realisierung des Reschedulings wurde daher ein Packing- bzw. Unpacking-Pass vor bzw. nach dem Scheduling-Pass eingefügt.

Danach ist zwar das *Rescheduling* funktional korrekt aber noch nicht besonders effizient. Es kann nur nicht parallelisierte Operationen in eine neue RSIW-Instruktion

einplanen aber bereits parallelisierte RSIW-Instruktionen bleiben unverändert. Zur Steigerung der Effizienz wurde daher ein Instruction Splitting Pass vor dem Scheduling integriert. Dieser Pass versucht parallelisierte Instruktionen in nicht parallelisierte aufzuteilen, um den Freiheitsgrad fürs Scheduling zu erhöhen. Beim Aufteilen muss beachtet werden, dass sich die Semantik nicht verändert. Die Semantik kann sich ändern, wenn eine Gegenabhängigkeit innerhalb der Instruktion zwischen zwei Operationen besteht. In diesem Fall muss die Reihenfolge, in der eine Operation aus der Instruktion herausgezogen wird, beachtet werden. Wenn sogar ein Zyklus von Gegenabhängigkeiten innerhalb der Instruktion besteht, gibt es keine korrekte Reihenfolge und es dürfen alle beteiligten Operationen nicht auseinandergezogen werden. Nach dem Instruction Splitting sind die meisten Operationen (Zyklen von Gegenabhängigkeiten sind selten) nicht parallelisiert und das folgende Scheduling hat einen hohen Freiheitsgrad zum Finden eines effizienten Schedules.

6.5.2.8 NOP Insertion und Code Emission

Vor der *Code Emission* muss die Unpacked-Repräsentation noch in die Assembler-Repräsentation umgewandelt werden. Dies wird im NOP Insertion Pass durchgeführt. In der Assembler-Repräsentation wird die Clusternummer nicht mehr im Opcode kodiert sondern ist implizit durch die Position der Operation innerhalb der Instruktion gegeben. Damit jede Operation an der korrekten Stelle ist, werden bei Bedarf NOP-Operationen in die Operationsliste eingefügt. Danach wird die Clusternummer aus dem Opcode entfernt.

Das NOP Insertion ist als Vorverarbeitung für die *Code Emission* zu sehen. Durch die Vorverarbeitung konnten die Code-Emission-Klassen des LLVM-Compilers als Basis für die Assemblerausgabe wiederverwendet werden. Es wurden nur noch die *Code Emission* um die Möglichkeit zur Ausgabe von parallelen Operationen durch die Platzierung des || Tokens zwischen paralleler Operationen einer RSIW-Instruktion erweitert.

6.5.3 Unterstützung des statischen mixed-ISA Programmiermodells

Bei dem statischen mixed-ISA Programmiermodell wird eine Anwendung so kompiliert, dass vor jeder Ausführung die ISA individuell festgelegt werden kann. Dazu kompiliert der Compiler mehrere ISA-Alternativen einer Anwendung in eine mixed-ISA ausführbare Datei. Das LLVM-Backend so erweitert, dass dieses für eine Übersetzungseinheit in einem Durchlauf automatisch Assemblercode für mehrere ISAs erzeugt. Diese Erweiterungen werden in diesem Abschnitt vorgestellt.

6.5.3.1 Function Cloning

Um gleichzeitig Code für verschiedene ISAs erzeugen zu können, wurde im Compiler ein neuer **Function Cloning** Pass am Anfang des LLVM-Backends eingefügt. Der Pass arbeitet auf der LLVM-Zwischendarstellung und dupliziert jede Funktion für jede zu kompilierende ISA. Dabei wird die Anzahl an Funktionen mit der Anzahl an ISAs multipliziert. Die Ziel-ISA, mit der eine Funktion kompiliert werden soll, wird dabei in der Klasse jeder Funktion gespeichert. Globale Variablen sind von der Duplizierung ausgeschlossen.

Jede Funktion braucht einen eindeutigen Funktionsnamen. Dies ist nach der Duplikation nicht mehr gegeben, da mehrere Funktionen mit gleichem Funktionsnamen existieren. Daher wurde zur eindeutigen Identifikation jeder ISA-Variante der Funktionsnamen mit einem ISA-Präfix erweitert. Diese Erweiterung wird dann bis zum Linker weiterverwendet. So wird der ISA-Name gefolgt von einem Punkt vor den Funktionsnamen geschrieben. Der Punkt wurde als Trennzeichen ausgewählt, weil dieser kein erlaubtes Zeichen eines Funktionsnamen in der C/C++-Programmiersprache darstellt. Z.B. wird Funktionsname `test` in `RSIW2.test` umgeschrieben, um die *RSIW 2* ISA-Variante der Funktion eindeutigen identifizieren zu können.

Neben den Funktionen werden auch Referenzen auf Funktionen (z.B. zum Aufruf einer Funktion) mit dem ISA-Präfix erweitert. Dabei ist entscheidend in welcher Funktion sich eine Funktionsreferenz befindet. Jede Funktionsreferenz wird mit der ISA, der Funktion in der sich die Referenz befindet, erweitert. Damit wird sichergestellt, dass jede Funktion nur andere Funktionen mit der gleichen ISA aufruft. Z.B. wird ein Aufruf der Funktion `prinft` innerhalb `RSIW22.test` durch `RSIW22.printf` ersetzt.

6.5.3.2 Steuerung der Ziel-ISAs

Im Vergleich zu einem gewöhnlichen Backend, das Code für eine ISA erzeugt, müssen bei einem mixed-ISA Backend eine Menge von ISAs angegeben werden, für die Code erzeugt werden soll. Innerhalb des LLVM-Compilers kann die ISA eines Backends (häufig ein Familie von Mikroprozessoren) durch die Subtarget-Einstellung kontrolliert werden. Z.B. verwendet das ARM-Backend die Subtarget-Einstellung um zwischen der gewöhnliches 32 Bit ISA und der platzoptimierten 16 Bit Thumb ISA umzuschalten. Die Subtarget-Einstellung wird für gewöhnlich über die Kommandozeile gesteuert und ist während der Kompilierung konstant. Während der mixed-ISA Kompilierung muss die Subtarget-Einstellung allerdings geändert werden und daher wurden die Subtarget-Einstellung in konstante und nicht konstante Variablen unterteilt.

Die konstanten Variablen werden durch die Kommandozeile gesetzt und beinhalten eine Liste aus Ziel-ISAs. Die Liste kann entweder auf eine ISA für klassische single-ISA Kompilierung oder auf mehrere ISAs für mixed-ISA Kompilierung gesetzt werden. Um mehrere ISAs anzugeben, kann man in der Kommandozeile eine kom-

maseparierte Liste von ISAs, eine ISA-Gruppe oder eine Mischung aus beiden angeben. Z.B. gibt der Kommandozeilenparameter `-mcpu=RSIW2` eine ISA an während `-mcpu=RSIW2,RSIW22,RSIW222,RSIW2222` mehrere ISAs auswählt.

Die nicht konstante Variablen beinhaltet die aktuell aktive ISA sowie ihrer Eigenschaften (z.B. Anzahl an Slots, Anzahl an Cluster, usw.). Sie werden während der Kompilierung immer dann gewechselt, wenn eine neue Funktion mit einer neuen ISA kompiliert wird. Das LLVM-Backend ist so organisiert, dass eine Funktion zuerst sämtliche Passes von *Instruction Selection & Scheduling* bis *Code Emission* durchläuft, bevor die nächste Funktion verarbeitet wird. Daher wird im mixed-ISA Backend die aktuelle ISA nur innerhalb des *Lowering*-Passes, dem ersten ISA-sensitiven Pass einer Funktion, geändert.

6.5.3.3 ISA-spezifischer Code

Das CoreGen-Werkzeug muss zur Realisierung der Benutzer-Retargierbarkeit Teile des Backend-Quellcodes erzeugen. Der erzeugt Quellcode des Backends besteht aus td- und C++-Dateien. Zur Bekanntgabe der verschiedenen ISAs der ADL wird innerhalb der td-Dateien eine Liste mit Subtargets erzeugt. Zusätzlich wird in C++ die Tabelle mit Eigenschaften der Subtargets erzeugt, die innerhalb der td-Dateien nicht spezifiziert werden können.

Für die *Befehlsauswahl* erzeugt das CoreGen-Werkzeug innerhalb der td-Dateien eine Liste von Befehlen. Für das mixed-ISA Backend wird nun für jede ISA eine eigene Liste von Befehlen erzeugt, die zur eindeutigen Identifizierbarkeit den ISA-Namen als Präfix haben. Innerhalb der td-Dateien kann man für eine Gruppe von Befehlen Vorbedingungen definieren, die abhängig von dem aktuellen Subtarget bzw. ISA sind. Dies wird für gewöhnlich für die Aktivierung oder Deaktivierung von Befehlen verwendet, die nur in bestimmten ISA-Dialekten vorkommen oder nicht vorkommen. In diesem Fall wird dieser Mechanismus zum Umschalten zwischen den ISAs innerhalb der ADL verwendet. Genauso wie die Liste der Befehle, wird auch für jede ISA separat der Registersatz und Registerklassen definiert. Die Registerklassen werden innerhalb der ISA-spezifischen Befehle verwendet. Somit wird implizit nach der *Befehlsauswahl* die richtige Registerklasse verwendet.

Neben den td-Dateien werden auch etliche Funktionen in C++, wie z.B. das Einfügen von Load/Store-Operationen für die *Registerzuteilung*, erzeugt. Jede einzelne dieser Funktionen muss für eine mixed-ISA Realisierung des Backends ebenfalls Realisierungen für sämtliche beschriebenen ISAs innerhalb der ADL bieten. So generiert CoreGen für eine generierte Funktion sämtliche Implementierung für alle beschriebenen ISAs. Die jeweilige Implementierung wird dann innerhalb der Funktion mittels einer Abfrage der aktuellen ISA und eine Switch-Case-Anweisung ausgewählt.

6.5.3.4 ISA-spezifische Passauswahl

Abhängig von der Ziel-ISA kann eine unterschiedliche Zusammensetzung der Passes im Backend notwendig sein. Entscheidend ist hier die Klasse der ISA. Innerhalb der ADL ist die Beschreibung von RISC-, VLIW- und Clustered-VLIW-Prozessoren und deren ISAs möglich. Dieses flexibilität der ADL wird auch mittels der rekonfigurierbaren RSIW-ISA der Kahrisma-Architektur ausgeschöpft. Bei einem single-ISA Backend würde man die Auswahl der Passes bei der Initialisierung treffen. Nach der Initialisierung kann die Auswahl an Passes nicht mehr geändert werden. Da bei einem mixed-ISA Backend für jede Funktion eine andere Auswahl benötigt wird, musste hier eine andere Lösung gefunden werden.

Pass	RISC	VLIW	Clustered VLIW
Instruction Selection & Scheduling	X	X	X
If Conversion	-	X	X
Clustering	-	-	X
Parallelizing Scheduling	-	X	X
Cluster Removal	-	-	X
Packing	-	X	X
Register Allocation	X	X	X
Prologue/Epilogue Insertion	X	X	X
Unpacking	-	X	X
Reclustering	-	-	X
Rescheduling	-	X	X
Nop Insertion	-	X	X
Nop Insertion	-	X	X
Code Emission	X	X	X

Tabelle 6.2: ISA-spezifische Pass-Auswahl

Bei dem realisierten mixed-ISA Backend werden daher bei der Initialisierung sämtliche Passes für die komplexeste ISA, der Clustered-VLIW-Prozessoren, erzeugt. Jeder Pass wird somit für alle ISAs, unabhängig ob dieser von der ISA benötigt wird, aufgerufen. Die Logik, ob ein Pass für eine ISA benötigt wird oder nicht, wurde stattdessen von der Initialisierung in den jeweiligen Pass verlagert. Ein Pass wird für jede Funktion aufgerufen und entscheidet anhand der ISA der jeweiligen Funktion, ob er durchgeführt wird oder nicht. Tabelle 6.2 zeigt die Auswahl der Passes abhängig von der ISA-Klasse an.

6.5.3.5 Mixed-ISA Code Emission

Bei der *Code Emission* wird die Operationsliste in eine Assemblerausgabe überführt. Da jede Funktion für eine andere ISA kompiliert werden kann, muss ebenfalls in der

Assemblerausgabe spezifiziert werden, welche ISA gerade verwendet wird. Zu diesem Zweck wurde eine `.target`-Pseudoassemblerdirektive eingeführt. Diese Direktive ermöglicht die Ausgabe von mixed-ISA Assemblerdateien. Durch die Ausgabe von `.target <isaname>` vor jeder Funktion innerhalb der *Code Emission* wird dadurch dem Assembler die aktuelle ISA bekanntgegeben.

6.5.4 Unterstützung des dynamischen mixed-ISA Programmiermodells

Bei dem dynamischen mixed-ISA Programmiermodell kann der Entwickler innerhalb der C/C++-Programmiersprache festlegen, für welche ISA eine Funktion kompiliert werden soll. Der Compiler muss dann automatisch Code zum Umschalten der ISA bei einem Funktionsaufruf generieren. Innerhalb des Umschaltcodes ändert sich die ISA im Programm durch eine spezielle Assembler-Instruktion. Dadurch muss der Compiler, im Gegensatz zum statischen mixed-ISA Programmiermodell, mit unterschiedlichen ISAs innerhalb einer Funktion umgehen können. Dies hat weitreichende Änderungen im Backend-Design zur Folge, die im Folgenden vorgestellt werden.

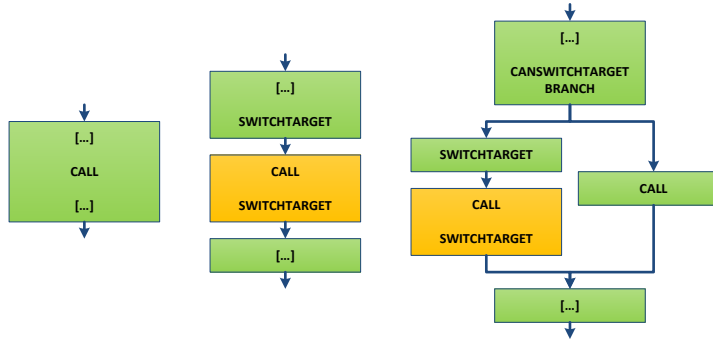
6.5.4.1 Function Cloning

Der Function Cloning Pass, der für das statische mixed-ISA Programmiermodell eingeführt wurde, wird auch im dynamischen mixed-ISA Programmiermodell zur Duplizierung der Funktionen verwendet. Beim dynamischen mixed-ISA Programmiermodell hängt es allerdings vom ISA-Funktionsmodus (siehe 6.2.2) ab, für welche ISAs eine Funktion dupliziert werden muss. Bei einem MUST-Funktionsmodus wird die Funktion nur für die angegebene ISA kompiliert während bei allen anderen Funktionsmodi die Funktion für sämtliche ISAs kompiliert wird.

6.5.4.2 Switch Target Insertion

Zum Einfügen des Umschaltcodes zwischen den verschiedenen ISAs wurde ein neuer *Switch Target Insertion* Pass in das Backend integriert. Hauptaufgabe des Passes ist das Hinzufügen von Code zum Umschalten der ISA. Dazu wird jeder Funktionsaufruf in jeder (ggf. duplizierten) Funktion überprüft, ob ein ISA-Wechsel durchgeführt wird. Bei einem ISA-Wechsel werden sog. SWITCHTARGET-Operationen vor und nach dem Funktionsaufruf hinzugefügt. Diese Operationen triggern vor dem Funktionsaufruf eine Änderung bzw. Rekonfiguration der ISA auf die ISA der aufgerufenen Funktion und stellen nach dem Funktionsaufruf die alte ISA wieder her. Falls der ISA-Wechsel nur optional durchgeführt werden soll (bei einem SHOULD-Funktionsmodus), wird vorher noch mittels einer CANSWITCHTARGET-Operation überprüft, ob die ISA gewechselt werden kann. In einem If-Else-Konstrukt wird dann, abhängig

von dem Ergebnis der Überprüfung, die Funktion mit oder ohne ISA-Wechsel aufgerufen.



(a) Ohne ISA-Wechsel (b) Mit ISA-Wechsel (c) Optionaler ISA-Wechsel

Abbildung 6.7: Einfügen der SWITCHTARGET- und CANSWITCHTARGET-Operationen

Ohne Beschränkung der Allgemeinheit wurde im Backend-Design die Entscheidung getroffen, dass ein Basisblock im Backend nur Instruktionen einer ISA enthalten darf. Dadurch kann die aktuelle ISA im Basisblock gespeichert werden und dies reduziert die Komplexität vieler Passes im Backend bei der Behandlung von mixed-ISA Funktionen. Dies hat im Switch Target Pass zur Folge, dass bei einem ISA-Wechsel in einem Basisblock dieser in zwei Basisblöcke gesplittet werden muss. Die Operation direkt nach der SWITCHTARGET-Operation wird in einer anderen ISA ausgeführt und muss daher in einen neuen Basisblock platziert werden. Abbildung 6.7 zeigt, wie der Kontrollflussgraph durch den Umschaltcode umgeschrieben wird. Ohne ISA-Wechsel ist in Abbildung 6.7(a) ein CALL direkt in einem Basisblock. Bei einem nicht optionalen ISA-Wechsel (Abbildung 6.7(b)) werden die SWITCHTARGET-Operationen um CALL hinzugefügt und entsprechend der ISA der Basisblock aufgeteilt. Bei einem optionalen ISW-Wechsel (Abbildung 6.7(c)) muss zusätzlich ein If-Else-Konstrukt mittels einer Sprung-Operation realisiert werden.

6.5.4.3 Mixed-ISA Behandlung auf Basisblock-Granularitätsebene

Durch das Hinzufügen von Umschaltcode bei Funktionsaufrufen muss das gesamte Backend die Kompilierung von unterschiedlichen ISAs auf Basisblock-Granularitäts-

ebene unterstützten. Im Backend durchläuft eine Funktion sämtliche Passes bevor die nächste Funktion übersetzt wird. Daher musste für die Unterstützung der statischen mixed-ISA Programmierbarkeit die aktuelle ISA im Backend nur vor der Verarbeitung jeder Funktion umgeschaltet werden. Bei der Basisblock-Granularitätsebene muss das Umschalten der aktuellen ISA allerdings während der Kompilierung einer Funktion innerhalb der jeweiligen Passes bei Bedarf durchgeführt werden. Dies hängt davon ab, ob ein Pass sensitiv auf die aktuelle ISA-Einstellung ist. Manche Passes, die über die Basisblockgrenzen hinaus agieren, bedürfen sogar einer speziellen Anpassung, die über das Umschalten hinausgeht. Im Folgenden werden die Änderungen pro Pass beschrieben:

Bei den Passes *Instruction Selection & Scheduling*, *Clustering*, *Parallelizing Scheduling*, *Cluster Removal*, *Packing*, *Unpacking*, *Reclustering*, *Rescheduling*, *Nop Insertion* und *Code Emission* wurde die aktuelle ISA anhand des Basisblocks geändert und eine ISA-spezifische Pass-Aktivierung auf Basisblock-Granularitätsebene implementiert. Im Folgenden werden sämtliche weitergehenden Erweiterung im Backend beschrieben.

6.5.4.4 Instruction Selection (dt. *Befehlsauswahl*)

Im *Switch Target Insertion* Pass wurden SWITCHTARGET- und CANSWITCHTARGET-Operationen hinzugefügt. Diese beiden Operationen sind plattformunabhängige Operationen, die in der *Befehlsauswahl* durch plattformspezifische Operationen ersetzt werden müssen. Dafür kann man in der ADL Operationen definieren, die die beiden plattformunabhängigen Befehle ersetzen.

6.5.4.5 If Conversion

Die *If Conversion* arbeitet über Basisblockgrenzen hinaus und bedarf einer speziellen Behandlung von mixed-ISA Kontrollflussgraphen. Durch die Bedingung, dass ein Basisblock nur Instruktionen einer ISA enthalten darf, muss in diesem Pass sichergestellt werden, dass nicht zwei Basisblöcke mit unterschiedlichen ISAs zusammengeführt werden. Die *If Conversion* in LLVM behandelt nur Basisblöcke deren Sprungverhalten analysierbar ist. Daher wurden sämtliche Basisblöcke mit einer SWITCHTARGET-Operation am Ende als nicht analysierbar gekennzeichnet. Dadurch führt LLVM keine *If Conversion* mehr über Basisblöcke durch, bei denen die ISA gewechselt wird.

6.5.4.6 Scheduling

Beim *Scheduling* und *Rescheduling* muss sichergestellt werden, dass die SWITCHTARGET-Operation immer am Ende eines Basisblocks steht, weil nach dieser Operation der Prozessor eine andere ISA ausführt. Dies konnte durch die Kennzeichnung der

SWITCHTARGET-Operation als Terminator sichergestellt werden. Diese Kennzeichnung wird für sämtliche Sprung-Befehle verwendet, damit diese am Ende eines Basisblocks eingeplant werden.

6.5.4.7 Register Allocation (dt. *Registerzuteilung*)

Ein kritischer Punkt bei der mixed-ISA Unterstützung ist die *Registerzuteilung*. Die *Registerzuteilung* operiert über Basisblockgrenzen hinaus. Jede ISA kann eine unterschiedliche Anzahl an Register besitzen, so dass bei einem ISA-Wechsel ein Teil der Registerinhalte verloren gehen können. In der ADL können die Register einer ISA frei gewählt werden. Die einzige Bedingung ist, dass der Stack- und Frame-Pointer bei einem ISA-Wechsel erhalten bleiben muss.

Im Backend wurden daher sämtliche SWITCHTARGET-Operationen so definiert, dass sie implizit sämtliche Register schreiben. Dadurch weiß die *Registerzuteilung*, dass sie keine Register über die Grenze einer SWITCHTARGET-Operation verwenden darf. Sämtliche verwendeten Register werden dann vorher auf den Stack geschrieben und später wieder vom Stack geladen. Dabei ist es unerheblich, in welcher ISA ein virtuelles Register benötigt wird. Der Datentransfer bei einem ISA-Wechsel findet ausschließlich über den Stack statt.

Eine Sonderbehandlung musste für die Stack- und Frame-Pointer eingeführt werden. In ISAs mit geclusterten Registerspeicher verwaltet der Compiler für jedes Cluster einen eigenen Stack- und Frame-Pointer. Sämtliche Stack- und Frame-Pointer müssen immer den gleichen Wert enthalten. Da sich je nach ISA die Anzahl an Cluster ändern kann, müssen bei einem ISA-Wechsel neu hinzugekommene Stack- und Frame-Pointer initialisiert werden. Zu diesem Zweck werden bei Bedarf der Stack- und Frame-Pointer aus dem ersten Cluster in die neuen Cluster transferiert.

6.5.4.8 Code Emission

Bei der *Code Emission* muss bei jedem Basisblock, der eine andere ISA wie der vorherige hat, eine `.target`-Pseudoassemblerdirektive ausgegeben werden. Dadurch wird dem Assembler bekannt gegeben, dass die folgenden Instruktionen für eine andere ISA assembliert werden müssen.

6.5.5 Unterstützung des automatischen mixed-ISA Programmiermodells

Beim automatischen mixed-ISA Programmiermodell wählt der Compiler automatisch eine passende ISA-Partitionierung für eine Anwendung aus. Dazu wurde das Konzept

der profile-guided Compilation verwendet. Dabei wird eine Profiling einer Anwendung zunächst mit Hilfe des Simulators erzeugt, basierend auf diesen Informationen eine Partitionierung berechnet und die Anwendung abschießend mit der Partitionierung kompiliert. Die Berechnung der Partitionierung findet dabei nicht im Backend des Compilers sondern in einem separaten Werkzeug, dem ISA-Partitionierer, statt (siehe Abschnitt 6.9). Das Backend muss dafür die Anwendung zunächst für das Profiling und in einem zweiten Schritt anhand der berechneten Partitionierung kompilieren können. Zu diesem Zweck wurde ein neuer *ISA Partitioning*-Pass in das Backend am Anfang integriert, der beide Schritt, gesteuert über die Kommandozeile, erlaubt.

6.5.5.1 ISA Partitioning

Der *ISA Partitioning*-Pass hat die Aufgabe eine ISA-Partitionierung festzulegen. Der Programmierer kann über die Erweiterung der C/C++-Programmiersprache pro Funktion eine ISA und ISA-Funktionsmodus festlegen. Beim AUTO-Funktionsmodus soll der Compiler automatisch einer passende ISA für die Funktion auswählen. Wenn keine Angaben für eine Funktion gemacht werden, ist die Funktion im AUTO-Funktionsmodus. Im *ISA Partitioning* wird jetzt jede AUTO-Funktion durch einen STAY-, MUST- oder SHOULD-Modus ersetzt. Dazu gibt es folgende Partitionierungsmöglichkeiten, die über die Kommandozeile ausgewählt werden.

STAY Sämtliche AUTO-Funktionen werden als STAY gekennzeichnet, so dass die ISA nicht gewechselt wird. Dies ist der Standard und entspricht dem dynamischen mixed-ISA Programmiermodell.

PROFILE Die Anwendung soll fürs ISA-Profiling kompiliert werde. Sämtliche Funktionen (nicht nur AUTO) werden als STAY gekennzeichnet. Bei dieser Einstellung ändert sich die ISA innerhalb einer Anwendung nie und diese kann für alle ISA zum Erzeugen des Profilings simuliert werden. Zusätzlich werden sämtliche ISA-Einstellungen für den ISA-Partitionierer in eine Datei gespeichert.

PARTITIONER Die ISA-Partitionierung, die durch den ISA-Partitionierer festgelegt wurde, wird aus einer Datei geladen und die ISA-Einstellungen sämtliche Funktionen werden anhand den Vorgaben dieser Datei gesetzt.

6.6 Mixed-ISA Binärwerkzeuge

Die Binärwerkzeuge des Softwareframeworks bestehen aus einem Assembler und Linker. Der Assembler übersetzt die Assemblerdateien zunächst in Objektdateien. Mehrere Objektdateien werden dann im Linker zu einer ausführbaren Datei zusammengelinkt. Insbesondere der Assembler muss zum einen benutzer-retargierbar anhand der ADL sein und zum anderen mit mehreren ISAs gleichzeitig umgehen können. Im Folgenden werden die beiden Werkzeuge erklärt.

Eine erste Version des Assemblers wurde in meiner Diplomarbeit [119] realisiert. Dieser war damals noch in den Core-Simulator integriert und auf einen einfachen RISC-Befehlssatz beschränkt. Dieser wurde dann innerhalb dieser Arbeit als eigenständiges Programm realisiert und um den Linker ergänzt.

6.6.1 Mixed-ISA Assembler

Der Assembler wurde komplett neu entwickelt. Innerhalb der ADL wird der Assemblersyntax und das Binärformat für jede Operation und Instruktion festgelegt. Diese Daten verwendet der Assembler als Grundlage. Dazu erzeugt das CoreGen-Werkzeug aus der ADL eine ISA-Tabelle, eine Register-Tabelle und pro ISA eine Tabelle mit Operationen, die sämtliche notwendigen Information enthalten.

6.6.1.1 Mixed-ISA Assemblersprache

Als Alleinstellungsmerkmal unterstützt der Assembler die Verarbeitung von einer mixed-ISA Assemblersprache, bei der die ISA durch die `.target` Pseudoassemblerdirektive in der Eingangssprache geändert werden kann. Ein Beispiel einer Funktion zum Wechseln der ISA ist in Quelltext 6.2 zu sehen. Die Funktion `init_switch` ist in der ISA RSIW2 definiert. Nach dem Prologue-Code wird durch den Assemblerbefehl SWITCHTARGET die ISA gewechselt. Die folgende Instruktion wird daraufhin in der neuen ISA ausgeführt. Daher muss auch im Assembler die ISA auf RSIW2222 umgestellt werden. Die `init`-Funktion wird dann in RSIW2222 aufgerufen und danach wird die ISA zurück auf RSIW2 gewechselt.

Quelltext 6.2: Beispiel einer mixed-ISA Assemblerdatei

```
.globl      RSIW2.init_switch
.align      1
.type       RSIW2.init_switch,@function
.target     RSIW2               ! Set RSIW2 ISA within assembler
RSIW2.init_switch:
    SUB %SP0, %SP0, 4           ! Prologue code
    ST32 [0 + %SP0], %RETREG

    SWITCHTARGET 0              ! Instruction to switch target to RSIW2222
.target     RSIW2222            ! Set RSIW2222 ISA within assembler
    MOV %SP1, %SP0              ! Setup stack pointers of RSIW2222
    MOV %SP2, %SP0
    MOV %SP3, %SP0

    CALL RSIW2222.init          ! Call init function with RSIW2222 ISA

    SWITCHTARGET 3              ! After return switch back to RSIW2
```

```
.target      RSIW2                ! Set RSIW2 ISA within assembler

    LD32 %RETREG, [0 + %SP0]     ! Epilogue code
    ADD %SP0, %SP0, 4
    RET
```

6.6.1.2 Aufbau

Der Assembler ist in die folgenden Phasen eingeteilt:

Lexical Analysis Zunächst wird eine lexikalische Analyse durchgeführt. Dabei wird die Assemblerdatei in Tokens unterteilt. Die lexikalische Analysis ist noch unabhängig von der ISA.

Syntax Analysis Der Syntax der Assemblerdateien ist zeilenorientiert. Typischerweise besteht eine Assembleranweisung aus einer Zeile. Um die Lesbarkeit zu erhöhen, ist es allerdings möglich, VLIW-Instruktionen bestehend aus mehreren Operationen über mehrere Zeilen zu verteilen. Daher wird zunächst die Tokenliste nach Anweisungen gesplittet.

Danach wird der Typ der Anweisung bestimmt. Eine Anweisung kann eine Symboldefinition (Symbol gefolgt von einem Doppelpunkt), eine Pseudoassemblerdirektive (Symbol beginnend mit Punkt) oder eine Instruktion sein. Das Ergebnis der syntaktischen Analyse ist eine Direktivenliste.

Bei den Pseudoassemblerdirektiven wurde sich beim Assembler an die Direktiven des GNU Assembler aus den GNU Binutils [106] orientiert, da diese Direktiven ohne große Anpassungen von der *Code Emission* des LLVM-Compilers verwendet werden. Als besondere Pseudoassemblerdirektive wurde `.target <targetname>` integriert, die ein Umschalten der aktuellen ISA im Assembler erlaubt. Diese stellt in der Abarbeitung im Assembler eine Sonderrolle dar. Während alle anderen Pseudoassemblerdirektiven zu der Direktivenliste hinzugefügt werden, ändert `.target` die aktuelle aktive ISA im Assembler. Dies hat direkte Auswirkungen auf die Syntaxanalyse der Instruktionen und legt fest, welche Instruktionen vom Assembler erkannt werden.

Eine Instruktion kann aus mehreren Operationen bestehen, wobei die Operationen durch das „||“-Token getrennt sind. Daher wird als erstes die Tokenliste einer Instruktion nach Operationen gesplittet. Jede Operation fängt mit einer Mnemonik an gefolgt von den Operanden. Für die Tokenliste der Operanden wird ein Syntaxbaum aufgestellt. In der ADL wird für jede Operanden einer Operation der Assemblersyntax als Zeichenkette angegeben. In einem Vorverarbeitungsschritt werden die Zeichenketten ebenfalls mit Hilfe der gleichen lexikalischen und syntaktischen Analyse in einen Syntaxbaum transformiert. Die Operation wird nun mit sämtlichen Operationen der aktuell ausgewählten ISA

verglichen. Dabei muss der Mnemonik identisch sein sowie der Syntaxbaum der Operanden mit dem Syntaxbaum einer passenden Operation kompatibel sein. Im Syntaxbaum der Operationsspezifikation sind Felder als Platzhalter angegeben. Wenn beim Syntaxbaumvergleich ein Platzhalter gefunden wird, muss überprüft werden, ob das korrespondierende Feld mit den Daten des anderen Syntaxbaums kompatibel ist. Bei einem Register-Feld muss im anderen Syntaxbaum ein Register angegeben sein, das im Feld enthalten ist. Bei einem Immediate-Feld muss im korrespondierenden Syntaxbaum eine passende Zahl, Symbol oder eine Addition aus Symbol und Zahl stehen. Nach einer erfolgreichen Erkennung einer Instruktion wird die Operation in die Direktivenliste hinzugefügt.

Semantic Analysis Bei der semantischen Analyse wird die Direktivenliste aus der syntaktischen Analyse ausgewertet. Als Ergebnis erzeugt die semantische Analyse einen Speicherbereich für jedes Segment, das durch den Assembler beschrieben wird. Z.B. enthält das `.text`-Segment den Code sämtlicher Instruktionen während im `.data`-Segment die Variablen gespeichert werden. Für jede Direktive wird zunächst dessen exakte Position im Segment bestimmt. Die erste Direktive eines jeden Segments wird an Position 0 platziert. Jede Direktive hat ein wohldefinierte Länge, die die aktuelle Position im Segment für die folgenden Direktiven erhöht. Am Ende ist die Position und das Segment jeder Direktive festgelegt und die Gesamtgröße der Segmente bekannt.

Nachdem die Positionen festgelegt sind, kann die Symbolliste erzeugt werden. In der Symbolliste wird für jedes Symbol seine exakte Position gespeichert. Dabei werden sämtliche Pseudoassemblerdirektiven ausgewertet, die die Eigenschaften eines Symbols verändern. Z.B. wird durch die `.globl`-Pseudoassemblerdirektive die Sichtbarkeit eines Symbols auf Global festgelegt.

Als nächster Schritt kann das Binärformat jeder Instruktion und Daten festgelegt werden. Dabei ist eine besondere Behandlung von Symbolen notwendig. So kann ein Immediate oder Datenwort von einem oder mehrere Symbolen abhängen. Bei einer Differenzberechnung zwischen zwei Symbolen ist das Immediate exakt berechenbar. Ansonsten müssen bei Symbolen ein Eintrag in eine Realokationstabelle vorgenommen werden. Diese Tabelle beschreibt, wie einer Instruktion oder Daten im Segmentspeicher verändert werden müssen, wenn ein Symbol beim Linken aufgelöst wird.

Im letzten Schritt werden die Daten eines Segments erzeugt, indem alle Binärformate von Instruktionen und Daten an die richtige Position im Segmentspeicher kopiert werden.

Output Generation Bei der Output Generation werden dann die Daten eines Segments, die Symboltabelle und Realokationstabelle in ein ELF-Format (siehe 6.6.2.1) überführt. Dafür wurde die ELFIO-Bibliothek [107] verwendet.

6.6.2 Linker

Für den Linker des Softwareframeworks wurde der GNU Linker der GNU Binutils [106] um die Kahrisma-Zielarchitektur erweitert. Der Linker unterstützt daraus sämtliche, durch die ADL spezifizierbaren, ISAs. Der Linker liest mehrere Objektdateien des mixed-ISA Assemblers ein und generiert dadurch eine ausführbare Datei.

6.6.2.1 ISA-invariantes ELF-Format

Für gewöhnlich ist sowohl der Linker als auch das ELF-Format abhängig von der ISA. Dies liegt darin begründet, dass der Linker Immediates von Operationen im Speicher umschreiben muss. Für jedes Operationsformat einer ISA ist daher im Linker ISA-spezifischer Realokationscode zum Umschreiben des Immediates vorhanden. Genauso befinden sich dann in der ELF-Datei ISA-spezifische Realokationseinträge, die definieren, wie eine Operation im Speicher umgeschrieben werden muss.

Damit nicht für jede ADL-Beschreibung ein neuer Linker und ELF-Format generiert werden muss, wurde daher ein allgemeines, ISA-invariantes ELF-Format mit allgemeinem Realokationscode entwickelt. Dabei wurde der Umstand ausgenutzt, dass in der ADL ein Immediate-Feld immer zusammenhängend sein muss. Es hat eine festgelegte Bit-Startposition und Bit-Länge. Diese beiden Parameter werden in jedem Realokationseintrag vom mixed-ISA Assembler kodiert und vom Linker umgesetzt. Dadurch konnte auf ein ISA-spezifisches ELF-Format und Linker verzichtet werden.

6.7 Core-Simulator

Der Core-Simulator ist eine benutzer-retargierbarer, zyklenapproximierender, mixed-ISA *Befehlssatzsimulator* (ISS, engl. *Instruction Set Simulator*), der sämtliche ISAs einer ADL-Spezifikation emulieren kann. Als Eingabe verwendet er eine ausführbare ELF-Datei, die vorher vom Linker erzeugt wurde.

Eine erste Version des Core-Simulators wurde in meiner Diplomarbeit [119] realisiert. Dieser wurde dann innerhalb dieser Arbeit stark erweitert.

Der Simulator hat folgende Ziele innerhalb des Softwareframeworks:

1. Er wird für die Validierung des Compiler und der Binärwerkzeuge in Verbindung mit der simulierten Anwendung verwendet. Nur wenn der Compiler, Assembler und Linker eine Anwendung korrekt übersetzt haben und der Simulator korrekt funktioniert, kann dieser korrekte Ergebnisse für eine Anwendung generieren.
2. Der Core-Simulator erzeugt zyklenapproximierende Performanzergebnisse. Diese können durch dynamische Programmanalysetechniken, wie z.B. Profiling,

auf Teile einer Anwendung abgebildet werden. Dies ist insbesondere für die automatische Partitionierung einer Anwendung durch den ISA-Partitionierer wichtig.

3. Der Core-Simulator kann Trace-Dateien generieren. Trace-Dateien enthalten das exakte Verhalten des Prozessors für jede ausgeführte Instruktion während der Simulation einer Anwendung. Es kann für die Validierung anderer Implementierungen der ISA(s) der ADL-Beschreibung, wie z.B. die Kahrisma-Hardwareimplementierung auf *Registertransferebene* (RTL, engl. *Register Transfer Level*), verwendet werden.
4. Er kann zur Fehlersuche in Anwendungen verwendet werden. Während der Compilerentwicklung passiert es häufiger, dass fehlerhafter Code erzeugt wird. In diesem Fall bietet der Simulation die Auflösung von Instruktionsadressen auf Assembler- oder Quellcodezeilen, einer Instruktionspointerhistorie sowie Trace-Dateien zur Fehlersuche an.

Im Folgenden wird auf die Realisierung des mixed-ISA Simulators genauer eingegangen. Zunächst wird in Abschnitt 6.7.1 das Design vorgestellt und in Abschnitt 6.7.2 der Simulationsablauf mit der Hauptschleife erklärt. Zur Performanzoptimierung besitzt der Simulator einen Decode Cache (Abschnitt 6.7.3). Die Simulation von parallelen Operationen, wie sie in der RSIW-ISA vorkommen, wird in Abschnitt 6.7.4 erläutert. Zur Realisierung der automatische ISA-Partitionierung unterstützt der Simulator das Profiling anhand Debugginginformationen (Abschnitt 6.7.6). Abschnitt 6.7.7 beschreibt die zyklenapproximativen Modelle der Simulation, die für eine möglichst gute Partitionierung der Anwendung eingesetzt werden können. Abschnitt 6.7.8 fasst das Kapitel zusammen.

6.7.1 Design

Als Grunddesign des Core-Simulators wurde die Technik der Interpretation ausgewählt. Im Gegensatz zur dynamischen oder statischen Kompilierungstechniken ist die Interpretation zwar langsamer hat aber den Vorteil, dass man in der Hauptschleife Berechnungen zur Approximation der Performanz der Architektur integrieren kann. Die Zyklenapproximation muss für jede Instruktion durchgeführt werden. Statische oder dynamische Kompilierungstechniken würden in diesem Fall keinen Sinn ergeben, weil ihr Performanzvorteil dadurch resultiert, dass sie mehrere Instruktionen auf Basisblockebenen zusammen kompilieren und optimieren können.

Abbildung 6.8 zeigt das Design des Simulators. Das CoreGen-Werkzeug erzeugt Fragmente des Simulationsquellcodes einer ADL-Beschreibung. Dabei wird für den Simulator eine Registertabelle, mehrere Operationstabellen sowie eine Simulationsfunktion pro Operation generiert. Zur gleichzeitgen Unterstützung mehrerer ISAs ist zusätzlich eine ISA-Tabelle vorhanden, die jeweils eine Operationstabelle referenziert. Während der Simulation ist dann immer nur die Operationstabelle der aktuellen ISA aktiv.

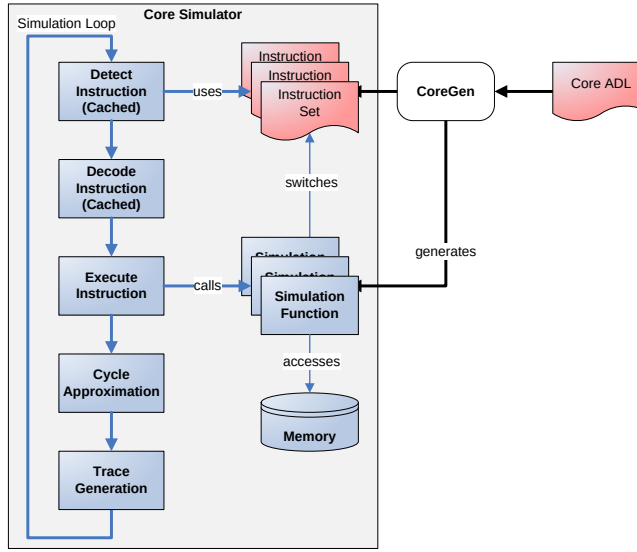


Abbildung 6.8: Design der zyklenapproximativen mixed-ISA Core-Simulator (Veröffentlicht [132])

Jede Operationstabelle enthält eine Liste mit sämtlichen Operationen einer ISA. Jeder Eintrag enthält den Name, die Größe in Bits, Felder, implizite Register und einen Pointer zur Simulationsfunktion. Die Felder repräsentieren die Kodierung des Instruktionsworts der Operation, z.B. die Kodierung und Position des Opcodes, Immediates oder der Ziel- und Quellregister. Sämtliche Register, die durch die Operation verwendet werden, ergeben sich aus der Dekodierung von Registerfeldern sowie den impliziten Registern. Implizite Register einer Operation werden, unabhängig von deren Operanden, immer gelesen oder geschrieben. Z.B. verändert ein Sprungbefehl implizit den *Befehlszeiger* (IP, engl. *Instruction Pointer*). Für jede Operation ist eine Simulationsfunktion vorhanden, die für die Ausführung der Operation aufgerufen wird. Diese wird aus den Simulationsquellcode in der ADL (siehe Abschnitt 5.3.6.1) generiert.

6.7.2 Simulationsablauf

Der Simulator bekommt als Eingange die kompilierte ausführbare Datei einer Anwendung im ELF-Format. Die ELF-Datei wird zunächst mittels eines ELF-Loaders in den simulierten Speicher des Prozessors geladen. Die Startadresse wird extrahiert und verwendet um den IP des Prozessors zu initialisieren. Danach wird die Simulationsschleife ausgeführt, die aus folgenden Phasen besteht:

Detect Instruction Als erstes wird in der Simulationsschleife zunächst die Instruktion erkannt. Dazu wird das Instruktionswort geladen und für jede Operation einer Instruktion die Operationstabelle der aktuellen ISA durchsucht. Dabei muss für eine Operation jedes konstante Feld (z.B. der Opcode) im Instruktionswort übereinstimmen. Zusätzlich muss jedes Feld eine erlaubte Kodierung haben.

Decode Instruction Nachdem jede Operation einer Instruktion erkannt wurde, wird die Instruktion dekodiert indem alle Felder einer Operation extrahiert werden. Diese werden dann in der sog. *Decode Struktur* gespeichert, um während der Simulation darauf schnell zugreifen zu können.

Execute Instruction Zum Ausführen einer Instruktion wird die generiert Simulationsfunktion aufgerufen. Dabei werden Teile der *Decode Struktur* direkt an die Simulationsfunktion übergeben.

Cycle Approximation Nach der Ausführung kann optional eine Zyklenapproximation durchgeführt werden. Dabei unterstützt der Simulator mehrere Zyklenapproximationsmodelle. Diese werden in Abschnitt 6.7.7 näher beschrieben.

Trace Generation Ebenfalls kann nach der Ausführung einer Instruktion optional das Prozessorverhalten in eine Trace-Datei gespeichert werden. Für jede ausgeführte Operation wird die Zyklennummer, der Name der dekodierten Operation, die Debugging-Informationen (Namen und Zeilennummer der Assembler- und C-Quelldatei), die Werte aller dekodierten Felder (z.B. Opcode), die Werte sämtlicher Ein- und Ausgabeoperanden sowie die Nummern der Ein- und Ausgaberegister festgehalten.

6.7.3 Decode Cache

Die beiden Phasen *Detect Instruction* und *Decode Instruction* in der Hauptschleife sind der Flaschenhals in interpretierenden Simulatoren. Zur Kompensation wurde daher in den Core-Simulator ein Cache eingeführt. Dieser enthält die *Decode Struktur* sämtlicher erkannten und dekodierten Instruktionen und wird anhand des IPs adressiert. Dadurch muss jede Instruktion an einem IP nur einmal erkannt und dekodiert werden. Wegen dem Prinzip der Programmlokalität kann die Anzahl erkannter und dekodierter Instruktionen drastisch reduziert werden. Durch den Cache wird die Zeit, die zum Erkennen und Dekodieren benötigt wird, unerheblich für die Simulationsgeschwindigkeit.

Zur Realisierung des Caches wurde der *unordered_map*-Container der Boost-Bibliothek [108] verwendet. Dieser Implementiert eine Hashtabelle und hat eine durchschnittliche Zeitkomplexität nahe $O(1)$ beim Cachezugriff. Um auch den Cachezugriff aus der Hauptschleife weitestgehend zu eliminieren, wurde zusätzlich eine Instruktionsvorhersage in den Simulation integriert. Die Grundidee besteht darin, dass bei den meisten Instruktionen (außer Sprungbefehle) die folgende Instruktion immer gleich ist. Auch bei Sprungbefehlen ist die folgende Instruktion häufig identisch, was z.B. bei der Sprungvorhersage in Prozessoren mit 1-Bit-Prädikator ausgenutzt wird.

Zur Realisierung der Instruktionsvorhersage wird in der *Decode Struktur* der IP und Decode-Struktur-Pointer auf die folgende Instruktion gespeichert. Diese werden immer aktualisiert, wenn sich die folgende Instruktion geändert hat. Um die *Decode Struktur* der aktuellen Instruktion zu bekommen, wird zunächst dessen IP mit dem vorhergesagten IP der vorherigen Instruktion vergleichen. Bei Übereinstimmung kann der Decode-Struktur-Pointer der vorherigen Instruktion verwendet werden. Ansonsten wird ein Cachezugriff durchgeführt und der folgende IP und Decode-Struktur-Pointer der vorherigen Instruktion aktualisiert. Dadurch können ein Großteil der vergleichsweise teuren Cachezugriffe in der Simulationsschleife eliminiert werden und die durchschnittliche Zeit zum Erkennen und Dekodieren von Instruktionen wird unerheblich.

6.7.4 Simulation von parallelen Operationen

Zur Simulation einer RISC-Instruktion, bestehend aus einer einzigen Operation, genügt ein Aufruf der Simulationsfunktion, die durch CoreGen generiert wurde. Die Ausführung von parallelen Operationen einer VLIW-Instruktion ist dagegen komplexer. Für jede Operation existiert eine Simulationsfunktion. Dabei ist es wichtig, dass die Quellregister sämtlicher Operationen einer Instruktion geladen werden, bevor ein Zielregister geschrieben wird. Daher ist es nicht einfach möglich die Simulationsfunktionen der Operationen einer Instruktion nacheinander aufzurufen. Eine Lösung wäre die Generierung von Simulationsfunktionen für jede mögliche Kombination an Operationen in einer Instruktion. Allerdings würde in diesem Fall die Anzahl an Simulationsfunktionen exponentiell mit der Anzahl an parallelen Operationen wachsen.

Stattdessen wurde im Core-Simulator eine Lösung gewählt, bei der nur eine Simulationsfunktion pro Operation verwendet wird, die allerdings die anderen Simulationsfunktionen der Instruktion rekursiv aufruft. In der Simulationsfunktion wird nach dem Ausführen und vor dem Schreiben der Zielregister die Simulationsfunktion der nächsten Operation in der Instruktion aufgerufen. Dies wird bis zur letzten Operation einer Instruktion fortgesetzt. Bei dieser linearen Rekursion werden zuerst alle Quelloperanden geladen und die Operationen ausgeführt, bevor alle Zieloperanden geschrieben werden.

6.7.5 Mixed-ISA Unterstützung

Um die Rekonfigurierbarkeit der Prozessorarchitektur simulieren zu können, kann der Simulator die ISA während der Laufzeit verändern. Dafür wurde der Zustand des Prozessors (bestimmt durch den Register- und Hauptspeicher) um eine Variable zur Speicherung der aktuellen ISA erweitert. Jede ISA wird durch eine eindeutige Nummer identifiziert, die durch das CoreGen-Werkzeug vergeben wird. Über die Kommandozeile kann die initiale ISA festgelegt werden. Andernfalls startet der Simulator mit der Standard-ISA, wie sie in der ADL angegeben ist.

Jede ELF-Datei hat im Header eine Startadresse angegeben, die den Einsprungpunkt der Anwendung festlegt. Bei einer mixed-ISA Anwendung ist allerdings ein Einsprungpunkt nicht ausreichend. Stattdessen brauch man pro ISA einen eigenen Einsprungpunkt. Daher ist der Boot-Code in der ELF-Datei so organisiert, dass der Einsprungpunkt nicht auf den Startcode sondern auf eine Sprungtabelle zeigt. In der Sprungtabelle ist dann für jede ISA ein separater Einsprungpunkt pro ISA-spezifischen Startcode hinterlegt. Die Sprungtabelle wird beim Starten des Simulators anhand der initialen ISA adressiert und somit der ISA-spezifische Einsprungpunkt geladen. Falls die ISA in der ELF-Datei nicht vorhanden ist, ist der Einsprungpunkt auf NULL gesetzt und der Simulator kann eine Fehlermeldung ausgeben.

Zum Wechsel der ISA während der Simulation kann im Simulationscode, definiert in der ADL, eine spezielle Funktion aufgerufen werden, die die aktuelle ISA im Simulator ändert. Bei der RSIW-ISA ist dafür die neue `SWITCHTARGET`-Instruktion vorhanden. Diese Instruktion akzeptiert einen Operanden (z.B. ein Immediate) und wechselt die ISA anhand des Wertes des Operanden. Die nächste Instruktion wird dann mit den Einstellungen der neuen ISA (d.h. hauptsächlich einer neuen Operationstabelle) erkannt und dekodiert.

6.7.6 Profiling und Debugging

Zum Debugging und zur Generierung von Statistiken kann der Simulator Speicheradressen einer Instruktion zu ihrer korrespondierenden Zeile der Assemblerdatei, Zeile der C/C++-Quellcodedatei sowie ihrem Funktionsnamen zuordnen. Für die Zuordnung auf Assemblerdateien sowie deren Zeilennummer speichert der Assembler Zuordnungsinformationen in einer speziellen Sektion in der ELF-Datei. Zusätzlich ist der Compiler in der Lage, Debugging-Informationen in die Assemblerausgabe einzubetten. Diese werden im DWARF-Format [109] gespeichert. Ein DWARF-Reader innerhalb des Simulators kann die Quellcodedateinamen und -zeilennummern extrahieren. Weiterhin sind in den Symboltabellen der ELF-Datei die Start- und Endadressen sämtlicher Funktionen gespeichert.

Während der Simulation wird für jede dekodierte Instruktion innerhalb der *Decode Struktur* die Häufigkeit ihrer Ausführung gezählt. Am Ende der Simulation können die Informationen ausgewertet werden. Die Zähler auf Instruktionsebene können mit

Hilfe der Debugging-Informationen auf Funktionen, Zeilen der Quellcodedateien und Zeilen der Assemblerdateien abgebildet werden.

6.7.7 Modelle zur Zyklenapproximation

Neben der funktionalen Simulation unterstützt der Simulator verschiedenen Modelle zur Approximation der Ausführungszeit einer Anwendung auf Mikroarchitekturebene. Im Gegensatz zu einem taktgenauen Simulator wird hierbei die Mikroarchitektur nicht exakt im Simulator modelliert. Stattdessen werden die Zyklen basierend auf einem heuristischen Modell approximiert, das einen Tradeoff zwischen Genauigkeit und Simulationsgeschwindigkeit bietet. Momentan werden im Core-Simulator drei verschiedenen Zyklenmodelle unterstützt, die im Folgenden beschrieben werden: *Instruction-Level Parallelism* (ILP), *Atomic Instruction Execution* (AIE) und *Dynamic Operation Execution* (DOE).

6.7.7.1 Instruction-Level Parallelism (ILP)

Das ILP-Zyklenmodell führt eine schnelle Messung des theoretischen ILPs einer Anwendung durch. Der theoretische ILP versucht die obere Schranke für die Anzahl an Operationen pro Zyklus für eine Architektur mit unlimitierten Ressourcen zu bestimmen. Er sagt die Performanz von ungeclusterten, out-of-order RSIW-Prozessorinstanzen mit unlimitierten Anzahl an parallelen Operationen, unlimitierte Anzahl an Renaming-Registern und einer idealen Speicherarchitektur (mit 100% L1-Cache-Trefferrate und unlimitierte Anzahl an parallelen Speicherzugriffen) voraus. In einer solchen theoretischen Architektur wird die Parallelität auf Befehlsebene durch echte Datenabhängigkeiten limitiert. Als Eingang wird die RISC-ISA (RSIW1) verwendet. Die RISC-Instruktionen werden durch den Simulator in der vom Compiler vorgegebenen Reihenfolge simuliert. Allerdings würde es eine solche theoretische Architektur erlauben, sämtliche parallele Operationen in einer Instruktion so früh wie möglich auszuführen. Zur Imitation dieses Verhaltens wird für jede Operation ein individueller Start- und Fertigstellungszyklus berechnet. Dieser ist unabhängig der Simulationsreihenfolge, sondern wird hauptsächlich über die Datenabhängigkeiten berechnet. Daher kann eine Instruktion, die später in der Simulationsreihenfolge ist, einen niedrigeren Fertigstellungszyklus haben. Dies entspricht dem Ausführungsmodell von out-of-order Prozessoren, die ebenfalls die Operationen außerhalb der Reihenfolge ausführen können.

Zur Modellierung von echten Datenabhängigkeiten wird zu jedem Register der Zyklus des letzten Schreibzugriffs gespeichert. Dieser ist durch den Fertigstellungszyklus der letzten Operation gegeben, die das Register geschrieben hat. Der Startzyklus einer Operation ist abhängig von seinen Quellregistern. Er wird auf das Maximum der Schreibzugriffszyklen sämtlicher Quellregister gesetzt. Der Fertigstellungszeitpunkt ist dann durch den Startzyklus plus der Operationsverarbeitungszeit gegeben. Bei

VLIW-Prozessoren können (bei einem einfachen Compiler) nur Operationen bis zum nächsten Sprungbefehl parallel eingeplant werden. Daher muss der Startzyklus einer Operation zusätzlich größer als der Fertigstellungszyklus der letzten Sprungoperation sein. Für Speicheroperationen wurde ein pessimistisches Modell verwendet, das davon ausgeht, dass sämtliche Speicherzugriffe voneinander abhängen. Dieses Modell geht von einem Compiler aus, der keine Aliasanalyse von Speicheroperation durchführt. Ein Lade/Speicher-Operation ist in diesem Modell immer abhängig von der letzten Speicher-Operation und kann daher frühestens zum Startzyklus der letzten Speicheroperation ausgeführt werden.

6.7.7.2 Atomic Instruction Execution (AIE)

Im AIE-Zyklenmodell wird davon ausgegangen, dass sämtliche Operationen einer Instruktion im gleichen Zyklus angestoßen werden. Die folgende Instruktion kann nur angestoßen werden, wenn alle Operationen der vorherigen Instruktion fertig ausgeführt wurden. Dies entspricht dem klassischen Ausführungsmodell bei VLIW-Prozessoren. Innerhalb des Simulators wird daher die Ausführungszeit einer Instruktion durch das Maximum sämtlicher Ausführungszeiten der individuellen Operationen berechnet.

6.7.7.3 Dynamic Operation Execution (DOE)

Das DOE-Zyklenmodell approximiert die Performanz der Kahrisma-Architektur (siehe Abschnitt 3.5.2.6), die dieses Ausführungsmodell in der Hardware umsetzt. Im Gegensatz zum AIE-Modell muss nicht jede Operation einer Instruktion im gleichen Zyklus angestoßen werden. Stattdessen können die Slots einer RSIW-Instruktion gegenseitig driften. Eine Operation in einem Slot kann angestoßen werden, wenn die vorherige Operation des Slots angestoßen wurden und die echten Datenabhängigkeiten über die Quellregister eingehalten sind.

Innerhalb des Simulators werden die echten Datenabhängigkeiten des DOE-Modells genauso wie im ILP Modell über den letzten Schreibzugriff der Register modelliert. Zusätzlich wird für jeden Slot der Startzyklus der letzten Operation hinterlegt. Innerhalb eines Slots können die Operationen nur innerhalb der Reihenfolge angestoßen werden. Daher muss der Startzyklus einer Operation mindestens der Startzyklus der letzten Operation plus eins sein. Durch die Addition der Eins wird sichergestellt, dass maximal eine Operation pro Zyklus und Slot angestoßen werden kann.

Mit diesem einfachen Modell kann die Performanz der Kahrisma-Architektur ohne exakte Simulation der Mikroarchitekturpipeline approximiert werden. Dieses Modell ist aus folgenden Gründen heuristisch:

- Die Ressourcenconstraints (wenn z.B. ein Multiplikator zwischen zwei Slots geteilt wird) werden nicht berücksichtigt.

- Der Drift zwischen den Slots ist innerhalb der Hardware auf einen maximalen Wert beschränkt. Dieser maximale Wert wird im Simulator nicht modelliert.
- Die Speicheroperationen werden im Simulator in der Programmreihenfolge und nicht in der out-of-order Reihenfolge der Hardware ausgeführt.

6.7.7.4 Approximation des Speichermodells

Innerhalb des Simulators wird die Dauer von Speicherzugriffen approximiert. Dabei wird die Approximation in der Reihenfolge des Befehlsstroms und nicht in der Ausführungsreihenfolge der Hardware durchgeführt. Zur Approximation der Verzögerung wird die Speicherhierarchie mit drei Arten von Modulen modelliert: Speicher, Caches und Verbindungsbegrenzung. Jedes Modul hat das gleiche Interface, das aus einer Funktion zur Berechnung des Fertigstellungszyklus eines Speicherzugriffs besteht. Innerhalb eines Cache- und Verbindungsbegrenzungmoduls ist ein Pointer zum nächsten Modul der Speicherhierarchie hinterlegt. Ein Speichermodul befindet sich dann am Ende dieser Kette. Ein Cachemodul reicht z.B. den Speicherzugriff im Falle eines Cache-Misses an das nächste Modul weiter. Innerhalb der Berechnung des DOE- oder AIE-Zyklenmodells wird zur Bestimmung der Dauer von Speicheroperationen die Funktion des ersten Moduls in der Hierarchie aufgerufen. Dabei wird die Speicheradresse, der Zugriffstyp (Lesen oder Schreiben), der Slot der Operation und der Startzyklus als Parameter an die Funktion übergeben. Als Ergebnis liefert die Funktion den Fertigstellungszyklus des Speicherzugriffs.

Im Folgenden werden die drei Module vorgestellt:

Speicher Das Speichermodul ist das einfachste Modul. Es kann mit der Dauer eines Speicherzugriffs konfiguriert werden. Es berechnet den Fertigstellungszyklus durch die Addition des Startzykluses mit der Zugriffsdauer.

Cache Das Cachemodul modelliert einen n-Wege assoziativen Cache. Dieser verwendet die Write-Back-Schreibstrategie und *Least Recently Used* (LRU) Verdrängungsstrategie. Die Zeilengröße, Assoziativität, Cachegröße und Zugriffsdauer sind variable gehalten.

Zur Berechnung des Fertigstellungszyklus wird der *aktuelle Zyklus* als interne Variable eingeführt und mit dem Startzyklus initialisiert. Dieser wird schrittweise in der Berechnung erhöht, bis es am Ende der Berechnung als Fertigstellungszyklus zurückgegeben wird. Die Berechnung gliedert sich in folgende Schritte:

- Der aktuelle Zyklus wird um die Zugriffsdauer auf den Cache erhöht. Dies repräsentiert die Zyklen, die für den Cachezugriff benötigt werden. Beim Cachezugriff wird festgestellt, ob ein Cache-Hit oder -Miss aufgetreten ist.
- Bei einem Cache-Miss werden folgende Berechnungen ausgeführt:
  - Zunächst wird eine freie Cachezeile gesucht. Falls keine vorhanden

ist, wird nach dem LRU-Prinzip eine Cachezeile zur Verdrängung ausgesucht. Falls die Cachezeile als Dirty markiert ist, muss sie erst zurück geschrieben werden. Dazu wird ein Schreibzugriff zum nächsten Modul in der Hierarchie (z.B. ein zweiter Cache) initiiert, indem dessen Berechnungsfunktion aufgerufen wird. Dieser Schreibzugriff hat als Startzyklus den aktuellen Zyklus und der berechnete Fertigstellungszyklus wird als neuer aktueller Zyklus verwendet.

- Nachdem eine freie Cachezeile vorhanden ist, wird der Lesezugriff zum Füllen der Cachezeile initiiert und nach dem gleichen Prinzip die Berechnungsfunktion des folgenden Moduls mit der aktuellen Zyklus als Startzyklus aufgerufen.
- In der Cachezeile wird die Adresse als Tag sowie der aktuelle Zyklus hinterlegt. Der aktuelle Zyklus gibt an, wann die Cachezeile gefüllt wurde.

- Bei einem Cache-Hit werden folgende Berechnungen ausgeführt:
  - Da die Berechnungsfunktion mit Startzyklen außerhalb der Reihenfolge aufgerufen werden kann, wird bei jedem Cache-Miss der Zyklus hinterlegt, wann die Cachezeile gefüllt wurde. Bei einem Cache-Hit wird jetzt der aktuelle Zyklus mit dem Zyklus der Cachezeile verglichen und der aktuelle Zyklus darauf gesetzt, wenn der Zyklus der Cachezeile größer ist. Dadurch wird sichergestellt, dass die Zeiten eines Cache-Misses auch bei parallelen Speicherzugriffen für sämtliche Speicherzugriffe mitberechnet werden.
- Bei einem Schreibzugriff wird die Cachezeile als Dirty markiert.
- Der aktuelle Zyklus wird als Fertigstellungszyklus zurückgegeben.

Verbindungsbegrenzung Das Cachemodul berechnet nur die Dauer eines Cachezugriffs ohne dabei den Ressourcenconstraints eines Caches zu berücksichtigen. Ein Cache kann nur eine bestimmte Anzahl an Zugriffe pro Zyklus durchführen. Dies ist in Hardware durch die Anzahl an Ports des Caches begrenzt. Bei einem gepipelinten L1-Cache kann dieser einen Zugriff pro Zyklus und Port durchführen. Diese Begrenzung wird durch das Verbindungsbegrenzungsmodul modelliert, das vor einen Cache oder Speicher geschaltet werden kann. Es kann mit der maximalen Anzahl an Ports konfiguriert werden.

Das Verbindungsbegrenzungsmodul speichert, ob ein Port für einen Zyklus belegt ist. Bei einem Speicherzugriff wird für dessen Startzyklus ein freier Port gesucht. Der Startzyklus wird dabei so lange erhöht, bis ein freier Port vorhanden ist. Danach wird der Port und Zyklus als verwendet markiert und der Speicherzugriff an das nächste Modul (typischerweise ein Cache) mit dem aktualisierten Startzyklus weitergegeben.

Das Modul geht davon aus, dass für einen Port gleichzeitig eine bidirektionale

Kommunikation möglich ist. Daher wird das gleiche Prinzip, das für den Hinkanal und dem Startzyklus angewandt wurde, ebenfalls für den Rückkanal und den Fertigstellungszyklus angewandt.

6.7.7.5 Zyklenapproximation der Kahrisma-Architektur

Mit den drei Modulen des Speichermodells konnte im Core-Simulator die Speicherhierarchie der Kahrisma-Architektur nachgebaut werden. Diese besteht aus einem L1-Cache, einem L2-Cache und einem Hauptspeicher. Dementsprechend wurden zwei Cachemodule und ein Speichermodul verwendet. Zusätzlich wurde vor das L1-Cachemodul ein Verbindungsbegrenzungsmodul platziert, um den parallelen Cachezugriff zu begrenzen.

Mit dieser Konfiguration zusammen mit dem DOE-Pipelinemodell konnten gute Ergebnisse für die Vorhersage der Performanz der Kahrisma-Architektur erzielt werden. Die Genauigkeit wird genauer im Ergebniskapitel in Abschnitt 7.1.2.2 behandelt.

6.7.8 Zusammenfassung

Der Core-Simulator des Softwareframeworks ermöglicht die Simulation eines einzelnen Threads der Kahrisma-Architektur. Durch seine Benutzer-Retargierbarkeit durch die ADL unterstützt er dabei sämtliche ISA-Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur. Ferne wird eine mixed-ISA Simulation unterstützt, bei der während der Laufzeit die ISA gewechselt und somit eine Rekonfiguration von RSIW emuliert werden kann. Dadurch ermöglicht der Simulator mit seiner funktionalen Simulation eine Validierung des gesamten mixed-ISA Softwareframeworks.

Aufbauend auf der funktionalen Simulation unterstützt der Simulator verschiedene Modelle zur Zyklenapproximation. Dabei wird die Performanz einer Prozessorpipeline mit *Dynamic Operation Execution* (DOE)-Verhalten approximiert, wie es von der out-of-order Pipeline der Kahrisma-Architektur implementiert ist. Zusammen mit einer in-order Approximation der out-of-order Speicherzugriffe bietet der Simulator einen Tradeoff zwischen Performanz und Simulationsgenauigkeit.

Zur Generierung von Statistiken und zur dynamischen Programmanalyse kann der Simulator ein Profiling durchführen. Zusammen mit der Zyklenapproximation können die Profiling-Informationen für eine automatische ISA-Partitionierung der Anwendung durch den ISA-Partitionierer (siehe Abschnitt 6.9) verwendet werden. Der Core-Simulator ist somit ein wichtiges Werkzeug zur Realisierung der automatische mixed-ISA Programmierbarkeit.

6.8 System-Simulator

Der System-Simulator ermöglicht die Simulation auf der Kahrisma-Prozessorarchitektur auf Systemebene. Dabei wird die Simulation von mehreren virtuellen Prozessorinstanzen ermöglicht, was für die Entwicklung von parallelen mixed-ISA Anwendungen benötigt wird. Als Eingabe verwendet der Simulator eine parallele mixed-ISA Anwendung sowie eine Systembeschreibung innerhalb der System-ADL.

Der System-Simulator ist unter meiner Anleitung in der Diplomarbeit von Timo Sandmann [122] entstanden.

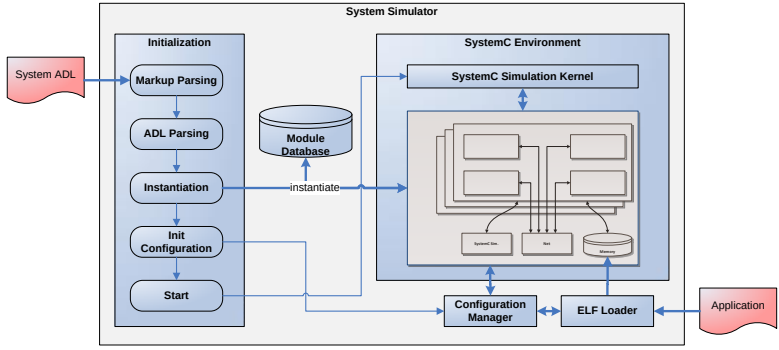


Abbildung 6.9: Aufbau des System-Simulators

Abbildung 6.9 zeigt das Design des System-Simulators. Der System-Simulator basiert auf der SystemC-Simulationssprache, die auf der Programmiersprache C++ aufsetzt und durch Makros und Klassen diese um Simulationskonzepte erweitert. Für gewöhnlich wird SystemC für die Simulation von einem statischen System verwendet. Innerhalb dieser Arbeit wird der Simulator allerdings benutzer-retargierbar anhand der Systembeschreibung in der System-ADL gehalten. Zusätzlich wurden Konzepte zur Simulation von rekonfigurierbaren Architekturen integriert.

Die Benutzer-Retargierbarkeit des System-Simulators wird durch einen bibliotheksbasierten Ansatz realisiert. Dabei wird in der System-ADL die Struktur des zu simulierenden Systems auf einer hohen Abstraktionsebene modelliert. In dieser Beschreibung ist das Verhalten eines Moduls nicht modelliert. Stattdessen ist innerhalb des Simulators eine Datenbank aus SystemC-Modulen vorhanden, die in der System-ADL-Beschreibung referenziert werden können. Während der Initialisierung der SystemC-Umgebung werden die SystemC-Module anhand der strukturellen Beschreibung innerhalb der System-ADL instanziiert und verschaltet. Dadurch wird ein flexibler Sys-

tem-Simulator ermöglicht, der die Designzeit-Flexibilität der Kahrisma-Architektur auf Systemebene unterstützt.

Als einzigartiges Feature unterstützt der System-Simulator die Simulation von rekonfigurierbaren Prozessorarchitekturen. Bei der Kahrisma-Architektur können verschiedene Module zu virtuellen Prozessorinstanzen zusammengeschaltet werden. Innerhalb der System-ADL kann dieses Prinzip durch die Beschreibung von Konfiguration modelliert werden. Die Konfigurationen werden parallel zu der strukturellen Beschreibung spezifiziert. Sie zeichnen sich durch einen Ressourcentemplate sowie ein Simulationsmodul aus, das durch eine Referenz in der Modulbibliothek spezifiziert ist.

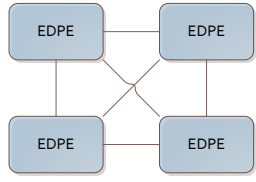
Eine Konfiguration kann in das simulierte Design geladen werden, wenn genügend freie Ressourcen vorhanden sind. Die benötigten Ressourcen einer Konfiguration sind im positionsunabhängigen Ressourcentemplate festgelegt. Dafür wird in jedem rekonfigurierbaren Module gespeichert, ob dieses bereits konfiguriert ist, also einer Konfiguration angehört. In diesem Fall ist es für die Konfiguration von neuen Konfigurationen geblockt. Ein Konfigurationsmanager ermöglicht im System-Simulator das Laden von neuen Konfigurationen. Dabei werden zuerst freie passende Ressourcen gesucht, die Ressourcen als konfiguriert markiert und das Konfiguration-Simulationsmodul instanziiert und mit den rekonfigurierbaren Modulen verbunden. Die rekonfigurierbaren Simulationsmodule haben dabei selbst keine Logik sondern fungieren nur als Wrapper für das Konfiguration-Simulationsmodul.

Abbildung 6.10(a) zeigt als Beispiel ein kleines Design einer Kahrisma-Prozessorarchitektur bestehend aus vier Prozessorressourcen (EDPEs), wie es in der System-ADL beschrieben werden kann. Abbildung 6.10(a) zeigt ein Ressourcentemplate einer RSIW22-Konfiguration, das auf das Design gemappt werden kann. Abbildung 6.10(a) zeigt sämtliche sechs Möglichkeiten, wie die Konfiguration auf das Design gemappt werden kann. Aus diesen sechs Möglichkeiten kann der Konfigurationsmanager eine heraussuchen, wenn das Laden einer Konfiguration angefordert wird. Dabei wird eine Instanz eines Core-Simulators als Konfiguration-Simulationsmodul erzeugt und mit den betroffenen EDPE-Modulen, die als Wrapper fungieren, verbunden.

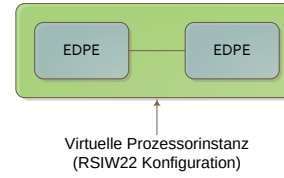
Abbildung 6.9 zeigt ebenfalls die Schritte der Initialisierung des System-Simulators, die im Folgenden vorgestellt werden:

Markup Parsing Als erstes wird die Datenbeschreibungssprache der System-ADL (siehe Abschnitt 5.2), genauso wie beim CoreGen-Werkzeug, geparst, interpretiert und in eine baumartige Datenstruktur umgewandelt. Bei der Interpretation werden konstante mathematische Ausdrücke ausgerechnet, Variablen aufgelöst und Funktionen und Kontrollstrukturen (if, for) ausgeführt.

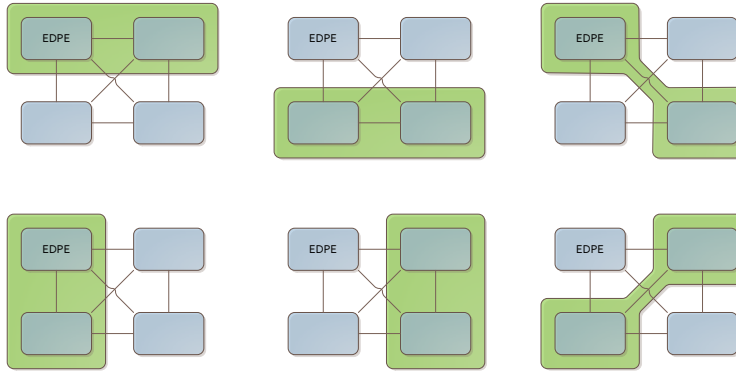
ADL Parsing analysiert den Baum und überführt dessen Inhalt in interne Datenstrukturen, die ähnlich den Sektionen der System-ADL aufgebaut sind. Dabei wird sowohl eine syntaktische als auch semantische Überprüfung der Daten im Baum vorgenommen. Bei der semantischen Überprüfung werden insbesondere



(a) Strukturelles Design bestehend aus vier Prozessorressourcen (EDPEs)



(b) Spezifikation des Ressourcentemplates einer RSIW22-Konfiguration



(c) Mappingmöglichkeiten der Konfiguration auf das Design

Abbildung 6.10: Beispiel der Beschreibung von Konfigurationen für den Simulator

die Referenzen zwischen den Sektionen überprüft. Zusätzlich werden für jede Konfiguration sämtliche möglichen Abbildungsvarianten ausgerechnet.

Instantiation verwendet die SystemC-Module gemäß der Spezifikation der System-ADL, die in den internen Datenstrukturen geladen wurden. Jedes Modul der System-ADL referenziert ein SystemC-Modul aus der Moduldatenbank. Das Modul wird in der Datenbank gesucht, in der SystemC-Umgebung instanziiert und mit den anderen Modulen gemäß der System-ADL-Beschreibung verbunden.

Init Configuration lädt die Boot-Konfiguration mit Hilfe des Konfigurationsmana-

gers in den Simulator. Dabei wird mittels des ELF-Loaders die ausführbare Datei in den Speicher der Simulation geladen.

Start Der SystemC-Kernel wird aufgerufen und die Ausführung wird diesem übertragen. Dadurch wird die Simulation gestartet.

6.9 ISA-Partitionierer

Der ISA-Partitionierer ist ein Werkzeug des mixed-ISA Softwareframeworks, das automatisch eine ISA-Partitionierung einer Anwendung berechnen kann, um einen Tradeoff zwischen Performanz und Ressourcen- bzw. Energieverbrauch zu ermöglichen. Zusammen mit dem mixed-ISA Compiler und Simulator ermöglicht der ISA-Partitionierer eine profile-guided Optimierung einer Anwendung und somit eine Realisierung des automatischen mixed-ISA Programmiermodells.

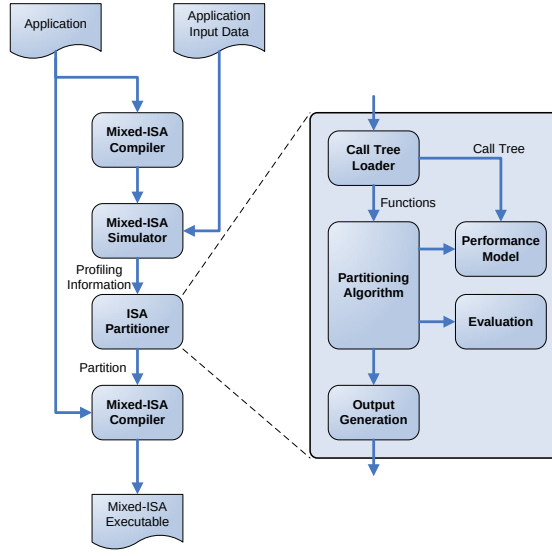


Abbildung 6.11: Verwendung und Aufbau des ISA-Partitionierers

Abbildung 6.11 zeigt, wie der ISA-Partitionierer innerhalb des mixed-ISA Softwareframeworks für die profile-guided mixed-ISA Optimierung verwendet werden kann.

Zur Datenakquise wird zunächst eine Anwendung mittels des mixed-ISA Compilers für mehrere ISAs kompiliert. Dabei wird das statische mixed-ISA Programmiermodell verwendet, bei dem sämtlichen ISAs in eine mixed-ISA ausführbare Datei kompiliert werden. Mit Hilfe des Simulators wird die Anwendung für eine Gruppe von ISAs simuliert und ISA-abhängige Profiling-Informationen gesammelt. Dabei können der simulierten Anwendung Eingangsdaten per Kommandozeile als Dateien zur Verfügung gestellt werden. Anhand der Profiling-Informationen und der Partitionierungsparameter wird die Anwendung mit dem ISA-Partitionierer partitioniert und die Partitionierung in einer Datei gespeichert. Bei einem erneuten Aufruf des Compilers wird die Anwendung nach den Daten aus der Partitionierungsdatei kompiliert und somit die automatische mixed-ISA Programmiermodell umgesetzt.

6.9.1 Eingangsdaten

Um eine Partitionierung der Anwendung auf Funktionsebene durchführen zu können, benötigt der ISA-Partitionierer genaue Informationen, wie lange eine Funktion in einer bestimmten ISA benötigt und wie häufig eine Funktion eine andere aufruft. Diese Informationen sind in den flachen Profiling-Informationen des Simulators vorhanden und sind ausreichend, wenn jede Funktion exakt einer ISA zugeordnet ist.

Wenn allerdings die ISA einer Funktion von der aufgerufenen Funktion abhängt, werden Profiling-Informationen benötigt, die die Aufrufhistorie einer Funktion berücksichtigen. Dies ist beim STAY-Funktionsmodus der Fall, bei dem die ISA der aufgerufenen Funktion gleich der ISA der aufrufenden Funktion ist. Der Simulator unterstützt die Generierung von hierarchischen Profiling-Informationen mittels Tracing. Dabei wird in einem Aufrufbaum die Dauer der Funktionsabarbeitung und Aufrufhäufigkeit in Abhängigkeit ihrer Aufrufhistorie aufgeschlüsselt. Die Unterstützung des STAY-Funktionsmodus ist insbesondere bei der Entwicklung von heuristischen Partitionierungsalgorithmen vorteilhaft, bei denen man einen Teil der Funktionen nicht beachten muss, indem man sie konstant auf STAY setzt.

Als Eingangsdaten für den ISA-Partitionierer werden hierarchische Profiling-Informationen im Simulator generiert. Dabei wird für jede ISA ein separater Simulationslauf initiiert. Als Ergebnis bekommt man für jede ISA einen Aufrufbaum (engl. *Call Tree*) sämtlicher verwendeter Funktionen. In dem Aufrufbaum ist für jede Funktion deren Zeitbedarf in Zyklen sowie deren Aufrufhäufigkeit enthalten. Abbildung 6.12 zeigt als Beispifel einen Aufrufbaum, wie er vom ISA-Partitionierer verwendet wird. In dem Baum werden im Knoten der Funktionsnamen sowie deren erforderlichen Zyklen abhängig von der ISA dargestellt. Die erste Zahl gibt die verbrauchten Zyklen der Funktion selbst und die zweite Zahl inklusive der aufgerufenen Funktionen wieder. Auf den Kanten steht, wie häufig eine Funktion von einer anderen aufgerufen wurde.

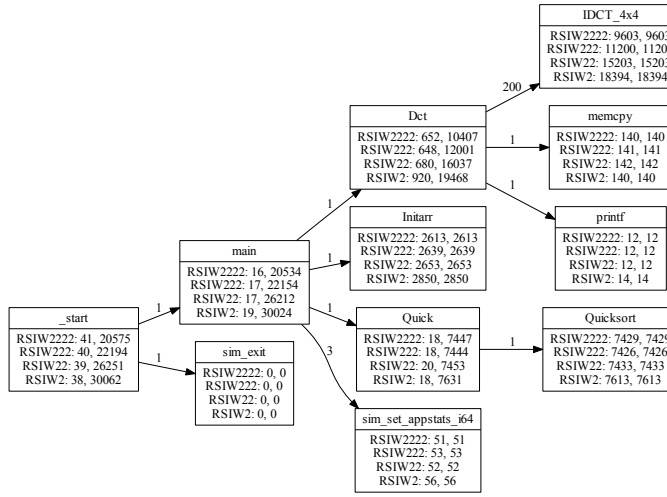


Abbildung 6.12: Beispiel eines Aufrufbaums als Eingabe für den ISA-Partitionierer

6.9.2 Aufbau

Abbildung 6.11 zeigt den Aufbau des ISA-Partitionierers. Zunächst lädt der *Call Tree Loader* die hierarchischen Profiling-Informationen der verschiedenen Simulationsläufe und kombiniert diese in einem einzigen Aufrufbaum, erweitert mit ISA-spezifischen Performanzinformationen. Danach führt das *Partitioning Algorithm* Modul den Partitionierungsalgorithmus aus, der entscheidet, welche Partitionen in welcher Reihenfolge überprüft werden sollen. Bei der Überprüfung berechnet zunächst *Performance Model* die erwartete Performanz der Anwendung anhand des Aufrufbaums und *Evaluation* evaluiert die Lösung anhand der Zielfunktion. Dabei wird die beste Lösung gespeichert. Am Ende der Partitionierung schreibt *Output Generation* das Ergebnis in eine Partitionierungsdatei.

Der ISA-Partitionierer ist sehr modular gehalten, so dass für die Module *Partitioning Algorithm*, *Evaluation* und *Output Generation* jeweils verschiedenen Typen zur Verfügung stehen, die verschieden kombiniert werden können. Diese werden in den nächsten Abschnitten erklärt.

6.9.3 Partitioning Algorithm

Das *Partitioning-Algorithm*-Modul entscheidet, in welcher Reihenfolge welche Partitionen evaluiert werden. Das Partitionierungsproblem ist im Allgemeinen NP-Schwer. Daher können aus Zeitgründen nicht alle möglichen Partitionen einer mixed-ISA Anwendung überprüft werden. Der Partitionierungsalgorithmus entscheidet innerhalb des ISA-Partitionierers welche Partitionierung in welcher Reihenfolge überprüft werden sollen. Es wurden drei Partitionierungsalgorithmen implementiert:

Random Der Random-Partitionierungsalgorithmus wählt zufällige Partitionierungen aus. Dabei kann der Seed und die Anzahl an Partitionierung angegeben werden.

Const Der Const-Partitionierungsalgorithmus wählt alle Partitionieren aus, bei denen sich die ISA nicht ändert. Dieser Algorithmus ist nur für Statistiken relevant.

Priority Enumeration Der Priority-Enumeration-Partitionierungsalgorithmus ist der Standard-Partitionierungsalgorithmus. Die Grundidee des Algorithmus ist auf das Prinzip der Programmlokalität zurückzuführen, das besagt, dass ein Großteil der Zeit eines Programms von einer kleinen Anzahl an Funktionen verursacht wird. Somit müssen für eine effiziente Partitionierung auch nur die Funktionen betrachtet werden, die am meisten Zeit verbrauchen. Dazu betrachtet der Algorithmus nur die Funktionen mit der meisten Ausführungszeit und setzt die restlichen Funktionen auf STAY. Für die betrachteten Funktionen werden sämtliche ISA-Kombinationen überprüft. Im Einzelnen geht der Algorithmus folgendermaßen vor:

1. Er speichert sämtliche Funktionen in dem *Func*-Vektor.
2. Er sortiert den *Func*-Vektor absteigend anhand der Ausführungszeit der Funktion inklusive ihrer Kinder. Danach steht die Funktion mit der größten Ausführungszeit ganz oben.
3. Er geht den Vektor der Reihe nach durch. Dabei wird in einer Variable gespeichert, wie viele Partitionen mit den Funktionen erzeugen werden können. Sobald eine Funktion die Anzahl an erlaubten Partitionen überschreitet, wird diese und sämtliche nachfolgenden Funktion aus dem *Func*-Vektor gelöscht. Alle gelöschten Funktionen werden auf STAY gesetzt.
4. Der Algorithmus enumeriert und überprüft sämtliche Partitionen, die mit den verbliebenen Funktionen realisiert werden können.

6.9.4 Performance Model

Die Partitionen, die vom Partitionierungsalgorithmus erzeugt werden, müssen innerhalb des ISA-Partitionierers evaluiert werden. Dafür ist es wichtig, die Performanz

und den Ressourcenverbrauch der Partitionierung zu bestimmen. Dies könnte durch Kompilierung und Simulation durchgeführt werden. Dies wäre allerdings viel zu Aufwändig, weil man im Idealfall mehrere Sekunden für eine Evaluation benötigen würde. Stattdessen wurde ein approximatives Performanzmodell verwendet, mit dem wesentlich mehr Partitionen in gleicher Zeit untersucht werden können.

Das Performanzmodell verwendet den Aufrufbaum erweitert mit Profiling-Informationen. Der Aufrufbaum wird rekursiv durchlaufen und dabei immer die ISA der aktuellen Funktion bestimmt. Die Dauer einer Funktion wird anhand ihrer ISA aus den Profiling-Informationen entnommen und zur Gesamtdauer addiert. Zusätzlich wird bei einem ISA-Wechsel die Aufrufhäufigkeit mit den Kosten für den ISA-Wechsel multipliziert und ebenfalls der Gesamtlaufzeit addiert. Die Kosten für einen ISA-Wechsel sind in einer Tabelle gespeichert, in der die Kosten für einen ISA-Wechsel für sämtliche ISA-Kombinationen gespeichert sind. Die Tabelle wurde zuvor mit Experimenten mit dem Compiler und Simulator erzeugt. Dadurch kann die Performanz und der Ressourcenverbrauch einer Partitionierung gut abgeschätzt werden.

6.9.5 Evaluation

Das Evaluation-Modul bekommt sämtliche überprüften Partitionieren übergeben. Das Modul hat einen Speicher, indem je nach Evaluationsmethode eine oder mehrere Lösungen gespeichert werden. Je nachdem wie viele Lösungen generiert werden, werden die Methoden in zwei Gruppen verteilt:

6.9.5.1 Mehrere Lösung

Evaluationsmethoden mit mehr als einer Lösung sind nur für Statistiken und für Feedback für den Entwickler relevant, um z.B. eine grafische Auswertung des Lösungsraums vornehmen zu können.

All Es werden sämtliche Partitionierungen gespeichert.

Pareto Es werden sämtliche pareto-optimalen Partitionen gespeichert. Diese Pareto-Menge kann später gespeichert und dem Endnutzer zur Verfügung gestellt werden, um eine geeignete Partitionierung auszuwählen.

6.9.5.2 Einzelne Lösung

Evaluationsmethoden mit einer einzigen Lösung versuchen eine effiziente Partitionierung zu bestimmen. Dazu kann man verschiedene Methoden bzw. Optimierungskriterien auswählen, die alle eine Kostenfunktion und Nebenbedingungen haben. Die Nebenbedingungen bestimmen, wann eine Lösung verworfen wird. Für alle akzeptierten Lösungen wird die Lösung gespeichert, die die Kostenfunktion minimiert.

Optimize Resources Es wird der Ressourcenverbrauch unter Angabe der maximalen Performanzreduktion in Prozent optimiert. Die Kostenfunktion ist gleich dem Ressourcenverbrauch. Als Nebenbedingung darf sich die Zyklenzahl nur um einen gewissen Prozentsatz verschlechtern.

Dabei wird zur Initialisierung die performanteste Lösung sämtlicher konstanten Partitionierungen gespeichert. Bei der Evaluation werden dann nur Lösungen betrachtet, die maximal n Prozent langsamer sind als die performanteste Lösung. Darunter wird die Lösung mit den niedrigsten Ressourcenverbrauch gespeichert.

Optimize Performance Es wird die Performanz unter Angabe der maximalen Ressourcenerhöhung in Prozent optimiert. Die Kostenfunktion ist gleich der Zyklenanzahl. Als Nebenbedingung darf sich der Ressourcenverbrauch nur um einen gewissen Prozentsatz verschlechtern.

Factor Der Ressourcenverbrauch und die Zyklenanzahl (Performanz) werden jeweils mit einem konstanten Faktor multipliziert und die Summe daraus minimiert. Es gibt keine Nebenbedingung.

6.9.6 Output Generation

Bei der Output Generation werden sämtliche Lösungen, die vom Evaluation-Modul gespeichert wurden, ausgegeben. Dabei stehen zwei Methoden für die Output Generation zur Verfügung:

Partition Es wird die Partitionierung für den mixed-ISA Compiler gespeichert. Dies ist nur in Kombination mit einer Evaluationsmethode möglich, die ein Ergebnis produziert.

Statistics Es werden die Performanz- und Ressourcenverbrauchergebnisse sämtlicher Ergebnisse in eine Datei geschrieben. Dies kann vom Endbenutzer zur Auswahl einer sinnvollen ISA-Partitionierung verwendet werden.

6.10 Zusammenfassung

In diesem Kapitel wurde die Realisierung des **mixed-ISA Softwareframeworks** vorgestellt, das die *Programmierbarkeit* der rekonfigurierbaren Kahrisma-Prozessorarchitektur zur dynamischen Ausnutzung von verschiedenen ILP/TLP-Charakteristika in Anwendungen umsetzt. Nur durch die Programmierbarkeit der neuartigen rekonfigurierbaren Features der Kahrisma-Prozessorarchitektur können diese auch in aktuelle Software, geschrieben in der C/C++-Hochsprache, Einzug halten und ermöglicht somit einen Einsatz der Technologie über das akademische Umfeld hinaus.

Die grobgranular rekonfigurierbare Kahrisma-Prozessorarchitektur ermöglicht eine dynamische Konfiguration von virtuellen Prozessorinstanzen, die sich durch die Anzahl an Ressourcen sowie ihrer Performanz unterscheiden können. Durch die Wahl der virtuellen Prozessorinstanz kann somit dynamisch zur Laufzeit auf die Anforderungen und Charakteristika von Anwendungen zur Optimierung des Ressourcen-/Energiebedarfs sowie der Performanz eingegangen werden. Eine effiziente Umsetzung dieser Flexibilität in Hardware erforderte allerdings die Verwendung der rekonfigurierbaren RSIW-Befehlssatzarchitektur als Interface zwischen Hard- und Software. Im Unterschied zu klassischen Befehlssatzarchitekturen hat eine rekonfigurierbare Befehlssatzarchitektur unterschiedliche, konfigurationsabhängige Ausprägungen. Klassische Software- und Compilerframeworks sind für die Programmierung nicht ausreichend, weil diese immer nur eine Konfiguration unterstützen könnten. Daher wurde innerhalb dieser Arbeit ein mixed-ISA-fähiges Softwareframework realisiert, das sämtliche Anforderungen der Kahrisma-Architektur erfüllt. Der Begriff „mixed-ISA“ bedeutet hierbei, dass das Softwareframework mit variablen *Befehlssatzarchitekturen* (ISAs, engl. *Instruction Set Architecturs*) umgehen kann. Auf diese Weise werden sämtliche Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur umgesetzt.

Der Programmierer einer rekonfigurierbaren Prozessorarchitektur benötigt die Möglichkeit der Steuerung der Konfiguration aus der Software heraus. Zu diesem Zweck wurden drei **mixed-ISA Programmiermodelle** definiert, die von dem Softwareframework umgesetzt werden: das statische, das dynamische und das automatische mixed-ISA Programmiermodell. Beim statischen mixed-ISA Programmiermodell kann die ISA bzw. Konfiguration beim Start gewählt werden und bleibt während der Laufzeit einer Anwendung unverändert. Beim dynamischen mixed-ISA Programmiermodell kann man die ISA bzw. Konfiguration zur Laufzeit manuell steuern, um auf Veränderungen der Charakteristika innerhalb einer Anwendung eingehen zu können. Beim automatischen mixed-ISA Programmiermodell wird eine Anwendung automatisch anhand vorgegebener Optimierungskriterien durch Wechsel der ISA bzw. Konfiguration zur Laufzeit optimiert.

Das **statische mixed-ISA Programmiermodell** wurde innerhalb des Softwareframeworks durch einen *mixed-ISA Compiler* und *mixed-ISA Binärwerkzeuge* umgesetzt. Als Basis für den Compiler wurde die LLVM-Compiler-Infrastruktur verwendet. Dazu wurde ein neues benutzer-retargierbares, mixed-ISA LLVM-Backend realisiert. Die Benutzer-Retargierbarkeit wurde durch das *CoreGen-Werkzeug* umgesetzt, das mittels Metaprogrammierung Teile des Quellcodes des Backends generiert. Das Backend musste für die Codegenerierung von Clustered-VLIW-Befehlssatzarchitekturen erweitert werden, die in bestimmten Konfigurationen der RSIW-Befehlssatzarchitektur enthalten sind. Für das statische mixed-ISA Programmiermodell wurde das Backend so erweitert, das es in einem Durchlauf sämtliche Funktion anhand der ISA dupliziert und diese dann abhängig von der ISA bzw. Konfiguration kompiliert. Als Ausgabe erzeugt es eine mixed-ISA Assemblerdatei, die verschiedene mixed-ISA Implementierungen pro Funktion enthält. Die Binärwerkzeuge, der Assembler und Linker, wurden

um die Möglichkeit der Verarbeitung von mixed-ISA Anwendungen erweitertet und generieren eine mixed-ISA ausführbare Datei.

Zur Realisierung des **dynamischen mixed-ISA Programmiermodells** wurde eine *mixed-ISA Erweiterung der C/C++-Programmiersprache* spezifiziert, die eine Steuerung der ISA in einer Anwendung zur Laufzeit mittels mixed-ISA Funktionsattributen ermöglicht. Das Frontend verarbeitet die mixed-ISA Funktionsattribute und speichert sie in der LLVM-Zwischendarstellung. Auf diese Weise werden die mixed-ISA Einstellung dem Backend zur Verfügung gestellt. Dieses dupliziert die Funktionen dann anhand der Funktionsattribute und fügt bei Bedarf Code zum Umschalten der ISA ein. Insbesondere die automatische Generierung des Umschaltcodes erforderte die Unterstützung von verschiedenen ISAs auf Basisblock-Granularitätsebene, die durch eine Erweiterung nahezu sämtlicher Backend-Passes erreicht wurde.

Für das **automatische mixed-ISA Programmiermodell** wurde eine *profile-guided Optimierung* der ISA in das Softwareframework integriert. Dabei wird eine Anwendung zunächst mit dem statischen mixed-ISA Programmiermodell kompiliert und ISA-abhängige, hierarchische Profiling-Informationen durch den Simulator erzeugt. Anhand dieser Informationen berechnet dann ein neues Werkzeug, der *ISA-Partitionierer*, eine ISA-Partitionierung. Der ISA-Partitionierer löst das NP-schwere Partitionierungsproblem mittels eines heuristischen Algorithmus unter Ausnutzung der Programmlokalität. Innerhalb des Algorithmus wird eine Partitionierung durch ein mixed-ISA Performanzmodell evaluiert, das die Performanz anhand der ISA-abhängigen Profiling-Informationen abschätzt. Der Entwickler kann dabei über Optimierungsparameter die Partitionierung steuern. Als Ergebnis speichert er die berechnete ISA-Partitionierung in einer Datei. Bei einem erneuten Durchlauf des Compilers wird die ISA-Partitionierungsdatei in Kombination mit der dynamischen mixed-ISA Codegenerierung verwendet, um die finale mixed-ISA optimierte Anwendung zu generieren.

Das mixed-ISA Softwareframework wird durch einen **Core- und System-Simulator** komplettiert. Der Core-Simulator unterstützt die mixed-ISA Simulation einer virtuellen Prozessorinstanz, wie sie in der mixed-ISA Architekturbeschreibungssprache spezifiziert ist. Dabei kann sowohl die Start-ISA gewählt als auch die ISA zur Laufzeit gewechselt werden. Er integriert ein zyklenapproximatives Performanzmodell, das in der Lage ist, die Performanz der virtuellen Prozessorinstanzen zu approximieren ohne die Prozessorpipeline im Detail zu simulieren. Dadurch wird ein guter Tradeoff zwischen Performanz und Genauigkeit erzielt. Damit kann der Core-Simulator sowohl für die Validierung des Softwareframeworks als auch für die Generierung der Profiling-Informationen für die profile-guided Optimierung des automatischen mixed-ISA Programmiermodells verwendet werden.

Der System-Simulator simuliert die Kahrisma-Prozessorarchitektur auf Systemebene und kann, im Gegensatz zum Core-Simulator, mehrere virtuelle Prozessorinstanzen gleichzeitig simulieren. Dabei verwendet der System-Simulator die System-ADL als Eingabe, die das Design der simulierten Prozessorarchitektur festlegt. Der Simulator basiert auf der SystemC-Simulationssprache und kann die rekonfigurierbaren Eigen-

schaften der Kahrisma-Architektur simulieren. Für die Simulation des Verhaltens einer virtuellen Prozessorinstanz werden auf verschiedene Instanzen des Core-Simulators zurückgegriffen. Dadurch ermöglicht der System-Simulator die Entwicklung von parallelen mixed-ISA Anwendungen.

7 Ergebnisse

In diesem Kapitel werden die Ergebnisse des mixed-ISA Softwareframeworks und der Kahrisma-Architektur vorgestellt. Die Ergebnisse sind in zwei Sektionen unterteilt. Zunächst werden in Abschnitt 7.1 alle Ergebnisse für Ein-Prozessor-Anwendungen vorgestellt. Hierbei liegt der Fokus auf einer einzelnen, virtuellen Prozessorinstanz der Kahrisma-Architektur. Danach wird in Abschnitt 7.2 auf die Ergebnisse für Multi-Prozessor-Anwendungen eingegangen. Abschnitt 7.3 charakterisiert die Realisierung des Softwareframework und fasst die erzielten Ergebnisse zusammen.

7.1 Ein-Prozessor-Anwendungen

In diesem Abschnitt wird zunächst der Versuchsaufbau beschrieben (Abschnitt 7.1.1). Dies beinhaltet die Core-ADL für die Kahrisma-Architektur sowie die verwendeten Ein-Prozessor-Anwendungen. Danach werden Ergebnisse für den Core-Simulator (Abschnitt 7.1.2), die Benutzer-Retargierbarkeit des Softwareframeworks (Abschnitt 7.1.3), das statische mixed-ISA Programmiermodell (Abschnitt 7.1.4), das dynamischen mixed-ISA Programmiermodell (Abschnitt 7.1.5) und das automatischen mixed-ISA Programmiermodell (Abschnitt 7.1.6) vorgestellt.

7.1.1 Versuchsaufbau

7.1.1.1 Core-ADL

Zum Generieren der Ergebnisse wurde eine ADL-Beschreibung entwickelt, die ein Obermenge der RSIW-Befehlssatzarchitektur (siehe Abschnitt 3.4) umsetzt. Tabelle 7.1 zeigt sämtliche RSIW-Konfigurationen, wie sie in der Core-ADL spezifiziert wurden. Dabei wird für jede Konfiguration der Konfigurationsname, die Anzahl an Slots (d.h. die Anzahl an parallelen Operationen innerhalb einer RSIW-Instruktion), die Anzahl an Cluster, die Anzahl an Register sowie die Anzahl an benötigten Ressourcen (d.h. Anzahl an EDPEs) angegeben.

RSIW2, RSIW22, RSIW222 und RSIW2222 sind praktisch relevante Konfigurationen, wie sie von der Kahrisma-Architektur umgesetzt werden können.

RSIW1 benötigt die gleiche Anzahl an Ressourcen wie RSIW2, verwendet allerdings

Konfiguration	Slots	Cluster	Register	Ressourcen
RSIW1	1	1	32	1
RSIW2	2	1	32	1
RSIW22	4	2	2x32	2
RSIW222	6	3	3x32	3
RSIW2222	8	4	4x32	4
RSIW4	4	1	64	-
RSIW6	6	1	96	-
RSIW8	8	1	128	-

Tabelle 7.1: RSIW-Konfigurationen

nur einen Issue-Slot innerhalb der EDPE. Obwohl nicht alle Ressourcen verwendet werden, kann dadurch Energie und Instruktionsspeicher eingespart werden. Die Konfiguration ist ebenfalls von theoretischer Relevanz, da sie einen RISC-Prozessor repräsentiert und somit nützlich als Vergleich ist.

RSIW4, RSIW6 und RSIW8 sind theoretische single-cluster Versionen von RSIW22, RSIW222 und RSIW2222. Sie diene als Vergleich und haben deswegen die gleiche Anzahl an Registern.

7.1.1.2 Anwendungen

Zur Generierung wurden verschiedene Anwendungen auf das Softwareframework portiert. Für diese Anwendungen wurde das gesamte Softwareframework validiert. Die Anwendungsmenge beinhaltet die folgenden Applikationen:

Cjpeg JPEG-Encoder [110] (aus dem MiBench-Benchmark [111])

FFT KissFFT [112], eine einfach *keep-it-simple-stupid* (kiss) *Schnelle Fourier-Transformation* (FFT, engl. *Fast Fourier Transform*)-Implementierung mit fixed-point Datentyp

Quicksort Quicksort-Sortieralgorithmus [113], der zufallsgenerierte Integer-Zahlen sortiert

Dhrystone Dhrystone-Benchmark [114]

AES Eine komplett ausgerollte *Advanced Encryption Standard* (AES)-Implementierung [115]

DCT Eine 4x4 integer *Diskrete Kosinustransformation* (DCT, engl. *Discrete Cosine Transform*)-Approximation, wie sie im H.264-Videocodec [116] zum Einsatz kommt

7.1.2 Evaluation des Core-Simulators

Der Core-Simulator ist ein wichtiger Bestandteil des Softwareframeworks. Wie in Kapitel 6.7 beschrieben, erlaubt der Core-Simulator eine Approximation der Zyklen einer Anwendung. Mit dieser Approximation wird er als Grundlage für die Ergebnisse in diesem Kapitel verwendet. Dabei ist die Approximation ein Tradeoff zwischen Performanz und Simulationsgenauigkeit. Diese beiden Parameter werden in den folgenden beiden Abschnitten evaluiert.

7.1.2.1 Performanz

Zur Performanzmessung wurde die Cjpeg-Anwendung verwendet. Diese wurde für eine RSIW1-Konfiguration kompiliert. Sämtliche Performanzmessungen des Simulators wurden auf einem einzelnen Kern eines Intel Xeon X5680 Prozessors bei 3,33 GHz durchgeführt. Der Simulator wurde mit GCC und O3-Optimierungen kompiliert.

Der Simulator verwendet das Konzept von interpretativen Simulatoren. Allerdings wurde durch einen Decode-Cache der zeitaufwändige Dekodiervorgang beschleunigt. Zusätzlich wurde durch eine Instruktionsvorhersage der Zeitaufwand für den Cache-Zugriff weiter reduziert.

Um den Einfluss des Decode-Cache sowie der Instruktionsvorhersage analysieren zu können, wurde zunächst der Simulator ohne Cache kompiliert. Dabei wurde eine schlechte Performanz von 0,177 *Million Instructions per Second* (MIPS) erzielt. Durch die Aktivierung des Decode-Caches konnten 99,991% der dekodierten Instruktionen eingespart und die Performanz auf 16,7 MIPS erhöht werden. Durch die Instruktionsvorhersage konnten wiederum 99,2% der Cache-Zugriffe verhindert werden, wodurch die Simulationsgeschwindigkeit auf 29,5 MIPS gesteigert werden konnte. Wenn man den Overhead für die Instruktionsvorhersage ignorieren (dieser beläuft sich auf 2 Speicherzugriffe und einen Vergleich) kann man ein lineares Gleichungssystem aufstellen und somit die Kosten für die einzelnen Komponenten des Simulators aufschlüsseln. Diese sind in Abbildung 7.1 aufgeführt.

Die bisherigen Ergebnisse waren für eine rein verhaltensorientierte Simulation. Durch die Aktivierung der Zyklenapproximation müssen während der Simulation zusätzliche Berechnungen durchgeführt werden, die die Performanz reduzieren. Die 29,5 MIPS verringern sich je nach Zyklenmodell unterschiedlich. Bei ILP konnte immer noch 18,3 MIPS, bei AIE 18,9 MIPS und für DOE 15,3 MIPS erzielt werden. Zusätzlich wurde der Anteil des Speichermodells auf die Zyklenapproximation untersucht. Daher stellte sich heraus, dass das Speichermodell nur 9,5 ns benötigt und somit nur einen vergleichsweise kleinen Anteil an der Zyklenapproximation hat, obwohl in unserem Test jede vierte Instruktion auf den Speicher zugegriffen hat.

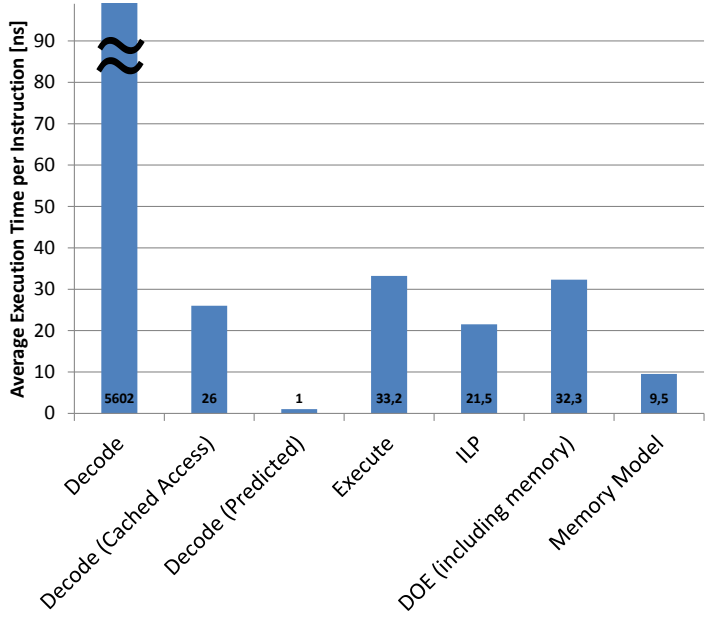


Abbildung 7.1: Performanz der einzelnen Komponenten des Core-Simulators (Veröffentlicht [132])

7.1.2.2 Genauigkeit

Neben der Performanz ist die Genauigkeit ein sehr wichtiges Kriterium für den Simulator, weil dies die Güte der gesamten Ergebnisse in dieser Arbeit beeinflusst. Das DOE-Zyklenmodell im Core-Simulator approximiert die Zyklen der Kahrisma-Architektur so genau wie möglich. Daher wurden die Zyklen des Core-Simulators mit den Ergebnissen der RTL-Hardwaresimulation verglichen. Da der Core-Simulation keine Sprungvorhersage approximieren kann, wurde in der Hardware eine perfekte Sprungvorhersage angenommen. Bei dem DOE-Zyklenmodell stehen insbesondere die Laufzeiteffekte, die durch ein Driften der Slots gegeneinander resultieren, im Vordergrund. Daher wurde die Anwendung mit der höchsten Parallelität auf Befehlsebene, die DCT, als Testanwendung ausgewählt.

Tabelle 7.2 zeigt die benötigten Zyklen für die RSIW-ISA für eine unterschiedliche Anzahl an Issue-Slots. Dabei konnte das DOE-Zyklenmodell mit enthaltener Speicherap-

Konf.	Zyklen der RTL-Simulation	Zyklen der Approximation	Fehler
RSIW1	21768	22062	1.4%
RSIW2	14111	13922	1.4%
RSIW4	9774	9878	1.1%
RSIW8	7774	7992	2.8%

Tabelle 7.2: Genauigkeit des DOE-Zyklenmodells im Core-Simulator (Veröffentlicht [132])

proximation die Zyklen mit einem Fehler von bis zu 2,8% vorhersagen. Dabei benötigte die RTL-Simulation pro Instruktion im Durchschnitt 8 ms. Somit ist der zyklenapproximative Core-Simulation ungefähr 100000 Mal schneller als die RTL-Simulation schafft es aber nahezu die gleiche Zyklenanzahl zu berechnen.

7.1.3 Benutzer-Retargierbarkeit des Softwareframeworks

Eine wichtige Anforderung an das Softwareframework war die Unterstützung der Designzeit-Flexibilität der Hardware (siehe Kapitel 4.2), die sich aus den Zielen ergeben haben (siehe Kapitel 3.2). Dadurch wird es möglich, zur Designzeit bereits einen Tradeoff zwischen Flächenbedarf der Hardware und Performanz der Software erzielen zu können. Hierbei ist es wichtig, dass das Softwareframework die Designparameter der Hardware sowie sämtliche daraus folgende Änderungen unterstützt, z.B. die Binärkodierungen der Operationen. Daher wurde das gesamte Softwareframework mittels einer Architekturbeschreibungssprache (siehe Kapitel 5) benutzer-retargierbar gehalten.

In diesem Abschnitt wird nun die Benutzer-Retargierbarkeit des Softwareframeworks zur Realisierung einer *Entwurfsraumexploration* (DSE, engl. *Design-Space Exploration*) eingesetzt:

7.1.3.1 Auswirkung der Anzahl der Register auf die Performanz

Zur Demonstration der Benutzer-Retargierbarkeit und Flexibilität des Softwareframeworks wurde die Registeranzahl als Designparameter der Kahrisma-ISA variiert. Es wurde die RISC-Kahrisma-ISA (also RSIW1) als einfachste Ausprägung als Grundlage genommen. Die Registeranzahl hat massive Auswirkungen auf die Befehlssatzarchitektur. So ist die Kodierung einer Operation von der Registeranzahl abhängig, weil es die Breite der Felder für die Operanden bestimmt. Weiterhin ist das Application Binary Interface abhängig von der Registeranzahl, weil sich die Nummern des Stack- und Frame-Pointers ändern sowie die Anzahl an Register angepasst wird, die von einer Funktion nicht verändert werden dürfen.

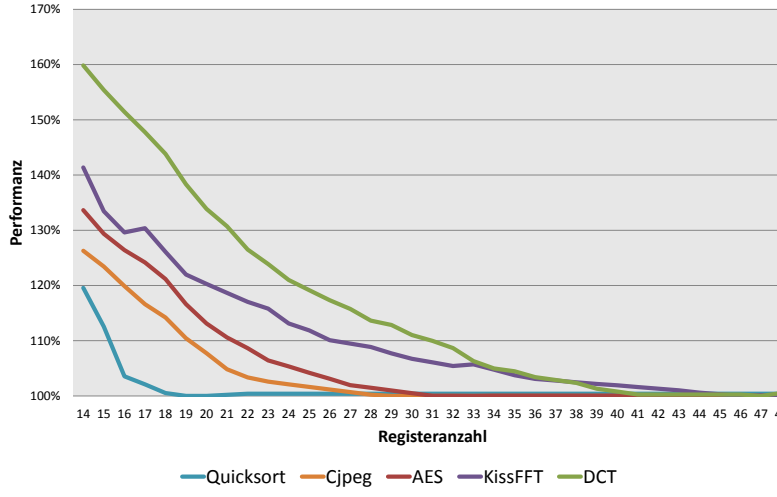


Abbildung 7.2: Einfluss der Registeranzahl auf die Performanz einer RISC-Architektur
(Veröffentlicht [131])

Basierend auf der ADL-Beschreibung wurde das Softwareframework retargiert. Für jede untersuchte Registeranzahl wurde mittels PHP-Skripte eine separate ADL-Beschreibung erzeugt und daraus jeweils unterschiedliche Varianten des LLVM-Compilers, Assemblers und Core-Simulators mit Hilfe des CoreGen-Werkzeugs generiert. Die verschiedenen Versionen des Softwareframeworks wurden dann mit einer Menge von Applikationen aus unterschiedlichen Anwendungsdomänen evaluiert.

Für jede Version des Softwareframeworks wurde die Menge an Anwendungen kompiliert und simuliert. Abbildung 7.2 zeigt den Performanzeinbruch dieser Anwendungen, wenn die Registeranzahl schrittweise von 48 auf 14 (X-Achse) reduziert wird. Die Y-Achse zeigt die relative Anzahl an ausgeführten Instruktionen in Prozent an. Das beste (kleinste) Ergebnis einer Anwendung (typischerweise mit 48 Registern) wurde dabei auf 100% normiert und jeder höhere Wert gibt den Overhead in Prozent an, der durch die Registerreduktion verursacht wird.

Wie der Abbildung entnommen werden kann, ist der Performanzeinbruch stark von der jeweiligen Anwendung abhängig. So ist die kontrollflussdominate Quicksort-Applikation nur in der Lage, maximal 19 Register effizient auszulasten, während die datenflussdominante FFT von bis zu 46 Registern profitieren kann. Allgemein scheinen datenflussdominante Anwendungen eine größere Anzahl von Registern im LLVM ef-

fizient auslasten zu können.

7.1.3.2 Geschwindigkeit der Retargierung

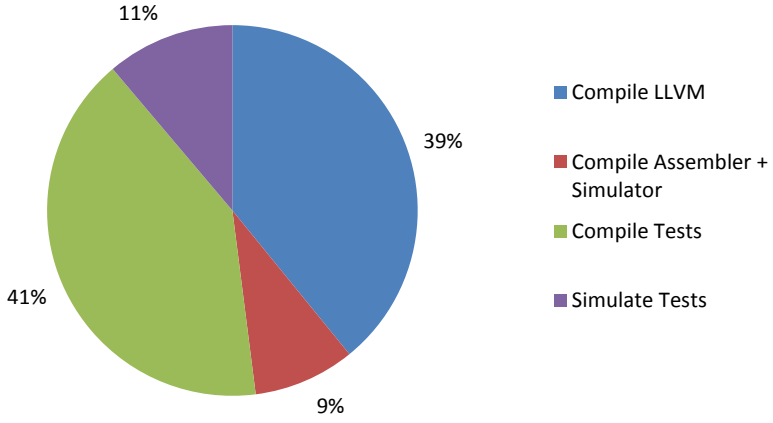


Abbildung 7.3: Prozentuale Zeitverteilung zum Durchführungen der Retargierung während der Entwurfsraumexploration
(Veröffentlicht [131])

Bei der Durchführung einer Entwurfsraumexploration ist die Zeit, die für die Evaluation eines Entwurfs aufgebracht werden muss, ein wichtiger Faktor. Auf einem 2x4-kerne Intel-Xeon-X5355-Prozessor mit 2,66 GHz wurden im Durchschnitt 277 Sekunden (03:47) für die Kompilierung der veränderten Teile des Softwareframeworks (Compiler, Assembler, Simulator) sowie zur Kompilierung und Simulation der Testapplikationen benötigt. In Abbildung 7.3 sind die einzelnen Schritte für die Evaluierung eines Entwurfs aufgeführt. In diesem Fall werden 109 Sekunden (48%) für die Retargierung des Softwareframeworks benötigt. Die größte Anwendung (nach der Anzahl der Quellcodezeilen) ist in diesem Fall der JPEG-Encoder, der alleine zum Kompilieren bereits 66 Sekunden (29%) benötigt. Die Simulation fällt dank der hohen Simulationsgeschwindigkeit von über 10 MIPS relativ kurz aus.

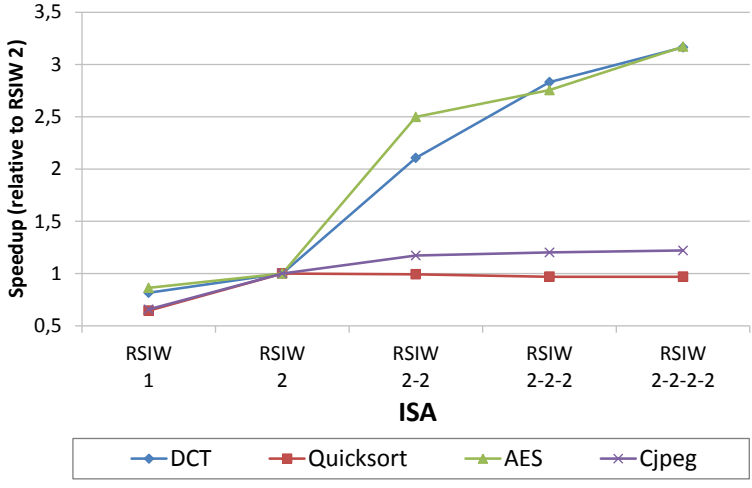


Abbildung 7.4: Beschleunigung durch Skalierung der Kahrisma-Ressourcen (Veröffentlicht [133])

7.1.4 Statisches mixed-ISA Programmiermodell

7.1.4.1 Beschleunigung durch Skalierung der Kahrisma-Ressourcen

Abbildung 7.4 zeigt die Performanz für die verschiedenen Kahrisma-Konfigurationen pro Anwendung. Dabei wird die Beschleunigung durch die Skalierung der Ressourcen (EDPEs) auf der X-Achse angezeigt. Die Performanz einer Anwendung auf der Y-Achse ist relativ zur RSIW2-Konfiguration gegeben, die jeweils auf 1 normiert wurde.

Aus den Ergebnissen lässt sich ablesen, dass die Performanzsteigerung durch zusätzliche parallele Ressourcen stark von der Anwendung abhängt. Die DCT und AES bieten viel nutzbare Parallelität auf Befehlsebenen an und dadurch werden diese Anwendungen um bis zu Faktor 3,2 beschleunigt, wenn vier EDPEs verwendet werden. Für zwei EDPEs ist sogar eine superlinearer Speedup von 2,1 (DCT) bzw. 2,5 (AES) ersichtlich. Die Superlinearität ist auf die zusätzlichen Register der zweiten EDPE zurückzuführen, wodurch weniger Variablen auf dem Stack abgelegt werden müssen. Somit können die DCT und AES effizient die RSIW2222-Konfiguration ausnutzen.

Im Gegensatz dazu hat die kontrollflussdominate Quicksort-Anwendung sogar eine negative Beschleunigung. Bei Quicksort ist der Overhead durch die Inter-Clus-

ter-Kommunikation höher als die Beschleunigung durch mehr parallele Operationen. Daraus resultiert, dass RSIW2 ausreichend für die Quicksort-Anwendung ist und jede weitere Ressourcen die Ausführung ineffizient gestalten.

Zum Vergleich mit RISC-Prozessoren wurde noch die RSIW1-Konfiguration als zusätzliche Option aufgeführt. Diese könnte zum Einsparen von Energie relevant sein, weil dann weniger Ressourcen innerhalb der Architektur verwendet werden. Überraschenderweise profitieren kontrollflussdominate Anwendungen stärker von einem zweiten Issue-Slot als datenflussdominante. Es hat den Anschein, dass für AES und DCT die Anzahl an verfügbaren Registern der limitierende Faktor in diesem Fall darstellt.

7.1.4.2 Performanzeinbußen aufgrund geclusterter Registerspeicher

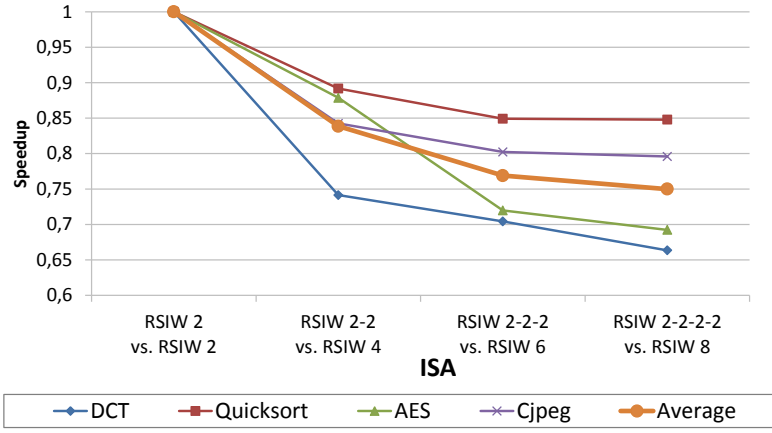


Abbildung 7.5: Geclusterten im Vergleich zu ungeclusterten Registerspeicher (Veröffentlicht [133])

Abbildung 7.5 zeigt die Performanzeinbußen, die durch den geclusterten Registerspeicher verursacht werden, der für die Realisierung der Flexibilität und der Rekonfigurierbarkeit der Hardware benötigt wird. Dazu wurden die geclusterten mit den theoretischen ungeclusterten RSIW-Konfigurationen verglichen. Die ungeclusterten Konfigurationen haben jeweils die gleiche Anzahl an Register und Issue-Slots. Z.B. haben die RSIW2222- und RSIW8-Konfigurationen jeweils 128 Register und 8 Issue-Slots. RSIW2 hat von vornherein nur ein Cluster. Deswegen ist die „geclusterte“ und

ungeclusterte Variante identisch und es wurde der Speedupfaktor 1,0 in der Abbildung verwendet.

DCT und AES haben eine hohe Auslastung sämtlicher Cluster und benötigen deswegen viel Inter-Cluster-Kommunikation. Dadurch sind ihre Performanzeinbußen große im Vergleich zu Quicksort und Cjpeg. Quicksort z.B. führt die meisten Operationen in der ersten EDPE aus, wodurch signifikant weniger ICC benötigt wird.

Im Durchschnitt lässt sich eine Performanzeinbußen durch Clustering von 15% bei 2 EDPEs und von 25% bei 4 EDPEs beobachten. Dies bestätigt die Faustregel von HP [117], die besagt, dass eine Aufsplittung des Registerspeichers in zwei bzw. vier Cluster in Performanzeinbußen von 15-20% bzw. 25-30% bei VLIW-Prozessoren resultiert. Diese Faustregel ist anscheinend auch für Prozessoren mit Dynamic Operation Execution Ausführungsmodell (siehe Abschnitt 3.5.2.6) gültig. Allerdings werden hier nur die Auswirkungen auf die Ausführungszyklen betrachtet. Eine positive Wirkung des Clusterings durch eine Reduktion des kritischen Pfades und somit eine Erhöhung der Taktfrequenz bleibt unberücksichtigt.

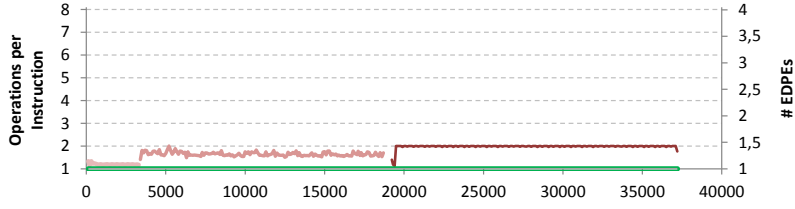
7.1.5 Dynamisches mixed-ISA Programmiermodell

Zur Demonstration und Evaluation des dynamischen mixed-ISA Programmiermodells des Softwareframeworks wurde eine Beispielanwendung entwickelt, die einen Quicksort und eine DCT durchführen muss. Diese Anwendung wurde in drei Phasen unterteilt: Initialisierung, Quicksort und DCT. Quicksort und DCT führen den gleichnamigen Algorithmus aus, während Initialisierung die Eingangsdaten für beide generiert.

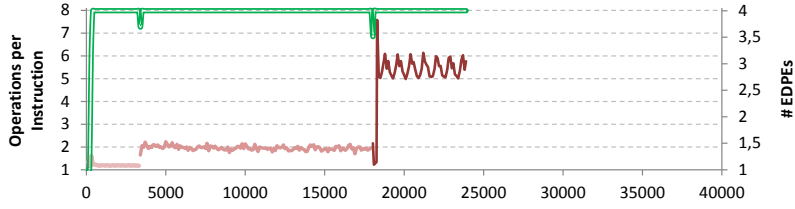
Mittels der mixed-ISA Erweiterung der C/C++-Programmiersprache wurde die Anwendung so entwickelt, dass man über die Kommandozeile steuern kann, welche Phase mit welcher ISA bzw. Konfiguration ausgeführt werden soll. Jede Phase wird durch einen Funktionsaufruf gestartet. Die Funktion jeder Phase hat den STAY-Modus. Zusätzlich gibt es noch für jede Phase eine Switch-Funktion, die den MUST-Modus hat und als Wrapper für die eigentliche Funktion der Phase dient. Dadurch kann in der main-Funktion, abhängig von einem Kommandozeilenparameter, jede Phase entweder direkt oder mittels der Switch-Funktion aufgerufen werden. Nur wenn die Switch-Funktion verwendet wird, ändert sich die ISA bzw. Konfiguration. Ansonsten bleibt sie gleich.

Die main-Funktion wird immer in RSIW2 aufgerufen. Abhängig von der Kommandozeile werden die Phasen wahlweise in RSIW2 oder RSIW2222 ausgeführt. Nach der Ausführung einer Phase wird immer in die RSIW2-Konfiguration zurückgekehrt, bevor die nächste Phase anfängt. Auf diese Weise wurde eine Anwendung im dynamischen mixed-ISA Programmiermodell entwickelt, die eine dynamische Steuerung der Rekonfiguration erlaubt.

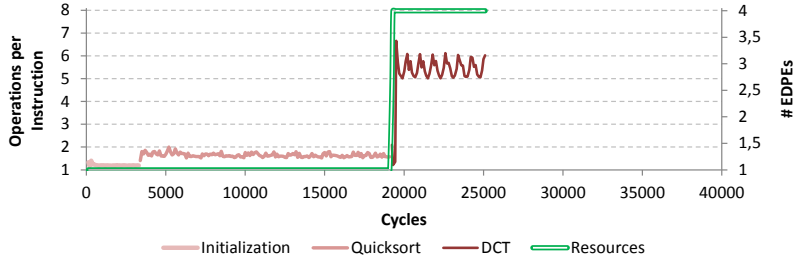
Abbildung 7.6 zeigt die Operationen pro Instruktion sowie die Anzahl an EDPEs der



(a) Statische mixed-ISA Ausführung: RSIW2



(b) Statische mixed-ISA Ausführung: RSIW2222



(c) Dynamische mixed-ISA Ausführung: RSIW2 (Initialisierung und Quicksort) und RSIW2222 (DCT)

Abbildung 7.6: Drei mixed-ISA Szenarien zur Ausführung einer synthetischen Anwendung im dynamischen mixed-ISA Modell
(Veröffentlicht [133])

Anwendung für drei Ausführungsszenarien (Abbildung 7.6(a) - 7.6(c)) über die Zeit. Die Zeit ist auf der X-Achse in Zyklen angegeben und die Werte von jeweils 100 Zyklen wurden gemittelt. Die linke Y-Achse zeigt die Operationen pro Instruktion als einfache rote Linie. Die jeweils aktuelle Phase wird durch die Helligkeit der Linie angegeben. Die Operationen pro Instruktion repräsentieren, wie viel Parallelität der Compiler jeweils in den Algorithmen über die Zeit extrahieren konnte. Die rechte Y-Achse zeigt den Ressourcenverbrauch in EDPEs als doppelte grüne Linie.

Es sind drei Ausführungsszenearien abgebildet. Im ersten Szenario werden alle Phasen in RSIW2 ausgeführt, im zweiten alle in RSIW2222 und im dritten wird nur die DCT in RSIW2222 ausgeführt. Das erste Szenario ist wesentlich langsamer (35%) als das zweite Szenario, benötigt allerdings nur eine anstatt vier EDPEs. Wie im Vergleich zwischen Szenario 1 und 2 ersichtlich wird, kann nur die DCT vier EDPEs effizient auslasten. Für Initialisierung und Quicksort sind die zusätzlichen Ressourcen nahezu nutzlos. Daher wurde im dritten Szenario eine effiziente Ressourcenauslastung durch die Rekonfiguration des dynamischen Programmiermodells angestrebt und somit nur die DCT in RSIW2222 mit vier EDPEs ausgeführt.

Auf diese Art und Weise ist das dritte Szenario 4,9% langsamer als Szenario zwei. Dafür benötigt es durchschnittlich 57% weniger Ressourcen (1,7 anstatt 4,0) als das zweite Szenario. Die nicht verwendeten Ressourcen können entweder zum Energiesparen abgeschaltet oder können für die Beschleunigung von parallel laufenden Anwendungen verwendet werden. Dadurch ermöglicht die Flexibilität der Hardware in diesem Fall einen guten Tradeoff zwischen Performanz und Ressourcen- bzw. Energieverbrauch zu erzielen.

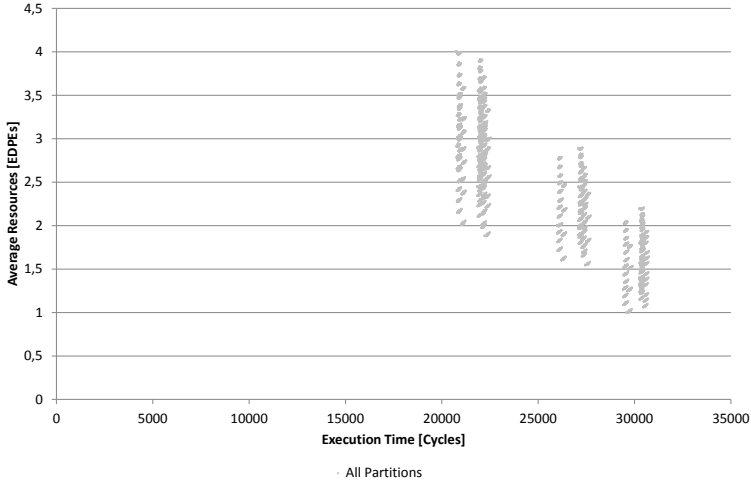
7.1.6 Automatische mixed-ISA Programmiermodell

7.1.6.1 Synthetische DCT/Quicksort-Beispielanwendung

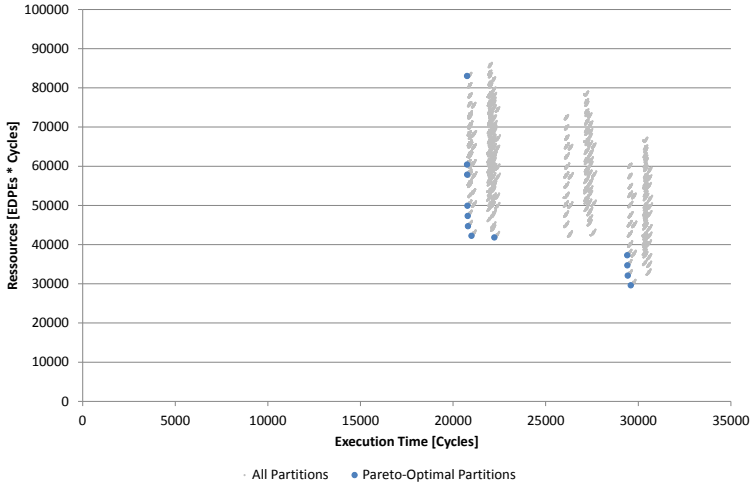
Zur Evaluation des ISA-Partitionierers und des automatischen mixed-ISA Programmiermodells wurde zunächst die DCT/Quicksort-Beispielanwendung aus dem dynamischen mixed-ISA Programmiermodell untersucht. Dadurch kann das automatische direkt mit dem dynamischen Programmiermodell verglichen werden.

Zur Generierung der Ergebnisse wurde zunächst die Anwendung im statischen mixed-ISA Programmiermodell für RSIW2, RSIW22, RSIW222 und RSIW2222 kompiliert und zur Generierung der Profiling-Ergebnisse simuliert. Der Aufbau der Profiling-Informationen für diese Anwendung wurden beispielhaft im Realisierung-Kapitel in Abbildung 6.12 gezeigt. Basierend auf den Profiling-Informationen wurde eine Menge von Lösungen für die Partitionierung berechnet. Dabei kam der Priority-Enumeration-Partitionierungsalgorithmus zum Einsatz.

Abbildung 7.7(a) zeigt sämtliche Partitionen als graue Punktewolke, die der Partitionierungsalgorithmus für die Anwendung untersucht hat. Auf der X-Achse ist die Aus-



(a) Durchschnittlicher Ressourcenverbrauch



(b) Absoluter Ressourcenverbrauch

Abbildung 7.7: Untersuchter Lösungsraum und Paretomenge der DCT/Quicksort-Beispielanwendung

führungszeit in Zyklen gegeben und auf der Y-Achse wird der durchschnittliche Ressourcenverbrauch in EDPEs angezeigt. Da der Priority-Enumeration-Partitionierungsalgorithmus eine Heuristik ist und nicht sämtliche Partitionen untersucht, entspricht dies einer Untermenge des gesamten Lösungsraums. Da bei dem Algorithmus Funktionen mit einer hohen Ausführungszeit bevorzugt werden, ist davon auszugehen, dass die Untermenge des Lösungsraums repräsentativ ist.

Allerdings bietet der durchschnittliche Ressourcenverbrauch nur eine bedingte Aussagekraft. Bedeutend ist der Ressourcenverbrauch über die Zeit. Dieser ist in Abbildung 7.7(b) dargestellt, in der der durchschnittliche Ressourcenverbrauch mit der Ausführungszeit multipliziert wurde. Innerhalb des Lösungsraums wurden zusätzlich sämtliche pareto-optimalen Partitionen als größere blaue Punkte dargestellt. Diese repräsentieren sämtliche relevanten Lösungen aus dem Lösungsraum. Es lässt sich anhand der Pareto-Menge erkennen, dass sich mittels der dynamischen Rekonfigurierbarkeit der Kahrisma-Architektur und dem dynamischen mixed-ISA Programmiermodell ein flexiblen Tradeoff zwischen Performanz und Ressourcenverbrauch einer Anwendung erzielt werden kann.

In der Pareto-Menge in Abbildung 7.7(b) sind die beiden seitlichen Konfigurationen mit dem höchsten bzw. niedrigsten Ressourcenverbrauch mit 4 bzw. 1 EDPE jeweils eine Partition ohne Rekonfiguration. Dazwischen lässt sich ein Tradeoff zwischen Performanz und Ressourcenverbrauch auswählen. Auffallend ist, dass man bei dieser Anwendung mit einer minimalen Reduktion der Performanz viele Ressourcen einsparen kann. So ist eine Halbierung des durchschnittlichem Ressourcenverbrauchs mit minimalem Performanzverlust möglich.

7.1.6.2 Genauigkeit des Performanzmodells im ISA-Partitionierer

Ein wichtiger Aspekt zur Bewertung des ISA-Partitionierers ist die Genauigkeit des eingesetzten Performanzmodells, das auf Profiling-Informationen beruht. Zur Überprüfung der Genauigkeit der Lösungen wurden dabei einige Lösungen mit der berechneten Partitionierung kompiliert, simuliert und das Simulationsergebnis mit dem Ergebnis des Performanzmodells verglichen. Der Kompilier- und Simulationsvorgang ist wesentlich Aufwändiger als die Modellberechnung. Dies war auch überhaupt erst die Motivation ein approximatives Performanzmodells zu verwenden, mit dem dann allerdings wesentlich mehr Partitionen in gleicher Zeit untersucht werden können. Zur Überprüfung konnte daher nicht der gesamte untersuchte Lösungsraum des ISA-Partitionierers verwendet werden. Stattdessen wurde die Pareto-Menge auf Genauigkeit untersucht.

Abbildung 7.1.6.2 zeigt die bereits bekannte Pareto-Menge der DCT/Quicksort-Beispielanwendung als blaue Punkte. Zusätzlich wurden die Ergebnisse vom Compiler und Simulator als rote Punkte eingezeichnet. Wie gut zu erkennen ist, liegen die roten Punkte ziemlich nah bei den blauen Punkten. Der maximale Fehler für diese Ergebnisse liegt bei 3%. Weiterhin ließ sich feststellen, dass der Fehler bei Partitionen mit einer

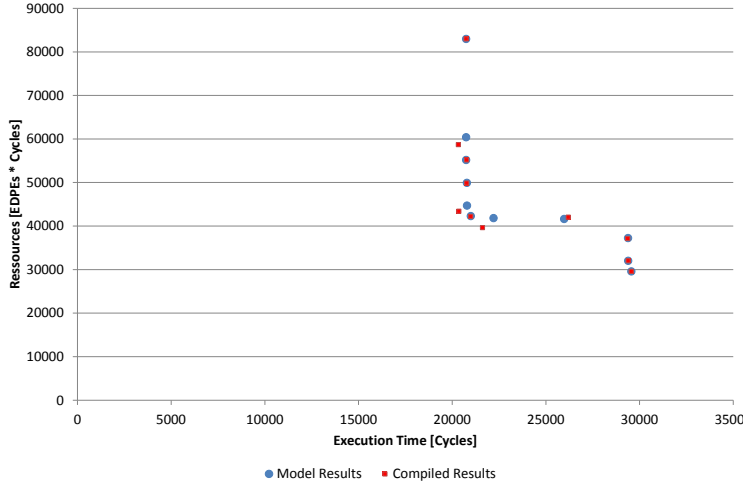


Abbildung 7.8: Genauigkeit des Performanzmodells im ISA-Partitionierer bei der DCT/Quicksort-Beispielanwendung

höheren Anzahl von ISA-Wechseln (wie z.B. in der Mitte der Grafik) zunimmt. Dies ist darin zu begründen, dass bei einem ISA-Wechsel sämtliche Register auf den Stack geschrieben werden müssen und der Performanzverlust hierbei von der Registerbenutzung in der jeweiligen Funktion abhängt. Dies konnte im Modell nicht exakt vorhergesagt werden.

Durch den niedrigen Fehler bei der Abschätzung der Performanz im ISA-Partitionierer lässt sich schlussfolgern, dass das automatische mixed-ISA Programmiermodell basierend auf den Profiling-Informationen in Kombination mit dem Priority-Enumeration-Partitionierungsalgorithmus funktioniert und zur mixed-ISA Optimierung einer Anwendung herangezogen werden kann.

7.2 Mixed-ISA Multi-Prozessor-Anwendungen

Bei den bisherigen Ergebnissen wurde die Rekonfigurierbarkeit und mixed-ISA Programmierbarkeit der flexiblen Prozessorarchitektur nur für Ein-Prozessor-Anwendungen betrachtet. In diesem Abschnitt wird nur die Rekonfigurierbarkeit der Architektur für Multi-Prozessor-Anwendungen betrachtet. Dabei wurde als Grundlage der System-Simulator verwendet, der eine systemweite Simulation der Kahrisma-Ar-

chitektur erlaubt.

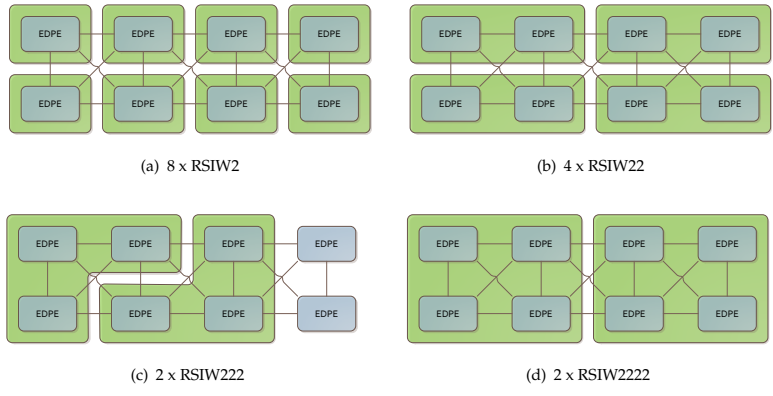


Abbildung 7.9: Verschiedene homogene Konfigurationsmöglichkeiten für eine Multiprozessoranwendung auf 8 EDPEs

Zur Evaluation wurde für eine Multi-Prozessor-Anwendung eine Entwurfsraumexploration durchgeführt, bei der die Konfiguration, die Problemgröße sowie die Anzahl an EDPE-Ressourcen verändert wurden. Dabei kamen nur homogene Konfigurationen zum Einsatz, so dass jede virtuelle Prozessorinstanz einer Anwendung jeweils die gleiche Konfiguration hat. Durch die Konfiguration und Anzahl an EDPE-Ressourcen ergeben sich automatisch die Anzahl an virtuellen Prozessorinstanzen. Dies ist am Beispiel von acht EDPE-Ressourcen in Abbildung 7.2 zu sehen. Dabei wurde für jede mögliche Konfiguration das Mapping von virtuellen Prozessorinstanzen auf EDPE-Ressourcen dargestellt. So kann man bei 8 EDPE-Ressourcen wahlweise 8 RSIW2-Prozesse, 4 RSIW22-Prozesse, 2 RSIW222-Prozesse oder 2 RSIW2222-Prozesse konfigurieren. Da 8 nicht durch 3 teilbar ist, bleiben bei der RSIW222-Konfiguration 2 EDPE-Ressourcen ungenutzt. Tabelle 7.3 listet alle möglichen Konfigurationen bis zu 16 EDPE-Ressourcen auf. Ein Bindestrich bedeutet, dass die Konfiguration bei der EDPE-Ressourcenanzahl nicht möglich ist. Ein Anführungszeichen in der Tabelle bedeutet, dass die Konfiguration mit den Ressourcen ineffizient ist, weil Ressourcen verschwendet werden. Wenn man in diesem Fall in der Tabelle bis zur nächsten Zahl nach oben geht, kommt man auf eine effiziente Konfiguration.

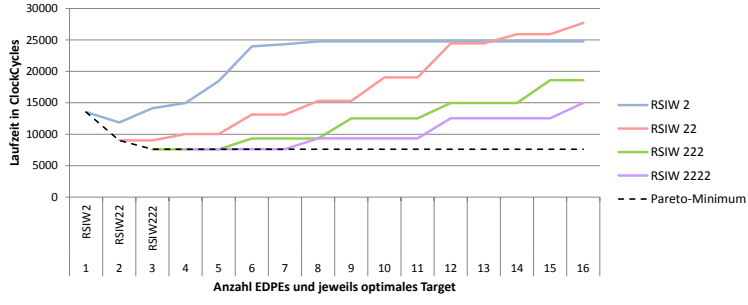
Als Anwendung für die Entwurfsraumexploration wurde eine parallele MPI-Version der Matrixmultiplikation verwendet. Zur Erhöhung der Cache-Effizienz wurde eine modifizierte Matrixmultiplikation implementiert, bei der die rechte Eingangsmatrix vor der Berechnung transponiert wird, so dass möglichst viele Speicherzugriffe line-

EDPE-Ressourcen	RSIW2	RSIW22	RSIW222	RSIW2222
1	1	-	-	-
2	2	1	-	-
3	3	"	1	-
4	4	2	"	1
5	5	"	"	"
6	6	3	2	"
7	7	"	"	"
8	8	4	"	2
9	9	"	3	"
10	10	5	"	"
11	11	"	"	"
12	12	6	4	3
13	13	"	"	"
14	14	7	"	"
15	15	"	5	"
16	16	8	"	4

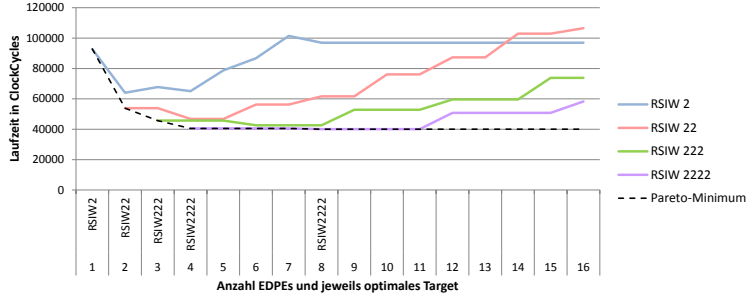
Tabelle 7.3: Anzahl an homogenen Konfigurationen in Abhängigkeit der EDPE-Ressourcenanzahl

ar hintereinander liegen. Zur Kompilierung der MPI-Anwendung wurde eine MPI-Bibliothek für die Kahrisma-Architektur entwickelt, die eine Untermenge des MPI-Standards unterstützt. Die Matrixmultiplikation wurde dabei so geschrieben, dass sowohl die Matrixgröße als auch die Anzahl an Prozessen variable ist. Durch die Verwendung des statischen mixed-ISA Programmiermodells kann beim Starten der Anwendung die Konfiguration bzw. ISA der Anwendung gewählt werden.

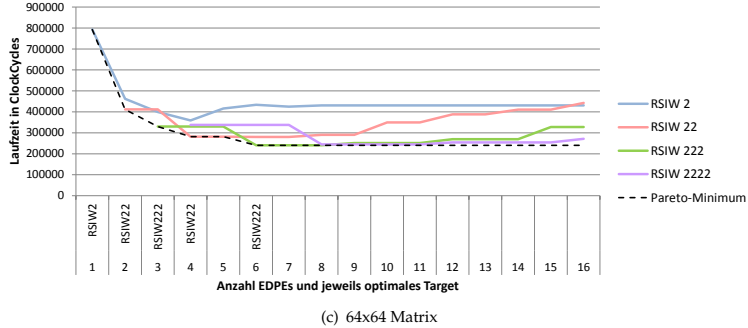
Abbildung 7.10 und 7.11 zeigen die Performanz der Matrixmultiplikation für sämtliche Konfigurationen in Abhängigkeit der Anzahl an verwendeten Ressourcen für Matrixgrößen zwischen 16x16 und 512x512. Die Y-Achse zeigt die Laufzeit der Anwendung in Zyklen an. Auf der X-Achse ist die Anzahl an Ressourcen dargestellt. Innerhalb des Diagramms wird die Laufzeit für jede der vier RSIW-Konfiguration als Linie in unterschiedlichen Farben dargestellt. Aus der Anzahl an EDPE-Ressourcen und der Konfiguration ergibt sich automatisch die Anzahl an Prozessen mit Hilfe von Tabelle 7.3. In der X-Achse ist zusätzliche für jede Anzahl an EDPE-Ressourcen die performanteste Konfiguration eingetragen. Wenn das Feld leer ist, können die EDPE-Ressourcen nicht effizient ausgenutzt werden. In diesem Fall reicht für die gleiche Performanz eine niedrigere Anzahl an EDPEs aus und man findet auf X-Achse durch Reduktion der Ressourcen eine effizientere Konfiguration. Die jeweils performanteste Konfiguration ist ebenfalls im Diagramm durch das Pareto-Minimum als gestrichelte Linie eingezeichnet. Diese zeigt jeweils die beste Performanz für die maximale Anzahl an EDPE-Ressourcen, wenn die Konfiguration flexibel gewählt werden kann.



(a) 16x16 Matrix

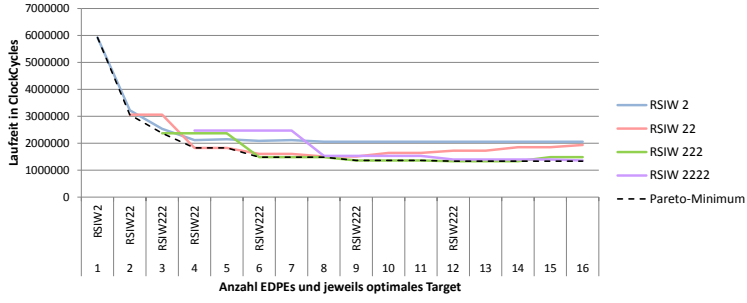


(b) 32x32 Matrix

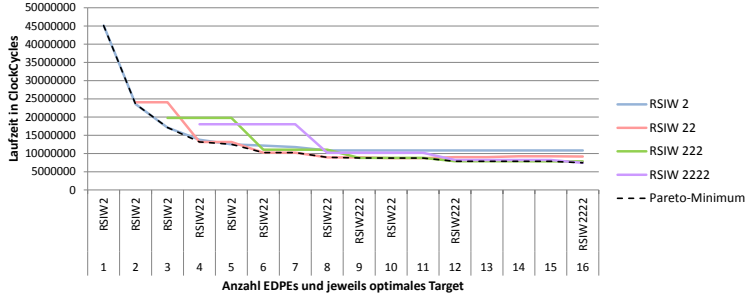


(c) 64x64 Matrix

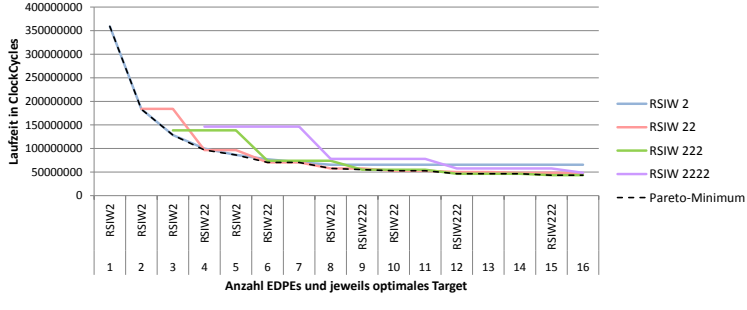
Abbildung 7.10: Performanz der Matrixmultiplikation in Abhängigkeit der Ressourcen und Konfiguration (1)
(Quelle [125])



(a) 128x128 Matrix



(b) 256x256 Matrix



(c) 512x512 Matrix

Abbildung 7.11: Performanz der Matrixmultiplikation in Abhängigkeit der Ressourcen und Konfiguration (2)
(Quelle [125])

Die MPI-Version der Matrixmultiplikation bietet sowohl ausreichend Parallelität auf Befehls- als auch Threadebene. Daher ist interessant zu sehen, unter welchen Voraussetzungen es sich lohnt, mehr ILP oder TLP auszunutzen. Im Folgenden wird abhängig von der Ressourcenanzahl beschrieben, welche Konfiguration am performantesten ist:

1 EDPE Bei nur einer EDPE-Ressource gibt es keine Wahlfreiheit, daher kann immer nur RSIW2 ausgeführt werden.

2 EDPEs Bei zwei EDPEs kann schon zwischen RSIW2 und RSIW22 gewählt werden. Ab einer Matrixgröße von 128x128 ist der Overhead von TLP und ILP nahezu gleich.

3 EDPEs Bei drei EDPEs kann man zwischen RSIW2 und RSIW222 wählen. Hier ist ab 256x256 der Overhead von TLP niedriger als von ILP.

4 EDPEs Bei vier EDPEs hat man drei Wahlmöglichkeiten 1 x RSIW2222, 2 x RSIW22 oder 4 x RSIW2. Bei einer sehr kleinen Matrix von 16x16 lohnen sich 4 EDPEs überhaupt nicht. Bei 32x32 ist RSIW2222 am performantesten. Danach ist eine Mischung zwischen ILP und TLP am besten, wobei es sich ab 256x256 die Waage gibt.

5 EDPEs Bei fünf EDPEs ist nur eine Konfiguration mit 5 Threads möglich. Dies ist wieder ab 256x256 sinnvoll.

6 EDPEs Bei sechs EDPEs kann man entweder zwei oder drei Prozesse wählen. Hier ist ebenfalls ab 256x256 die 3-Prozess-Variante schneller.

ab 8 EDPEs Ab acht EDPEs profitieren nur noch sehr große Matrizen und es ist meistens eine Mischung aus TLP und ILP am schnellsten.

Allgemein lässt sich sagen, dass eine geeignete Multiprozessor-Konfiguration bei der Matrixmultiplikation abhängig von der Problemgröße ist. Umso größer die Problemgröße ist umso weniger fällt der Multiprozessor-Overhead ins Gewicht. Bei ILP hingegeben profitiert die Anwendung bereits ab einer vergleichsweise niedrigen Problemgröße. Ebenso ist die Konfiguration abhängig von der Anzahl an verfügbaren Ressourcen. So wird bei großen Matrizen und wenig Ressourcen zunächst nur TLP ausgenutzt während bei einer hohen Ressourcenanzahl eine Kombination aus ILP und TLP die besten Ergebnisse erzielt.

Im Vergleich zu statischen, nicht-rekonfigurierbaren MPSoCs bietet die Kahrisma-Architektur den Vorteil, dass sie sich dynamische auf die ILP/TLP-Charakteristiken einer Anwendung einstellen kann. Wie an den Ergebnissen gezeigt wurde, sind diese Charakteristika nicht nur von dem Algorithmus an sich sondern ebenfalls von der Problemgröße und der Anzahl an verfügbaren Ressourcen abhängig. Wenn ein statisches MPSoC jeweils nur eine Linie in den Diagrammen repräsentiert, gibt es bei jeder untersuchten Problemgröße mindestens einen Fall, in dem die Rekonfiguration der Architektur Performanzvorteile bringt. Dies verdeutlicht, dass die Entscheidung zwischen Komplexität von Kernen und Kernanzahl zur Designzeit nur für einen fest-

gelegten Algorithmus und Problemgrößen getroffen werden kann. Ansonsten sind statische, nicht-rekonfigurierbar MPSoCs zwangsläufig ineffizient.

7.3 Charakterisierung & Zusammenfassung

Es wird immer schwieriger die steigende Anzahl an Transistoren innerhalb eines Prozessors effizient durch Software auszunutzen. In der Vergangenheit war die Mikroarchitektur eines Prozessors für die automatisch Ausnutzung von *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) eines sequentiellen Befehlsstroms verantwortlich. Da diese Form der Parallelität nur bis zu einem gewissen Grad skaliert, wird bei heutigen MPSoCs noch zusätzliche *Parallelität auf Threadebene* (TLP, engl. *Thread-Level Parallelism*) ausgenutzt. Dabei entscheidet das Design eines MPSoCs für welche ILP/TLP-Charakteristika dieses optimiert ist. Allerdings sind die ILP/TLP-Charakteristika stark von der Anwendung bzw. eines Algorithmus innerhalb einer Anwendung abhängig. Daher haben aktuelle MPSoCs den Nachteil, dass sie nicht für eine breite Masse an Anwendungen optimiert werden können.

Daher wurde in Kapitel 3 die Kahrisma-Architektur, ein rekonfigurierbares MPSoC, vorgestellt, die sich dynamisch an wechselnde ILP/TLP-Anforderungen anpassen kann. Dabei können mittels einer neuartigen rekonfigurierbaren Prozessorpipeline sog. EDPEs zu virtuellen Prozessorinstanzen zusammengeschaltet werden. Die Anzahl an EDPEs pro virtueller Prozessorinstanz ist dabei variabel. Der verfügbare ILP pro virtueller Prozessorinstanz ist proportional zu der EDPE-Anzahl. Genauso ist der verfügbare TLP des MPSoCs abhängig von der Anzahl der virtuellen Prozessorinstanzen. Da die Anzahl an EDPEs beschränkt ist, kann man somit bei dieser Architektur deren ILP/TLP-Charakteristik partiell dynamisch zur Laufzeit umschalten. Durch diesen neuen Freiheitsgrad hat die Architektur signifikante Vorteile gegenüber den Stand der Technik, bei dem die ILP/TLP-Charakteristik statisch zur Designzeit festgelegt werden muss.

Die Flexibilität in der Mikroarchitektur konnte allerdings nicht transparent zur Software realisiert werden. Stattdessen benötigt die Architektur eine spezielle rekonfigurierbare RSIW-Befehlssatzarchitektur, die von klassischen Softwareframeworks nicht unterstützt wird. Um auch die Programmierbarkeit der Prozessorarchitektur gewährleisten zu können, wurde in dieser Arbeit folglich ein mixed-ISA Softwareframework realisiert, das mit einer variablen Befehlssatzarchitektur umgehen kann und somit die gleiche Flexibilität wie die Hardware zur Laufzeit aufweist. Neben der Flexibilität zur Laufzeit bietet die Hardware zusätzliche Flexibilität zur Designzeit an, die eine Entwurfsraumexplorationen ermöglicht. Die Designzeit-Flexibilität wurde im Softwareframework in dieser Arbeit berücksichtigt.

Zur Verwirklichung der Programmierbarkeit von Architekturen mit variablen Befehlssatzarchitekturen wurden zunächst drei Programmiermodelle definiert: das statische, dynamische und automatische mixed-ISA Programmiermodell. Beim statischen Pro-

grammiermodell hat man zunächst die Freiheit beim Start einer Anwendung jeweils eine geeignete Befehlssatzarchitektur auszuwählen, die dann für die Laufzeit der Anwendung gleich bleibt. Das dynamische und automatische mixed-ISA Programmiermodell geht einen Schritt weiter und erlaubt zusätzlich einen Wechsel der Befehlssatzarchitektur während der Laufzeit. Unter Auswahl eines geeigneten Programmiermodells kann ein Programmierer mixed-ISA Anwendung in C/C++ für Architekturen mit variablen Befehlssatzarchitekturen im allgemeinen entwickeln und für diese kompilieren. Im Fall der Kahrisma-Architektur kann sich eine Anwendung dadurch vor und/oder während der Laufzeit auf unterschiedliche ILP/TLP-Charakteristika anpassen.

Die mixed-ISA Programmiermodelle werden durch das Softwareframework umgesetzt, das auf einer mixed-ISA *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*) basiert. Diese ADL wird zum einen zur Unterstützung von variablen Befehlssatzarchitekturen im Framework benötigt. Zum anderen gewährleistet sie eine Erweiterbarkeit der Prozessorarchitektur. Für die rekonfigurierbare Befehlssatzarchitektur der Kahrisma-Architektur unterstützt die ADL insbesondere die gleichzeitige Beschreibung von variablen Befehlssatzarchitekturen, die dann jeweils einer Konfiguration der rekonfigurierbaren Befehlssatzarchitektur entsprechen. Die ADL wird dann zur Realisierung der Retargierbarkeit der verschiedenen Komponenten des Frameworks verwendet.

Das **statische mixed-ISA Programmiermodell** wurde innerhalb des Softwareframeworks durch einen *mixed-ISA Compiler* und *mixed-ISA Binärwerkzeuge* umgesetzt. Dazu wurde ein neues benutzer-retargierbares, mixed-ISA Compiler-Backend realisiert. Die Benutzer-Retargierbarkeit wird durch das *CoreGen-Werkzeug* umgesetzt, das mittels Metaprogrammierung Teile des Quellcodes des mixed-ISA Compilers und der Binärwerkzeuge generiert. In Abschnitt 7.1.3 wurde die Retargierbarkeit des mixed-ISA Softwareframeworks durch Modifikation der Registeranzahl innerhalb der Befehlssatzarchitektur untersucht. Dabei konnte innerhalb einer Entwurfsraumexploration gezeigt werden, dass die Sensitivität auf Änderung der Registeranzahl innerhalb einer Befehlssatzarchitektur sehr stark von der jeweiligen Anwendung abhängt.

Zur Unterstützung der rekonfigurierbaren RSIW-Befehlssatzarchitektur musste das Backend für die Codegenerierung von Clustered-VLIW-Befehlssatzarchitekturen erweitert werden, die in bestimmten Konfigurationen der RSIW-Befehlssatzarchitektur enthalten sind. Für das statische mixed-ISA Programmiermodell wurde das Backend so erweitert, dass es in einem Durchlauf sämtliche Funktion dupliziert und somit mehrfach, je ISA bzw. Konfiguration, kompiliert. Als Ausgabe erzeugt es eine mixed-ISA Assemblerdatei, die Informationen über jeweils verwendete ISA enthält. Die Binärwerkzeuge, der Assembler und Linker, wurden um die Möglichkeit der Verarbeitung von mixed-ISA Anwendungen erweitertet und generieren eine mixed-ISA ausführbare Datei. In Abschnitt 7.1.4 wurde das statische mixed-ISA Programmiermodell mit verschiedenen Anwendungen demonstriert. Dazu wurden die Anwendungen einmalig kompiliert und danach jeweils mit unterschiedlichen Konfigurationen simuliert. Dabei hat sich herausgestellt, dass der Verfügbare ILP sehr stark pro Anwendung va-

riiert. Durch eine Erhöhung der Ressourcen innerhalb einer Konfiguration hat z.B. der Quicksort-Algorithmus fast nicht profitiert während z.B. die DCT um Faktor 3,2 bei einer Vervierfachung der Ressourcen beschleunigt wurde. Damit konnte zum einen die grundlegende Programmierbarkeit der Kahrisma-Architektur gezeigt werden. Zum anderen ist bereits die Rekonfiguration der Architektur im statischen mixed-ISA Programmiermodell vorteilhaft, weil dadurch je nach Anwendung Faktor 4 an Ressourcen eingespart oder eine um bis zu Faktor 3,2 höhere Performanz erzielt werden kann.

Zur Realisierung des **dynamischen mixed-ISA Programmiermodells** wurde eine *mixed-ISA Erweiterung der C/C++-Programmiersprache* spezifiziert, die eine Steuerung der ISA in einer Anwendung zur Laufzeit mittels mixed-ISA Funktionsattributen ermöglicht. Das Frontend im Compiler verarbeitet die mixed-ISA Funktionsattribute und stellt sie dem Compiler-Backend innerhalb der LLVM-Zwischendarstellung zur Verfügung. Dieses dupliziert die Funktionen dann anhand der Funktionsattribute und fügt bei Bedarf Code zum Umschalten der ISA ein. In Abschnitt 7.1.5 wurde das dynamische mixed-ISA Programmiermodell anhand einer Beispielanwendung evaluiert. Dazu wurde eine Beispielanwendung entwickelt, die einen Quicksort- und eine DCT-Algorithmus durchführt. Mittels der mixed-ISA Erweiterung der C/C++-Programmiersprache wurde die Anwendung so entwickelt, dass man über die Kommandozeile steuern kann, welcher Teil der Anwendung zur Laufzeit mit welcher ISA bzw. Konfiguration ausgeführt werden soll. Dabei konnte gezeigt werden, dass ein Umschalten der ISA bzw. Konfiguration zur Laufzeit einer Anwendung im Vergleich zu statischen Architekturen oder dem statischen mixed-ISA Programmiermodell vorteilhaft ist. So konnten 57% der Ressourcen bei einem geringen Performanzverlust von 4,9% eingespart werden.

Für das **automatische mixed-ISA Programmiermodell** wurde eine *profile-guided Optimierung* der ISA in das Softwareframework integriert. Dabei wird eine Anwendung zunächst mit dem statischen mixed-ISA Programmiermodell kompiliert und ISA-abhängige, hierarchische Profiling-Information durch den Simulator erzeugt. Anhand dieser Informationen berechnet dann ein neues Werkzeug, der *ISA-Partitionierer*, eine ISA-Partitionierung. Der ISA-Partitionierer löst das NP-schwere Partitionierungsproblem mittels eines heuristischen Algorithmus unter Ausnutzung der Programmlokalität und einem mixed-ISA Performanzmodell. In Abschnitt 7.1.6 wurde das automatische mixed-ISA Programmiermodell anhand einer Beispielanwendung evaluiert. Dabei wurden die Performanz und der Ressourcenverbrauch aller untersuchten Partitionierungen basierend auf dem Performanzmodell im ISA-Partitionierer dargestellt. Die Pareto-Menge darauf liefert alle sinnvollen ISA-Partitionierungen. Es konnte gezeigt werden, dass der ISA-Partitionierer automatisch eine Menge von ISA-Partitionierungen generieren kann, aus diesen der Entwickler eine geeignete Partitionierung mit einem Tradeoff zwischen Ressourcen/Energie und Performanz auswählen kann.

In Abschnitt 7.2 wurde für eine Multi-Prozessor-Anwendung eine Entwurfsraumexploration durchgeführt, bei der die Konfiguration, die Problemgröße sowie die Anzahl paralleler Prozesse verändert wurde. Dabei kamen nur homogene Konfigurationen zum Einsatz, so dass jede virtuelle Prozessorinstanz einer Anwendung jeweils

die gleiche Konfiguration aufweist. Als Anwendung für die Entwurfsraumexploration wurde eine parallele MPI-Version der Matrixmultiplikation verwendet. Zur Kompilierung der MPI-Anwendung wurde eine MPI-Bibliothek für die Kahrisma-Architektur entwickelt, die eine Untermenge des MPI-Standards unterstützt. Durch die Verwendung des statischen mixed-ISA Programmiermodells kann pro Prozess die Konfiguration bzw. ISA der Anwendung gewählt werden und über die MPI-Bibliothek kann die Anzahl an Prozessen gesteuert werden. Es konnte gezeigt werden, dass die dynamische Ausnutzung von ILP und TLP selbst bei parallelen Anwendungen mit beiden Parallelitätsformen im Vergleich zu statischen, nicht-rekonfigurierbaren MPSoCs vorteilhaft ist. Dabei konnte gezeigt werden, dass eine effiziente Konfiguration nicht nur vom Algorithmus an sich, sondern auch von der Problemgröße sowie der Anzahl an verfügbaren Ressourcen abhängt. Dies verdeutlicht, dass die Entscheidung zwischen Komplexität von Kernen und Kernanzahl zur Designzeit nur für eine festgelegten Algorithmus und Problemgrößen getroffen werden kann. Ansonsten sind statische, nicht-rekonfigurierbare MPSoCs zwangsläufig ineffizient.

Das mixed-ISA Softwareframework wird durch einen **Core- und System-Simulator** komplettiert. Der Core-Simulator unterstützt die mixed-ISA Simulation einer virtuellen Prozessorinstanz, wie sie in der mixed-ISA Architekturbeschreibungssprache spezifiziert ist. Dabei kann sowohl die Start-ISA gewählt als auch die ISA zur Laufzeit gewechselt werden. Er integriert ein zyklenapproximatives Performanzmodell, das in der Lage ist, die Performanz der virtuellen Prozessorinstanzen zu approximieren ohne die Prozessorpipeline im Detail zu simulieren. Der System-Simulator verwendet mehrere Core-Simulatoren zur Simulation eines Gesamtsystems. In Abschnitt 7.1.2 wurde der Tradeoff des Core-Simulators zwischen Performanz und Genauigkeit evaluiert. Dabei ist die zyklenapproximative Simulation ungefähr 100000 Mal schneller als die RTL-Simulation aber kann mit einem geringen Fehler die Performanz der Kahrisma-Pipeline approximieren.

8 Ausblick

Das entwickelte Softwareframework bietet interessante Möglichkeiten weiterführender Untersuchungen und Verbesserungen.

8.1 Verwendung für andere Prozessorarchitekturen

Im Rahmen dieser Arbeit wurde das mixed-ISA Softwareframework zusammen mit der Kahrisma-Architektur zur dynamischen Ausnutzung von ILP/TLP-Charakteristika eingesetzt. Das Prozessorframework bietet allerdings noch weitergehende Möglichkeiten zur Programmierung von anderen Prozessorarchitekturen.

8.1.1 Prozessoren mit Codekompression

Es gibt Prozessoren mit variablen Befehlssatzarchitekturen, bei denen die Befehlssatzarchitektur zur Reduktion der Codegröße umgeschaltet werden kann. So bieten einige ARM-Prozessoren die Möglichkeit an, die ISA zwischen der allgemeinen 32 Bit ISA und der platzoptimierten 16 Bit Thumb ISA umzuschalten. Bei der Auswahl der ISA ist somit ein Tradeoff zwischen Performanz und Codegröße gegeben. Dieser Tradeoff kann durch das mixed-ISA Framework effizient ausgenutzt und die Ergebnisse dieser Arbeit darauf übertragen werden.

8.1.2 Deaktivierung von Prozessorressourcen

Bei vielen Prozessorarchitekturen werden nicht sämtliche Ressourcen für die Laufzeit einer Anwendung benötigt. Durch die Deaktivierung von Ressourcen, die nur minimal die Performanz beeinflussen, kann der Energieverbrauch erheblich reduziert werden. Am effizientesten ist eine solche Deaktivierung, wenn dies bereits durch den Compiler berücksichtigt werden kann, so dass er durch einen Wechsel der ISA weniger Ressourcen verwendet. Zum Beispiel können bei einem VLIW-Prozessor nicht benötigte Issue-Slots oder Register deaktiviert werden. In beiden Fällen ändert sich die ISA in Abhängigkeit der aktivierten Issue-Slots oder Register. Durch den Einsatz des mixed-ISA Softwareframework kann der Tradeoff zwischen Performanz und Energieverbrauch flexibel durch Einsatz der vorgestellten mixed-ISA Programmiermodellen

evaluiert werden. Dadurch enstehend weitere Möglichkeiten zum Einsatz des vorgestellten mixed-ISA Softwareframeworks über die Kahrisma-Architektur hinaus.

8.2 Semi-Automatische Parallelisierung von Matlab-ähnlichen Code

Die Kahrisma-Architektur und das Softwareframework werden im Rahmen des ALMA EU-Projekts [137] als eine Zielarchitektur für die semi-automatische Parallelisierung von Scilab-Code verwendet. Scilab ist eine Alternative zu Matlab. Dabei wird insbesondere die Kahrisma-Architektur an die Kommunikationsanforderungen der automatischen Parallelisierung der Eingangssprache optimiert. Weiterhin werden die System-ADL und der System-Simulator für die Simulation von MPSoC-Architekturen erweitert. Zusätzlich wird die System-ADL um verhaltensorientierten Informationen erweitert, die für eine Retargierung des gesamten Parallelisierungsframework eingesetzt werden kann.

8.3 Verbesserung der Clustered-VLIW-Performanz

Innerhalb dieser Arbeit wurde ein retargierbares mixed-ISA Compiler-Backend entwickelt, das unter anderem Code für Clustered-VLIW-Prozessoren erzeugen kann. Dieses Backend setzt viele bekannte Compileroptimierungen für diesen Prozessortyp um. Allerdings wurde die Codegenerierung von VLIW- und Clustered-VLIW-Prozessoren über Jahrzehnte erforscht und weiterentwickelt und somit besteht im Compiler-Backend immer Raum für Optimierungen. Zur Verbesserung der Codequalität können folgenden Optimierungen durchgeführt werden:

- Unterstützung von If-Konversion für verschachtelte If-Anweisungen
- Untersuchung unterschiedlicher Clustering-Verfahren
- Integration des Modulo-Schedulings für Clustered-VLIW-Prozessoren
- Verwendung von aggressiveren Loop-Unrolling-Methoden
- Unterstützung von globalen Scheduling-Verfahren

8.4 Erhöhung des ILPs durch spekulative Sprünge

Die Kahrisma-Mikroarchitektur hat eine interessante Prozessorpipeline. Sie basiert auf einer superskalaren Pipeline, die jedoch von einer VLIW-artigen Befehlssatzarchitektur programmiert wird. Superskalarität hat gegenüber VLIW den Vorteil, dass zur

Laufzeit innerhalb der Pipeline mehr ILP extrahiert werden kann, da mittels spekulativer Ausführung der Kontrollfluss bei der Parallelisierung nicht beachtet werden muss. Compiler für VLIW-Prozessoren können allerdings nur sehr limitiert über Basisblock-Grenzen hinaus optimieren.

Spekulative Sprünge könnten hier Abhilfe schaffen. Im Gegensatz zu normalen Sprüngen erfordern die spekulativen Sprünge kein Eingangsregister, das mitteilt, ob ein solcher Sprung genommen oder nicht genommen wird. Stattdessen wird diese Entscheidung anhand der Sprungvorhersage getroffen. Dadurch ist es möglich, dass die spekulativen Sprünge früher im Basisbock, bevor die eigentliche Sprungvorhersage berechnet wurde, ausgeführt werden können. Der eigentliche Sprung wird dann später ausgeführt und überprüft, ob die Spekulation korrekt durchgeführt wurde. Falls dies nicht der Fall war, werden sämtlichen Befehle, die seit dem spekulativen Sprung ausgeführt wurden, verworfen und es wird der Basisblock, der den spekulativen Sprung enthielt, normal ausgeführt. Dazu benötigt die Mikroarchitektur allerdings die Möglichkeit der spekulativen Ausführung von Instruktionen, wie es in der Kahrisma-Architektur der Fall ist. Der Compiler könnte dann nachfolgende Basisblöcke analysieren und ggf. spekulative Sprünge zusammen mit optimierten Basisblöcken einfügen.

8.5 Optimierung von OpenMP-Programmen

OpenMP (Open Multi-Processing) ist eine Programmierschnittstelle für die Shared-Memory-Programmierung in C++, C und Fortran auf Multiprozessor-Computern. Bei OpenMP-Programmen findet die Parallelisierung häufig auf Schleifenebene statt, so dass eine Schleife auf unterschiedliche Threads verteilt wird. Somit kann innerhalb von Schleifen TLP ausgenutzt werden, während außerhalb der Schleifen die parallelen Threads ungenutzt sind. Daher findet OpenMP häufig bei numerischen Applikationen, dessen zentrale Hauptschleife parallelisiert werden kann, eingesetzt.

In Kombination mit der rekonfigurierbaren Kahrisma-Architektur und einer Erweiterung des mixed-ISA Softwareframeworks könnte auch hier die dynamische Ausnutzung von ILP und TLP effizient verwendet werden. So könnten in OpenMP-Schleifen die Architektur auf die Ausnutzung von TLP konfiguriert werden während außerhalb der Schleifen der einzelne Thread von zusätzlichem ILP profitieren könnte. Dadurch wäre ein wesentlich feingranularerer Einsatz von OpenMP möglich und man könnte selbst kleine Schleifen automatisch parallelisieren.

8.6 Automatische Hardware-Generierung von Prozessoren

Im Rahmen dieser Arbeit lag der Fokus der entwickelten ADLs auf der Simulator- und Compilergenerierung. Insbesondere die System-ADL könnte noch für die Gene-

rierung von MPSoC-Architekturen erweitert werden. Dadurch würde ein generisches Werkzeug zur Exploration von applikationsspezifischen MPSoCs entstehen.

Genauso könnte die Core-ADL für die Hardwaregenerierung von applikationsspezifischen Prozessorkernen innerhalb des MPSoCs erweitert werden. Der aktuelle Stand der Technik besteht darin, dass innerhalb einer ADL eine strukturelle Beschreibung der Mikroarchitektur vorhanden ist und dadurch einer Hardwaregenerierung ermöglicht wird. Diese Methode ist allerdings unflexibel, wenn man den Typ der Mikroarchitektur wechseln möchte (z.B. von RISC auf Superskalarität). Statt die Mikroarchitektur in der ADL zu beschreiben, könnte man diese stattdessen im Generatormodul vorgeben, dafür aber verschiedenen Generatormodule für unterschiedliche Mikroarchitekturen anbieten. Dadurch würde man zwar Flexibilität verlieren, aber man könnte relativ schnell die Mikroarchitektur je nach Performanzanforderung wechseln. Denkbar wäre hier z.B. ein Generator für einen FSM-Prozessor, einer 5-stufigen RISC-Pipeline, 7-stufigen RISC-Pipeline, n-fache VLIW-Pipeline oder superskalare Pipeline.

8.7 Just-in-time-Kompilierung

Ein Nachteil des aktuellen Verfahrens für das statische mixed-ISA Programmiermodell liegt darin, dass für jede Konfiguration unterschiedlicher Programmcode erzeugt werden muss, wodurch die ausführbare Datei um ein Vielfaches mehr Speicherplatz benötigt. Dies kann durch die Verwendung von Just-in-time-Kompilierung verhindert werden, bei der das Programm zunächst in eine Zwischendarstellung (wie z.B. Java-Bytecode, *Common Intermediate Language* (CIL) des .NET-Frameworks oder LLVM-Bitcode) übersetzt wird und die endgültige Kompilierung erst bei Bedarf, also Just-in-time, bei der Ausführung für die benötigte Funktion und Konfiguration durchgeführt wird.

9 Zusammenfassung

Es wird immer schwieriger, die steigende Anzahl an Transistoren innerhalb eines Prozessors effizient durch Software auszunutzen. In der Vergangenheit war die Mikroarchitektur eines Prozessors für die automatisch Ausnutzung von *Parallelität auf Befehlsebene* (ILP, engl. *Instruction-Level Parallelism*) eines sequentiellen Befehlsstroms verantwortlich. Da diese Form der Parallelität nur bis zu einem gewissen Grad skaliert, wird bei Multiprozessorsystemen zusätzlich *Parallelität auf Threadebene* (TLP, engl. *Thread-Level Parallelism*) ausgenutzt.

Heutige MPSoCs sind statisch und haben somit eine feste Anzahl an Prozessoren und Funktionseinheiten pro Prozessor. Dadurch ist die ausnutzbare ILP und TLP durch das Design festgelegt und nur Anwendungen mit diesen ILP/TLP-Charakteristiken werden effizient ausgeführt. Daher wurde die Kahrisma-Architektur vorgestellt, die die Beschränkung von heutigen Multiprozessorsystemen mittels grobgranularer Rekonfiguration aufhebt. Die Prozessoren der Architektur sind nicht mehr statisch, sondern es können zur Laufzeit virtuelle Prozessorinstanzen mit unterschiedlicher Anzahl an Funktionseinheiten konfiguriert werden. Dadurch wird eine dynamische Ausnutzung von ILP und TLP ermöglicht und die Architektur kann sich zur Laufzeit auf unterschiedliche Parallelitätsanforderungen der Anwendungen anpassen. Die Kahrisma-Mikroarchitekur setzt die aufgestellten Ziele bezüglich (1) der Wiederverwendbarkeit von Ressourcen zur adaptiven Ausnutzung von ILP und TLP, (2) die partiell dynamische Rekonfiguration von ILP und TLP zur Laufzeit einer Anwendung, (3) die funktionale und praktische Skalierbarkeit des Konzeptes, (4) die Interruptfähigkeit zur Unterstützung von komplexen Steuer- und Betriebssystemfunktionen und (5) die Designzeit-Flexibilität um.

Eine effiziente Umsetzung dieser Flexibilität in der Mikroarchitektur erfordert allerdings die Verwendung der rekonfigurierbaren RSIW-Befehlssatzarchitektur als Interface zwischen Hard- und Software. Im Unterschied zu klassischen Befehlssatzarchitekturen hat eine rekonfigurierbare Befehlssatzarchitektur unterschiedliche, konfigurationsabhängige Ausprägungen. Klassische Software- und Compilerframeworks sind für die Programmierung nicht ausreichend, weil diese immer nur eine Konfiguration unterstützen können. Daher wurde innerhalb dieser Arbeit ein **mixed-ISA Softwareframework** realisiert, das die Programmierung von C/C++-Anwendungen mit variablen Befehlssatzarchitekturen ermöglicht und anhand der Kahrisma-Architektur demonstriert. Das Framework besteht aus einem mixed-ISA Compiler, mixed-ISA Binärwerkzeugen und mixed-ISA Simulatoren. Es zeichnet sich gegenüber dem Stand der Technik durch den gleichzeitigen Umgang mit mehreren (variablen) Befehlssatz-

architekturen aus. Der Begriff „mixed-ISA“ bedeutet hierbei, dass das Softwareframework mit mehrere *Befehlssatzarchitekturen* (ISAs, engl. *Instruction Set Architecturs*) gleichzeitig umgehen kann. Auf diese Weise werden sämtliche Konfigurationen der rekonfigurierbaren RSIW-Befehlssatzarchitektur unterstützt.

Zur Verwirklichung der Programmierbarkeit von Architekturen mit variablen Befehlssatzarchitekturen wurden zunächst drei neuartige Programmiermodelle definiert: das statische, dynamische und automatische mixed-ISA Programmiermodell. Beim statischen Programmiermodell hat man zunächst die Freiheit beim Start einer Anwendung jeweils eine geeignete Befehlssatzarchitektur auszuwählen, die dann für die Laufzeit der Anwendung gleich bleibt. Das dynamische und automatische mixed-ISA Programmiermodell geht einen Schritt weiter und erlaubt zusätzlich einen Wechsel der Befehlssatzarchitektur während der Laufzeit. Die drei mixed-ISA Programmiermodelle werden von dem mixed-ISA Softwareframework umgesetzt, das dadurch die Entwicklung und Kompilierung von mixed-ISA Anwendung in C/C++ für Architekturen mit variablen Befehlssatzarchitekturen im allgemeinen erlaubt. Im Fall der Kahrisma-Architektur kann sich eine Anwendung dadurch vor und/oder während der Laufzeit auf unterschiedliche ILP/TLP-Charakteristika anpassen.

Das mixed-ISA Softwareframework basiert dabei auf einer compiler- und simulationsorientierten *Architekturbeschreibungssprache* (ADL, engl. *Architecture Description Language*). Diese ADL wird für die Unterstützung der Rekonfiguration der Prozessorarchitektur sowie deren Designzeit-Flexibilität benötigt. Darauf aufbauend gewährleistet sie eine Erweiterbarkeit der Prozessorarchitektur. Zur Beschreibung der Rekonfiguration wurde die ADL in zwei Teile aufgeteilt: die Core-ADL und System-ADL. Zunächst wird innerhalb der **Core-ADL** die Befehlssatzarchitektur jeder Konfiguration spezifiziert. Dazu bietet die Core-ADL die Möglichkeit, mehrere Befehlssatzarchitekturen gleichzeitig zu spezifizieren und ist somit mixed-ISA-fähig. Durch die Verwendung von zentralen Sektionen, die von den verschiedenen ISA-Beschreibung gemeinsam verwendet werden, kann sowohl der Spezifikationsaufwand einer Konfiguration reduziert als auch das Verhalten im Falle einer Rekonfiguration modelliert werden. Damit unterscheidet sich die Core-ADL grundlegend von bekannten ADLs, die auf die Beschreibung einer ISA limitiert sind. Innerhalb des mixed-ISA Frameworks wird sie für die Generierung des Compiler, Assemblers und Core-Simulators eingesetzt. Durch die mixed-ISA-fähigkeit der Core-ADL hebt sich diese grundlegend von bekannten ADLs ab, die auf die Beschreibung einer ISA limitiert sind. Während die Core-ADL das Verhalten sämtlicher möglicher Konfigurationen enthält, beschreibt die **System-ADL** die Kahrisma-Prozessorarchitektur innerhalb der strukturellen Domäne auf der höchsten Abstraktionsebene. Sie enthält sowohl eine strukturelle Beschreibung des rekonfigurierbaren und statischen Teils des Gesamtsystems als auch der Konfigurationen. Die Struktur einer Konfiguration wird durch ein Template aus benötigten Ressourcen des Gesamtsystems festgelegt. Das Verhalten einer Konfiguration ist nur durch einen Verweis auf die Core-ADL gegeben. Dadurch wird eine benutzer-retargierbare Simulation der Kahrisma-Architektur auf Systemebenen im Softwareframework ermöglicht.

Das **statische mixed-ISA Programmiermodell** wurde innerhalb des Softwareframeworks durch einen *mixed-ISA Compiler* und *mixed-ISA Binärwerkzeuge* umgesetzt. Als Basis für den Compiler wurde die LLVM-Compiler-Infrastruktur verwendet. Dazu wurde ein neues, benutzer-retargierbares, mixed-ISA LLVM-Backend realisiert. Die Benutzer-Retargierbarkeit wurde durch das *CoreGen-Werkzeug* umgesetzt, das mittels Metaprogrammierung aus der Core-ADL Teile des Quellcodes des Compiler-Backends generiert. Abhängig von der Konfiguration der RSIW-Befehlssatzarchitektur ähneln diese RISC-, VLIW- oder Clustered-VLIW-Befehlssatzarchitekturen. Daher musste das Backend für die Codegenerierung insbesondere von Clustered-VLIW-Befehlssatzarchitekturen erweitert werden. Für das statische mixed-ISA Programmiermodell wurde das Backend so erweitert, dass es in einem Durchlauf sämtliche Funktionen dupliziert und somit mehrfach, je ISA bzw. Konfiguration, kompiliert. Als Ausgabe erzeugt es eine mixed-ISA Assemblerdatei, die Assembler-Instruktionen sämtlicher verwendeter ISAs enthält. Die Binärwerkzeuge, bestehend aus Assembler und Linker, wurden um die Möglichkeit der Verarbeitung von mixed-ISA Anwendungen erweitertet und generieren eine mixed-ISA ausführbare Datei. Durch die Kompilierung von sämtlichen Konfigurationen in eine Datei kann beim Starten der Anwendung die Konfiguration bzw. ISA frei ausgewählt und somit der ausnutzbare ILP eingestellt werden.

Zur Realisierung des **dynamischen mixed-ISA Programmiermodells** wurde eine *mixed-ISA Erweiterung der C/C++-Programmiersprache* spezifiziert, die eine Steuerung der ISA in einer Anwendung zur Laufzeit mittels mixed-ISA Funktionsattributen ermöglicht. Das Frontend im Compiler verarbeitet die mixed-ISA Funktionsattribute und stellt sie dem Compiler-Backend innerhalb der LLVM-Zwischendarstellung zur Verfügung. Dieses dupliziert die Funktionen dann anhand der Funktionsattribute und fügt bei Bedarf Code zum Umschalten der ISA ein. Insbesondere die automatische Generierung des Umschaltcodes erforderte die Unterstützung von verschiedenen ISAs auf Basisblock-Granularitätsebene im Compiler-Backend, die durch eine Erweiterung nahezu sämtlicher Backend-Passes erreicht wurde. Dadurch kann der Entwickler für jede Funktion eine Konfiguration festlegen. Zur Laufzeit wird beim Aufruf einer Funktion deren Konfiguration durch Rekonfiguration sichergestellt.

Für das **automatische mixed-ISA Programmiermodell** wurde eine *profile-guided Optimierung* der ISA in das Softwareframework integriert. Dabei wird eine Anwendung zunächst mit dem statischen mixed-ISA Programmiermodell kompiliert und ISA-abhängige, hierarchische Profiling-Informationen durch den Simulator erzeugt. Anhand dieser Informationen berechnet dann ein neues Werkzeug, der *ISA-Partitionierer*, eine ISA-Partitionierung. Der ISA-Partitionierer löst das NP-schwere Partitionierungsproblem mittels eines heuristischen Algorithmus unter Ausnutzung der Programmlokalität. Innerhalb des Algorithmus wird eine Partitionierung durch ein mixed-ISA Performanzmodell evaluiert, das die Performanz anhand der ISA-abhängigen Profiling-Informationen abschätzt. Der Entwickler kann dabei die Partitionierung über Optimierungsparameter steuern. Bei einem erneuten Durchlauf des Compilers wird die ISA-Partitionierung in Kombination mit der dynamischen mixed-ISA Codegenerie-

rung verwendet, um die finale mixed-ISA optimierte Anwendung zu generieren.

Das mixed-ISA Softwareframework wird durch einen **Core-Simulator** und einen **System-Simulator** komplettiert. Der Core-Simulator unterstützt die mixed-ISA Simulation einer virtuellen Prozessorinstanz, wie sie in der mixed-ISA Architekturbeschreibungssprache spezifiziert ist. Dabei kann sowohl die Start-ISA gewählt als auch die ISA zur Laufzeit gewechselt werden. Er integriert ein zyklenapproximatives Performanzmodell, das in der Lage ist, die Performanz der virtuellen Prozessorinstanzen zu approximieren ohne die Prozessorpipeline im Detail zu simulieren. Der System-Simulator verwendet mehrere Core-Simulatoren zur Simulation eines Gesamtsystems.

Es konnte gezeigt werden, dass die dynamische Ausnutzung von TLP und ILP innerhalb einer grobgranular rekonfigurierbaren Prozessorarchitektur nicht nur in der Hardware möglich ist, sondern aus Software-Sicht auch die Programmierbarkeit durch das entwickelte mixed-ISA Softwareframework gewährleistet werden kann. Dazu wurden alle drei aufgestellten mixed-ISA Programmiermodelle zunächst für Ein-Prozessor-Anwendungen evaluiert. Durch die Kompilierung mit dem statischen mixed-ISA Programmiermodell konnte die Konfiguration und damit der Ressourcenverbrauch einer Anwendung beim Start dynamisch eingestellt werden. Dadurch konnte ein maximaler Speedup-Faktor von 3,2 durch eine Vervierfachung der Ressourcen bei Anwendungen mit hohem ILP erzielt werden. Dabei kann der Tradeoff zwischen Performanz und Ressourcenverbrauch flexibel gewählt werden. Mit dem dynamischen mixed-ISA Programmiermodell konnten Anwendungen, bestehend aus Teilen mit hohem und niedrigem ILP, weiter optimiert werden. Durch die Rekonfiguration während der Ausführung von Algorithmen mit niedrigem ILP auf Konfigurationen mit weniger Ressourcen konnte der durchschnittliche Ressourcenverbrauch bei einem geringen Performanzverlust von 4,9% um 57% reduziert werden. Weiterhin wurde mit dem automatischen mixed-ISA Programmiermodell gezeigt, dass eine automatische ISA-Partitionierung einer Anwendung möglich ist und somit dem Endnutzer flexibel einen Tradeoff zwischen Performanz und Ressourcen- bzw. Energieverbrauch wählen kann.

Weiterhin wurde eine Multi-Prozessor-Anwendung im statischen mixed-ISA Programmiermodell für das Kahrisma-MPSoC untersucht, die sowohl ausreichend ILP als auch TLP bietet. Dabei wurden die verfügbaren Ressourcen, Problemgröße und verwendeten Parallelitätsformen variiert. Es hat sich herausgestellt, dass eine effiziente Konfiguration zwischen ILP und TLP nicht nur vom Algorithmus an sich, sondern auch von der Problemgröße sowie der Anzahl an verfügbaren Ressourcen abhängt. Dies verdeutlicht, dass die Entscheidung zwischen Komplexität und Anzahl von Prozessoren eines MPSoCs zur Designzeit nur für einen festgelegten Algorithmus und eine festgelegte Problemgröße getroffen werden kann. Andernfalls sind statische, nichtrekonfigurierbare MPSoCs zwangsläufig ineffizient.

In dieser Arbeit wurde gezeigt, dass mit neuen Programmiermodellen für Prozessoren mit verschiedenen Befehlssatzarchitekturen und einem Softwareframework, das diese Programmiermodelle umsetzt, die Programmierung der Kahrisma-Architektur unter

dynamische Ausnutzung von ILP und TLP möglich ist und effizient eingesetzt werden kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABI	Application Binary Interface (dt. *Binärschnittstelle*)
ADL	Architecture Description Language (dt. *Architekturbeschreibungssprache*)
AES	Advanced Encryption Standard
AIE	Atomic Instruction Execution
ALU	Arithmetic Logical Unit (dt. *Arithmetisch-logischen Einheit*)
AR	Assembly Representation (dt. *Assembly-Repräsentation*)
AST	Abstract Syntax Tree (dt. *Abstrakter Syntaxbaum*)
CAS	Cycle-Accurate Simulator (dt. *Taktgenauer Simulator*)
CIL	Common Intermediate Language
CISC	Complex Instruction Set Computers
DAG	Directed Acyclic Graph (dt. *Gerichteter azyklischer Graph*)
DCT	Discrete Cosine Transform (dt. *Diskrete Kosinustransformation*)
dest	Destination Operand (dt. *Zieloperanden*)
DOE	Dynamic Operation Execution
DOL	Distributed Operation Layer
DSE	Design-Space Exploration (dt. *Entwurfsraumexploration*)
EDPE	Encapsulated Datapath Element
ELF	Executable and Linkable Format
EPIC	Explicit Parallel Instruction Computing
FFT	Fast Fourier Transform (dt. *Schnelle Fourier-Transformation*)
FSM	Finite State Machine (dt. *Endlicher Automat*)
GPP	General-Purpose Prozessoren
GPR	General Purpose Register (dt. *Allzweckregister*)
HDL	Hardware Description Language (dt. *Hardwarebeschreibungssprache*)
HMDES	High-Level Machine Description
IA-32	32 bit Intel Architecture
IA-64	64 bit Intel Architecture
ICC	Inter-Cluster Communication (dt. *Inter-cluster Kommunikation*)
ILP	Instruction-Level Parallelism (dt. *Parallelität auf Befehlsebene*)
ILP	Integer Linear Programming
IP	Instruction Pointer (dt. *Befehlszeiger*)
ISA	Instruction Set Architecture (dt. *Befehlssatzarchitektur*)
ISDL	Instruction Set Description Language

ISE ..... Instruction Set Extraction (dt. *Befehlssatzextraktion*)
ISR ..... Interrupt-Service Routine (dt. *Unterbrechungsbehandlungsroutine*)
ISS ..... Instruction Set Simulator (dt. *Befehlssatzsimulator*)
LISA ..... Language for Instruction Set Architecture
LLVM ..... Low-Level Virtual Machine
LLVM-IR ..... LLVM Intermediate Representation (dt. *LLVM-Zwischendarstellung*)
LP ..... Linear Programming (dt. *Lineare Programmierung*)
LRU ..... Least Recently Used
LSB ..... Least Significant Byte
LSQ ..... Load/Store Queue
MAC ..... Multiply and Accumulate
MAML ..... Machine Markup Language
MC ..... Modulus Cycle
MIMOLA ..... Machine Independent Microprogramming Language
MIPS ..... Million Instructions per Second
MMU ..... Memory Management Unit
MOB ..... Memory Order Buffer
MPSoC ..... Multi-Prozessor System on Chip
MSB ..... Most Significant Byte
NoC ..... Networks-on-Chip
NUAL ..... Non-Unit Assumed Latencies
OOE ..... Out-of-Order Execution
PE ..... Processing Elements (dt. *Verarbeitenden Elemente*)
PE-Level ..... Processing Element Level (dt. *Prozessor-Ebene*)
PR ..... Packed Representation (dt. *Packed-Repräsentation*)
PROPAN ..... Postpass Retargetable Optimizer and Analyser
RADL ..... Retargetable Architecture Description Language
RISC ..... Reduced Instruction Set Computer
RSIW ..... Run-Time Scalable Issue-Width Processors
RTL ..... Register Transfer Level (dt. *Registertransferebene*)
SHAPES ..... Scalable Hardware/Software Architecture Platform for Embedded Systems
SIC ..... Single Instruction Computer (dt. *Eininstruktionscomputer*)
src ..... Source Operand (dt. *Quelloperanden*)
SSA ..... Static Single Assignment Form (dt. *Static-Single-Assignment-Darstellung*)
td ..... Target Description
TDL ..... Target Description Language
TLP ..... Thread-Level Parallelism (dt. *Parallelität auf Threadebene*)
UAL ..... Unit Assumed Latencies
UDL/I ..... Unified Design Language
UML ..... Unified Modeling Language
UR ..... Unpacked Representation (dt. *Unpacked-Repräsentation*)
VHDL ..... Very High Speed Integrated Circuit Hardware Description Lan-

	guage
VLIW	Very Long Instruction Word
WPPA	Weakly Programmable Processor Arrays

Literaturverzeichnis

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[3] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[4] Fred J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)(abstract only). In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] SPARC International Inc. The SPARC architecture manual version 9. Technical Report SAV09R1459912, SPARC International Inc., 1994.

[6] David Patterson and John LeRoy Hennessy. *Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle*. Oldenbourg Wissenschaftsverlag, vollständig überarbeitete auflage edition, March 2011.

[7] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1 –58, 2008.

[8] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948 –960, September 1972.

[9] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual combined volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C. Technical Report 325462-045US, January 2013.

[10] Advanced Micro Devices. AMD64 architecture programmer's manual volume 3: General-purpose and system instructions. Technical Report 24594 v3.19, September 2012.

[11] David A. Patterson and Carlo H. Sequin. RISC i: A reduced instruction set VLSI computer. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, page 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[12] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. *SIGARCH Comput. Archit. News*, 11(3):140–150, June 1983.

[13] T. Bonny and J. Henkel. FBT: filled buffer technique to reduce code size for VLIW processors. In *IEEE/ACM International Conference on Computer-Aided Design, 2008. ICCAD 2008*, pages 549 –554, November 2008.

[14] Michael S. Schlansker, B. Ramakrishna Rau, and Multitemplate. EPIC: an architecture for instruction-level parallel processors. Technical report, 2000.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 5 edition, September 2011.

[16] J.H. Crawford. The i486 CPU: executing instructions in one clock cycle. *IEEE Micro*, 10(1):27 –36, February 1990.

[17] D. Alpert and D. Avnon. Architecture of the pentium microprocessor. *IEEE Micro*, 13(3):11 –21, June 1993.

[18] James E. Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, AFIPS '64 (Fall, part II), page 33–40, New York, NY, USA, 1965. ACM.

[19] RM Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[20] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70 –83, October 2000.

[21] B.R. Rau. Dynamically scheduled VLIW processors. In *, Proceedings of the 26th Annual International Symposium on Microarchitecture, 1993*, pages 80 –92, December 1993.

[22] Mark Smotherman. *Understanding EPIC Architectures and Implementations*. 2002.

[23] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing. A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, January 2005.

[24] Andrei Sergeevich Terechko. *Clustered VLIW Architectures: a Quantitative Approach*. PhD thesis, 2007.

[25] Andrei Terechko, Erwan Le Thenaff, Manish Garg, Jos Van Eijndhoven, and Henk Corporaal. Inter-cluster communication models for clustered VLIW processors. In *In 9th International Symposium on High Performance Computer Architecture*, page 298–309. IEEE Computer Society, 2003.

[26] Wei Qin and Sharad Malik. Architecture description languages for retargetable compilation. In *in The Compiler Design Handbook: Optimizations & Machine Code Generation*, page 535–564. CRC Press, 2002.

[27] Wei Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. 2004.

[28] Prabhat Mishra, Aviral Shrivastava, and Nikil Dutt. Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):626–658, June 2004.

[29] Wei Qin and S. Malik. A study of architecture description languages from a model-based perspective. In *Sixth International Workshop on Microprocessor Test and Verification, 2005. MTV '05*, pages 3–11, 2005.

[30] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *Computers and Digital Techniques, IEE Proceedings -*, 152(3):285 – 297, May 2005.

[31] Prabhat Mishra, editor. *Processor description languages : applications and methodologies*. The Morgan Kaufmann series in systems on silicon. Elsevier Morgan Kaufmann, Amsterdam, 2008. Includes bibliographical references and index. - Formerly CIP.

[32] United States Dept of Defense VHSIC Program Office, United States Office of the Director of Defense Research, Engineering, and Acquisition United States. Office of the Under Secretary of Defense. *Very High Speed Integrated Circuits (VHSIC): Final Program Report, 1980-1990*. VHSIC Program Office, 1990.

[33] Samir Palnitkar. *Verilog; hdl: a guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[34] Rainer Leupers and Peter Marwedel. Retargetable code generation based on structural processor description. *Design Automation for Embedded Systems*, 3:75–108, 1998. 10.1023/A:1008807631619.

[35] R. Leupers and P. Marwedel. Retargetable generation of code selectors from hdl processor models. In *European Design and Test Conference, 1997. ED TC 97. Proceedings*, pages 140 –144, mar 1997.

[36] Hiroki Akaboshi. *A study on design support for computer architecture design*. Phd thesis, Dept. of Information Systems, Kyushu University, Japan, 1996.

[37] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[38] A. Fauth and A. Knoll. Automated generation of dsp program development tools using a machine description formalism. In *Int. Conf. on Audio, Speech and Signal Processing*, pages 457–460, 1993.

[39] Peter Marwedel and Gert Goossens, editors. *Code Generation for Embedded Processors [Dagstuhl Workshop, August 31 - September 2, 1994]*. Kluwer, 1995.

[40] F. Löhr, A. Fauth, and M. Freericks. *SIGH/SIM: An Environment for Retargetable Instruction Set Simulation*. Bericht (Technische Universität Berlin. Fachbereich 20, Informatik). Technische Universität Berlin, Fachbereich 13, Informatik, 1993.

[41] Target Compiler Technologies. http://www.retarget.com.

[42] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[43] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 510–515, New York, NY, USA, 1998. ACM.

[44] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 927–932, New York, NY, USA, 1999. ACM.

[45] John C. Gyllenhaal, Wen-mei W. Hwu, and B. Ramakrishna Rau. HMDES version 2.0 speci

cation. Technical Report IMPACT-96-3, 1996.

[46] The MDES User Manual. http://www.trimaran.org, 1997.

[47] S Aditya, V Kathail, and BR Rau. *Elcor's Machine Description System: Version 3.0*. Number HPL-98-128 in HP Laboratories Technical Report. 1998.

[48] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM.

[49] Exploration framework using EXPRESSION ADL. http://www.ics.uci.edu/ express.

[50] Prabhat Mishra, Peter Grun, Nikil Dutt, and Alex Nicolau. Memory subsystem description in EXPRESSION. Technical Report ICS Technical Report # 00-31, University of California, Irvine, Oct 2000.

[51] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *in Proceedings of the IEEE Workshop on VLSI Signal Processing*, pages 127–136, 1996.

[52] Lisa projektseite. http://servus.ert.rwth-aachen.de/lisa/.

[53] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):815–834, 2000.

[54] Coware inc. http://www.coware.com.

[55] Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and

Hans van Someren. A methodology and tool suite for c compiler generation from adl processor models. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21276, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Daniel Kästner. *Retargetable postpass optimisation by integer linear programming*. PhD thesis, Universität des Saarlandes – Naturwissenschaftlich-Technische Fakultät I, 2000.

[57] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. *SIGPLAN Not.*, 41(7):145–152, June 2006.

[58] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, page 13–22, New York, NY, USA, 2007. ACM.

[59] Brandner, Florian. *Compiler Backend Generation from Structural Processor Models*. PhD thesis, Vienna University of Technology, 2009.

[60] W3C XML Working Group. The XML 1.0 standard (5th edition). W3C recommendation, Network Theory Ltd, 2008.

[61] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Paphos, January 2009.

[62] Florian Brandner. Precise simulation of interrupts using a rollback mechanism. In *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '09, page 71–80, New York, NY, USA, 2009. ACM.

[63] Florian Brandner. Completeness of automatically generated instruction selectors. In *2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 175 –182, July 2010.

[64] International Business Machines Corp. Book i: PowerPC user instruction set architecture. PowerPC Architecture Book Version 2.02, January 2005.

[65] MIPS Technologies, Inc. Volume II: the MIPS32® instruction set. MIPS32® Architecture For Programmers MD00086, Revision 2.60, January 2009.

[66] SPARC International Inc. The SPARC architecture manual version 8. Technical Report SAV080SI9308, 1992.

[67] ARM Limited. ARM architecture reference manual. Technical Report ARM DDI 0100I, July 2005.

[68] Intel Corporation. MCS@51 (Intel 8051) MICROCONTROLLER FAMILY USER'S MANUAL. Technical Report 272383-002, February 1994.

[69] Microchip Technology Inc. PIC16F84A data sheet - 18-pin enhanced FLASH/EEPROM 8-bit microcontroller. Technical Report DS35007B, 2001.

[70] MOTOROLA INC. MOTOROLA m68000 FAMILY programmer's reference manual. Technical report, 1992.

[71] Altera Corporation. Nios II custom instruction user guide. Technical Report UG-N2CSTNST-2.0, January 2011.

[72] OPENCORES.ORG and Authors. OpenRISC 1000 architecture manual. Technical Report Rev 1.0, January 2003.

[73] Texas Instruments. TMS320C62x DSP CPU and instruction set reference guide. Technical Report SPRU731A, May 2010.

[74] Ricardo Santos, Rodolfo Azevedo, and Guido Araujo. The 2D-VLIW architecture. Technical Report IC-06-06, Institute of Computing, University of Campinas, March 2006.

[75] Ricardo Santos, Rodolfo Azevedo, and Guido Araujo. 2D-VLIW: an architecture based on the geometry of computation. In *International Conference on Application-specific Systems, Architectures and Processors, 2006. ASAP '06*, pages 87 –94, September 2006.

[76] Distributed operation layer homepage. http://www.tik.ee.ethz.ch/~shapes/dol.html.

[77] Kai Huang, Wolfgang Haid, Iuliana Bacivarov, Matthias Keller, and Lothar Thiele. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. *ACM Trans. Embed. Comput. Syst.*, 11(1):8:1–8:23, April 2012.

[78] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Seventh International Conference on Application of Concurrency to System Design, 2007. ACSD 2007*, pages 29 –40, July 2007.

[79] Dirk Fischer, Jürgen Teich, Ralph Weper, Uwe Kastens, and Michael Thies. Design space characterization for Architecture/Compiler co-exploration. In *IN ACM SIG PROCEEDINGS INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMS (CASES 2001*, page 108–115. ACM Press, 2001.

[80] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, and Jürgen Teich. MAML: an ADL for designing single and multiprocessor architectures. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, Systems on Silicon, page 295–327. Morgan Kaufmann, June 2008.

[81] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Rainer Schaffer, and Jürgen Teich. MAML – an architecture description language for modeling and simulation of processor array architectures, part i. Technical Report 03-2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, March 2006.

[82] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Jürgen Teich, Rainer Schaffer, and Renate Merker. An architecture description language for massively parallel processor architectures. In *IN GI/ITG/GMM-WORKSHOP 2006 - METHODEN*, page 11–20, 2006.

[83] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jurgen Teich. A highly parameterizable parallel processor array architecture. In *IEEE International Conference on Field Programmable Technology, 2006. FPT 2006*, pages 105 –112, December 2006.

[84] Cristiano C. de Araujo, Edna Barros, Rodolfo Azevedo, and Guido Araujo. Processor centric specification and modelling of MPSoCs. In *Forum on specification and Design Languages*, pages 303–315, 2005.

[85] Sejong Oh and Yunheung Paek. A quantitative comparison of two retargetable compilation approaches. In *2003 International Conference on Parallel Processing, 2003. Proceedings*, pages 29 –36, October 2003.

[86] Free Software Foundation (FSF). GCC, the GNU compiler collection - GNU project. http://gcc.gnu.org/.

[87] Chris Lattner. The LLVM compiler infrastructure project.

[88] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27:68–76, 1991.

[89] Christopher W Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1:213–226, 1992.

[90] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In *COMPUTERS AND CHEMICAL ENGINEERING*, page 22–8. Springer, 2003.

[91] A. Shrivastavai, N. Dutti, A. Nicolaut, and E. Earlier. Operation tables for scheduling in the presence of incomplete bypassing. In *International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004*, pages 194 – 199, September 2004.

[92] Sanghyun Park, A. Shrivastava, N. Dutt, E. Earlie, A. Nicolau, and Yunheung Paek. Automatic generation of operation tables for fast exploration of bypasses in embedded processors. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, page 6 pp., March 2006.

[93] Hewlett-Packard Labs. VEX toolchain. http://www.hpl.hp.com/downloads/vex/.

[94] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.

[95] Trimaran: A compiler and simulator for research on embedded and EPIC archi-

tectures - version 4.0. Technical report, April 2007.

[96] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. HPL-PD architecture specification: Version 1.1. Technical report, 2000.

[97] LLVM Project. LLVM language reference manual. http://www.llvm.org/docs/LangRef.html.

[98] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 21(5):895–913, 1999.

[100] Ralf König. *Grobgranular rekonfigurierbare Mikroarchitekturen zur dynamischen Erzeugung heterogener Prozessorinstanzen in Chip-Multiprozessoren*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2012. Karlsruhe, KIT, Diss., 2012.

[101] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. TIS Committee, http://refspecs.freestandards.org/elf/elf.pdf, May 1995.

[102] The SCO Group. System v application binary interface - DRAFT. http://www.sco.com/developers/gabi/latest/contents.html, October 2010.

[103] D. Crockford. JSON: the fat-free alternative to XML. In *Proceedings of XML 2006, USA*, 2006.

[104] International Organization for Standardization. Information technology - syntactic metalanguage - extended BNF. Technical Report ISO/IEC 14977, ISO/IEC, 2001.

[105] John R. Ellis. *Bulldog: a compiler for vliw architectures (parallel computing, reduced-instruction-set, trace scheduling, scientific)*. PhD thesis, Yale University, New Haven, CT, USA, 1985. AAI8600982.

[106] Free Software Foundation (FSF). GNU binutils - GNU project.

[107] Serge Lamikhov-Center. ELFIO - c++ library for reading and generating ELF files.

[108] Boost Community. Boost c++ libraries.

[109] DWARF debugging information format - version 4, June 2010.

[110] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1992.

[111] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE*

International Workshop, WWC '01, page 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[112] M. Borgerding. Kiss FFT, a fast fourier transform based up on the principle, "Keep it simple, stupid.", 2013.

[113] C. a. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962.

[114] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[115] Joan Daemen. *The design of Rijndael: the wide trail strategy explained*. Springer, Berlin; New York, 2001.

[116] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, July 2003.

[117] Paolo Faraboschi, Giuseppe Desoli, and Joseph A. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, HP Laboratories Cambridge, December 1998.

Betreute studentische Arbeiten

[118] Patrick Rieder. *KAHRISMA - Erweiterung eines retargierbaren, ADL-basierten Compiler-Framework zur Unterstützung von Prozessorarchitekturen mit geclusterten Registerspeichern*. Diplomarbeit, Karlsruher Institut für Technologie, May 2011.

[119] Timo Stripf. *Konzeption und Implementierung eines Compiler Frameworks zur Personalisierung eines grobgranular rekonfigurierbaren Prozessortemplates*. Diplomarbeit, Karlsruher Institut für Technologie, September 2007.

[120] Timo Sandmann. *KAHRISMA - Entwicklung eines parametrisierbaren SystemC Simulators für eine rekonfigurierbare Prozessorarchitektur*. Studienarbeit, Karlsruher Institut für Technologie, December 2009.

[121] Patrick Rieder. *KAHRISMA - Erweiterung eines LLVM Back-End zur Unterstützung von Clustered VLIW Architekturen*. Studienarbeit, Karlsruher Institut für Technologie, April 2010.

[122] Timo Sandmann. *KAHRISMA - Entwicklung eines ADL-basierten systemweiten mehrprozessor Simulators*. Diplomarbeit, Karlsruher Institut für Technologie, January 2011.

[123] Michael Schöffler. *KAHRISMA - High-Level Programmierung von Anwendungen mit gemischten Befehlssatzarchitekturen*. Masterarbeit, Karlsruher Institut für Technologie, November 2011.

[124] Achim Lösch. *KAHRISMA - Portierung der LibC auf die Kahrisma Architektur*. Studienarbeit, Karlsruher Institut für Technologie, December 2011.

[125] Achim Lösch. *Untersuchung von Anwendungen auf Parallelität auf Thread- und Befehlssatzebenen für die Kahrisma Architektur*. Diplomarbeit, Karlsruher Institut für Technologie, December 2012.

[126] Stefan Brähler. *Design and Implementation of an ADL based parameterizable MPSoC Simulator*. Diplomarbeit, Karlsruher Institut für Technologie, July 2013.

Eigene Veröffentlichungen

Konferenzbeiträge

[127] R. Koenig, T. Stripf, and J. Becker. A novel recursive algorithm for bit-efficient realization of arbitrary length inverse modified cosine transforms. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 604 –609, March 2008.

[128] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel. KAHRISMA: a novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 819 –824, March 2010.

[129] Ralf Koenig, Timo Stripf, Jan Heisswolf, and Juergen Becker. A scalable microarchitecture design that enables dynamic code execution for variable-issue clustered processors. pages 150–157. IEEE, May 2011.

[130] R. Koenig, T. Stripf, J. Heisswolf, and J. Becker. Architecture design space exploration of run-time scalable issue-width processors. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 77 –84, July 2011.

[131] T. Stripf, R. Koenig, and J. Becker. A novel ADL-based compiler-centric software framework for reconfigurable mixed-ISA processors. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 157 –164, July 2011.

[132] T. Stripf, R. Koenig, and J. Becker. A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 21 –26, March 2012.

[133] T. Stripf, R. Koenig, P. Rieder, and J. Becker. A compiler back-end for reconfigurable, mixed-ISA processors with clustered register files. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 462 –469, May 2012.

[134] Timo Stripf, Oliver Oey, Thomas Bruckschloegl, Ralf Koenig, Michael Huebner, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Gerard Rauwerda, Kim Sunesen, Steven Derrien, Daniel Menard, Olivier Sentieys, Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos, Diana Goehringer, Thomas Perschke, Dimitrios Kritharidis, Nikolaos Mitas, and Juergen Becker. A flexible approach for compiling scilab to reconfigurable multi-core embedded systems. In *2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*

(ReCoSoC), July 2012.

[135] J. Becker, T. Stripf, O. Oey, M. Huebner, S. Derrien, D. Menard, O. Sentieys, G. Rauwerda, K. Sunesen, N. Kavvadias, K. Masselos, G. Goulas, P. Alefragis, N.S. Voros, D. Kritharidis, N. Mitas, and D. Goehringer. From scilab to high performance embedded multicore systems: The ALMA approach. In *2012 15th Euromicro Conference on Digital System Design (DSD)*, pages 114 –121, September 2012.

[136] Timo Stripf, Oliver Oey, Thomas Bruckschloegl, Ralf Koenig, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Jordy Potman, Kim Sunesen, Steven Derrien, Olivier Sentieys, and Juergen Becker. A compilation- and simulation-oriented architecture description language for multicore systems. In *2012 IEEE 15th International Conference on Computational Science and Engineering (CSE)*, pages 383 –390, December 2012.

[137] Timo Stripf, Oliver Oey, Thomas Bruckschloegl, Juergen Becker, Gerard Rauwerda, Kim Sunesen, George Goulas, Panayiotis Alefragis, Nikolaos S. Voros, Steven Derrien, Olivier Sentieys, Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos, Dimitrios Kritharidis, Nikolaos Mitas, and Thomas Perschke. Compiling scilab to high performance embedded multicore systems. *Microprocessors and Microsystems*, July 2013.

Patente

[138] R. Koenig, T. Stripf, and J. Becker. *Reconfigurable processor architecture*. 2010. Published: Patent (Pending) EP2363812.