Spezifikation, Simulation und Validierung von Prozessoren

Dissertation

Schriftliche Arbeit zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
an der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Dennis Klassen

Paderborn, Mai 2013

Datum der mündlichen Prüfung:

1. Juni 2013

Gutachter:

Prof. Dr. Uwe Kastens, Universität Paderborn
Prof. Dr.-Ing. Ulrich Rückert, Universität Bielefeld

Danksagung

An dieser Stelle bedanke ich mich bei all jenen Personen, die mir die Fertigstellung der Arbeit ermöglicht und mich während meiner Arbeit unterstützt haben.

Als erstes danke ich meinem Doktorvater Professor Uwe Kastens dafür, dass er mich nach meiner Diplomarbeit in seine Arbeitsgruppe aufgenommen hat und mir damit die Möglichkeit gegeben hat diese Arbeit zu schreiben. Dabei hat er mich über die ganze Zeit hinweg unterstützt und mit konstruktiven Vorschlägen und Diskussionen geleitet. Des weiteren danke ich Professor Ulrich Rückert für die Übernahme des Korreferats und seinen wertvollen und konstruktiven Hinweisen für die Fertigstellung der Arbeit.

Ein besonderer Dank gilt Dr. Michael Thies für seine Hilfsbereitschaft und langjährige Zusammenarbeit. Durch sein umfangreiches Wissen und seine Erfahrung habe ich in fachlichen Diskussionen sehr viel gelernt, was zum Inhalt und der Qualität dieser Arbeit beigetragen hat. Weiterhin danke ich Dr. Bastian Cramer für den Support des Werkzeugsystems DEViL und für die Hilfestellung bei kniffligen Problemen.

An dieser Stelle danke ich meiner ersten Deutschlehrerin in Deutschland Klara Heinemann. Sie hat bei der Suche nach verschollenen Kommata und anderen sprachlichen Inkonsistenzen bei der Korrektur dieser Arbeit geholfen.

Abschließend danke ich meinen Eltern für die Unterstützung und Rückhalt. Sie haben mich auf meinem Bildungsweg begleitet und damit die Grundsteine für mein Studium gelegt.

Inhaltsverzeichnis

1 Einleitung

Der Einsatz anwendungsspezifischer Prozessoren erfährt, nicht zuletzt durch die Verbreitung der mobilen Systeme, immer größere Verbreitung. Dabei werden in verschiedenen Domänen neben der Optimierung der Software auch Prozessoren für die jeweiligen Anwendungen optimiert. Die Ziele der Optimierung der Prozessoren können vielfältig ausfallen: Ausführungsgeschwindigkeit, Stromverbrauch, Parallelität, Chipfläche usw. können im Fokus der Entwicklung liegen. Meistens werden Vorschläge für die Optimierung in der Simulation erarbeitet und erprobt. Durch steigende Integration erhöht sich insbesondere die Komplexität der Entwürfe, an denen in der Regel unterschiedliche Entwicklergruppen, wie Nachrichtentechniker, Programmierer oder Schaltungstechniker, beteiligt sind. Dabei besitzen die Entwickler unterschiedliche Erwartungen an die Beschreibungsmittel und bringen unterschiedliches Wissen über das System mit. Wie in einem vergleichbaren Umfeld zum Prozessorentwurf bereits festgestellt, wird „die Beschreibung komplexer Zusammenhänge und Abläufe umso schwieriger, je unterschiedlicher die Gesprächspartner sind“ [Kla10].

Die Verkürzung der Entwicklungszyklen und die Zusammenarbeit unterschiedlicher Entwicklergruppen soll dabei durch Werkzeugsysteme im Prozessorentwurf begünstigt werden. Die speziell entwickelten Werkzeugsysteme unterstützen die Entwurfsprozesse dadurch, dass die Darstellungen oder Strukturen der Werkzeuge auf die Problematik der Anwendungsdomäne angepasst sind. Sie erlauben somit die Entwicklung auf einem hohen Abstraktionsniveau.
Aus der Betrachtung der Problemstellung im Prozessorentwurf lassen sich für die Ausgestaltung der werkzeuggestützten Prozesse die Spezifikation, die Simulation und die Validierung von Entwürfen als Teilprobleme extrahieren. Ein Werkzeugsystem für die Vereinigung der Prozesse, in dem die Spezifikation und die Validierung der Entwürfe auf einem hohen Abstraktionsniveau ermöglicht werden, ist wünschenswert.

Die Grundidee beim Einsatz von Werkzeugsystemen ist es, die Beschreibung der Lösungen durch die Verwendung der Begriffe aus der Domäne des Prozessorentwurfs auf ein höheres Abstraktionsniveau zu bringen. Damit sind die Anforderungen an die Werkzeuge hoch, da das Abstraktionsniveau, die Beschreibungsmittel und die Ausdrucksfähigkeit für die Akzeptanz eines Werkzeugs entscheidend sind. Einige Werkzeuge sind für spezifische Aufgaben bei der Beschreibung von Prozessoren auf einem hohen Abstraktionsniveau spezialisiert und unterstützen nur die Spezifikation bestimmter

Teilaspekte eines Prozessors. Andere unterstützen ein breites Spektrum von Prozessoraspekten und geben damit in ihren Beschreibungsmitteln das hohe Abstraktionsniveau teilweise auf. Bei der Entwicklung eines Werkzeugs oder einer Sprache muss für die auftretenden Problemstellen ein Kompromiss zwischen dem Abstraktionsniveau und der Ausdrucksfähigkeit gefunden werden.

Bei der steigenden Komplexität der Entwürfe wird es immer schwieriger, Korrektheitsaussagen über deren Komponenten zu treffen. Für die Lokalisierung von Entwurfsfehlern wird eine Reihe von Methoden beschrieben, wie z. B. Validierung oder Verifikation. Die Verifikation findet in der Praxis aufgrund der hohen Komplexität der Modelle keine Anwendung. Bereits Sofiène Tahar hat im Resümee ihrer Arbeit „Eine Methode zur formalen Verifikation von RISC-Prozessoren“ [Tah94] festgestellt:

> *„Vollständig verifizierte RISC-Chips gibt es bis heute noch nicht.“*

Deshalb wird stattdessen bis heute die Validierung von Prozessoren mittels Simulation verwendet. Alexander Pretschner und Jan Philipps stellen in ihrer Arbeit „Methodological Issues in Model-Based Testing“ [PP05] fest:

> *„When a single model for both code generation and test case generation [is] chosen, this redundancy is lacking. In a sense, the code (or model) would be tested against itself. This is why no automatic verdicts are possible.“*

Auf dieser Grundlage verwenden bekannte Validierungswerkzeuge, basierend auf den Methoden des modellbasierten Testens, abstrakte Modelle für die automatische Erzeugung von Testfällen für die Validierung. Das modellbasierte Testen hat seinen Ursprung in der Softwareentwicklung, wodurch die Werkzeuge, sogar für die Validierung von Prozessoren, in der Regel für die Beschreibung der Modelle UML Notationen benutzen. Eine vollständig angepasste Validierung auf einem höheren Niveau benötigt Modelle mit den Begriffen aus dem Prozessorentwurf.

Ziele der Arbeit In dieser Arbeit wird eine Entwicklungsumgebung vorgestellt, die die Spezifikation und Validierung von Prozessoren auf einem hohen Abstraktionsniveau ermöglicht. Für die Spezifikation von Prozessoren wird eine visuelle Sprache entwickelt, die formale Spezifikationen von Prozessoren beschreibt. Die entwickelte Sprache erfüllt dabei folgende drei Aspekte:

Darstellung – Die Sprache beschreibt die Prozessorkonstrukte auf einem hohen Abstraktionsniveau und ist für Benutzer aus unterschiedlichen Gruppen zugänglich. Hierzu erfolgt die Visualisierung durch bekannte Darstellungen aus der Domäne.

Struktur – Für die Entwicklung von Prozessoren werden Simulatoren benötigt, die aus den Prozessorspezifikationen generiert werden.

Ausdrucksfähigkeit – Die Sprache wird für den Entwurf realer Prozessoren eingesetzt. Die Ausdrucksfähigkeit der Sprachkonstrukte unterstützt ein dafür ausreichendes Spektrum von Prozessorkonstrukten, womit die Beschreibung bereits vorhandener oder neuer Prozessoren möglich ist.

Die Sprachkonstrukte und die Struktur der Sprache erlauben die Validierung von Prozessorentwürfen. Hierfür werden statische und dynamische Validierungsmethoden entwickelt, welche die Validierung von Prozessorentwürfen aus der Spezifikation ermöglichen. Die statischen Validierungsmethoden sind in die Entwicklungsumgebung integriert und können dort auf die formale Spezifikation von Prozessoren angewendet werden. Bei der dynamischen Validierung stehen zwei Aspekte im Vordergrund. Zum einen werden für die dynamische Validierung Prozessorsimulatoren mit einer ausreichenden Simulationsgenauigkeit und -geschwindigkeit erzeugt. Zum anderen werden Methoden vorgestellt, mit denen aus der formalen Spezifikation des Prozessors Testfälle für die Validierung der Spezifikation automatisch generiert werden.

Methodischer Beitrag dieser Arbeit Diese Arbeit stellt einen methodischen Beitrag zur Entwicklung von Prozessoren dar. Als Ausgangspunkt werden existierende Werkzeuge für die Spezifikation von Prozessoren untersucht. Darüber hinaus werden die zentralen Konzepte aus dem Prozessorentwurf und charakteristische Prozessorarchitekturen genauer betrachtet. Die Arbeit stützt sich auf die bekannten Kategorisierungen der Sprachen für den Prozessorentwurf hinsichtlich Struktur und Entwicklungsziele. Diese Kategorisierung wird um die Aspekte des Abstraktionsniveaus und der Ausdrucksfähigkeit bei der Prozessorbeschreibung erweitert. Aus diesen Aspekten wird die methodische Vorgehensweise bei der Entwicklung der visuellen Sprache hergeleitet. Bewährte Ansätze werden dabei übernommen und in die neue Sprache integriert.

Außerdem leistet diese Arbeit einen methodischen Beitrag zur Validierung von Prozessorentwürfen auf einem hohen Abstraktionsniveau. Hierfür werden aktuelle Validierungsmethoden auf ihre Anwendbarkeit im Prozessorentwurf untersucht und dazu vorhandene Validierungswerkzeuge genauer betrachtet. Dabei werden für die dynamische Validierung die wohlbekannten Konzepte und Methoden zur Modellierung und Generierung von Testfällen übernommen.

Der entwickelte Ansatz zeigt, wie die Spezifikation und Validierung von Prozessoren durch visuelle Mittel vorangetrieben wird. Dabei wird gezeigt, dass die Konzepte der visuellen Sprache eine große Anzahl von Prozessorkonstrukten abdecken können, ohne dabei das hohe Abstraktionsniveau der Spezifikation aufzugeben. Mit der Sprache wird eine methodische Vorgehensweise für die Spezifikation von Prozessoren beschrieben, die den Entwurf einer Prozessorspezifikation und die Generierung von Simulatoren mit einem geringen Spezifikationsaufwand erlaubt. Außerdem wird gezeigt, wie die Validierung bereits während der Entwicklung einer Prozessorspezifikation anhand des Entwurfs durchgeführt werden kann.

Die hier entwickelte Sprache steht nicht in direkter Konkurrenz zu den bereits vorhandenen Ansätzen zur Spezifikation von Prozessoren, da sich die Vorgehensweise bei der Spezifikation und die Entwicklungsziele deutlich unterscheiden.

Der entwickelte Ansatz für die Validierung wird in zwei Bereiche unterteilt: die statischen und die dynamischen Methoden. Aus der Untersuchung des Umfelds für die Validierung von Prozessoren resultiert ein Fehlermodell, dass systematisch die Fehlerquellen in Prozessorspezifikationen auf einem hohen Abstraktionsniveau erfasst. Dieses Fehlermodell wird für die Entwicklung der statischen und dynamischen Validierungsmethoden verwendet.

Wie bereits erwähnt zeigt diese Arbeit, wie die statische Analyse bei dem Entwurf von Prozessoren eingesetzt werden kann. Dabei wird methodisch beschrieben, wie die Analysemethoden angewendet werden und welche Inkonsistenzen in den Prozessorspezifikationen durch diese Methoden aufgedeckt werden können.

Weiter wird gezeigt, wie die Konzepte des modellbasierten Testens systematisch in die Domäne des Prozessorentwurfs übertragen werden können. Konzepte aus den verschiedenen Werkzeugen oder Arbeiten für die Validierung von Prozessoren können teilweise für die Prozessorvalidierung übernommen werden. Durch das Wissen aus der Domäne besteht die Möglichkeit, diese Konzepte anzupassen und zu erweitern. Daraus resultieren Methoden für die dynamische Validierung von Prozessoren. Zusätzlich wird gezeigt, dass die Generierung eines Simulators und die Ableitung der Testfälle für die dynamische Validierung aus derselben Spezifikation mit der unbedingt erforderlichen Redundanz möglich ist.

Ein weiterer Beitrag dieser Arbeit ist ein Konzept für die automatische Generierung von Testfällen, bei dem für die Prozessorentwickler die Möglichkeit besteht, die Testfälle zu bearbeiten oder einzusehen. Auch hierfür wird eine visuelle Sprache vorgestellt, mit der die Definition der Testfälle mittels Darstellungen aus dem Prozessorentwurf vorgenommen wird.

Neben den bisher genannten methodischen Beiträgen wird in dieser Arbeit gezeigt, dass die beschriebenen Konzepte für die Spezifikation und Validierung realer Prozessoren geeignet sind, indem einige Beispielprozessoren spezifiziert und validiert werden. Dazu wird das Werkzeugsystem anhand von Beispielen evaluiert und konkrete Simulatoren für die spezifizierten Prozessoren automatisch erzeugt. Neben der Simulation, werden auch die Validierungsmethoden evaluiert, um die Einsatzfähigkeit der entwickelten Systeme und Konzepte zu demonstrieren.

Struktur der Arbeit Im zweiten Kapitel werden die Grundlagen dieser Arbeit vorgestellt. Dabei stehen die wesentlichen Konzepte aus den Bereichen Prozessorentwurf und Prozessorarchitektur im Vordergrund. Außerdem werden die grundlegenden Methoden aus dem Gebiet der Validierung mit dem Schwerpunkt Prozessorvalidierung erläutert.

Auf die verwandten Konzepte aus den Bereichen Prozessorspezifikation und Validierung wird im dritten Kapitel eingegangen. Es gibt einen Überblick der wesentlichen Eigenschaften der verwandten Sprachen und Werkzeuge aus der Forschung und stellt die entscheidenden Unterschiede zu den Konzepten in dieser Arbeit heraus.

Das vierte Kapitel beschreibt die Konzepte der entwickelten visuellen Sprache *ViCE-UPSLA* für die Spezifikation von Prozessoren. Dazu werden das Einsatzgebiet und die Aufgaben der Sprache und des Werkzeugsystems genau beschrieben. Für den Entwurf der Sprache werden den einzelnen Prozessorkonstrukten treffende Visualisierungen im Werkzeugsystem zugeordnet.

Im fünften Kapitel werden die Validierungsmethoden basierend auf dem Fehlermodell beschrieben. Dabei werden die Aufgaben der Validierung und die eingesetzten Methoden erläutert. Eine weitere Sprache *TFS* unterstützt die Konstruktion und Visualisierung der Testfallspezifikationen zur automatischen Generierung der Testfälle.

Das sechste Kapitel beschreibt die hier verwendeten Konzepte bei der Umsetzung der Generatoren, Simulatoren und der automatischen Testfallgenerierung. Dabei wird insbesondere auf die Erzeugung zyklengenauer Prozessorsimulatoren bei vorgegebener Mikroarchitektur eingegangen. Außerdem wird der Prozess der Testfallgenerierung aus der Prozessorspezifikation beschrieben.

Im siebten Kapitel werden die Sprache *ViCE-UPSLA* und die Validierungsmethoden evaluiert. Deren Bewertung basiert auf etablierten Klassifikationsschemata. Anhand von realen Prozessoren wird die Mächtigkeit der Sprache und die resultierende Simulationseffizienz untersucht. Es zeigt sich, dass das entwickelte Werkzeugsystem für die Spezifikation, die automatische Generierung von Simulatoren und Testfällen realer Prozessoren geeignet ist.

Das achte Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf zukünftige Erweiterungen der Sprache und Verbesserungen am Validierungsprozess für Prozessoren.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Konzepte aus der Domäne des Prozessorentwurfs zum Verständnis der vorliegenden Arbeit erläutert. Hierfür werden als Erstes die Begriffe des Abstraktionsniveaus sowie die verschiedene Entwurfsszenarios bei der Entwicklung von Prozessoren in Abschnitt 2.1 erläutert. In Abschnitt 2.2 werden die Prozessorkonstrukte auf einem hohen Abstraktionsniveau beschrieben, damit werden grundlegende Begriffe für die Beschreibung der Sprache *ViCE-UPSLA* zusammengefasst. Die Vielfalt realer Prozessoren erlaubt es, den Architekturen der Prozessoren Eigenschaften zuzuordnen, womit die Prozessoren klassifiziert werden können. In Abschnitt 2.3 werden verschiedene Klassen von Prozessoren vorgestellt, um die unterschiedlichen Designphilosophien bei dem Prozessorentwurf zu verdeutlichen. Der Abschnitt 2.4 beschreibt die Simulation und die verschiedenen Arten für die Implementierung von Prozessorsimulatoren. In dieser Arbeit wird die Simulation bei der Validierung von Prozessoren eingesetzt. Die Grundlagen für die Validierungsmethoden werden in Abschnitt 2.5 vorgestellt und für die Begriffe aus dem Prozessorentwurf adaptiert.

2.1 Prozessorentwurf

Der Entwurfsprozess eines Prozessors beginnt mit einem informellen Entwurf. Bis zu einem fertigen Chip wird eine Reihe von Entwicklungsschritten durchlaufen. Dabei werden beim Entwurf eines Prozessors angefangen bei der abstrakten Strukturierung der Architektur, über die Modellierung des Verhaltens, bis zur Synthese des Chips verschiedene Aufgaben gelöst. Für die Beschreibung einer systematischen Vorgehensweise müssen die Aufgaben, die Ziele und die Abstraktionsebene für die einzelnen Entwicklungsschritte definiert werden. Für die Einordnung der Sprache *ViCE-UPSLA* in den Entwurfsprozess werden in diesem Abschnitt die Abstraktionsebenen aus dem Prozessorentwurf und die unterschiedlichen Entwurfsszenarios vorgestellt.

2.1.1 Entwurfsprozess

Unabhängig von der Entwicklungsphase werden für die Durchführung eines Entwicklungsschrittes eine Systemspezifikation und die Anforderungen an das System verwendet, die der Entwurf erfüllen muss. Für die Erzeugung des Entwurfes werden Werkzeuge eingesetzt, mit denen Spezifikationen erzeugt werden. Anhand der Anforderungen

an das System wird der Entwurf auf seine Konsistenz zu der Systemspezifikation, die als Grundlage für den Entwurfsschritt verwendet wurde, überprüft. Im idealen Entwurfsprozess wird die erzeugte Spezifikation aus einer Entwicklungsphase als Systemspezifikation für die nächste Entwicklungsphase verwendet.

In Abbildung 2.1 wird symbolisch der Entwurfsablauf in einer Entwicklungsphase dargestellt. Der Prozessorentwickler verwendet eine informelle Beschreibung des Prozessors, um mit Hilfe eines Werkzeugs eine Spezifikation des Prozessors für den nächsten Entwurfsschritt zu erzeugen. Aus den Anforderungen an das System werden Benchmarks erzeugt, mit denen der Entwurf auf die Erfüllung der Anforderung geprüft werden soll. Beschreiben die Benchmarks (z. B. Programme), die auf dem Zielprozessor ausgeführt werden sollen, wird ein Simulator des Prozessors benötigt, der in der Regel aus der erzeugten Spezifikation durch das Werkzeugsystem generiert wird. Durch die Anwendung der Benchmarks im Simulator werden Ergebnisse gesammelt, die in den Entwurf des Prozessors einfließen. Wird eine Entwicklungsphase abgeschlossen, kann der Entwurf in der nächsten Entwicklungsphase analog fortgeführt werden.

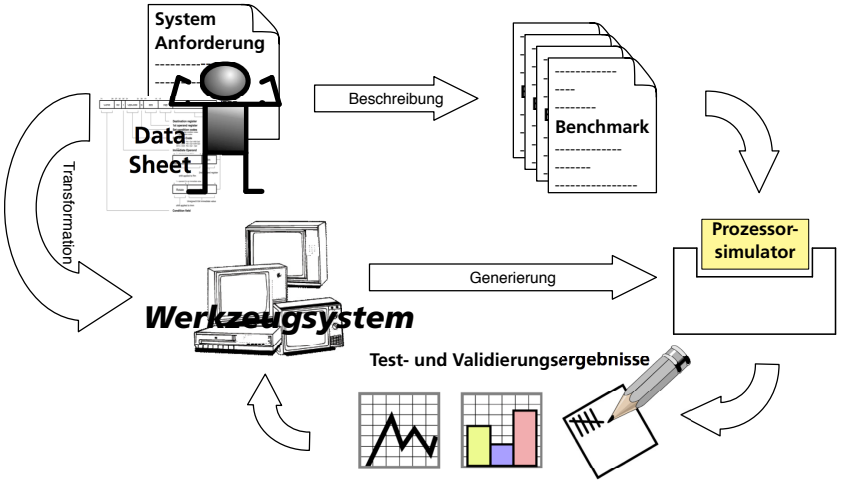


Abbildung 2.1: Entwurfsablauf.

2.1.2 Abstraktionsebenen und Entwurfsziele im Prozessorentwurf

Wie bereits beschrieben wird der Entwurfsprozess durch eine Reihe von Entwicklungsphasen auf unterschiedlichen Abstraktionsebenen durchgeführt. Die Beschreibung der Abstraktionsebenen wird für die Einordnung der Werkzeuge und der Entwurfsziele benötigt. Die Abstraktionsebenen im Prozessorentwurf können anhand des Y-Diagramms, wie in Abbildung 2.2 dargestellt, beschrieben werden. Die Definition der

Abstraktionsebenen und Sichten in einem Y-Diagramm wurde durch Daniel D. Gajski und Robert H. Kuhn [GK83] eingeführt, um die verschiedenen Modelle, die beim Prozessorentwurf entstehen, nach ihrem Abstraktionsniveau einzuordnen und damit Entwurfsmethodik zu beschreiben. Das Y-Diagramm beschreibt drei Sichten auf den Prozessor: Verhaltens-Sicht, die das Verhalten der einzelnen Komponenten des Prozessors von der Systemspezifikation bis zu Differenzialgleichungen beschreibt; die Struktur-Sicht beschreibt auf der höchsten Abstraktionsebene die Ressourcen des Systems und kann bis zum Schaltungsschema verfeinert werden; die Geometrische-Sicht beschreibt auf der äußersten Ebene die grobe Einteilung der Elemente auf dem Chip. Diese Sicht beschreibt die geometrischen Strukturen in einem Chip bis zu dem fertigen Layout aller Prozessor-Zellen, wonach der Prozessor letztendlich gefertigt wird.

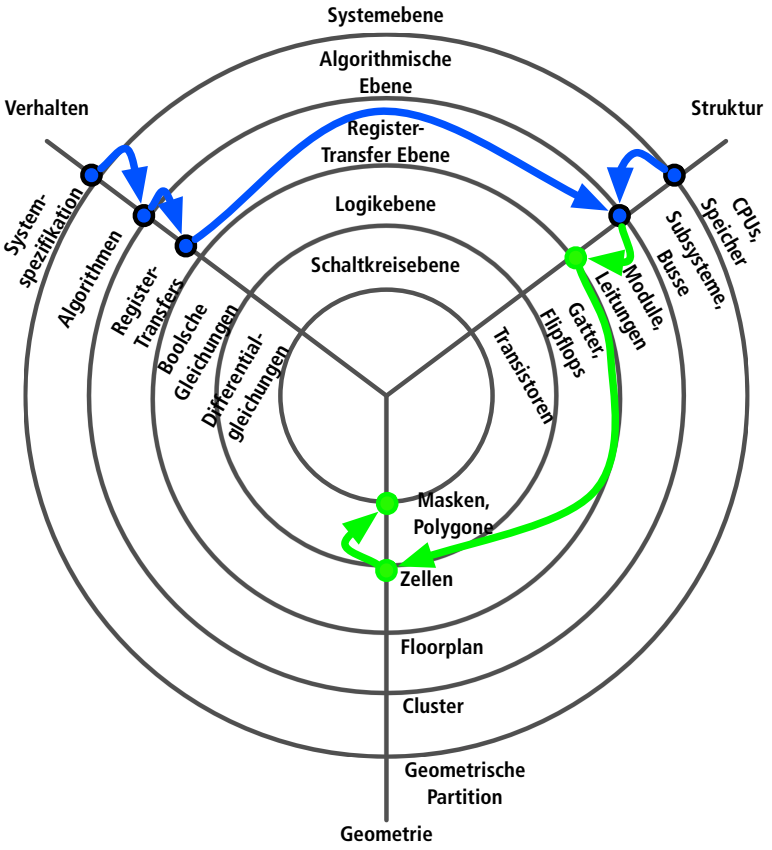


Abbildung 2.2: Y-Diagramm nach Daniel D. Gajski and Robert H. Kuhn [GK83] mit einem Entwurfsverlauf von Dieter Wecker [Wec08].

Dieter Wecker verwendet das Y-Diagramm in seinem Buch „Einführung in den Entwurfsprozess“ [Wec08] für die Definition des Entwurfsverlaufs durch die Abstraktionsebenen und Sichten. Der Entwurfsverlauf wird in Abbildung 2.2 durch Pfeile dargestellt. Damit wird beschrieben, dass die Entwicklung eines Prozessors auf der Systemebene aus der Sicht des Verhaltens und der Struktur parallel begonnen wird. In der ersten Phase wird z. B. beschrieben, nach welchem Prinzip (SISD, SIMD usw.) die Instruktionsverarbeitung im Prozessor erfolgen soll. In der algorithmischen Ebene werden der Instruktionssatz, die Anzahl der Pipelinestufen und die Adressierung des Prozessors festgelegt. Nach der Abbildung 2.2 wird der Entwurf des Prozessors aus der Sicht des Verhaltens und der Struktur bis zur Register-Transfer-Ebene vorangetrieben. Abschließend wird der Entwurf aus der geometrischen Sicht durchgeführt, womit die Synthese des Prozessors abgeschlossen wird.

Durch die Aufteilung des Entwurfsverlaufs definiert Wecker[Wec08] zwei verschiedene Ziele bei der Entwicklung, wodurch sich die Spezifikation der Komponenten eines Prozessors und die verwendeten Werkzeuge in den einzelnen Phasen unterscheiden. Wecker teilt die Aufgaben in:

- Verifikation, Model Checking und Simulation (in Abbildung 2.2 blau dargestellt)
- Realisierung, Synthese, Layout (in Abbildung 2.2 grün dargestellt)

Diese Aufteilung wird später dazu verwendet, die Werkzeuge zu klassifizieren und den Ansatz von *ViCE-UPSLA* einzuordnen.

2.1.3 Entwurfsszenarios

Der Abschnitt 2.1.2 beschreibt die Vorgehensweise beim Prozessorentwurf von den Anforderungen bis zur Fertigung eines Chips. Die Anforderung und die Einsatzgebiete der Prozessoren sind sehr vielfältig, sodass die Entwicklung von speziellen Prozessoren für bestimmte Aufgaben vorangetrieben wird. In diesem Zusammenhang wird oft über ASIP (Application-Specific Instruction-set Processor) [HL10] Prozessoren gesprochen. Aktuelle Prozessoren wie ARM oder CoreVA sind meist sehr komplex, sodass eine Neuentwicklung eines Prozessors kaum rentabel ist. Für die Reduzierung des Entwicklungsaufwandes werden bereits vorhandene Architekturen verwendet. Für den Entwurf eines anwendungsspezifischen Prozessors werden die Methoden des ISE (Instruktionssatzerweiterung) [KLST04] oder DSE (Design Space Exploration oder Entwurfsraumexploration) [JDP+10, JSPR10] angewendet.

Die Instruktionssatzerweiterung wird durchgeführt, um z. B. die Geschwindigkeit der Datenverarbeitung eines Prozessors zu steigern, ohne grundlegende Architektur des Prozessors zu verändern. Dabei wird für spezifische Anwendungen der Instruktionssatz so angepasst, dass eine Beschleunigung der Verarbeitung im Prozessor gesteigert wird. Eine Methode dazu wird in „Feedback Driven Instruction-Set Extension“ [KLST04] beschrieben.

Bei der Entwurfsraumexploration wird die Mikroarchitektur eines Prozessors erweitert. Neben der Erhöhung der Anzahl von Pipelinestufen, womit der Instruktionsdurchsatz gesteigert werden kann oder der Vergrößerung des Registersatz, wird bei DSE die parallele Verarbeitung von Instruktionen umgesetzt. Dabei werden z. B. die Ressourcen der Mikroarchitektur und der Datenpfad erweitert, sodass mehrere Instruktionen in unabhängigen Funktionseinheiten verarbeitet werden.

2.2 Prozessorarchitektur

Die grundlegenden Konzepte und Begriffe aus dem Bereich des Prozessorentwurfs werden von John L. Hennessy und David A. Patterson in dem Buch „Computer Architecture. A Quantitative Approach“ [HP06] oder Christian Siemers in seinem Skript „Rechnerarchitektur I/II“ [Sie12] zusammengefasst, indem verschiedene Prozessorarchitekturen und ihre Konstrukte auf verschiedenen Abstraktionsebenen beschrieben werden. Unter dem Begriff Prozessorarchitektur werden, abhängig von der Entwicklungsphase, unterschiedliche Eigenschaften für einen Prozessor angegeben. So kann z. B. für einen Prozessor der Instruktionssatz, Datenpfad oder die Chipfläche und der Stromverbrauch betrachtet werden. In diesem Abschnitt werden die für das Verständnis des Ansatzes dieser Arbeit notwendigen Prozessorkonstrukte auf einem hohen Abstraktionsniveau zusammengefasst.

2.2.1 Mikroarchitektur

Die Mikroarchitektur eines Prozessors beschreibt abstrakt die Struktur des Prozessors. In dieser Arbeit wird der Begriff der Mikroarchitektur für die zusammenfassende Betrachtung der Pipelinestruktur und des Datenpfads eines Prozessors verwendet. Diese Komponenten werden in folgenden Unterabschnitten detailliert erläutert. Auf die verschiedenen Ausprägungen der Mikroarchitekturen für unterschiedliche Prozessoren wird in Abschnitt 2.3 eingegangen.

2.2.1.1 Datenpfad

Auf einem hohen Abstraktionsniveau wird der Datenpfad eines Prozessors durch drei Komponenten angegeben: Funktionsblöcke, Busse und Lese- und Schreib-Ports. Dabei werden die Funktionsblöcke und die Ports als Knoten eines Datenflussgraphen verwendet und durch die Busse als gerichtete Kanten miteinander verknüpft. In Abbildung 2.3 wird ein einfacher Datenpfad dargestellt.

Die Funktionsblöcke in einer Mikroarchitektur werden z. B. durch arithmetisch-logische Einheiten (kurz ALU) oder Multiplexer (kurz MUX) dargestellt. Die Multiplexer beschreiben einfache Funktionsblöcke, mit denen z. B. die Adressierungsmethoden des Prozessors im Datenpfad beschrieben werden. Die Ausprägung der Spezifikation

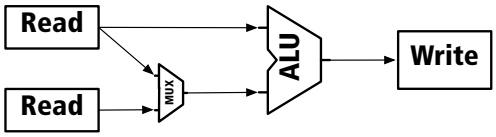


Abbildung 2.3: Beispiel eines einfachen Datenpfads.

einer ALU kann je nach Entwurfsstadium unterschiedlich sein. In frühen Entwicklungsphasen beschreiben die ALUs in einem Datenpfad die Knoten, in denen die Berechnungen der Instruktionen durchgeführt werden. In späteren Entwicklungsphasen, z. B. für die Synthese des Prozessors, werden die ALUs durch eine Menge von Operationen beschrieben, die in dem Funktionsblock durchgeführt werden können. Dabei werden z. B. arithmetische und logische Operationen angegeben.

Die Lese- und Schreib-Ports beschreiben die Anzahl der Registerwerte, die bei der Ausführung einer Instruktion geladen oder geschrieben werden können.

2.2.1.2 Pipeline

Die Ausführung einer Instruktion setzt sich aus einer Reihe von Aktionen zusammen, die separat betrachtet werden können. Diese Eigenschaft wird für die Erhöhung des Instruktionsdurchsatzes ausgenutzt, indem die Aktionen der Instruktionen in Schritte aufgeteilt werden. Die Pipeline wird durch Pipelinestufen des Prozessors beschrieben. Durch die Angabe eines Datenpfads werden die Ressourcen der einzelnen Pipelinestufen definiert. Für die Beschreibung einer Pipeline werden z. B. die Stufen *Fetch-Decode-Load-Execute-Store* verwendet. In Abbildung 2.4 werden eine Pipeline mit den durchführbaren Aktionen und die überlappende Ausführung von Instruktionen dargestellt. Damit wird jede Instruktion in fünf Schritten ausgeführt, so dass nach der Bearbeitung der Instruktion in der Stufe Fetch, die Bearbeitung in der Stufe Decode fortgeführt wird. Sobald die Pipelinestufe Fetch durch eine Instruktion wieder freigegeben wird, kann die nachfolgende Instruktion die Bearbeitung in dieser Stufe beginnen usw. Damit können mehrere Instruktionen nach dem Fließbandprinzip überlappend in der Pipeline ausgeführt werden.

2.2.1.3 Bypass

Die Erhöhung des Durchsatzes durch die überlappende Ausführung bedingt eine Latenz von mehreren Taktzyklen für die Ergebnisse der Instruktionen. Dadurch können Datenkonflikte (Data-Hazards [HP06]) zwischen den Instruktionen entstehen. Die Datenkonflikte entstehen dann, wenn z. B. eine nachfolgende Instruktion das noch nicht

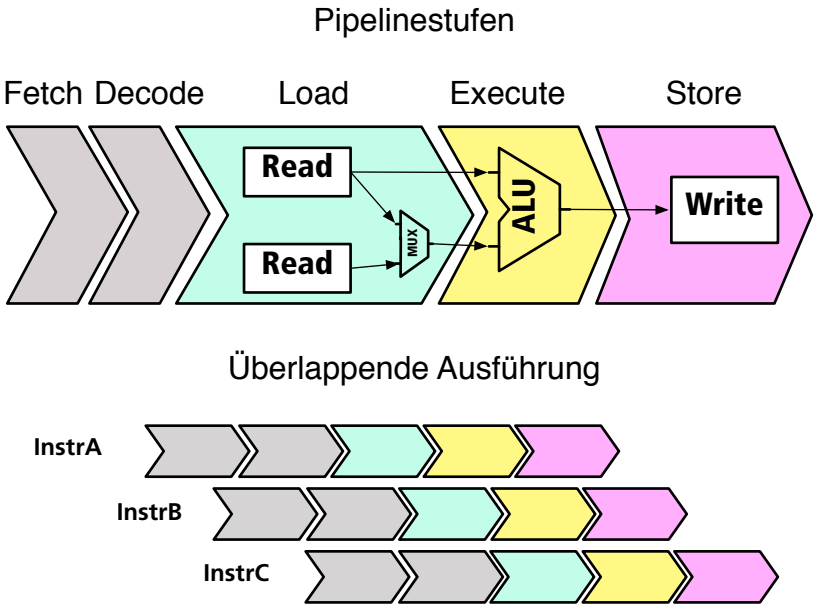


Abbildung 2.4: Mikroarchitektur eines Prozessors mit einer fünfstufigen Pipeline und die Überlappende Ausführung der Instruktionen (A,B,C) in der Pipeline.

zurückgeschriebene Ergebnis ihres Vorgängers benötigt. In Abbildung 2.5 wird eine Konfliktsituation zwischen zwei Instruktionen dargestellt. Die Instruktion `InstrB` wird nach der Instruktion `InstrA` ausgeführt. Die Instruktion `InstrA` verwendet die Operanden `y,z` und schreibt das Ergebnis in den Operanden `x`, während die Instruktion `InstrB` die Operanden `w,x` verwendet und das Ergebnis in den Operanden `v` schreibt. Die Instruktion `InstrB` benötigt das Ergebnis von `InstrA`. Bei einer überlappenden Verarbeitung wird der Konflikt durch den Operanden `x` ausgelöst.

Für die Auflösung der Datenkonflikte werden Bypässe in die Pipelinestruktur zwischen den Funktionsblöcken eingesetzt. Damit werden die Werte direkt nach der Berechnung an die nachfolgenden Instruktionen weitergeleitet, bevor sie zurück in die Register geschrieben wurden. Wie in Abbildung 2.6 dargestellt löst der eingesetzte Bypass die Datenkonflikte in der Pipelinestruktur aus Beispiel 2.5, indem der geladene `x`-Wert bei der Ausführung der `InstrB` durch den neu berechneten `x`-Wert in der `InstrA` ausgetauscht wird.

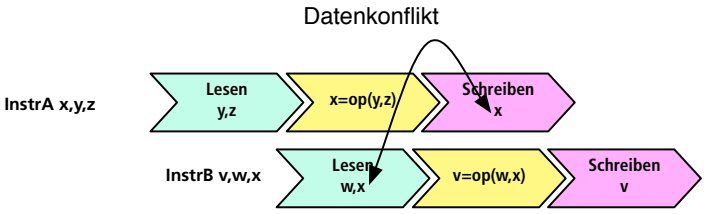


Abbildung 2.5: Datenkonflikt, ausgelöst durch den Operanden `x`.

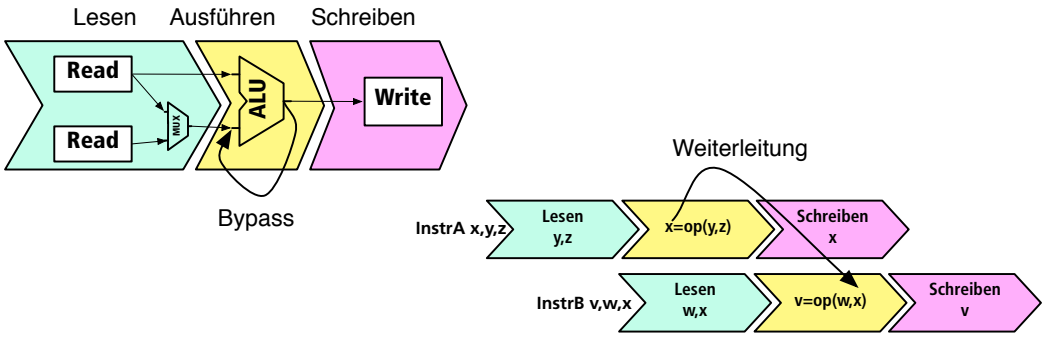


Abbildung 2.6: Weiterleitung eines Wertes durch einen Bypass zur Auflösung des Datenkonfliktes.

2.2.1.4 Interlocking

Bei der Ausführung von Instruktionen in einer Pipeline können Ressourcenkonflikte (Resource-Hazard [HP06]) entstehen, wenn mehrere Instruktionen zum selben Zeitpunkt eine Ressource der Mikroarchitektur, z. B. eine ALU, benötigen. Für die Auflösung der Ressourcenkonflikte wird das Prinzip des Interlockings in den Pipelines angewendet. Damit ist die Pipeline in der Lage, die Ausführung einer Instruktion anzuhalten, bis die benötigte Ressource zur Verfügung steht.

In Abbildung 2.7 wird das Prinzip des Interlock-Mechanismus dargestellt. Die Ausführung der Instruktionen `InstrA` und `InstrB` kann nacheinander gestartet werden. Die Instruktion `InstrA` wird in der *Execute* Stufe in zwei Taktzyklen ausgeführt, wodurch ein Ressourcenkonflikt zwischen den Instruktionen besteht, wenn beide Instruktionen die ALU in der *Execute* benötigen. Der Interlock-Mechanismus hält die Ausführung der Instruktion `InstrB` solange an, bis die benötigte Ressource zur Verfügung steht.

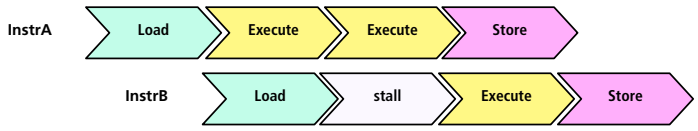


Abbildung 2.7: Anhalten der Ausführung einer Instruktion bei besetzen Ressourcen.

2.2.2 Instruktionssatz

Auf einem hohen Abstraktionsniveau wird der Instruktionssatz eines Prozessors durch die Menge der Instruktionen angegeben. Für die Strukturierung des Instruktionssatzes werden die Instruktionen oft in Instruktionsklassen [HP06], wie Integer, Floating-point oder Load-/Store-Instruktionen, eingeteilt. Auf der nächsten Abstraktionsebene werden in technischen Dokumentationen [Lim87, Mic94] die Eigenschaften der Instruktionen in mehreren Abschnitten für verschiedene Aspekte angegeben. Dabei werden z. B. die Kodierung, die verwendeten Ressourcen oder das zeitliche Verhalten der Instruktionen spezifiziert. Im Folgenden werden die Konstrukte für die Beschreibung von Instruktionen zusammengefasst.

Eine Instruktion wird durch ihre Struktur und das Verhalten charakterisiert. Für die Beschreibung der Struktur einer Instruktion werden meist ihre Binärkodierung und das Assemblerformat angegeben. Eine Instruktion wird in der Regel eindeutig durch den Operationscode in der Binärcodierung und den Bezeichner im Assemblerformat identifiziert, diese Eigenschaft wird jedoch in einigen realen Instruktionssatzarchitekturen verletzt. Außerdem werden bei der Kodierung der Instruktion die Operanden und deren Adressierungsmethoden angegeben. Die Adressierungsmethoden werden im folgenden Abschnitt 2.2.3 ausführlich beschrieben.

In Abbildung 2.8 wird ein Ausschnitt aus dem ARM-Datenblatt [Lim95] dargestellt, in dem die Kodierung für die arithmetischen und logischen Instruktionen abgebildet ist. Die Instruktionen besitzen ein `Conditional Field` für die bedingte Ausführung der Instruktion sowie drei Operanden `Rn`, `Rd` und `Operand 2`. Außerdem werden die Steuerbits `I` und `S` angegeben. Die Steuerbits beschreiben optional das Verhalten der Instruktionen, z. B. wird durch das Steuerbit `I` die Wahl der Adressierungsmethode für `Operand 2` gesteuert.

Die Verhaltensbeschreibung von Instruktionen kann auf einem hohen Abstraktionsniveau durch die Berechnungsvorschriften angegeben werden. In Abbildung 2.8 wird neben dem `Operation Code` die Operation der jeweiligen Instruktionen angegeben. Dabei wird z. B. für die Instruktion `AND` neben dem Operationscode das Ergebnis `Rd` aus der logischen UND-Verknüpfung von `OP1` und `OP2` beschrieben. Auf einer niedrigeren Abstraktionsstufe kann das Verhalten der Instruktionen zyklengenau durch eine ope-

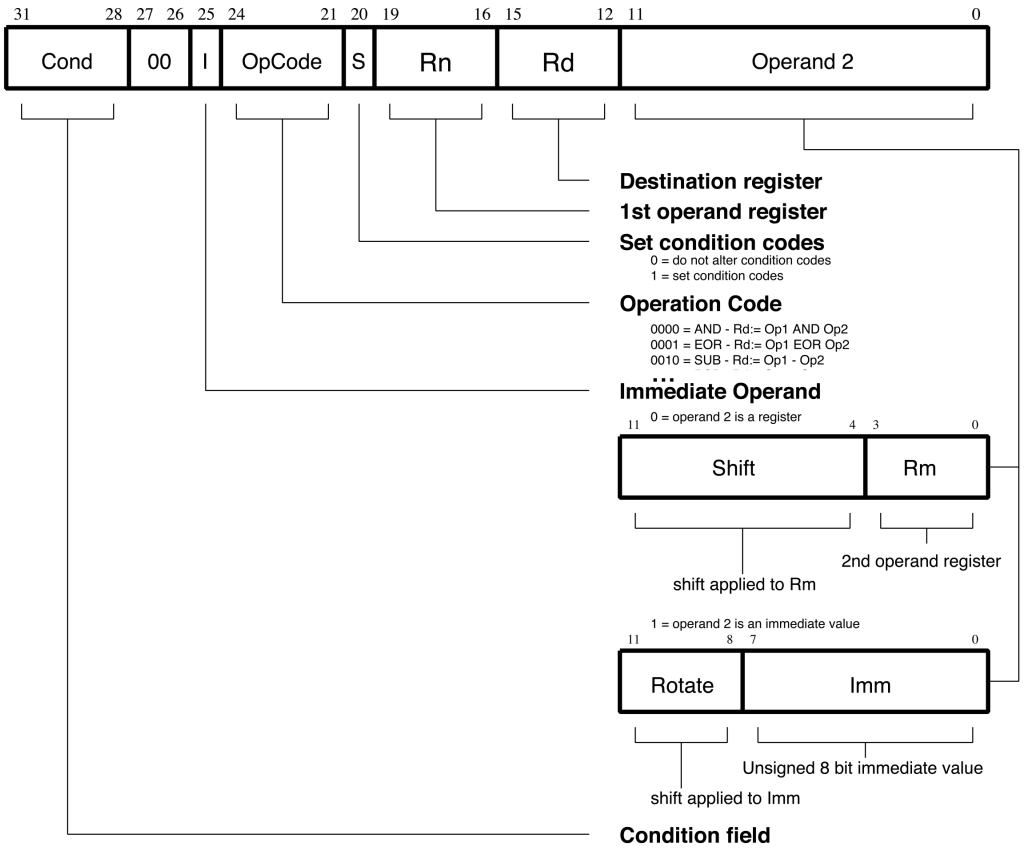


Abbildung 2.8: Ausschnitt aus dem ARM-Datenblatt [Lim87].

rationale Beschreibung angegeben werden. Die operationale Beschreibung beschreibt neben der Berechnungsvorschrift für das Ergebnis der Instruktion die Reihenfolge für die Ausführung von Aktionen, die durch eine Instruktion ausgelöst werden. Die operationale Beschreibung wird oft verknüpft zu der Mikroarchitektur angegeben und beschreibt die Wege einer Instruktion durch den Datenpfad des Prozessors.

2.2.3 Adressierungsmethoden

Die Adressierungsmethoden beschreiben das Laden oder Speichern der Werte, die bei der Berechnung in einer Instruktion verwendet oder erzeugt werden. Für die Beschreibung der Adressierungsmethoden werden in dieser Arbeit die Begriffe für die *Befehlsoperanden* und die *interpretierten Operanden* definiert. Die Begriffe werden anhand der Auswertung eines Instruktionswortes erklärt.

Ein Instruktionswort kann z. B. wie folgt angegeben sein `add reg0, reg1, reg2`. Damit wird angegeben, dass die Instruktion `add` ausgeführt werden soll und die Summe der gespeicherten Werte aus den Registern `reg1` und `reg2` im Register `reg3` speichern soll. Die angegebenen Operanden `reg0, reg1, reg2` sind Befehlsoperanden und beschreiben Adressen im Registersatz des Prozessors, an denen sich die Werte für die Berechnung befinden. Mit einer Adressierungsmethode wird ein Wert aus einem Register geladen und in den interpretierten Operand gespeichert. In der Hardware werden hierfür die Pipelineregister verwendet. Die Berechnung der Ergebnisse in der Instruktion wird auf den interpretierten Operanden durchgeführt. Damit beschreibt eine Adressierungsmethoden eine Funktion, mit der die Werte aus dem Registersatz geladen oder gespeichert werden.

Der Abbildung 2.8 kann entnommen werden, dass die Befehlsoperanden `Rn` und `Rd` direkt adressiert werden, indem Werte aus dem Befehlswort Registeradressen des Prozessors beschreiben. Der Operand `Operand 2` wird durch indirekte Adressierung beschrieben, indem in der Adressierung `Shift_Rm` der geladene Wert durch einen Shift Operator um einige Bits verschoben wird. Analog dazu wird in der Adressierung `Rotate_Imm` ein unmittelbarer Wert rotiert.

Bei den Adressierungsmethoden wird zwischen direkten und indirekten Adressierungsmethoden unterschieden. Die direkten Adressierungsmethoden zeichnen sich dadurch aus, dass die Werte für die interpretierten Operanden mit der Adresse aus den Befehlsoperanden unverändert geladen werden. Die indirekten Adressierungsmethoden führen zusätzliche Berechnungen durch, womit z. B. beim Laden des Wertes die Adresse aus dem Befehlsoperanden zusätzlich manipuliert wird, um z. B. bestimmte Adressbereiche zu erreichen. Außerdem gibt es bei den indirekten Adressierungsmethoden den Unterschied zwischen `Pre load` und `Post load` Methoden. Aus den unterschiedlichen Berechnungsvorschriften ergeben sich verschiedene Klassen für die Einordnung der Adressierungsmethoden. In Tabelle 2.1 wird eine Übersicht über die Adressierungsmethoden dargestellt.

Tabelle 2.1: Klassen von Adressierungsmethoden `Addr(instrOpd)`.

Befehlsoperand	Direkt	Indirekt	
		Pre load	Post load
Register	a) `load(reg)`	b) `load(f(reg))`	c) `f(load(reg))`
Immediate	d) `(imm)`	-	e) `f(imm)`
Register&Immediate	-	f) `load(f(reg,imm))`	g) `f(load(reg),imm)`

Die Benutzung eines einzelnen Befehlsoperanden als Parameter in einer Adressierungsmethode ist die Voraussetzung für eine direkte Adressierungsmethode. Wird der

Wert aus dem entsprechenden Registeroperanden oder immediate Operanden direkt verwendet, handelt es sich um eine direkte Adressierungsmethode (Tabelle 2.1a),b)). Die direkte Verwendung von Werten aus den Befehlsoperanden bedeutet, dass z. B. der Wert eines immediate Operanden bei der Simulation unverändert in die Berechnung der Instruktion übernommen wird.

Bei indirekten Adressierungsmethoden wird eine Berechnung durchgeführt. Die *Pre* oder *Post load* Methoden definieren die Reihenfolge, in der die Aktionen der Adressierungsmethode durchgeführt werden.

Die Spezifikation einer *Pre load* Methode gibt an, dass zuerst die Befehlsoperanden ausgewertet werden, wodurch die Adresse des Zielregisters bestimmt wird. Das Ergebnis aus der Berechnung beschreibt eine Registeradresse, in der sich der Wert für den interpretierten Operanden befindet. Diese Methode kann z. B. eingesetzt werden, um verschiedene Adressräume zu beschreiben (Tabelle 2.1b),f)).

Bei einer *Post load* Adressierungsmethode wird erst der Wert aus dem Register geladen und anschließend durch die Berechnungsvorschrift der Adressierungsmethode verändert. Das Ergebnis der Berechnung ist der Wert für den interpretierten Operanden, siehe (Tabelle 2.1c),e)). Eine der häufig verwendeten *Post load* Methoden ist z. B. das Verschieben des geladenen Wertes um einige Bitstellen.

2.2.4 Register und Registersatz

Die Register sind Speicherstellen, welche bei der Ausführung von Programmen zur Speicherung der Zwischenwerte oder für das Halten von Variablen verwendet werden. Die Register werden meistens in Registerbänken zusammengefasst. Für die Spezifikation einer Registerbank werden die Anzahl, die Breite und die Bezeichner der Register angegeben.

Der Registersatz eines Prozessors beschreibt allgemein alle physikalischen Register und Registerbänke eines Prozessors. Bei der Spezifikation werden die physikalischen Register oft in Gruppen eingeteilt, wobei z. B. Register für den System-Mode oder User-Mode, wie bei ARM- oder PowerPC-Prozessoren [Lim95, Mic94], angegeben werden. Die Aufteilung der Register stellt dar, welche Register in welchen Modi des Prozessors für den Programmierer sichtbar sind. Z. B. wird die Organisation des Registersatzes für den ARM-Prozessor in Tabelle 2.2 dargestellt.

Für die Beschreibung verschiedener Konfigurationen des Registersatzes werden architektonische Register angegeben. Die architektonischen Register [DHTK07] beschreiben durch Aliasing alternative Anordnungen der physikalischen Registerbänke, wie z. B. in der Dokumentation des *PowerPC* [Mic94] zu finden.

Tabelle 2.2: Register Organisation [Lim87].

user mode	svc mode	irq mode	fiq mode
R0			
...			
R7			
R8			R8_fiq
...			...
R12			R12_fiq
R13	R13_svc	R13_irq	R13_fiq
R14	R14_svc	R14_irq	R14_fiq
R15 (PC/PSR)			

2.3 Prozessorarchitekturen

Die Sprache *ViCE-UPSLA* soll dazu eingesetzt werden, verschiedene reale Prozessoren zu spezifizieren. Für die Erläuterung der Sprachkonstrukte für den Prozessorentwurf werden in diesem Abschnitt einige Design-Philosophien für unterschiedliche Prozessorarchitekturen vorgestellt. Damit soll die Vielfältigkeit der Prozessorarchitekturen gezeigt werden und zum Verständnis der entworfenen Sprachkonstrukte von *ViCE-UPSLA* dienen. Für die Vorstellung der realen Prozessorarchitekturen werden zunächst die verschiedenen Möglichkeiten für die Klassifizierung von Prozessoren eingeführt.

Die erste Klassifizierung kann anhand der Struktur des Instruktionssatzes von Prozessoren beschrieben werden [Jam95]. Dabei wird zwischen CISC (Complex Instruktion Set Computer) [NMEH81, SG79] und RISC (Reduced Instruktion Set Computer) [Kan88, Mic94, Lim95] unterschieden.

Die CISC-Prozessoren zeichnen sich in erster Linie durch einen großen und komplexen Instruktionssatz aus. Die Komplexität des Instruktionssatzes entsteht z. B. durch die unterschiedlichen Befehlsbreiten und komplexe Instruktionen. Außerdem besitzen die CISC-Prozessoren in der Regel einen großen Registersatz, auf den mit komplexen Adressierungsmethoden zugegriffen wird. Im Gegensatz zu CISC-Prozessoren besitzen die RISC-Prozessoren meist einen kleineren Instruktionssatz mit wohl strukturierten Instruktionen.

Die Klassifizierung durch den Instruktionssatz erfolgt lediglich anhand der Betrachtung der strukturellen Eigenschaften des Instruktionssatzes oder der Adressierungsmethoden. Die Klassifizierung von Prozessoren, abgesehen von der Struktur des Instruktionssatzes, kann durch die Betrachtung der Verarbeitung von Daten- und Instruktionsströmen nach Michael J.Flynn und Kevin W.Rudd [FR96] durchgeführt werden.

Hierbei wird auf die Anzahl der parallel verarbeiteten Daten- und Instruktionsströme eingegangen. Dabei wird zwischen folgenden Möglichkeiten unterschieden:

- SISD — Single Instruction Single Data
- SIMD — Single Instruction Multiple Data
- MIMD — Multiple Instruction Multiple Data
- MISD — Multiple Instruction Single Data

Die SISD-Verarbeitung entspricht der konventionellen Verarbeitung in einer Prozessorarchitektur, indem einen Strom von Instruktionen auf ein Strom von Eingabedaten angewendet wird. Bei SIMD-Prozessoren kommen die Instruktionsworte, wie bei SISD-Architekturen, aus einem Strom. Bei der Ausführung einer SIMD-Instruktion werden vektorisierte Datenströme verarbeitet, sodass durch eine Instruktion mehrere Datenströme parallel verarbeitet werden. In diesem Zusammenhang wird von Vektorinstruktionen gesprochen. MIMD-Architekturen beschreiben eine hohe Parallelität. Dabei werden Daten aus mehreren Datenströmen durch die entsprechende Anzahl von parallel ausgeführten Instruktionen verarbeitet. Nach diesem Prinzip können auch MISD-Architekturen beschrieben werden, wobei parallel ausgeführte Instruktionen auf einen Datenstrom angewendet werden.

2.3.1 ARM

Die ARM-Architektur wurde von der Firma Advanced RISC Machines entwickelt. Die ARM-Familie besitzt heute bereits die elfte Genration von ARM-Prozessoren, der erste *ARM1* (Acorn RISC Machine) wurde 1985 entwickelt. Der Instruktionssatz der ARM-Prozessoren wurde seit der siebten Generation (ARM7) bis zum heutigen *ARM11 Cortex* nicht verändert. In dieser Arbeit wird die ARM7-Architektur verwendet, die als *Open Access* Dokument [Lim95] veröffentlicht ist.

ARM ist eine klassische SISD/RISC-Architektur, sie wird aktuell in hoher Verbreitung in mobilen Geräten eingesetzt. Die 32-Bit Architektur des Prozessors besitzt eine charakteristisch übersichtliche Anzahl von wohl strukturierten Instruktionen. Die Instruktionen werden in Lade- und Speicher-Instruktionen, Sprunginstruktionen und die arithmetischen und logischen Instruktionen aufgeteilt.

Der Registersatz des Prozessors beschreibt 16 32-Bit Register. Das Register Nr. 15 ist das `Programmcounter` (PC) Register. Das Schreiben durch beliebige Instruktionen des Prozessors in das Register 15 bedingt einen Sprung, dennoch verfügt der ARM-Prozessor über die Sprungbefehle `Branch` (b) und `Branch with link` (bl). Mit diesen Instruktionen werden relative und absolute Sprünge umgesetzt.

Für den Zugriff auf den Registersatz des Prozessors werden neben den direkten Adressierungsmethoden zwei indirekte Adressierungsmethoden verwendet, wie in Abschnitt 2.2.3 vorgestellt. Als indirekte Adressierungsmethoden werden `Shift_Rm` und

`Rotate_Imm` eingesetzt. Bei der Adressierungsmethode `Shift_Rm` werden zwei Befehlsoperanden verarbeitet. Dabei wird ein unmittelbarer Wert aus dem Instruktionswort für die Berechnung des interpretierten Operanden verwendet, indem auf den geladenen Wert eine `Shift`-Operation angewendet wird. Für die `Shift` Operation kann dabei zwischen verschiedenen Methoden gewählt werden, z. B. `logical_shift_left` oder `arithmetical_shift_right` usw. Analog dazu werden für die Adressierungsmethode `Rotate_Imm` verschiedene Methoden für die Rotation des unmittelbaren Wertes aus dem Instruktionswort angegeben.

Zu den Eigenschaften des Prozessors zählt zum einen der Interlock-Mechanismus, wobei die Ausführung der Instruktionen bei unzureichenden Ressourcen angehalten wird. Zum anderen werden alle Instruktionen des Prozessors bedingt ausgeführt. Dazu wird in jeder Instruktion ein `Conditional Field` definiert, welches vor der Ausführung der Instruktion geprüft wird. Damit sind z. B. auch alle Sprungbefehle des Prozessors bedingt ausführbar.

2.3.2 CoreVA

Der CoreVA-Prozessor wurde in dieser Arbeit für die Evaluierung der Spezifikationssprache *ViCE-UPSLA* und der Validierungsmethoden verwendet. Für die Beschreibung der Vorgehensweise bei dem Entwurf der Spezifikation und der Validierung werden zunächst die Eigenschaften der Architektur und des Instruktionssatzes vorgestellt.

Die CoreVA-Architektur wurde in der Arbeitsgruppe Schaltungstechnik an den Universitäten Paderborn und Bielefeld entwickelt [JSPR10, Jun11, JD11]. Die Architektur basiert auf dem Эльбрус2000 (Elbrus2000) [Bab13, MCS13] Prozessor, der als Kern eine VLIW-Architektur besitzt, vergleichbar mit dem *Crouso*-Prozessor der Firma *Transmeta* [Mor06].

Die Mikroarchitektur des Prozessors wird durch sechs Pipelinestufen beschrieben, wie in Abbildung 2.9 dargestellt. Die Besonderheit der Mikroarchitektur sind vier inhomogene Funktionseinheiten in der Execute-Pipelinestufe des Prozessors. Damit beschreibt CoreVA eine typische VLIW (Very Long Instruction Word) Architektur [Pie95]. Diese Architektur erlaubt die Ausführung von VLIW-Befehlen. Die Befehle sind 128-Bit breit und werden nach dem MIMD-Prinzip verarbeitet. Dabei werden vier verschiedene Datenströme durch vier verschiedene Instruktionsströme bearbeitet.

Der Instruktionssatz des CoreVA-Prozessors beschreibt 32-Bit breite RISC-Instruktionen, vergleichbar mit dem Instruktionssatz des ARM-Prozessors. Zusätzlich zu den RISC/SISD-Instruktionen besitzt der Prozessor arithmetische und logische Vektorinstruktionen, welche nach dem Prinzip der SIMD-Verarbeitung parallel zwei 16-Bit Operationen durchführen. Diese Eigenschaft erlaubt die parallele Verarbeitung von zwei Datenströmen in nur einer ALU. Alle vier in der Architektur beschriebenen Funktionseinheiten verfügen über Vektorinstruktionen, womit die Verarbeitung auf bis zu acht parallel ausgeführte Operationen gesteigert wird. Der Instruktionssatz des

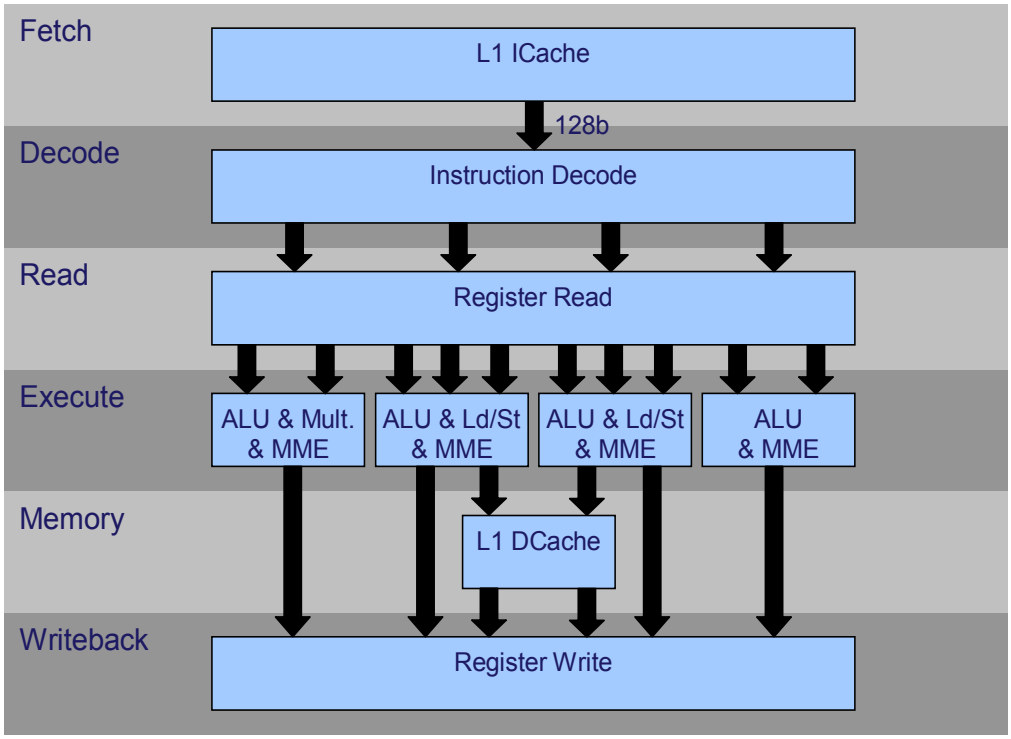


Abbildung 2.9: Mikroarchitektur des CoreVA VLIW Prozessors [JD11].

CoreVA-Prozessor wird ausführlich bei der Evaluierung der Sprache *ViCE-UPSLA* in Abschnitt 7 vorgestellt.

Die Adressierungsmethoden des Prozessors beschreiben, neben den direkten Adressierungsmethoden für den Zugriff auf den Registersatz, komplexe `Pre load` und `Post load` Methoden. Das Prinzip der Methoden wurde in Abschnitt 2.2.3 bereits beschrieben.

2.3.3 MIPS

Die MIPS Mikroprozessor Architektur (Microprocessor without Interlocked Piep Stages) [Kan88] wurde an der Universität Stanford [HKP+82] entwickelt. Die Mikroarchitektur des Prozessors wird durch übliche fünf Pipelinestufen beschrieben und der Instruktionssatz wird durch die 32-Bit logische, arithmetische, Sprung- und Speicher-Instruktionen definiert. Damit entspricht die MIPS Architektur dem typischen Bild

eines RISC-Prozessoren.

Der wesentliche Unterschied zu üblichen RISC-Prozessoren besteht in der einfachen Hardwareimplementierung. In der Mikroarchitektur des Prozessors wird auf die Interlock-Mechanismen in der Pipeline verzichtet. Damit wird die Implementierung der Hardware vereinfacht. Gleichzeitig können bei der Ausführung von Programmen Ressourcenkonflikte entstehen. Der Ansatz sieht vor, dass die Behandlung der Ressourcenkonflikte auf der Softwareseite durch Compiler durchgeführt wird.

2.4 Prozessorsimulator

Die Prozessorsimulatoren werden dazu verwendet, die Entwürfe der Prozessoren oder der Werkzeuge für die Prozessoren zu validieren. Für die Einordnung der Sprache *ViCE-UPSLA* und der damit erzeugten Simulatoren wird in diesem Abschnitt die Klassifikation der Simulatoren beschrieben. Hierzu werden unterschiedliche Simulationsebenen und Simulationsarten für die Simulatoren identifiziert. Die Simulationsebene beschreibt die Detailechtheit eines Simulators, mit der das Verhalten eines Prozessors nachgebildet wird. Die Simulationsart beschreibt die Umsetzung des Simulators, wie das Verhalten des Prozessors implementiert wird.

2.4.1 Simulationsebene

Die Simulationsebenen können in die Instruktionssatz- und die Mikroarchitektur-Simulation aufgeteilt werden, damit wird die Genauigkeit der Simulation im Vergleich zum echten Prozessor beschrieben. Die Wahl der Simulationsebene hängt von den Entwurfszielen in den jeweiligen Entwicklungsphasen ab. Die Genauigkeit der Simulation verhält sich in der Regel umgekehrt zu der Simulationsgeschwindigkeit.

Ein Instruktionssatzsimulator [Jer00, LEL99] imitiert das Verhalten der Instruktionen eines Prozessors, womit die Ausführung der Programme für den Zielprozessor ermöglicht wird. In einem Instruktionssatzsimulator werden zusätzlich zu dem Instruktionssatz in der Regel die Adressierungsmethoden und der Registersatz des Prozessors simuliert. Dabei wird von der überlappenden oder parallelen Verarbeitung in einer Pipeline abgesehen. Diese Art von Simulatoren kann in einer frühen Phase der Prozessorentwicklung eingesetzt werden, um die Eignung des Instruktionssatzes für die Anwendungen aus dem Einsatzgebiet zu prüfen. Die Simulation des Instruktionssatzes ist nur bedingt für die Validierung einer Prozessorspezifikation geeignet.

In einem Mikroarchitektursimulator [PC81] wird die Instruktionsausführung entsprechend der Mikroarchitektur des Prozessors durchgeführt. Damit können die Simulatoren für eine umfassende Validierung der Entwürfe eingesetzt werden. Ein Mikroarchitektur-Simulator kann zudem unterschiedliche Detailstufen des Prozessors abbilden. Ein Mikroarchitektursimulator kann z. B. dazu verwendet werden, die überlappende oder parallele Ausführung von Befehlen in einem Prozessor zu simulieren. Damit

ist die Validierung des Prozessors auf einer hohen Abstraktionsebene im Prozessorentwurf möglich. Die Mikroarchitektursimulatoren können mit einem Detaillierungsgrad bis zur Simulation der Schaltkreise des zukünftigen Prozessors erzeugt werden. Damit wird z. B. die Evaluierung des Stromverbrauchs oder der Taktfrequenz des Prozessors ermöglicht. Auf dieser Simulationsebene ist die Simulationsgeschwindigkeit in der Regel sehr niedrig.

2.4.2 Simulationsart

Mit *ViCE-UPSLA* werden Prozessoren spezifiziert und kompilierende Simulatoren aus der Spezifikation generiert. Für die Erläuterung der Generierung und der Funktionsweise der Simulatoren werden im Folgenden die gängigen Simulationsarten vorgestellt. Bei den Simulatoren wird zwischen interpretierenden, kompilierenden und Binärtransformation Simulatoren [ZTM95, LEL99, PD08] unterschieden.

Ein interpretierender Simulator basiert auf einem Modell des Prozessors, in dem die Programme bei der Simulation ausgeführt werden. Für die Erzeugung des Simulators werden die Systeme des Prozessors als ausführbare Einheiten nachgebildet und verarbeiten die Eingabedaten und Instruktionen. Bei der Verarbeitung von Instruktionen werden diese, wie in einem realen Prozessor, aus dem Speicher geladen, decodiert und ausgeführt. Der Spezifikationsaufwand für einen interpretierenden Simulator ist in der Regel sehr hoch. Durch das Laden und Dekodieren von Instruktionen nimmt die Simulationsgeschwindigkeit mit der steigenden Simulationsgenauigkeit ab. Das Prinzip des interpretierenden Simulation erlaubt flexible Ausdrucksmöglichkeiten mit hoher Genauigkeit. Diese Art von Simulation eignet sich für die Beschreibung von Mikroarchitektursimulatoren.

Die Binärtransformation-Simulatoren bilden die Instruktionen des Zielprozessors auf die Instruktionen des Simulationssystems ab. Diese Art von Simulation erfordert einen geringen Spezifikationsaufwand und erreicht sehr hohe Simulationsgeschwindigkeiten. Die Einschränkungen für den Einsatz solcher Simulatoren wird durch das verwendete Simulationssystem bedingt. Der Instruktionssatz des Simulators muss dem des Simulationssystems entsprechen. Damit besitzen die Simulatoren eine geringe Ausdrucksmöglichkeit und können in der Regel nur als Instruktionssatzsimulatoren eingesetzt werden.

Die kompilierenden Simulatoren beschreiben einen Mittelweg zwischen den interpretierenden und den Binärtransformation Simulatoren. Bei der Erzeugung eines kompilierenden Simulators wird ein dedizierter Simulator für ein spezielles Programm erzeugt. Die Genauigkeit der Simulation kann selektiv bestimmt werden, womit flexible Ausdrucksmöglichkeiten mit einem geringen Spezifikationsaufwand erreicht werden. Das Laden und Dekodieren der Instruktionen wird zur Übersetzungszeit durchgeführt, wodurch die Simulationsgeschwindigkeit sehr hoch ist. Durch den Einsatz optimierender Compiler für die Übersetzung der Simulatoren für das Simulationssystem wird die

Simulationsgeschwindigkeit zusätzlich gesteigert. Die ausführliche Beschreibung der Generierung und der Simulation mit kompilierenden Simulatoren wird in Abschnitt 6.1 beschrieben.

2.5 Prozessorvalidierung

Die Validierung soll sicherstellen, dass die Prozessorentwürfe in sich konsistent sind und den Anforderungen aus den für die Spezifikation verwendeten Dokumenten entsprechen. Für die Validierung von Prozessorspezifikationen werden statische und dynamische Methoden verwendet. Für die Beschreibung der Validierungsmethoden in Kapitel 5 werden an dieser Stelle die bekannten Validierungsmethoden vorgestellt und für die Begriffe aus dem Prozessorentwurf adaptiert.

Bei der Validierung wird zwischen der statischen Analyse und dem dynamischen Testen unterschieden [SL05]. Diese Unterscheidung wird in dieser Arbeit in Kapitel 5 ausführlich erläutert. Die statische Analyse überprüft die Konsistenz des Entwurfs durch Konsistenzbedingungen und wird in der Regel durch das Werkzeug, in dem die Spezifikation erzeugt wird, oder durch die übersetzenden Werkzeuge durchgeführt. Die Spezifikation wird auf Inkonsistenzen zwischen den Komponenten der Spezifikation durch Konsistenzbedingungen überprüft. Hierfür muss die Spezifikation durch eine formale Struktur beschrieben sein. Die dynamische Validierung wird durch das Ausführen von Eingabesequenzen (im Folgenden Testfall genannt) in dem Simulator des Prozessors durchgeführt. Die Schlussfolgerungen auf Inkonsistenzen in der Spezifikation werden aus den Ergebnissen der Simulation geschlossen.

In „Validierung von Entwurfsspezifikationen komponentenbasierter Software für verteilte, eingebettete Automatisierungssysteme mittels Simulation“ [Fle97] identifiziert W. Fleisch die Kriterien für die Validierung. Daraus lassen sich folgende Kriterien für die Validierung von Prozessorspezifikationen übernehmen:

- Konsistenz und Vollständigkeit.
- Struktur und Verhaltenseigenschaften.
- Syntaktische und semantische Eigenschaften.

2.5.1 Statische Validierungsmethoden

Die statische Analyse wird auf der Spezifikation des Prozessors durchgeführt. Für die textuellen Sprachen werden z. B. durch die Syntaxanalyse die Struktur der Sätze in einer Spezifikation überprüft. Für die Erzeugung der Spezifikation werden oft visuelle, domänenspezifische Sprachen verwendet. Eine visuelle Sprache mit einer soliden Struktur erlaubt in der Regel keine syntaktischen Fehler. Die statische Validierung

kann dennoch durch die Analyse der Struktur einer Spezifikation Inkonsistenzen aufdecken. Hierfür können folgende Beispiele angegeben werden:

- Vollständigkeit und Verwendungsnachweise können durch die Überprüfung der Querverweise in der Spezifikation geprüft werden.
- Überprüfung der Datentypen wird durch die Analyse der Definitions- und Verwendungsstellen beschrieben.
- Über- oder Unterschreitung der definierten Wertebereiche kann durch die Mengenkodierung überprüft werden.
- Durch die Grapheinbettung kann die Konsistenz größerer Strukturen aus mehreren Sprachkomponenten gezeigt werden.

Für die Beschreibung der Methoden für die statische Analyse werden Konsistenzbedingungen angegeben, indem z. B. Relationen zwischen den Prozessorkonstrukten definiert werden. Nach der Überprüfung einer Konsistenzbedingung durch eine statische Validierungsmethode ist sichergestellt, dass die Spezifikation bezüglich der geprüften Eigenschaften keine Inkonsistenzen enthält.

2.5.2 Dynamische Validierungsmethoden

Die dynamische Validierung wird durchgeführt, wenn die statische Validierung auf einer Spezifikation für bestimmte Eigenschaften nicht durchgeführt werden kann. Dies geschieht, wenn z. B. in einer Prozessorspezifikation Fremdcode eingesetzt wird oder die Anforderungen an die Spezifikation durch externe Dokumente vorgegeben sind.

Durch dynamisches Testen kann nicht die Fehlerfreiheit, sondern nur die Fehler, falls vorhanden, in einem Entwurf nachgewiesen werden. Um die Fehlerfreiheit nachzuweisen, wird ein vollständiger Test des Prozessors benötigt, was durch die Komplexität des Prozessors nicht umgesetzt werden kann. Sinnvoll ist es, die Testfälle so auszuwählen, dass möglichst viele Eigenschaften des Prozessors beim Testen überdeckt werden. Die Qualität der dynamischen Validierung hängt von der Auswahl der Testfälle und der Strategie für die Überdeckung ab.

Die Anforderungen an die Spezifikation können durch die Entwickler als eine Menge von Testfällen oder Benchmarks vorgegeben werden. Letztere werden oft durch externe Anbieter erstellt, so bietet die Firma *ACE*[ACE13] die „*SuperTest compiler test and validation suite*“, mit der die Compiler für Prozessoren validiert werden können.

Die Testfälle für die dynamische Validierung können auch aus der Spezifikation des Prozessors abgeleitet werden. Eine Methode für die Erzeugung der Testfälle ist z. B. der Äquivalenzklassentest [RBGW10, MD03]. Bei dem Äquivalenzklassentest werden die Konstrukte der Spezifikation nach ihren Eigenschaften zusammengefasst. Beim

Testen wird davon ausgegangen, dass die Testobjekte aus einer Äquivalenzklasse bei beliebigen Eingaben gleiche Ergebnisse bei der Simulation erzeugen.

Neben der geschickten Auswahl von Äquivalenzklassen oder der Verwendung von Benchmarks kann die Qualität des dynamischen Testens durch gezielt erzeugte Testfälle für eine konkrete Spezifikation gesteigert werden. Hierfür werden Ansätze mit automatischer Testfallgenerierung benötigt, um systematisch Testfälle für eine Prozessorspezifikation zu erzeugen. Hierzu werden einige Werkzeuge in Kapitel 3 vorgestellt. Für die automatische Generierung der Testfälle werden Modelle verwendet, aus denen die Testfälle generiert werden. Die Benutzung von Modellen zur Ableitung von Testfällen wird beim *modellbasierten Testen* (MBT) systematisch beschrieben [PP05, RBGW10, UL07, Sch07]. Im nächsten Abschnitt 2.5.3 werden die Elemente des modellbasierten Testens beschrieben und der Bezug zu der Prozessorvalidierung gezeigt. Dabei wird auf die Aufgaben und die Spezifikation einzelner Elemente eingegangen.

2.5.3 Modellbasiertes Testen

Der Begriff des modellbasierten Testens (MBT) wird in der Literatur unterschiedlich interpretiert. Eine abstrakte Definition wird in „Basiswissen Modellbasierter Test“ [RBGW10] wie folgt angegeben:

„Beim modellbasierten Testen wird die zu testende Software, ihre Umgebung oder der Test selbst über Modelle betrachtet […]. Diese Modelle können eigenständig, parallel zu Entwicklungsumgebung erstellt oder aus diesen abgeleitet werden.“

Aus dieser Definition folgt, dass sowohl das zu testende System als auch die Testfälle durch Modelle ausgedrückt werden [PP05, Sch07]. Der Prozess des modellbasierten Testens kann wie in Abbildung 2.10 [PP05] dargestellt werden. Ein abstraktes Modell beschreibt das Verhalten des Systems, aus dem sowohl die Eingabe für das System als auch das erwartete beobachtbare Verhalten in Form eines Testfalls abgeleitet werden kann. *Test Case Specification* beschreibt die Strategie, nach der aus dem Modell die verschiedenen Testfälle abgeleitet werden. Durch die Eingabedaten wird das Verhalten des Systems beobachtet und mit dem erwarteten Verhalten aus dem Testfall verglichen. Als Ergebnis der Validierung kann festgestellt werden, ob das System und das Modell unterschiedliches Verhalten aufweisen.

Die Begriffe des modellbasierten Testens werden bei der Beschreibung der dynamischen Validierungsmethoden in Abschnitt 5.5 verwendet, für das Verständnis der entwickelten Validierungsmethoden werden im folgenden Abschnitt die Szenarios des MBT nach Pretschner und Philipps [PP05] vorgestellt.

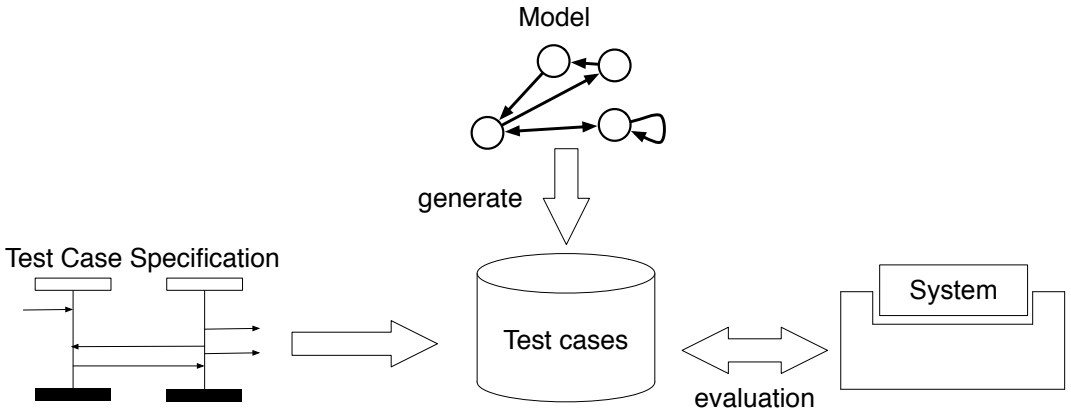


Abbildung 2.10: Modellbasiertes Testen [PP05].

2.5.3.1 MBT Szenarios

Die Vorgehensweise beim MBT unterschiedet sich nicht von der natürlichen Vorgehensweise beim Testen, indem die erwartete Ausgabe mit der erzeugten Ausgabe des Systems, bei einer entsprechenden Eingabe, verglichen wird. Die grundlegende Idee, die Testfälle aus einem Modell automatisch zu generieren, ist ebenfalls nicht neu. Das Konzept des modellbasierten Testens wird durch verschiedene Szenarios für die Erzeugung des Testmodells charakterisiert.

Die unterschiedliche Erzeugung der Modelle im Testprozess wird durch so genannte Szenarios ausgedrückt. Diese sollen den Informationsfluss zwischen den Modellen beim Testen verdeutlichen. In Abbildung 2.11 werden zwei Szenarios *Manual Modeling* und das *Common Modeling* von M.Pretschner und J.Philipps [PP05] dargestellt. Zu den Komponenten aus Abbildung 2.10 werden die *Requirements* (Anforderungen) und das *System Model* hinzugefügt. Die Pfeile zwischen den Komponenten stellen den Informationsfluss zwischen den Komponenten dar.

Das *Manual Modeling* Szenario beschreibt, dass das Systemmodell und das Testmodell unabhängig voneinander aus den Anforderungen an das System erzeugt werden. Die Testfälle und das System für die Validierung werden aus den jeweiligen Modellen generiert. Dieses Szenario setzt voraus, dass aus den Anforderungen zwei Modelle erzeugt werden. Bei dem *Common Modeling* Szenario wird das Testmodell aus dem Systemmodell abgeleitet. Wird das Testmodell und das System aus dem Systemmodell generiert, besteht nach M.Pretschner und J.Philipps keine Redundanz zwischen den Modellen, womit die Qualität der Generatoren validiert wird.

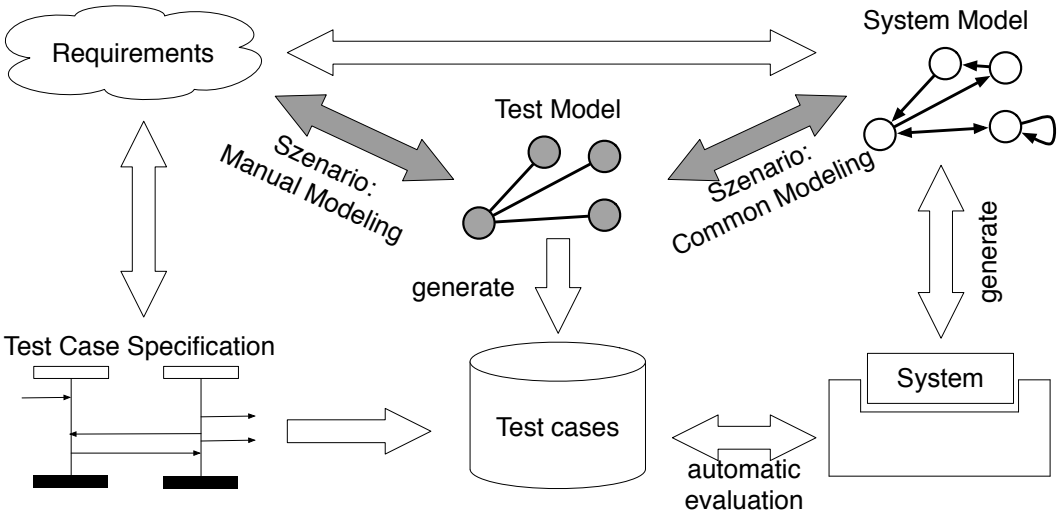


Abbildung 2.11: Manual Modeling und Common Model Szenario [PP05].

Übertragen auf den Prozessorentwurf und die dynamische Validierung mit *ViCE-UPSLA* können die Komponenten exakt benannt werden. Die *Requirements* stehen für die informelle Beschreibung des Prozessors, als Referenz-Dokumentationen oder als technische Datenblätter. Das *System Model* wird durch eine Prozessorspezifikation in *ViCE-UPSLA* repräsentiert. Aus der Prozessorspezifikation wird ein *System* als Simulator des Prozessors generiert. Die Prozessorspezifikation wird dazu verwendet, mit Hilfe eines Generators Testfallspezifikationen zu erzeugen. Die Testfallspezifikation in dieser Arbeit vereint die Eigenschaften von *Test Model* und *Test Case Specification* aus Beispiel 2.11, womit aus der Testfallspezifikation Testfälle für die Validierung erzeugt werden.

3 Verwandte Arbeiten

Im Grundlagenabschnitt 2.1 wurde die Vorgehensweise bei dem Prozessorentwurf vorgestellt. Dabei wurde beschrieben, dass in jeder Phase der Entwicklung von Prozessoren Werkzeuge für die Beschreibung von Prozessorspezifikationen verwendet werden. Hierfür existiert eine breites Spektrum von Sprachen und Werkzeugen, die im Prozessorentwurf eingesetzt werden. Die unterschiedlichen Sprachen und Werkzeuge werden in unterschiedlichen Phasen in der Entwicklung eingesetzt und verfolgen unterschiedliche Ansätze, um Prozessorspezifikationen zu beschreiben, und unterschiedliche Ziele für die Entwicklung. *ViCE-UPSLA* [Kla13] ist eine visuelle, domänenspezifische Sprache mit den Entwurfszielen der Spezifikation und der Validierung von Prozessoren. Für die Einordnung der Sprache in die Domäne des Prozessorentwurfs wird *ViCE-UPSLA* anderen Sprachen gegenübergestellt. In diesem Abschnitt wird zunächst die Klassifikation für die Sprachen und der Entwicklungsziele beschrieben. Damit werden anschließend einige Sprachen und deren Konzepte vorgestellt.

Das zweite Ziel dieser Arbeit ist die Validierung von Prozessorentwürfen auf einem hohen Abstraktionsniveau aus der Spezifikation. Bei der dynamischen Validierung werden die Ansätze aus MBT verwendet. Für die Einordnung der Methoden in dieser Arbeit werden einige Werkzeuge aus dem Bereich MBT und der Validierung von Prozessoren mit automatischer Testfallgenerierung vorgestellt.

3.1 Klassifikation der Prozessorspezifikationssprachen

Die Klassifikation der Sprachen wird nach P.Mishra und N.Dutt aus „Processor Description Languages“ [PD08] durchgeführt. In diesem Ansatz wird zwischen „Structural“ (strukturellen), „Behavioral“ (verhaltensbeschreibenden) und „Mixed“ (hybriden) Sprachen unterschieden. Diese Klassifikation der Sprachen wird entsprechend der Einteilung der Beschreibungssicht für die Prozessorkomponenten in dem Y-Diagramm aus Abschnitt 2.1.2 durchgeführt. Zusätzlich zu der Beschreibungssicht einer Sprache verfolgen die Sprachen ein Entwicklungsziel. Hierbei wird zwischen folgenden Zielen unterschieden:

Compilation-oriented — Entwicklung von Werkzeugen für den Prozessor

Simulation-oriented — Entwicklung von Simulatoren

Synthesis-oriented — Hardwaregenerierung

Validation-oriented — Validierung von Entwürfen

Zusätzlich zu der Einteilung durch die Ausrichtung und die Entwicklungsziele kann für die Sprachen das Abstraktionsniveau nach dem Y-Diagramm, auf dem die Prozessorspezifikation beschrieben wird, angegeben werden. Die Beschreibung des Abstraktionsniveaus ist für die Einsatzmöglichkeiten der Sprache in den verschiedenen Entwicklungsphasen eines Prozessors signifikant.

Im Folgenden werden einige Sprachen aus der Domäne und deren signifikante Eigenschaften vorgestellt. In Abschnitt 3.1.7 werden die vorgestellten Sprachen in einer Übersicht entsprechend ihrer Eigenschaften gegenübergestellt und der Ansatz der Sprache *ViCE-UPSLA* im Vergleich dazu eingeordnet.

3.1.1 UPSLA

Die Sprache *UPSLA* (Unified Processor Specification LAnguage) [KLST04] wurde an der Universität Paderborn entwickelt und besitzt den größten Verwandtschaftsgrad zu *ViCE-UPSLA*. Die Konzepte der Sprache *UPSLA* wurden als Basis bei der Entwicklung von *ViCE-UPSLA* übernommen. *UPSLA* wird für die automatische Generierung von zyklengenauen Simulatoren und optimierenden Compilerwerkzeugen für Prozessoren eingesetzt.

Mit der Sprache *UPSLA* wird eine Prozessorspezifikation durch die Definition von Elementen erzeugt. Dabei werden domänenspezifische Prozessorkonstrukte wie Instruktionen, Adressierungsmethoden sowie der Registersatz des Prozessors beschrieben. Der prozessorspezifische Teil in *UPSLA* kann aus einer Prozessorspezifikation in *ViCE-UPSLA* vollständig generiert werden. In *UPSLA* wird von den Konstrukten der Mikroarchitektur wie z. B. Pipelinestufen, Datenpfad oder Bypass-Struktur abstrahiert. Die Beschreibung der Komponenten erfolgt auf einem hohen Abstraktionsniveau in textueller Notation. Durch die flexiblen Ausdrucksmöglichkeiten wird die Sprache für DSE und ISE Entwurfsszenarios eingesetzt.

Für die Spezifikation einer Instruktion werden mit *UPSLA* die Kodierung, die Operanden und die Verhaltensbeschreibung der Instruktion angegeben. Die Kodierung der Instruktionen enthält die binäre Kodierung, sowie das Assemblerformat der Instruktion. Außerdem werden die von der Instruktion verwendeten impliziten Operanden und die Befehlsoperanden angegeben. Die Adressierungsmethoden werden durch Funktionsaufrufe definiert und den Befehlsoperanden aus der Instruktionskodierung zugeordnet.

Die Verhaltensbeschreibung der Instruktionen wird durch Ausführungszyklen spezifiziert. Dabei werden für jeden Zyklus die Operationen durch Funktionsaufrufe angegeben, womit die zyklengenaue Simulation ermöglicht wird. Die Funktionen für die Operationen der Instruktionen und für die Adressierungsmethoden werden, analog zu *ViCE-UPSLA*, in C implementiert.

Der Registersatz des Prozessors wird ebenfalls auf einem hohen Abstraktionsniveau angegeben und erlaubt die Spezifikation von komplexen Registersatzstrukturen mit architektonischen Registern.

3.1.2 nML

Die Sprache *nML* von A. Furth, J. Van Preat und M. Freerick [FPF95] ist eine hybride Sprache für die Spezifikation der Verhaltensbeschreibung von Instruktionen und deren Kodierung. Die Beschreibung der Spezifikation wird textuell und abstrahiert von den Prozessorkonstrukten durch eine attributierte Grammatik durchgeführt. Damit ist die kognitive Distanz zwischen den Konstrukten in der Spezifikation und den Begriffen aus dem Prozessorentwurf sehr hoch, womit der Zugang zu der Sprache für ungeübte Entwickler erschwert wird.

Mit dem Ansatz von nML wird demonstriert, wie die Konzepte der funktionalen Programmierung für die Beschreibung von Prozessorspezifikationen angewendet werden können. Die Spezifikation der Sprache ist dazu geeignet, Instruktionssatzsimulatoren zu beschreiben, um z. B. die Entwicklung von Werkzeugen für den Zielprozessor voranzutreiben.

3.1.3 ViDL

Die Sprache *ViDL* (Versatile ISA Description Language) [Dre12] ist eine Verhaltensbeschreibungssprache mit dem Ziel, VHDL Implementierungen [LWS94] des Prozessors und Instruktionssatzsimulatoren aus einer Spezifikation zu generieren.

Aus einer textuellen Verhaltensbeschreibung des Instruktionssatzes und der Beschreibung des Registersatzes werden aus der Prozessorspezifikation verschiedene Mikroarchitekturen, abhängig von der gewählten Taktfrequenz des Zielprozessors, generiert. Da die Mikroarchitektur des Prozessors aus der Verhaltensbeschreibung des Instruktionssatzes und der Taktfrequenz abgeleitet wird, kann kein direkter Einfluss auf die Struktur der Mikroarchitektur vorgenommen werden.

Für die Spezifikation eines Prozessors mit *ViDL* werden folgende Prozessorkonstrukte auf einem hohen Abstraktionsniveau angegeben:

- Speicher und Registersatz
- Ein- und Ausgabe-Ports
- Kodierung der Instruktionen
- Verhaltensbeschreibung und Semantik der Instruktionen

Für die Spezifikation des Registersatzes werden in einer Definition lediglich die Anzahl und die Breite der Register einer Registerbank benötigt. Die Beschreibung der Instruk-

tionen wird ebenfalls deklarativ durch die Angabe der binären Kodierung und der Semantik der Instruktionen in eigener *ViDL* Notation durchgeführt. Für die Generierung der Prozessoren oder Simulatoren mit *ViDL* werden keine zusätzlichen Funktionen wie bei *UPSLA* oder *ViCE-UPSLA* benötigt.

Die Sprache *ViDL* besitzt sehr flexible Ausdrucksmöglichkeiten und kann für die Spezifikation von realen Prozessoren verwendet werden. Damit kann die Sprache bei den Entwicklungsszenarios wie z. B. DSE oder ISE eingesetzt werden.

3.1.4 xADL

xADL [BPK13] ist eine strukturelle Sprache und beschreibt einen gegensätzlichen Ansatz zu *ViDL*, verfolgt dabei jedoch äquivalente Ziele. Die Spezifikation eines Prozessors mit *xADL* wird durch die strukturelle Beschreibung der Mikroarchitektur eines Prozessors durchgeführt. Aus der Spezifikation der Mikroarchitektur wird der Instruktionssatz des Prozessors extrahiert und eine VHDL Implementierung des Prozessors generiert. Im Gegensatz zu *ViDL* kann bei der Spezifikation eines Prozessors nur ein indirekter Einfluss auf den Instruktionssatz des Prozessors ausgeübt werden.

Die Beschreibung der Spezifikation wird durch ein visuelles Werkzeug *adlgen* unterstützt. Bei der Spezifikation werden Prozessorkonstrukte wie Registersatz, Speicher und Funktionseinheiten auf einem hohen Abstraktionsniveau angegeben. Für die Spezifikation des Datenpfads und der Pipeline werden Verknüpfungen zwischen den Komponenten der Mikroarchitektur definiert. In diesem Rahmen kann z. B. auch die Bypass-Struktur des Prozessors spezifiziert werden.

xADL ist besonders für die Durchführung von Entwurfsraumexplorationen für Prozessoren geeignet.

3.1.5 TIE

TIE (Tensilica Instruction Extension) [Inc06] ist eine kommerzielle, domänenspezifische Sprache für die Spezifikation von Instruktionssatzerweiterungen. Aus der Spezifikation werden zyklengenau Simulatoren und Compilerwerkzeuge für die Prozessoren generiert.

Für die Reduzierung des Spezifikationsaufwands wird für eine Prozessorspezifikation ein fest vorgegebener Kern eines Prozessors eingesetzt, auf dem die Instruktionssatzerweiterungen beschrieben werden. Die Spezifikation eines individuellen Instruktionssatzes oder der Mikroarchitektur ist mit *TIE* nicht vorgesehen. Durch die visuelle Notation wird die Erweiterung des Prozessors durch die Spezifikation ihres Verhaltens angegeben, womit die Sprache eine sehr hohe Akzeptanz bei Prozessorentwicklern hat.

Neben der Beschreibung von Instruktionssatzerweiterungen können mit der Sprache Prozessoren mit SIMD oder VLIW Architekturen beschrieben werden. Damit kann die Sprache für die Szenarios des DSE oder ISE eingesetzt werden.

3.1.6 LISA

Die Dokumentation von Synopsis [CoW08] erlaubt einen Überblick über die Sprache *Lisa*. *Lisa* ist eine kommerzielle und etablierte Sprache mit der Unterstützung von Entwicklungszielen wie Hardwaregenerierung, Validierung, Werkzeugentwicklung und Simulation. Das Konzept der Sprache ist hybrid und ermöglicht die Spezifikation des Prozessors aus der strukturellen und Verhaltenssicht. Mit der Sprache können eine Vielzahl von Prozessorkonstrukten und Eigenschaften auf einem niedrigen Abstraktionsniveau beschrieben werden. Damit besitzt die Sprache eine sehr hohe Ausdrucksmöglichkeit und erlaubt die Spezifikation von diversen realen Prozessoren.

Die Vorgehensweise bei der Spezifikation eines Prozessors verlangt als Erstes die Spezifikation der Mikroarchitektur und der Kodierung von Instruktionen. Die Spezifikation der Instruktionen und der Verhaltensbeschreibung wird auf dieser Basis durchgeführt. Diese Voraussetzung bedingt die Änderung der Mikroarchitektur zu einem spezifizierten Instruktionssatz nur mit einem sehr hohen Spezifikationsaufwand. Des Weiteren muss die Spezifikation der Verhaltensbeschreibung, anderes als z. B. bei *ViDL*, für die Generierung der Hardware und der Simulatoren in unterschiedlichen Modellen angegeben werden. Dies erfordert einen höheren Spezifikationsaufwand, ermöglicht jedoch gleichzeitig die Validierung zwischen den beiden Modellen.

Durch das breite Spektrum der Entwicklungsziele und Ausdrucksmöglichkeiten besitzt die Sprache eine hohe Komplexität für die Beschreibung von Prozessoren. Die meist textuellen oder tabellenbasierten Beschreibungen der Sprachkonstrukte sind auf einem niedrigen Abstraktionsniveau, was die Zugänglichkeit der Sprache für ungeschulte Entwickler erschwert.

3.1.7 Gegenüberstellung der Sprachen

Für die beschriebenen Sprachen wurden das Abstraktionsniveau und die Entwurfsziele für die verschiedenen Szenarios bei der Entwicklung von Prozessoren angegeben. In Abbildung 3.1 werden die Sprachen nach ihrem Abstraktionsniveau und der Beschreibungssichten aus dem Y-Diagramm eingeordnet.

Die Sprachen *TIE* und *UPSLA* sind Verhaltensbeschreibungssprachen, enthalten jedoch strukturelle Komponenten für die Spezifikation zyklengenauer Simulatoren und sind aus diesem Grund links neben *ViDL* angeordnet. *LISA*, *TIE* und *ViCE-UPSLA* sind hybride Sprachen und befinden sich in der Mitte. Wie bereits beschrieben, verfolgt die Sprache *xADL* einen gegensätzlichen Ansatz zu *ViDL* und befindet sich auf der gegenüberliegenden Seite der Skala.

Durch die visuelle Notation sind die Sprachkonstrukte von *TIE*, *xADL* und *ViCE-UPSLA* auf dem höchsten Abstraktionsniveau in diesem Vergleich. Die Sprachen *UPS-*

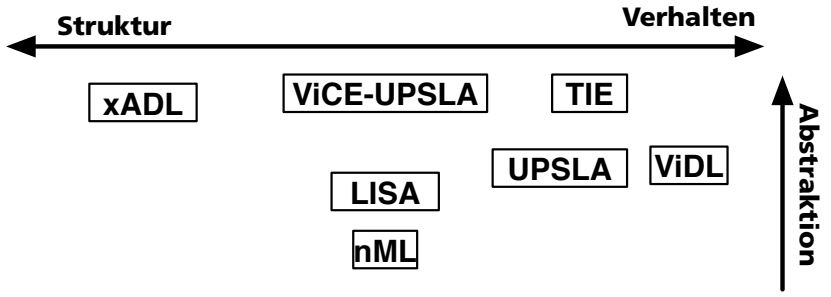


Abbildung 3.1: Einordnung der Sprachen nach ihrer Beschreibungsrichtung und Abstraktionsniveau.

LA und *ViDL* beschreiben die Prozessorkonstrukte in textueller Form auf vergleichbar hohem Abstraktionsniveau. Das Schlusslicht bildet die Sprache *nML*, welche die Prozessorkonstrukte durch die Komponenten aus Programmiersprachen realisiert.

3.2 Klassifikation von Validierungswerkzeugen

In dieser Arbeit werden Methoden für die dynamische Validierung von Prozessorspezifikationen mit automatischer Generierung von Testfällen beschrieben. Für die Erläuterung der Validierungsmethoden mit *ViCE-UPSLA* werden in diesem Abschnitt Werkzeuge aus dem Bereich der Prozessorvalidierung und MBT vorgestellt und mit den Ansätzen in dieser Arbeit verglichen. Die Werkzeuge für die Durchführung des modellbasierten Testens besitzen aktuell eine sehr hohe Popularität. In der Studie „Modellbasiertes Testen“ [GNRS09] werden kommerzielle Werkzeuge in einer Übersicht klassifiziert und verglichen. In diesem Abschnitt werden ausgewählte Werkzeuge aus der Studie kurz vorgestellt.

Die Klassifikation der Werkzeuge wird unter anderem durch die Aufgaben und Ziele bei der Validierung durchgeführt. Außerdem werden z. B. die Modellierungssprachen für die Beschreibung der Tests oder die Zusammenhänge zwischen System- und Testmodell bewertet. Damit werden die Werkzeuge in folgende drei Kategorien eingeteilt:

- Modellbasierte *Testdatengeneratoren* sind Werkzeuge für die Ableitung von Steuerinformationen und Testdaten bei manuell durchgeführten Tests.

- Modellbasierte *Testfalleditoren* sind Werkzeuge, die durch eine Spezifikationssprache die individuelle Gestaltung der Testfälle durch einen Testentwickler und die Generierung von Testfällen ermöglichen.

- Modellbasierte *Testfallgeneratoren*-Werkzeuge beschreiben das Systemverhalten eines Systems durch ein Modell und ermöglichen automatische Generierung von Testfällen.

Im Folgenden werden einige Werkzeuge vorgestellt und deren spezielle Eigenschaften erläutert.

3.2.1 Qtronic

Das Werkzeugsystem *Qtronic* wird von der Firma Conformiq [GNRS09, SÖ10] entwickelt und bietet ein *Testfallgenerator* Werkzeugsystem für die automatische Generierung von Testfällen. Die Systemmodelle werden in einer auf UML basierenden Modellierungssprache *Qtronic Modeler* ausgedrückt. Das Werkzeugsystem wird als Bindeglied zu einem beliebigen UML Modellierungstool eingesetzt, womit die automatische Erzeugung von Testmodellen und die Automatisierung der Testdurchführung ermöglicht werden. Die Modellierung für das erwartete Verhalten des Systems muss in einem separaten Modell parallel zu dem Systemmodell beschrieben werden.

Für das Testen werden konkrete Testfälle aus Klassendiagrammen oder Zustandsübergangsdiagrammen erzeugt. Für die Modellierung der Testfälle werden visuelle Beschreibungsmittel unterstützt. Die Tests werden auf der Stufe des Systemtests erzeugt, sodass das Testen einzelner Komponenten oder Module nicht unterstützt wird.

3.2.2 expecco

Von der Firma eXept wird das Werkzeugsystem *expecco* [Git08] entwickelt. Durch das Werkzeugsystem wird ein *Testfalleditor* zur Verfügung gestellt, er ermöglicht die visuelle Modellierung von UML-Aktivitätsdiagrammen für die Beschreibung von dezidierten Testmodellen. Für die Reduzierung des Spezifikationsaufwandes werden von der Firma eXept domänenspezifische Erweiterungen angeboten, womit die manuelle oder automatische Generierung und Ausführung von Soft- und Hardwaretests ermöglicht wird.

Die Generierung der Testfälle wird ausschließlich aus Testfallmodellen unabhängig von dem Systemmodell unterstützt. Die Beschreibung der Testfallmodelle wird durch Aktivitätsdiagramme vorangetrieben.

3.2.3 MMV: Metamodeling Based Microprocessor Validation Environment

Das Validierungswerkzeug MMV [DMK+06] wurde für die Validierung von Mikroprozessoren auf unterschiedlichen Abstraktionsebenen entwickelt. Der Ansatz basiert auf

einem *Metamodell*, aus dem die Testfälle mittels eines Generators für einen Mikroarchitektur- oder einen Instruktionssatzsimulator erzeugt werden.

Für die Modellierung des *Metamodells* werden visuelle domänespezifische Prozessorkonstrukte in Entity-Relationship-Modellen beschrieben. Die Validierung der Mikroarchitekturen wird durch die statische Analyse des *Metamodells* unterstützt. Aus dem *Metamodell* wird für die Erzeugung der Testfälle ein *modeling framwork* erzeugt, aus dem die Testfälle für die Validierung generiert werden.

In Abbildung 3.2 wird die Übersicht zu der MMV Modellierungs- und Validierungsumgebung dargestellt. Aus der Abbildung kann entnommen werden, dass die Modelle für den Prozessor, für die Validierung und die Simulatoren separat erzeugt werden müssen. Damit erlaubt MMV die Validierung von Prozessoren auf nahezu beliebigen Abstraktionsebenen.

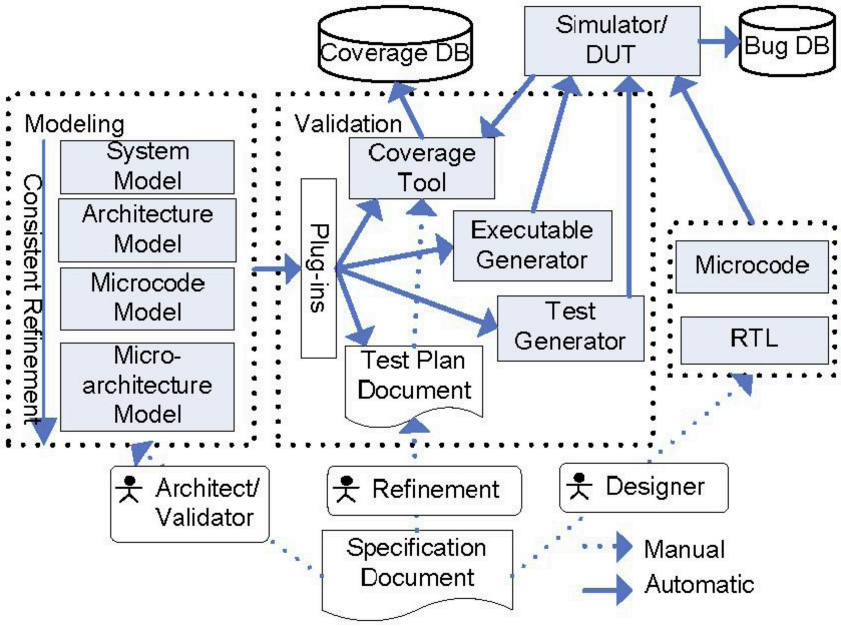


Abbildung 3.2: MMV Modellierungs- und Validierungsumgebung [DMK+06].

4 ViCE-UPSLA

In diesem Kapitel wird die Sprache *ViCE-UPSLA* [Kla13] vorgestellt. Durch die Konzepte der Sprache wird die Unterstützung der Entwicklung und Evaluierung von Prozessoren in einem Werkzeugsystem umgesetzt. Für die Beschreibung der Konzepte von *ViCE-UPSLA* werden die Anforderungen an das Werkzeugsystem definiert, indem dessen Einsatzszenarios im Prozessorentwurf beschrieben werden. Diese Vorgehensweise wurde bereits in einer abgeschlossenen Arbeit erfolgreich angewendet [CKK08]. Hierzu wird in einem einleitenden Abschnitt die Domäne des Prozessorentwurfs anhand eines Projekts vorgestellt. Für die Umsetzung des Entwurfs wird das Generatorsystem DEViL [Sch06] eingesetzt. DEViL ist besonderes dafür geeignet, Strukturediteren für anspruchsvolle visuelle Sprachen zu entwickeln.

4.1 Vorstellung der Domäne des Prozessorentwurfs

In den meisten Entwicklungsprojekten werden in einer sehr frühen Phase prototypische Lösungen der Ziel-Software und der Hardware getestet, um die Machbarkeit und die Leistungsfähigkeit der Systeme möglichst früh abzuschätzen. In dieser Phase der Entwicklung arbeiten verschiedene Arbeitsgruppen zusammen, um die Anforderungen für die späteren Entwicklungsphasen zu ermitteln. So wurde z. B. im *Easy-C-Projekt* (Enablers of Ambiente Services and Systems Part C - Wide Area Courage) [Bun10] für die Entwicklung der Software für *Protocol-Stacks* die Simulation der Software und der Hardware durchgeführt. Die verwendete Software und Hardware wurde nur prototypisch aus den Anforderungen an das spätere System umgesetzt. Die umgesetzte Software imitiert lediglich die verwendeten Algorithmen und die relevanten Eigenschaften des Codes, damit die Anforderungen an die Hardware abgeschätzt werden können. Mit einer umfassenden Simulation der Software und verschiedenen Konfigurationen der Zielprozessorarchitektur wurden Vorschläge für die Erweiterungen und Optimierungen des Prozessors erarbeitet.

Da der zukünftige Prozessor in dieser Entwicklungsphase lediglich in einer informellen Beschreibung vorliegt, wurde für die Erzeugung des Simulators eine formale Spezifikation erstellt und im Verlauf des Projektes bearbeitet. An der Umsetzung, Validierung und Evaluierung der Prozessorspezifikation sind Experten aus den Bereichen der Softwareentwicklung und Schaltungstechnik beteiligt. Wie in Abbildung 4.1 dargestellt, wurde ein dualer Ablauf bei der Entwicklung und Bewertung der Entwürfe angewendet. Dabei wurde parallel an der Synthese und der Simulation des Prozessors

entwickelt. Bei der Synthese des Prozessors wurde die Entwicklung von Simulatoren auf Registertransfer-Ebene vorangetrieben. Für die Beschreibung von Softwaresimulatoren wurde das Werkzeugsystem *UPSLA* (Unified Processor Specification Language) [KLST04] eingesetzt. Damit wurde, basierend auf einer formalen Spezifikation, die Entwicklung von Compilerwerkzeugen und Softwaresimulatoren vorangetrieben.

Aus der Motivation im Easy-C-Projekt wurde die Entwicklung einer visuellen Sprache auf einem hohen Abstraktionsniveau angestoßen, mit der die Prozessorentwickler aus unterschiedlichen Arbeitsgruppen bei der Entwicklung und Validierung der Prozessorentwürfe eine gemeinsame Basis erhalten. Das Ergebnis der Entwicklung ist die Sprache *ViCE-UPSLA*, die im Folgenden vorgestellt wird.

4.2 Anforderungen an das Werkzeugsystem

Die Anforderungen an ein System, das bei der Entwicklung von Prozessoren in frühen Phasen eingesetzt wird, können auf die Spezifikation, Simulation und die Validierung von Prozessoren auf einem hohen Abstraktionsniveau eingegrenzt werden. Dafür werden Werkzeuge benötigt, mit denen die Prozessoren hinreichend spezifiziert werden, mit dem Ziel, die Entwicklung von Simulatoren zu ermöglichen und den Prozessorentwickler bei der Validierung der Entwürfe zu unterstützen. Als eine zentrale Komponente des Werkzeugsystems wird eine domänenspezifische Sprache benötigt, mit der die Prozessorspezifikationen ausgedrückt werden. Die Anforderungen an die Sprache und die Einsatzmöglichkeiten des neuen Systems werden in diesem Abschnitt beschrieben.

Im Entwurfsprozess sind in der Regel Entwicklergruppen aus unterschiedlichen Bereichen wie Schaltungstechnik oder Softwareentwurf beteiligt. So, wie das Hintergrundwissen der beteiligten Experten an der Entwicklung unterschiedlich ist, sind auch die Vorstellungen und Anforderungen an das System und die Darstellung der Prozessorentwürfe unterschiedlich. Zur Überbrückung der kognitiven Distanz wird eine Spezifikationssprache benötigt, welche die bereits bekannten Strukturen und Darstellungen aus der Domäne aufgreift. Hierfür werden die informellen Beschreibungen von Prozessoren und andere vorliegende Dokumente und Werkzeuge bei dem Entwurf als Beispiele für die Sprachentwicklung verwendet.

Die Abstraktionsebene und die Darstellung der Sprache sowie die verwendeten Prozessorkonstrukte sind für die Akzeptanz seitens der Prozessorentwickler entscheidend. Die Abstraktionsebene der Prozessorkonstrukte, wie in Abschnitt 2.1.2 beschrieben, muss der frühen Entwicklungsphase entsprechen. Damit wird eine Notation für Prozessorarchitekturen benötigt, welche die Konstrukte des Prozessorentwurfs in frühen Phasen des Entwicklungsprozesses erfasst. Visuelle Sprachen finden unter solchen Anforderungen meistens sehr hohen Zuspruch. Für die Entwicklung der Sprache wurden in dieser Arbeit zur Analyse der Domäne neben den verschiedenen Unterlagen von Prozessorspezifikationen, z. B. ARM-Datasheet [Lim87] oder PowerPC User's Manual

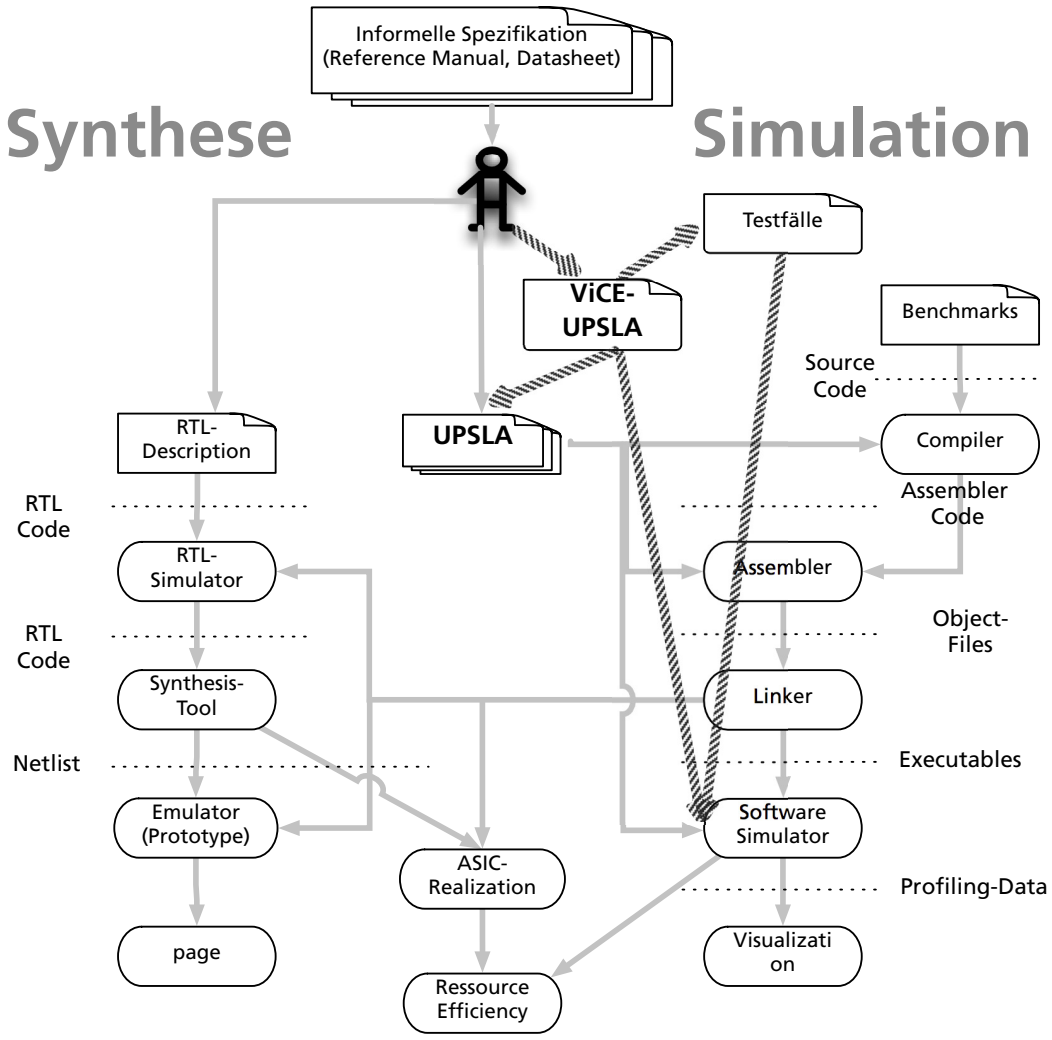


Abbildung 4.1: Integration von *ViCE-UPSLA* in den Entwurfsablauf.

[Mic94] usw., die Erfahrungen aus dem Easy-C-Projekt genutzt. Dabei wurde ermittelt, welche domänenspezifischen Konstrukte in der Sprache umgesetzt werden müssen und welches Abstraktionsniveau der Prozessorkonstrukte benötigt wird.

In Abbildung 4.1 wird *ViCE-UPSLA* in den Entwicklungsprozess eingeordnet. Die Sprache soll die Position zwischen der informellen Spezifikation des Prozessors und den weiterführenden Entwicklungsschritten einnehmen. Mit dem Werkzeugsystem sollen visuelle Spezifikationen von Prozessoren in zugänglicher Präsentation für alle be-

teiligten Entwicklergruppen beschrieben werden. Damit wird eine gemeinsame Basis für die Evaluierung und Validierung der Entwürfe geschaffen. Für die Nutzung der damit beschriebenen Prozessorspezifikationen müssen Generatoren integriert werden, mit denen die Simulation und Validierung aus der Spezifikation ermöglicht wird. Für die Entwicklung der Compilerwerkzeuge muss aus einer Prozessorspezifikation in *ViCE-UPSLA* die prozessorspezifischen Anteile für *UPSLA* generiert werden. Außerdem muss aus der Prozessorspezifikation die Generierung von Softwaresimulatoren und die Umsetzung statischer und dynamischer Validierungsmethoden ermöglicht werden. Für die Umsetzung der dynamischen Validierungsmethoden wird die Generierung individueller, an die Architektur des Entwurfs angepasster Testfälle beschrieben. Die statischen oder dynamischen Validierungsmethoden werden in dem Werkzeugsystem zusammen mit der Sprache *ViCE-UPSLA* in Form von Konsistenzprüfungen oder mit Hilfe von Code-Generatoren für die Erzeugung von Testfällen integriert. Die Methoden zur systematischen Validierung von Prozessoren aus der Spezifikation werden in Kapitel 5 vorgestellt.

In Abbildung 4.2 ist schematisch die Aufteilung der Werkzeuge und der Komponenten zur Spezifikation und Validierung von Prozessoren dargestellt. Das Werkzeugsystem soll die Sprache *ViCE-UPSLA* und Validierungsmethoden vereint umsetzen. Aus einer Prozessorspezifikation soll ein Prozessorsimulator mit Hilfe eines Simulator-Generators erzeugt werden. Die statische Analyse muss so integriert werden, dass der Benutzer bereits bei der Erzeugung eines Entwurfs unterstützt wird. Für die dynamischen Validierungsmethoden werden der generierte Prozessorsimulator und die Testfälle benötigt. Für die Erzeugung der Testfälle werden Methoden benötigt, mit denen aus der Prozessorspezifikation die Testfälle automatisch generiert werden können. Die dynamische Validierung muss dem Prozessorentwickler in einer angemessenen Form präsentiert werden, damit dieser die Vorgehensweisen bei der Validierung einsehen und gegebenenfalls nach seinen Wünschen anpassen kann. Hierfür wird eine weitere visuelle Sprache *TFS* benötigt, in der die **T**est**F**all**S**pezifikation ausgedrückt wird. In Kapitel 5 wird gezeigt, dass es sinnvoll ist, in der Spezifikation des Prozessors ergänzende Beschreibungen durch zusätzliches Wissen hinzuzufügen. Das zusätzliche Wissen wird nicht für die Generierung des Prozessorsimulators verwendet, sondern dient dazu, die statische und dynamische Validierung des Prozessors zu unterstützen.

4.3 Sprachkonzept

In diesem Abschnitt werden die Konzepte der visuellen Sprache *ViCE-UPSLA* [Kla13] vorgestellt. *ViCE-UPSLA* ist eine Sprache auf einem hohen Abstraktionsniveau, die eine einfache und systematische Spezifikation von Prozessoren ermöglicht. Die damit erstellten Spezifikationen werden für die Simulation und Validierung verwendet. Die Erzeugung von Simulatoren aus einer vollständigen Spezifikation wird automatisch

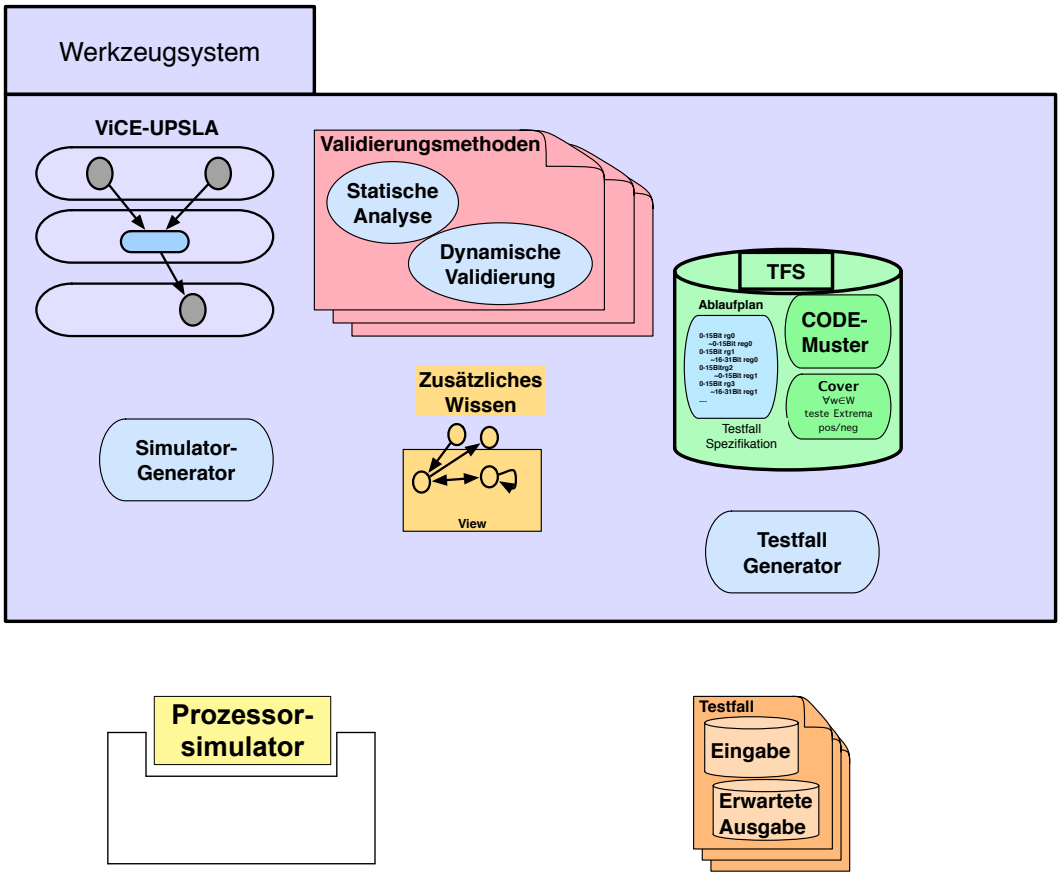


Abbildung 4.2: Struktur des Werkzeugsystems.

durch verschiedene Generatoren umgesetzt. Damit ist es möglich, in kurzer Zeit mehrere verschiedene Prototypen eines Prozessors zu erzeugen, was für die Durchführung von Instruktionssatzerweiterungen (ISE) oder Entwurfsraumexplorationen (DSE), wie in Abschnitt 2.1.1 beschrieben, geeignet ist. Darüber hinaus bietet die Struktur der Sprache eine Vielzahl von Möglichkeiten, statische und dynamische Konsistenzprüfungen zu integrieren. Die statischen Validierungsmethoden werden als Konsistenzprüfungen in der Werkzeugkette umgesetzt und ermöglichen die Überprüfung der statischen Eigenschaft direkt in der Prozessorspezifikation. Die dynamischen Validierungsmethoden können mit Hilfe von Code-Generatoren im Werkzeugsystem umgesetzt werden und damit die Testfälle als Assembler-Programme (vgl. Abbildung 4.1) für die Validierung aus der Prozessorspezifikation erzeugen. Die Validierungsmethoden werden in

Kapitel 5 beschrieben.

Bei der Beschreibung der Sprache wird für die Struktur ein Formalismus durch Wertebereiche und Mengenbeschreibungen angegeben. Die so beschriebenen Formalismen werden später bei der Beschreibung des Fehlermodells in Abschnitt 5.2 und bei der Entwicklung der Validierungsmethoden verwendet.

4.3.1 Anforderungen an die Sprache

Wie aus der Vorstellung der Domäne und den Anforderungen an das System folgt, wird eine Sprache benötigt, welche die Architektur eines Prozessors auf einem hohen Abstraktionsniveau formell beschreibt.

Neben dem geforderten Abstraktionsniveau, das durch die Verwendung in den frühen Entwicklungsphasen verlangt wird, muss die Sprache die bekannten Begriffe und Darstellungen aus dem Prozessorentwurf aufgreifen. Damit werden die Anforderungen an die Darstellung der Sprachkonstrukte beschrieben. Abschließend kann eine Anforderung abhängig von den Zielen bei der Entwicklung eines Prozessors oder Simulators formuliert werden. Um den exakten Rahmen für die Sprache zu definieren, werden im Folgenden die Anforderungen an die Visualisierung und Ausdrucksmöglichkeiten für die Prozessorkonstrukte und Strukturen genauer beschrieben.

Die Sprache muss die Prozessorspezifikation durch Konstrukte aus der Domäne visuell darstellen. In den informellen Dokumentationen zu den Prozessoren werden die Eigenschaften verschiedener Konstrukte wie z. B. die Mikroarchitektur, das Verhalten und das Timing der Instruktionen, die Adressierungsmethoden oder der Registersatz meistens unabhängig voneinander an verschiedenen Stellen der Dokumentation beschrieben. Für die Akzeptanz seitens der Prozessorentwickler und eine komfortable Bedienung der Sprache muss die Spezifikation der Prozessorkonstrukte an den erwarteten Stellen modelliert werden, sodass die wichtigen Spezifikationsdetails übersichtlich dargestellt werden. Außerdem müssen die Sprachelemente in einer abstrakten Struktur verknüpft werden, damit die Prozessorspezifikation als Modell für die Generierung eines Simulators oder für die Validierung der Prozessorspezifikation im Entwurfsprozess benutzt werden kann.

In den frühen Phasen kann die Prozessorentwicklung mit unterschiedlichen Ansätzen starten. So kann je nach Entwicklungszielen zunächst ein Instruktionssatzsimulator (vgl. Abschnitt 2.4.1) benötigt werden. Im fortgeschrittenen Entwicklungsprozess kann von dem Prozessorentwickler ein Mikroarchitektursimulator verlangt werden, wobei die Struktur der Mikroarchitektur definiert werden soll. Für diesen Zweck kann der Instruktionssatzsimulator durch eine vorgegebene Mikroarchitektur erweitert werden. Aus dieser Anforderung folgt, dass die Sprache zwei Szenarios für die Spezifikation von Prozessoren und die Erzeugung der o.g. Simulatoren unterstützen muss.

4.3.2 Gliederung der Sprache ViCE-UPSLA

In dem Grundlagenabschnitt 2.2 wurde die Beschreibung der Komponenten einer Prozessorarchitektur in vier Bereichen angegeben. Die Einteilung in Mikroarchitektur, Instruktionssatz, Adressierungsmethoden und Registersatz wird oft für die Gliederung der informellen Beschreibungen von Prozessoren oder in der Fachliteratur zur Prozessorentwicklung verwendet. Diese Bereiche werden als Ausgangspunkte für die Gliederung der Sprachkonstrukte und Sichten in *ViCE-UPSLA* übernommen.

Für die Generierung eines Instruktionssatzsimulators müssen die Komponenten des Instruktionssatzes sowie der Zugriff auf die Register oder Speicher des Prozessors beschrieben sein. Für den Instruktionssatz werden die strukturelle Spezifikation der Instruktionen und deren Verhaltensbeschreibung benötigt. Die strukturelle Spezifikation enthält die Kodierung und die Operanden der Instruktionen. Das Verhalten der Instruktionen wird durch die operationale Beschreibung ausgedrückt, indem die Anwendung von Operationen auf die Operanden beschrieben wird. In *ViCE-UPSLA* werden die Komponenten für einen Instruktionssatzsimulator wie in Definition 4.1 angegeben.

$$InstrCPU := RegSet \times ISA \times InstrForms \times AddrArt \tag{4.1}$$

Die Prozessorspezifikation `InstrCPU` wird durch den Registersatz *RegSet*, den Befehls- oder Instruktionssatz *ISA*, die Instruktionsformate *InstrForms*, die Adressierungsmethoden *AddrArt* ausgedrückt. Die Konzepte der Adressierungsmethoden und der Instruktionsformate werden manchmal mit der Beschreibung von Instruktionen angegeben. Zur Unterstützung einer systematischen Vorgehensweise bei dem Entwurf einer Prozessorspezifikation ist es sinnvoll, die Instruktionsformate mit Adressierungsmethoden in der Gliederung zu kapseln. Die Vorteile dieser Gliederung liegen in der Wiederverwendbarkeit der Konstrukte. Die Instruktionsformate und Adressierungsmethoden bilden damit eine Schnittstelle zwischen dem Instruktionssatz und dem Registersatz. Weitere Eigenschaften dieser Gliederung werden in den folgenden Abschnitten herausgestellt.

Bei der Erweiterung einer Prozessorspezifikation für die Simulation des Verhaltens mit einer Mikroarchitektur muss die Mikroarchitektur selbst spezifiziert und mit den bereits beschriebenen Komponenten verknüpft werden. Eine zu einer Mikroarchitektur erweiterte Spezifikation wird um die Spezifikation der Mikroarchitektur *MicroArch* ergänzt, wie in Definition 4.2 beschrieben.

$$MikroCPU := MicroArch \times RegSet \times ISA \times InstrForms \times AddrArt \tag{4.2}$$

Die visuelle Sprache wird durch verschiedene Sichten auf die Sprachkonstrukte des Prozessorentwurfs gegliedert. In Abbildung 4.3 sind die Screenshots aus dem Werkzeug für die vier Sichten der Sprache dargestellt. Durch die Zeiger werden die Abhängigkeit der Elemente und der logische Entwicklungsfluss für die Spezifikation eines Prozessors angedeutet.

Abbildung 4.3 wird in die Komponenten aufgeteilt, die für die Erzeugung eines Instruktionssatz- und eines Mikroarchitektursimulators benötigt werden. Für die Erzeugung eines Instruktionssatzsimulators müssen zunächst die Komponenten des Registersatzes spezifiziert werden. Diese werden sowohl in der Spezifikation des Instruktionssatzes als auch bei der Beschreibung der Adressierungsmethoden und der Instruktionsformate verwendet. Die Instruktionsformate spezifizieren die strukturellen Eigenschaften der Instruktionen und werden bei der Spezifikation der Instruktionen benutzt. Aus der Beschreibung dieser Komponenten wird ein Instruktionssatzsimulator erzeugt.

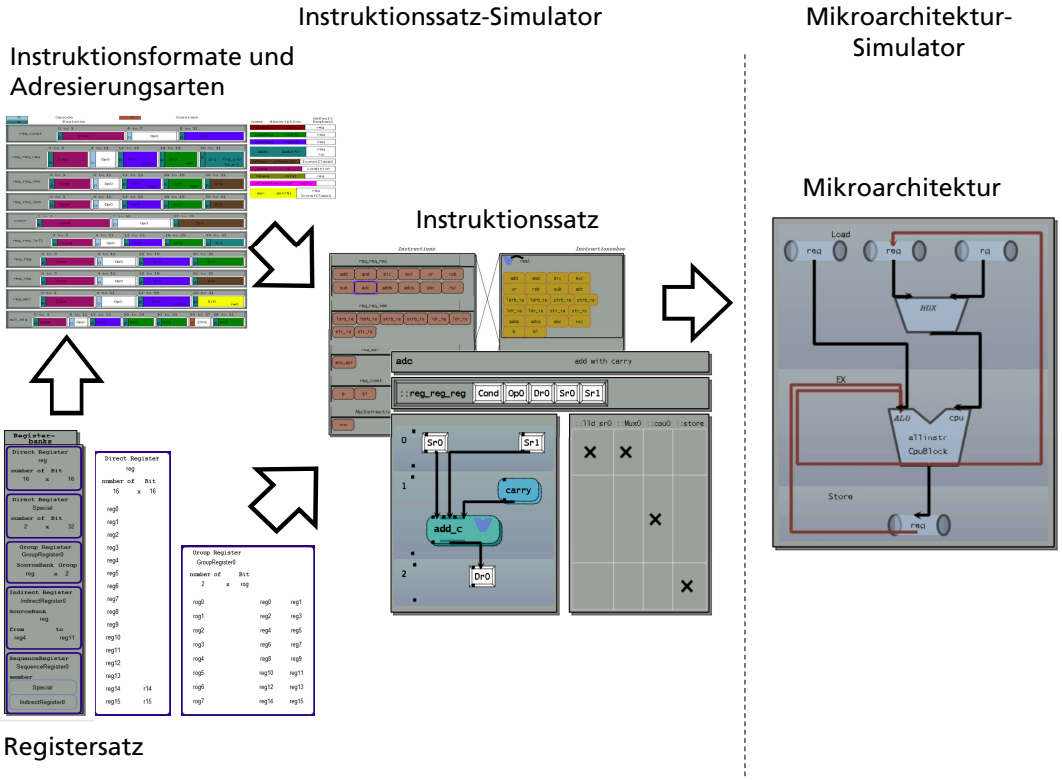


Abbildung 4.3: ViCE-UPSLA Gliederung.

Die Mikroarchitektur beschreibt die Pipeline-Struktur und den Datenpfad des Prozessors. Damit werden die Ressourcen wie z. B. ALU bzw. Lese- und Schreib-Ports beschrieben, welche wiederum mit dem Instruktionssatz bzw. Registersatz verknüpft sind. Die Verknüpfungen zwischen den Komponenten des Prozessors erzeugen eine zusammenhängende und ganzheitliche Prozessorspezifikation, welche zur automatischen Generierung eines Mikroarchitektursimulators genügt und für die Validierung verwen-

det werden kann. Durch die Verknüpfung der Sprachelemente soll der Spezifikationsaufwand verringert werden, indem bereits spezifizierte Konstrukte bei der Beschreibung neuer Komponenten berücksichtigt und als Bausteine für die Modellierung verwendet werden.

In den folgenden Abschnitten werden die Prozessorkonstrukte und die Visualisierung in der Sprache *ViCE-UPSLA* entsprechend der Vorgehensweise aus Abbildung 4.3 beschrieben.

4.3.3 Registersatz

Das Konzept des Registersatzes in *ViCE-UPSLA* umfasst Speicherstellen, die für die Simulation eines Prozessors auf einer hohen Simulationsebene benötigt werden, wie in Abschnitt 2.4.1 beschrieben. Dabei werden alle von dem Programmierer erreichbaren Register [Mic94, HP06] spezifiziert, Hilfsregister wie z. B. Pipelineregister werden nicht modelliert. Damit wird die Spezifikation des Registersatzes auf die Registerbänke und Register sowie die Wertebereiche, welche als Operanden bei der Ausführung von Instruktionen verwendet werden, eingegrenzt.

Bei der Spezifikation des Registersatzes in *ViCE-UPSLA* wird neben den physikalischen Registern das Konzept der architektonischen Register [DHTK07], wie in Abschnitt 2.2.4 beschrieben, verwendet. Der Registersatz *RegSet* wird beschrieben durch die physikalischen Register *PhysReg*, die architektonischen Register *ArchReg*, die Spezialregister *Spc* und Klassen von Wertebereichen oder Speicher *Icc*, wie in Definition 4.3 zusammengefasst.

$$RegSet := PhysReg \times ArchReg \times Spc \times Icc \tag{4.3}$$

Für die Umsetzung des Aliasing durch die architektonischen Register werden drei verschiedene Konzepte verwendet. Insgesamt stehen dem Prozessorentwickler folgende sechs Konstrukte zur Verfügung:

- *DirectRegister* wird für die Spezifikation der physikalischen Register verwendet.
- Die architektonischen Register *ArchReg* werden durch drei Konzepte beschrieben:
  - *GroupRegister*.
  - *IndirectRegister*.
  - *SequenceRegister*.
- *IconstClass* spezifiziert die Wertebereiche für die unmittelbaren Operanden.
- *Memory* wird für die Spezifikation von Speicher verwendet.

Die *DirectRegister* in *ViCE-UPSLA* beschreiben die physikalischen Register *PhysReg* eines Prozessors, wie in Definition 4.4 angegeben. Eine Registerbank wird durch einen Namen *Name* und einen Offset für die Register *Offset* sowie die Anzahl der Register *Length* definiert. Außerdem wird für die Register ihre Breite *Size* in Bits spezifiziert. Damit werden Speicherstellen spezifiziert, welche in der weiteren Spezifikation des Prozessors durch die Adressierungsmethoden verwendet werden können. Für die eindeutige Identifikation erhalten die einzelnen Register des Prozessors individuelle Bezeichner *Ident*, außerdem wird ihre Breite *Size* in Bits angegeben. Im Prozessorsimulator werden die Adressen der Speicherstellen anhand der Bezeichner lokalisiert. Während der Simulation werden in diesen Speicherstellen Werte gespeichert, womit der Systemzustand definiert wird.

$$\begin{aligned} PhysReg &:= Name \times Offset \times Length \times Size \\ Reg &:= Ident \times Size \times IsSpecial \end{aligned} \tag{4.4}$$

In den Instruktionen können unmittelbare Operanden (implicit operand) wie z. B. `carry`-Bit oder `PC` Register verwendet werden. Der Zugriff auf diese Art von Registern erfolgt unmittelbar ohne Adressierungsmethoden. Für die Spezifikation der Spezialregister werden ebenfalls die physikalischen Register verwendet. Diese werden erzeugt, indem zu einem physikalischen Register die Eigenschaft *IsSpecial* mit einem zusätzlichen Bezeichner hinzugefügt wird.

Das Konzept der architektonischen Registerbänke umfasst die Erzeugung von verschiedene Anordnungen der physikalischen Register. Diese werden benötigt, um z. B. verschiedene Modi der Registerbenutzung zu beschreiben, wie in den Abbildungen 4.4 und 4.5 dargestellt. Ohne die Beschränkungen durch Entwurfsrichtlinien sind die möglichen Neuanordnungen der physikalischen Register sehr umfangreich. Wie in der formalen Spezifikation 4.5 angegeben, werden die architektonischen Register durch eine beliebige Anordnung durch die Verwendung neuer Bezeichner erlaubt.

$$\begin{aligned} ArchReg &:= Name \times aReg \\ aReg &:= Alias \times \{(r_x, ..., r_y) \mid r \in Reg; x, y \in \mathbb{N}\} \end{aligned} \tag{4.5}$$

In *ViCE-UPSLA* sind für die Umsetzung der architektonischen Register drei Konzepte umgesetzt, die im Prozessorentwurf häufig verwendet werden und die meisten Konstellationen mit einem überschaubaren Spezifikationsaufwand ermöglichen. Dazu gehören die Konkatenation von Registerbänken (in *ViCE-UPSLA SequenceRegister*), die Konkatenation von Registern (in *ViCE-UPSLA GroupRegister*) und der Ausschnitt einer Registerbank (in *ViCE-UPSLA InDirectRegister*). Im Folgenden werden diese Konzepte vorgestellt.

Mit dem Ausschnitt einer Registerbank lässt sich eine beliebige Folge der Register in aufsteigender oder absteigender Reihenfolge beschreiben. Dieses Konstrukt beschreibt eine Referenz auf eine Ziel-Registerbank sowie auf das erste und das letzte Register dieser Bank. In Abbildung 4.4 stellt die Registerbank *UserMode* einen Teil der Registerbank *reg* dar und beinhaltet acht Register zwischen `reg0` und `reg15`.

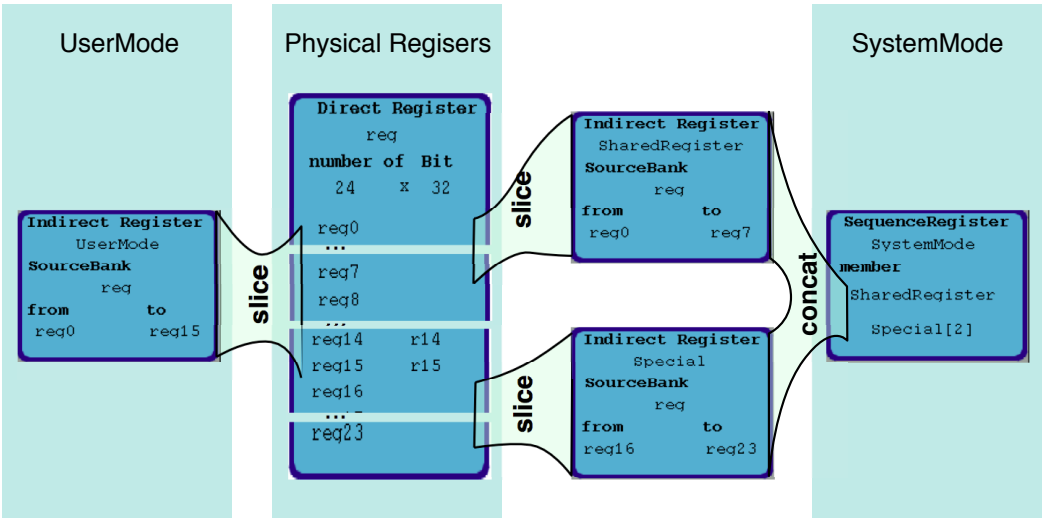


Abbildung 4.4: Registerbanksichten für den Benutzermodus und den Systemmodus mit jeweils 16 Registern.

Die Konkatenation der Registerbänke reiht die Register mehrerer Registerbänke aneinander und fasst diese in einer neuen zusammen. Dabei müssen nur die Ziel-Bänke in der entsprechenden Reihenfolge ausgewählt werden. In Abbildung 4.4 ist die Registerbank `SharedRegister` und die Registerbank `Special` mit Spezialregistern dargestellt, welche in der Registerbank `SystemMode` zusammengefasst werden.

Bei der Konkatenation von Registern werden Register einer physikalischen Registerbank zu größeren architektonischen Registern zusammengefasst. Dabei wird eine physikalische Registerbank referenziert und die Größe für die Gruppierung angegeben. Sind die Register aus der physikalischen Registerbank 16 Bit breit und werden sie durch die architektonische Registerbank in Zweiergruppen zusammengefasst, so sind die neuen architektonischen Register 32 Bit breit. Würden Dreiergruppen gebildet werden, würden diese 48 Bit breit werden usw. In Abbildung 4.5 werden die Register der Registerbank `reg` zu Paaren mit dem Offset `rog` in der Registerbank `GroupRegister0`

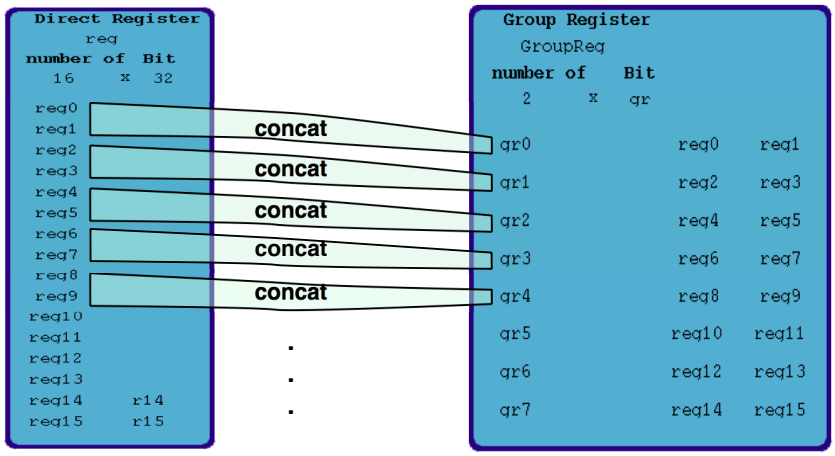


Abbildung 4.5: Erzeugung größerer Register.

zusammengefasst und sind somit 32 Bit breit.

Durch die Kombination der Abbildungsmöglichkeiten für die architektonischen Register kann eine Vielzahl verschiedener Anordnungen der physikalischen Register umgesetzt werden. Als Grundlage muss jedoch immer eine physikalische Registerbank vorhanden sein. In Abbildung 4.4 wird gezeigt, wie z. B. die User und System Mode Register spezifiziert werden. Dabei hat die physikalische Registerbank die Register `reg0-23`. Für den User Mode wird aus dieser Registerbank einfach ein Ausschnitt der Register `reg0-15` erzeugt und als Register `usr0-15` benannt. Für den System Mode werden die Ausschnitte der Registerbank `reg` in den Registerbänken `SharedRegister` und `Special` extrahiert und in der Registerbank `SystemMode` zusammengefasst. Dabei werden die Register `reg0-7` und `reg16-23` als Register `sm0-7` und `sm8-15` benannt.

4.3.4 Adressierungsmethoden und Instruktionsformate

Bei dem Entwurf eines Prozessors wird bereits in den frühen Phasen der Entwicklung die Spezifikationen der Instruktionskodierung festgelegt. Dabei wird z. B. die Entscheidung über die Größe der Operanden, die Breite der Instruktionen, oder ob es eine CISC- oder RISC-Architektur sein soll, gefällt. Anschließend wird die Kodierung für die Instruktionen festgelegt, welche meistens für Gruppen von Instruktionen angegeben wird. Es ist sinnvoll, zuerst die verschiedenen Instruktionsformate für die Instruktionen festzulegen, um anschließend eine systematische Spezifikation der Instruktionen zu erreichen.

In *ViCE-UPSLA* werden die Konzepte der Instruktionsformate und der Adressie-

rungsmethoden in einer Sicht dargestellt und durch eine flexible Kopplung zwischen diesen Konstrukten umgesetzt. Somit beschreiben die Instruktionsformate lediglich die statischen Eigenschaften einer Instruktion. Die Adressierungsmethoden beschreiben die dynamische Komponente zur Berechnung der Adressen bzw. Werte, die bei der Ausführung der Instruktion verwendet werden. In einer Spezifikation können entsprechend der Anzahl der Instruktionsgruppen beliebig viele Instruktionsformate oder Adressierungsmethoden spezifiziert werden. Eine Verknüpfung der Adressierungsmethoden mit den Instruktionsformaten vervollständigt die Spezifikation für die Kodierung und Benutzung von Instruktionen.

Bei der Beschreibung von Operanden für eine Instruktion wird in *ViCE-UPSLA* zwischen Befehlsoperanden und interpretierten Operanden differenziert. In *ViCE-UPSLA* werden in Instruktionsformaten die interpretierten Operanden angegeben, welche für die Berechnung der Ergebnisse der Instruktionen verwendet werden. Die Spezifikation der interpretierten Operanden wird durch die entsprechenden Adressierungsmethoden vervollständigt. Die Adressierungsmethoden beschreiben die Funktion für den Zugriff auf die Register des Prozessors, wie in Abschnitt 2.2 beschrieben. Die Befehlsoperanden werden in den Adressierungsmethoden definiert und als Parameter für die Berechnung der Adressen oder Werte verwendet. Das Prinzip für die Verwendung der Adressierungsmethoden wird in Abbildung 4.6 dargestellt. Abhängig davon, ob ein Wert geladen oder gespeichert wird, berechnet die Adressierungsmethode den Wert für den interpretierten Operanden oder das Ergebnis, das im Registersatz gespeichert wird.

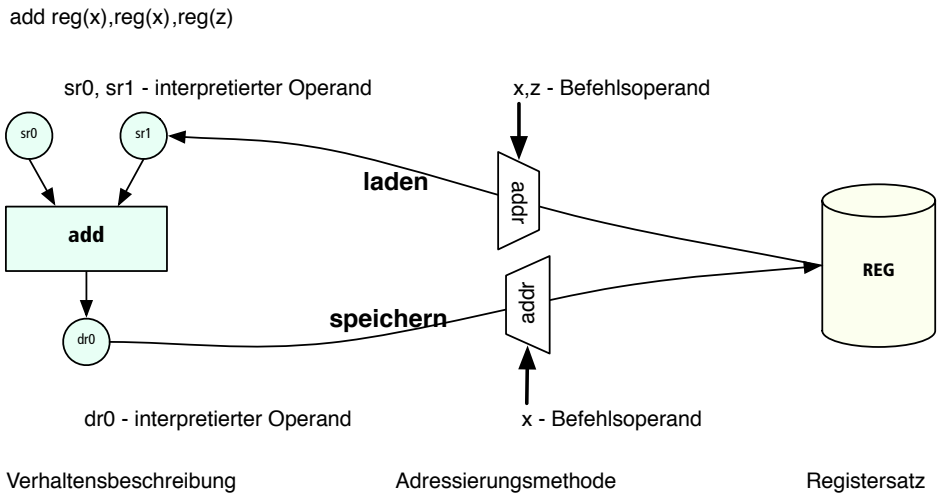


Abbildung 4.6: Verwendung der Adressierungsmethoden in Lese- und Schreibrichtung.

4.3.4.1 Instruktionsformate

Die Spezifikation der Instruktionsformate in *ViCE-UPSLA* wird durch eine domänentypische Visualisierung des Prozessorentwurfs ausgedrückt. Beispiele für die visuelle Darstellung sind in Abbildung 4.7 abgebildet. Die Aufgabe dieses Sprachkonstruktes ist es, eine systematische Spezifikation für die Kodierung der Instruktionen zu ermöglichen. In Abschnitt 4.3.5.1 wird beschrieben, wie die Instruktionsformate den Instruktionen zugeordnet werden und damit die Verknüpfung zu den Adressierungsmethoden beschreiben.

Definition 4.6 beschreibt formal die Komponenten der Instruktionsformate. Ein Instruktionsformat *InstrForm* wird zunächst in eine nicht leere Folge *Bitfields* von Feldern *Bitfield* eingeteilt. Für jedes Bitfeld wird seine Breite *Width*, Position *Position* und die Variable für die Art des Feldes zugeteilt. Die Bitfelder werden entweder durch einen Operationscode *Opc* oder einen interpretierten Operanden *intOpd* beschrieben. Die interpretierten Operanden *intOpd* werden durch einen Bezeichner und eine Adressierungsmethode spezifiziert. Bei der Spezifikation der interpretierten Operanden *intOpd* muss jedem Operanden die Berechnung seines Wertes *calculate* durch eine Adressierungsmethode zugeordnet werden.

$$\begin{aligned} InstrForm &:= Bitfields \\ Bitfield &:= Position \times Width \times BfVar \\ BfTyp &:= \{isOpc, isintOpd\} \\ BfVar &:= \{(isOpc, o) \mid o \in Opc\} \cup \{(isintOpd, p) \mid p \in intOpd\} \\ intOpd &:= Ident \times Addr \\ calculate &:= Addr \rightarrow intOpd \end{aligned} \tag{4.6}$$

Die Breite eines Instruktionsformats resultiert aus der Summe der Breiten seiner Bit-Felder. Dies erlaubt die Spezifikation von Instruktionsformaten mit unterschiedlichen Breiten und somit unterschiedlich langen Instruktionsworten in einer Architektur, womit z. B. CISC-Architekturen [SG79, NMEH81, HP06] beschrieben werden können.

Die Einteilung der Instruktionsformate in Bit-Felder spezifiziert zum einen die Breite der Instruktionen, zum anderen die Breite und somit auch die Wertebereiche der einzelnen Operanden und des Operationscodes. Abbildung 4.7 zeigt an einem Beispiel den generierten Code für die Instruktionskodierung aus der visuellen Spezifikation. Die Breite der Komponenten wird später für die Validierung der Spezifikation in Kapitel 5 verwendet, um z. B. die Konsistenz der Mengenkodierung zu überprüfen.

In einigen Prozessorarchitekturen wird das Konzept der Steuerbits verwendet, welche z. B. das Verhalten der Ausführung von Instruktionen oder die Auswahl der Adressierungsmethode beeinflussen, wie in den Grundlagen in Abschnitt 2.2 beschrieben. In *ViCE-UPSLA* werden die Steuerbits nicht explizit, sondern implizit als eine Erwei-

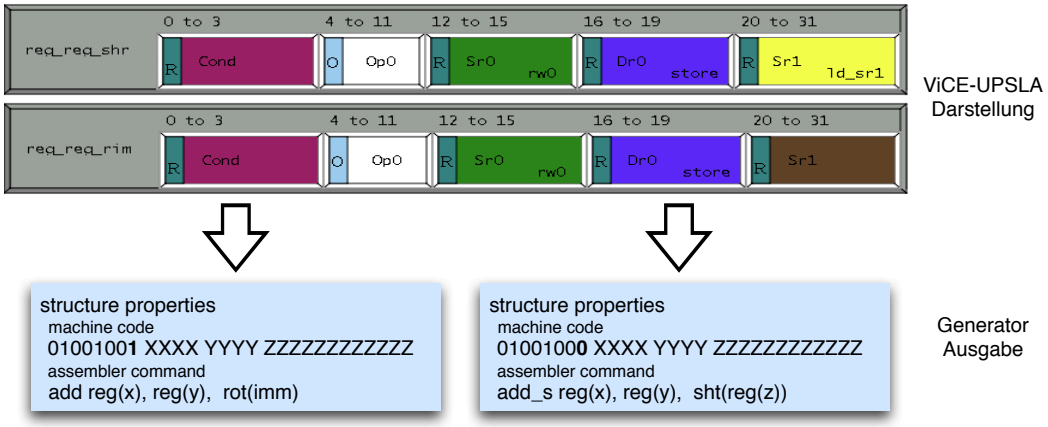


Abbildung 4.7: Zwei Instruktionsformate und die daraus resultierende Kodierung der Instruktionen am Beispiel des generierten Codes.

terung des Operationscodes vorgesehen. Dabei wird die Breite des Operationscodes um die Anzahl der Steuerbits erweitert. Daraus folgend werden die Instruktionen mit unterschiedlichen Konfigurationen in verschiedene Gruppen eingeteilt.

4.3.4.2 Adressierungsmethoden

Mit der Beschreibung des Operationscodes und der interpretierten Operanden sind die Elemente zur weiteren Spezifikation in den Instruktionen vollständig angegeben. Zu beachten ist, dass dieser Schritt entkoppelt von den Adressierungsmethoden durchgeführt wird. Damit die Befehlsoperanden aus der Spezifikation abgeleitet werden können, werden die Adressierungsmethoden, wie in Abbildung 4.8 dargestellt, mit den interpretierten Operanden im Instruktionsformat verknüpft. Die Adressierungsmethoden werden in der Spezifikation durch unterschiedliche Farben visuell voneinander abgegrenzt. Bei der Verknüpfung von interpretierten Operanden übernimmt letzterer die Farbe der zugeordneten Adressierungsmethode, sodass eine Zuordnung einfach erkannt wird.

Die formale Struktur der Adressierungsmethoden wird in Definition 4.7 angegeben. Die Adressierungsmethoden werden im Simulator oder im Prozessor den Instruktionen zugeordnet und beschreiben damit einen Teil der Instruktionssemantik. Eine Adressierungsmethode *Addr* wird durch eine Berechnungsvorschrift *Method* für die Berechnung der Werte und beliebig viele Befehlsoperanden *instrOpd* spezifiziert. Die Befehlsoperanden können vom Typ Registeroperanden *regOpd* oder unmittelbaren Operanden *immOpd* sein.

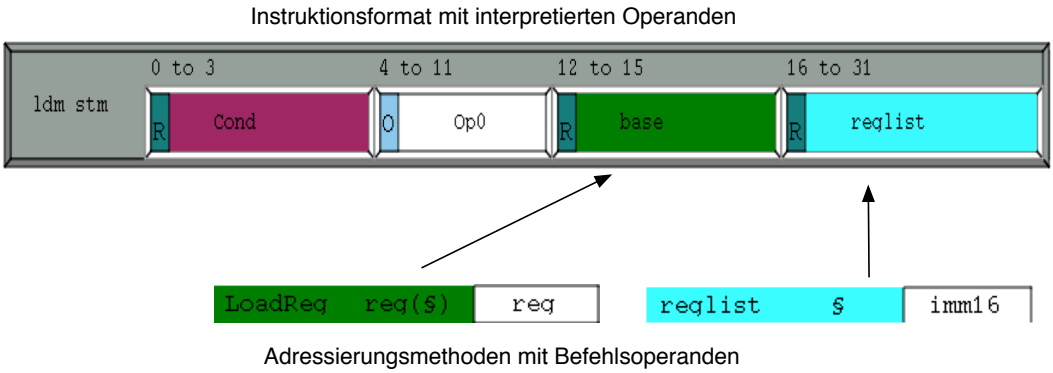


Abbildung 4.8: Visualisierung der Instruktionsformat und Adressierungsmethoden in ViCE-UPSLA.

$$\begin{aligned} Addr &:= Method \times instrOpd \\ OpdType &:= regOpd, immOpd \\ instrOpd &:= Length \times OpdVar \\ OpdVar &:= \{(regOpd, o) \mid o \in Register\} \cup \{(immOpd, o) \mid o \in IconstClass\} \end{aligned} \tag{4.7}$$

Aus der Sicht der Struktur beschreibt die Spezifikation der Adressierungsmethode in *ViCE-UPSLA* die Zuordnung zwischen den Bit-Feldern aus dem Instruktionsformat, den interpretierten Operanden und den Befehlsoperanden. Aus der Sicht der Verhaltensbeschreibung werden durch die Adressierungsmethoden die Berechnungsvorschriften ausgedrückt, die aus den Befehlsoperanden die interpretierten Operanden berechnen.

Die Befehlsoperanden werden in den Adressierungsmethoden angegeben und spezifizieren eine Verknüpfung zu einer Registerbank oder einem Wertebereich. Wird einem Befehlsoperanden ein Register oder eine Registerbank zugeordnet, handelt es sich um einen Register-Operanden *regOpd*. Diese Art von Operanden setzt voraus, dass vor oder nach der Auswertung der Adressierungsmethode der Wert des Operanden aus dem Register-Satz geladen wird. Wird einem Operanden ein Wertebereich zugeordnet, handelt es sich um einen unmittelbaren Operanden *immOpd*, der aus dem Instruktionswort entnommen wird.

Wie bereits beschrieben, können zwei Arten von Befehlsoperanden spezifiziert werden: die Register- und die unmittelbaren Operanden. In *ViCE-UPSLA* können die Adressierungsmethoden mehrere Befehlsoperanden enthalten. Anhand der unterschied-

lichen Kombinationen der Befehlsoperanden und der Benutzungsrichtungen können die Adressierungsmethoden, wie in Abschnitt 2.2.3 beschrieben, klassifiziert werden, wodurch den Methoden für die Simulation oder Generierung bestimmte Eigenschaften zugeordnet werden können.

Mit diesem Konzept lässt sich in *ViCE-UPSLA* ein breites Spektrum für die Beschreibung von Adressierungsmethoden umsetzen. Für die Erzeugung einfacher Instruktionssatz-Simulatoren können direkte Adressierungsmethoden umgesetzt werden. Das Konzept ist aber auch mächtig genug, um Architekturen mit komplexen Strukturen der Adressierung mit `Pre load` und `Post load` Methoden [SG79, Mot92] zu beschreiben, in denen die Adressierung durch eine große Anzahl von Befehlsoperanden und unterschiedlichen Berechnungsarten durchgeführt wird.

In diesem Abschnitt wurde die Modellierung der Adressierungsmethoden und der Instruktionsformate vorgestellt. Durch diesen Ansatz können die Instruktionsformate für die Spezifikation von Instruktionen gelöst von den Adressierungsmethoden betrachtet und spezifiziert werden. Damit lassen sich strukturiert die Instruktionsformate von bereits vorhandenen Architekturen wie RISC und CISC spezifizieren, sowie beliebig komplexe Adressierungsmethoden wie z. B. von Motorola 68000 Prozessoren verwendet.

4.3.5 Instruktionssatz

Der Instruktionssatz ist die zentrale Komponente für die Erzeugung eines Simulators. Damit ein Simulator für den Prozessor erzeugt werden kann, müssen zum einen die strukturellen Eigenschaften der Instruktionen angegeben werden, die die Benutzung der Instruktionen festlegen. Zum anderen muss das Verhalten der Instruktionen beschrieben sein. Die Spezifikation des Instruktionssatzes wird neben der Entwicklung des Prozessors von den Programmierern für die Erzeugung von Assemblerprogrammen, Compilern oder Simulatoren eines Prozessors verwendet. Hierfür werden die Eigenschaften der Instruktionssätze in Dokumenten wie z. B. „MOTOROLA M68000 FAMILY Programmer's Reference Manual“ [Mot92] zusammengefasst.

Die Spezifikation des Verhaltens einer Instruktion kann durch die operationale Beschreibung unterschiedlich detailliert angegeben werden. Für die Erzeugung eines einfachen Instruktionssatzsimulators werden in der operationalen Beschreibung der Instruktionen lediglich die Berechnungsvorschriften der Instruktion umgesetzt. Für die Simulation eines Prozessors mit einer vorgegebenen Mikroarchitektur muss die Spezifikation eine zyklengenaue operationale Beschreibung sowie einen Ressourcennutzungsplan enthalten.

Im Folgenden wird die Vorgehensweise für die Spezifikation eines Instruktionssatzes mit *ViCE-UPSLA* beschrieben. Dazu wird zunächst das Konzept der abstrakten Instruktionen in *ViCE-UPSLA* vorgestellt. Anschließend werden die Vorgehensweise

und die Komponenten für die Spezifikation der strukturellen Eigenschaften und die zyklusgenaue operationale Beschreibung des Instruktionsverhaltens vorgestellt. Für das Konzept der operationalen Beschreibung der Instruktionen werden die Ausdrucksmöglichkeiten für das Verhalten und die damit möglichen Rechnerarchitekturen in der Sprache *ViCE-UPSLA* vorgestellt.

4.3.5.1 Struktur des Instruktionssatzes

Die Vorstellungen über die spezifizierten Komponenten eines Instruktionssatzes variieren je nach Entwicklergruppe und Entwicklungszielen. Für die Schaltungstechniker sind z. B. die Eigenschaften der verwendeten Operatoren in den Instruktionen, oder welche Busbreiten verwendet werden, von Interesse. Für die Softwareentwickler sind die statischen Eigenschaften für die Benutzung der Instruktionen und die Erzeugung der Programme für die Prozessoren relevant. Diese Entwicklergruppe benötigt z. B. die Übersicht über die Instruktionen und deren Parameter. Bei der Entwicklung und Simulation von Prozessoren werden neben den statischen Eigenschaften der Instruktionen deren operationale Beschreibung, die Ressourcennutzung und das Zeitverhalten benötigt.

Das Konzept in *ViCE-UPSLA* beschreibt den Instruktionssatz, indem die strukturellen und Verhaltenseigenschaften der Instruktionen kombiniert angegeben werden. Die Spezifikation der strukturellen Eigenschaften wird mit der operationalen Beschreibung verknüpft angegeben und dargestellt. Das Verhalten der Instruktionen und die verwendeten Ressourcen werden dabei zyklengenau beschrieben. Damit bietet die Beschreibung für die unterschiedlichen Entwicklergruppen die relevanten Informationen.

Für die Strukturierung des Instruktionssatzes wird in dieser Arbeit das Konzept der abstrakten Instruktionen eingeführt. Damit lassen sich die Instruktionen des Prozessors klassifizieren und der Spezifikationsaufwand verringern. Der Instruktionssatz *ISA* wird, wie in (4.8) angegeben, durch die abstrakten und die konkreten Instruktionen beschrieben. Jede konkrete Instruktion *Instr* des Prozessors ist einer abstrakten Instruktion *Ais* zugeordnet. Die abstrakten Instruktionen beschreiben Äquivalenzklassen, indem sie Gruppen von konkreten Instruktionen zusammenfassen.

$$ISA := Instr \times Ais \tag{4.8}$$

Das Konzept der abstrakten Instruktionen sieht vor, dass sie die Äquivalenzklassen von Instruktionen beschreiben. Damit besitzen alle Instruktionen, die einer abstrakten Instruktion zugeordnet sind, bestimmte Eigenschaften. Die Äquivalenzklassen können nach verschiedenen Aspekten der Instruktionen, z. B. dem Instruktionsformat, der Ausführungsdauer, der Ressourcenbelegung usw., je nach Wünschen des Prozessorentwicklers gebildet werden. Das Konzept der abstrakten Instruktionen ermöglicht somit eine strukturierte Spezifikation des Instruktionssatzes. Die Äquivalenzklassen der In-

struktionen können z. B. für die Validierung benutzt werden, wie in Abschnitt 2.5.2 beschrieben.

Für die abstrakten Instruktionen kann sowohl eine strukturelle als auch eine operationale Beschreibung angegeben werden, die im Folgenden beschrieben werden. Bei der Erzeugung einer konkreten Instruktion werden die spezifizierten Eigenschaften der abstrakten Instruktion übernommen, indem eine abstrakte Instruktion als Schablone für die Erstellung einer konkreten Instruktion verwendet wird. Die Spezifikation der Instruktionen kann anschließend unabhängig von der abstrakten Instruktion weiter vervollständigt oder geändert werden.

4.3.5.2 Instruktion

Die Spezifikation einer Instruktion wird durch die strukturelle und operationale Beschreibung ausgedrückt. Die strukturelle Beschreibung beinhaltet im Wesentlichen die Spezifikation für die Benutzung einer Instruktion. Die operationale Beschreibung spezifiziert detailliert die Ausführungsschritte und die dabei ausgeführten Operationen. Ausführungsschritte werden durch Zyklen spezifiziert; verteilt auf die Zyklen der Instruktion werden die Operationen in einem Datenflussgraph angegeben. Außerdem kann für die Verhaltensbeschreibung die Belegung der Ressourcen angegeben werden.

In *ViCE-UPSLA* sind die Struktureigenschaften und die operationale Beschreibung eng miteinander verknüpft und werden in einer Sicht dargestellt. Für das bessere Verständnis der Sprachkonzepte und der Abstraktionsebenen in *ViCE-UPSLA* wird zuerst ein Beispiel vorgestellt. Abbildung 4.9 zeigt die Ansicht einer Instruktion sowie die Liste der verfügbaren Operatoren und der Instruktionsformate in der Spezifikation. In der Instruktionssicht wird die Aufteilung der visuellen Darstellung einer Instruktion in der Sprache *ViCE-UPSLA* gezeigt. Als Beispiel wird die Spezifikation einer *adc*-Instruktion, Addition mit *carry*-Bit, verwendet. Für die Instruktion werden die strukturellen Eigenschaften, die operationale Beschreibung und die genutzten Ressourcen dargestellt.

Die Liste der Operatoren sowie die der Instruktionsformate gibt eine Auswahl der Elemente für die Spezifikation, welche im Vorfeld durch den Prozessorentwickler spezifiziert wurden. Die vorgefertigten Bausteine können in der Spezifikation der Instruktionen direkt eingesetzt werden.

Die Darstellung der Instruktionen ist so aufgeteilt, dass die Instruktionsansicht mit dem Namen der Instruktion beginnt, in diesem Beispiel `adc`. Aus der Liste der Instruktionsformate wurde für diese Instruktion das Instruktionsformat `reg_reg_reg` eingesetzt. Wie bereits in Abschnitt 4.3.4 beschrieben, spezifiziert ein Instruktionsformat die Operanden und die Kodierung der Instruktion. Außerdem sind mit dem Instruktionsformat die Adressierungsmethoden für die Operatoren angegeben. Damit ist die Spezifikation der Struktureigenschaften der Instruktion vollständig angegeben.

In der operationalen Beschreibung sind in diesem Beispiel drei Taktzyklen für die

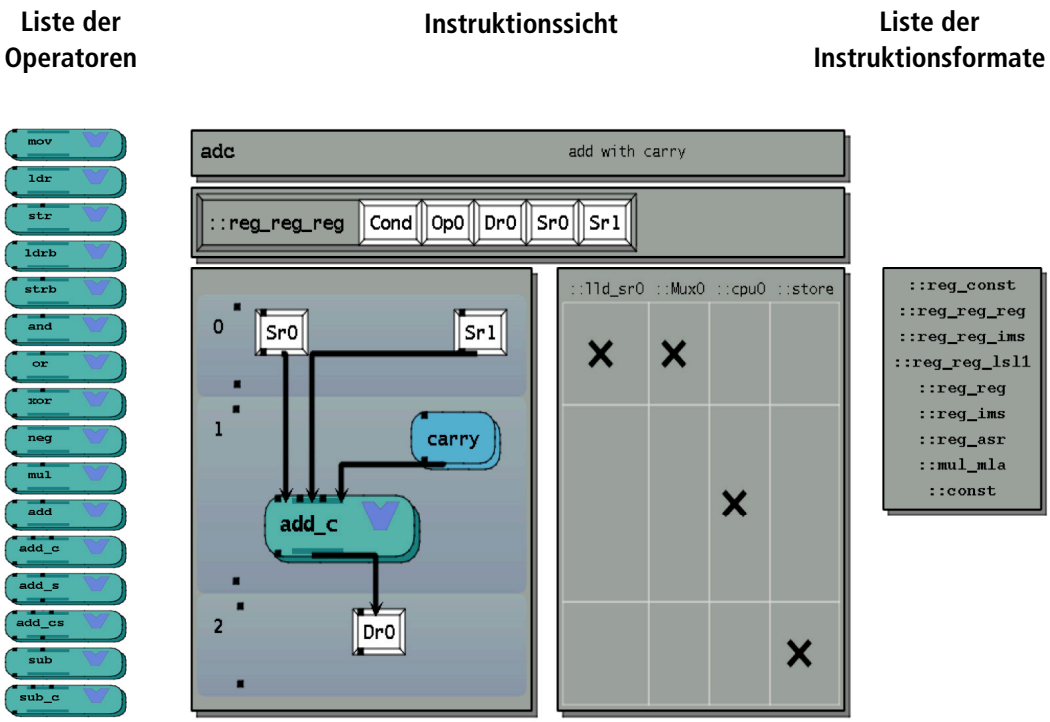


Abbildung 4.9: Sicht auf die visuelle Spezifikation einer Instruktion.

Instruktion angegeben; diese sind das Laden der Eingabeparameter, die Durchführung der Berechnung und das Speichern des Ergebnisses. Die Bezeichnung der Taktzyklen kann für eine bessere Übersicht durch den Benutzer frei gewählt werden. Für die Spezifikation sind die Reihenfolge und die Abgrenzung durch die Zyklen relevant. Die operationale Beschreibung im Beispiel 4.9 gibt an, dass im ersten Taktzyklus die Operanden `Sr0` und `Sr1` geladen werden. Für die Beschreibung dieser Operationen können die Operanden direkt aus dem angegebenen Instruktionsformat der Strukturbeschreibung übernommen werden. Im zweiten Taktzyklus wird zusätzlich ein unmittelbarer Operand `carry`-Bit geladen. Die Operation `add_c` beschreibt die Berechnung des Ergebnisses der Instruktion. Im letzten Taktzyklus ist das Ergebnis von `add_c` verfügbar und wird in den interpretierten Operand `Dr0` geschrieben.

Aus semantischer Sicht wurde das Verhalten dieser Instruktion durch den Operator `add_c` beschrieben, der aus der Liste bereits spezifizierter Operatoren ausgewählt wurde. Dieser Operator bekommt drei Eingabeparameter `Sr0`, `Sr1` und `carry` übergeben und speichert das Ergebnis in dem Operand `Dr0`.

Parallel zu den Taktzyklen der operationalen Beschreibung wird der Ressourcennutzungsplan angegeben. Im ersten Taktzyklus werden zwei Lese-Ports verwendet, im zweiten Taktzyklus die ALU und im dritten Taktzyklus ein Schreib-Port des Prozessors. Die Spezifikation der Ressourcen wird in Verbindung mit der Spezifikation der Mikroarchitektur erzeugt.

Für die Spezifikation einer Instruktion *instr* kann eine formale Beschreibung durch die Definition 4.9 angegeben werden.

$$instr := Ident \times OpCode \times ownsInstrForm \times Cycle \times DFG \times Resource \quad (4.9)$$

Damit werden die Instruktionen durch einen Bezeichner *Ident* und einen Operationscode *OpCode* identifiziert. Die Struktur der Instruktion wird durch ein ausgewähltes Instruktionsformat *ownsInstrForm* definiert. Die operationale Beschreibung wird durch die Ausführungszyklen *Cycle* und einen Datenflussgraphen *DFG* angegeben. Zusätzlich können die verwendeten Ressourcen durch *Resource* angegeben werden. Im Folgenden werden diese Eigenschaften und die Ausdrucksmöglichkeiten bei der Spezifikation genauer erläutert.

4.3.5.3 Struktureigenschaften

Wie bereits beschrieben, werden die Instruktionsformate *InstrForm* in einem früheren Schritt in einer eigenen Ansicht spezifiziert. Bei der Spezifikation der Struktureigenschaften einer Instruktion wird zunächst eine Verknüpfung *ownsInstrForm* zwischen einem Instruktionsformat und der Instruktion erstellt. Für eine konkrete Instruktion kann diese Verknüpfung aus der abstrakten Instruktion übernommen werden, falls diese dort spezifiziert wurde. Die visuelle Darstellung wurde bereits im Beispiel in Abbildung 4.9 gezeigt.

Das Konzept von *ViCE-UPSLA* sieht vor, dass der Prozessorentwickler bei der Spezifikation einer Instruktion lediglich durch die Verknüpfung eines Instruktionsformats die interpretierten Operanden der Instruktion festlegt. Diese werden später in der operationalen Beschreibung verwendet. Aus der Struktur der Sprache sind durch das Zuordnen des Instruktionsformats die statischen Eigenschaften der Instruktion vollständig beschrieben. In Abbildung 4.10 wird durch Pfeile der Pfad der Verknüpfungen für die Instruktion aus Beispiel 4.9 gezeigt. Der Pfad beginnt mit der Definition eines Instruktionsformats aus der Spezifikation einer Instruktion und verläuft bis zu den Konstrukten der Registerbänke und der Wertebereiche.

Die Verknüpfung eines Instruktionsformats legt die interpretierten Operanden mit den Adressierungsmethoden für die Instruktion fest. Die Bit-Felder der interpretierten Operanden und des Operationscodes sind im Instruktionsformat angegeben, womit die Binärkodierung der Instruktion spezifiziert ist. Der Operationscode kann automatisch generiert oder durch den Benutzer vorgegeben werden. Das Assemblerformat wird aus

dem Bezeichner der Instruktion und der Verknüpfung der interpretierten Operanden mit den Adressierungsmethoden abgeleitet. Bei der Generierung eines Simulators werden aus dieser Spezifikation z. B. die binäre Kodierung und der Assemblerbefehl der Instruktion in textueller Form erzeugt, wie im unteren Teil der Abbildung 4.10 dargestellt.

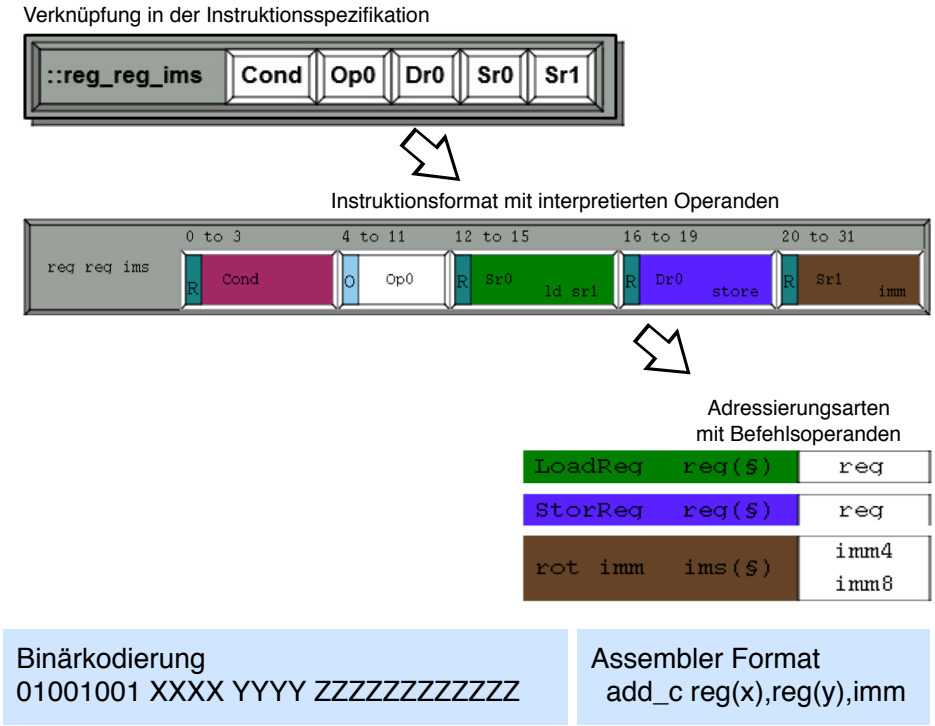


Abbildung 4.10: Ableitungspfad der Struktureigenschaften aus der Spezifikation.

4.3.5.4 Operationale Beschreibung der Instruktionen

Für die Simulation einer Instruktion müssen das Verhalten der Instruktion und die Berechnung der Ergebnisse angegeben sein. In *ViCE-UPSLA* wird diese Spezifikation durch eine operationale Beschreibung durchgeführt. Die ausgeführten Operationen und die Wertübergabe zwischen den Operationen werden in einem Datenflussgraph *DFG* angegeben, wie in Definition 4.10 angegeben. Die Spezifikation der Ausführungszyklen beschreibt die Reihenfolge und die Anzahl der Sequenzen für die Instruktion. Die einzelnen Operationen der operationalen Beschreibung werden den Ausführungszyklen zugeordnet, womit eine zyklusgenaue Spezifikation für die Simulation der Instruktions-

ausführung angegeben ist. Zusätzlich können zu der operationalen Beschreibung einer Instruktion die verwendeten Ressourcen aus der Mikroarchitektur angegeben werden.

Das Konzept der operationalen Beschreibung in *ViCE-UPSLA* erlaubt damit die Spezifikation von Instruktionen für verschiedene Mikroarchitekturen. Die Möglichkeiten bei der Spezifikation werden anschließend an einigen Beispielen für verschiedene Konfigurationen verdeutlicht.

Die Taktzyklen werden vergleichbar mit den Pipeline-Stufen einer Mikroarchitektur spezifiziert, indem sie, wie im Beispiel aus Abbildung 4.9 dargestellt, untereinander angeordnet werden. Damit beschreiben Taktzyklen sequenziell die Schritte der Instruktionen. Jede Instruktion kann in eine beliebige Anzahl von Ausführungszyklen aufgeteilt werden, die bei der Ausführung der Instruktion in der vorgegebenen Reihenfolge durchlaufen werden. Die durchgeführten Operationen in den Ausführungsschritten und die Übergabe der Werte zwischen den Operatoren oder Operanden der Instruktionen werden in Datenflussgraphen spezifiziert. Definition 4.10 für den Datenflussgraphen *DFG* wird durch die Knoten für Operatoren (*Operation*) sowie Operanden (*Operand*) beschrieben und durch die gerichteten Kanten *Flow* zwischen den Knoten verbunden.

$$DFG := Operand \times Operation \times Flow \tag{4.10}$$

Die Operanden beschreiben das Laden oder Speichern von Werten bei der Ausführung der Instruktion. Dabei wird bei der Spezifikation zwischen zwei Arten unterschieden: die interpretierten und die impliziten Operanden. Die Werte für die interpretierten Operanden, wie bereits aus der Spezifikation der Instruktionsformate und Adressierungsmethoden bekannt, werden durch die Adressierungsmethoden der Instruktion berechnet. Die impliziten Operanden beschreiben Operanden wie Carry- oder Conditional-Bits, die nicht aus dem Instruktionsformat der Instruktion abgeleitet, aber bei der Ausführung der Instruktion verwendet werden. Die Spezifikation und die Eigenschaften der impliziten Operanden wurden in Abschnitt 4.3.3 zu den Registerbänken erläutert. Auf die impliziten Operanden kann unmittelbar aus der Instruktion ohne Adressierungsmethoden zugegriffen werden.

Definition 4.11 gibt formal die Spezifikation der operationalen Beschreibung an. Durch die Zuordnung zu den Ausführungszyklen wird der Zyklus *Cycle* ihrer Anwendung festgelegt. Die Variante des Operanden, interpretiert oder implizit, wird durch *OpdVer* festgelegt.

$$
\begin{aligned}
OpdType &:= \{isOpi, isImplicit\} \\
Operand &:= Ident \times Cycle \times OpdVer \\
OpdVer &:= \{(isOpi, o) \mid o \in intOpd\} \cup \{(isImplicit, i) \mid i \in SpcReg\} \\
Operation &:= Ident \times Cycle \times Calulation \\
Flow &\subseteq Operand \times Operation \\
&\cup\, Operation \times Operation \cup Operation \times Operand
\end{aligned}
\tag{4.11}
$$

Die ALUs eines Prozessors besitzen eine festgelegte Menge von ausführbaren Operationen, z. B. Addition, Multiplikation usw. Für den Entwurf in den niedrigeren Abstraktionsebenen wird diese Menge durch den Entwickler fest vorgegeben. Das Konzept in *ViCE-UPSLA* imitiert diese Vorgehensweise, indem für eine Prozessorspezifikation eine Menge von Operatoren definiert wird. Durch die Verwendung und Kombination der Operationen in der operationalen Beschreibung werden Instruktionen mit unterschiedlichem Verhalten beschrieben. Die Spezifikation der Operatoren *Operation* wird in einer eigenen Ansicht durchgeführt. Dabei werden die Parameter und die Berechnungsvorschrift eines Operators beschrieben. Die Operatoren dieser Menge können für die Verhaltensbeschreibung in dem *DFG* beliebig eingesetzt werden.

Die benötigten Ressourcen können aus den verwendeten Operationen in der operationalen Beschreibung abgeleitet werden. Für die Spezifikation von zusätzlichen Konsistenzbedingungen oder zur Visualisierung der verwendeten Ressourcen können für die einzelnen Taktzyklen einer Instruktion die verwendeten Ressourcen aus der Mikroarchitektur explizit angegeben werden.

Wie bereits beschrieben, können Teile der Spezifikation für Instruktionen durch abstrakte Instruktionen vorweggenommen werden. Damit wird auch der Datenflussgraph aus den abstrakten Instruktionen in die konkreten Instruktionen übernommen. Da eine abstrakte Instruktion für eine Gruppe konkreter Instruktionen vorgesehen ist, kann die operationale Beschreibung nicht für jede Instruktion zutreffend sein, diese muss individuell für jede Instruktion angepasst werden.

Durch die operationale Beschreibung wird das Verhalten der Instruktionen ausgedrückt. Durch die Wahl der Menge der Operatoren und deren Verwendung in der operationalen Beschreibung kann das Verhalten identischer Instruktionen auf unterschiedliche Art beschrieben werden. Außerdem können durch Kombination mehrerer Operatoren komplexe Instruktionen erzeugt werden. Für die Präsentation der Mächtigkeit des Konzeptes in *ViCE-UPSLA* werden im Folgenden einige Spezifikationsbeispiele der operationalen Beschreibungen für Instruktionen angegeben. Dazu werden die dargestellten Beispiele der operationalen Beschreibung aus Abbildung 4.11 erläutert und verglichen.

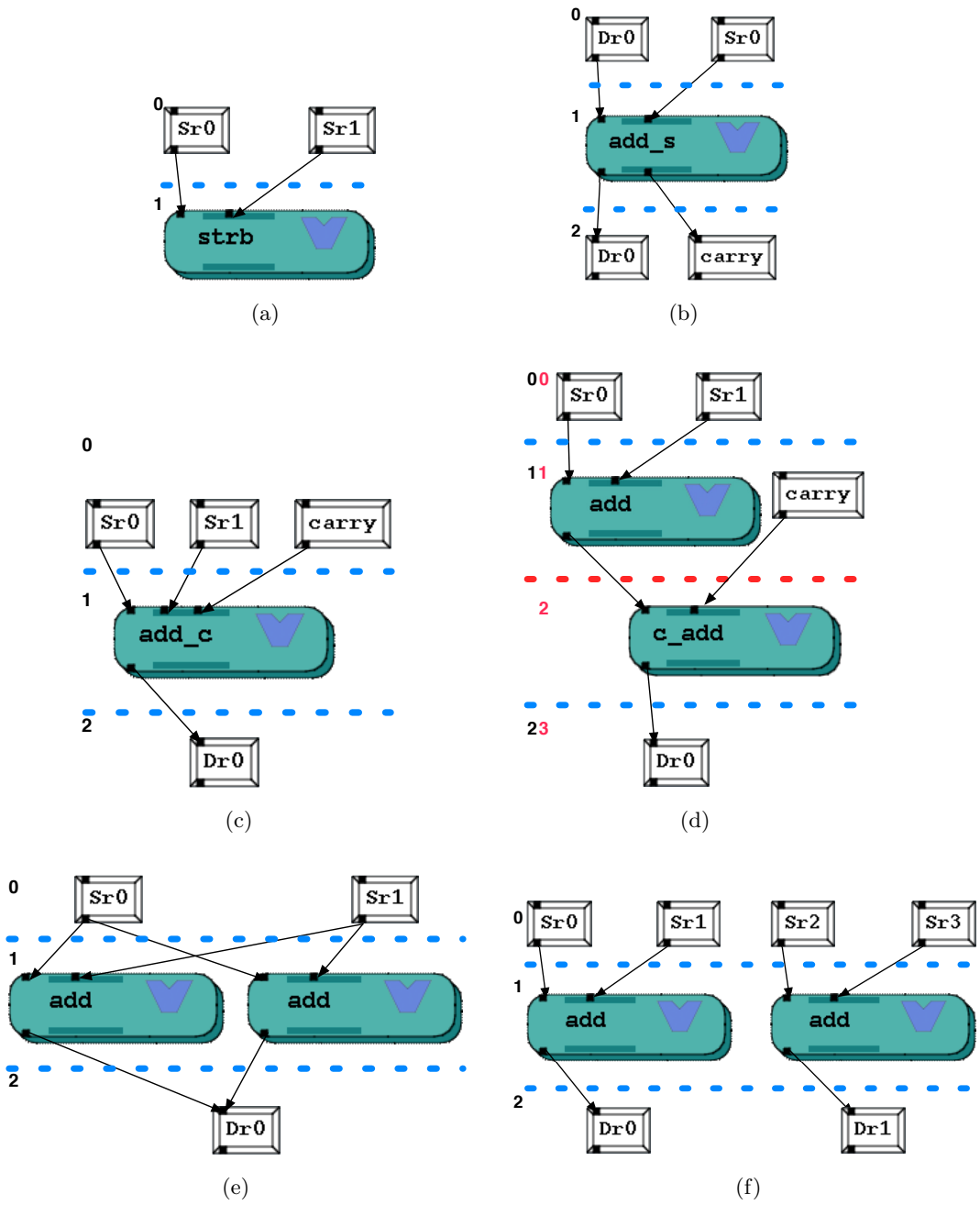


Abbildung 4.11: Operationale Beschreibung der Instruktionen.

In *ViCE-UPSLA* kann die operationale Beschreibung mit einem oder mehreren Operatoren angegeben werden. In den Beispielen in Abbildungen 4.11a, 4.11b und 4.11c sind operationale Beschreibungen mit je einem Operator abgebildet. Im Beispiel 4.11a ist die operationale Beschreibung einer Store-Instruktion dargestellt, welche einen Wert in den Speicher schreibt und somit keinen Ausgabeparameter besitzt. Die Instruktion wird in zwei Taktzyklen ausgeführt, die Aufteilung auf die Taktzyklen wird durch die blau-gestrichelte Linie angedeutet. Das Beispiel 4.11b zeigt die Beschreibung einer `add_s` Instruktion, bei der neben dem Ergebnis im Operand `Dr0` das `carry`-Bit gesetzt wird. Das Beispiel 4.11c beschreibt eine `add_c` Instruktion, bei der zusätzlich zu den Operanden `Sr0` und `Sr1` das `carry`-Bit gelesen wird.

Die Beispiele d), e) und f) zeigen Instruktionen mit mehreren Operatoren in der Verhaltensbeschreibung. Diese Beispiele sollen aufzeigen, wie mit den Konzepten von *ViCE-UPSLA* die Verkettung von Operatoren spezifiziert wird, wie im Beispiel d), oder welche Konfigurationen für die parallelen Strukturen in der operationalen Beschreibung möglich sind.

Nach der kurzen Vorstellung der dargestellten Beispiele werden im Folgenden die Ausdrucksmöglichkeiten bei der Spezifikation ausführlich erläutert und die Besonderheiten in den Beispielen aufgezeigt.

Wie bereits beschrieben, werden die Operatoren in *ViCE-UPSLA* durch den Prozessorentwickler vorgegeben. Sie besitzen eine beliebige Anzahl von Eingabe- und Ausgabeparametern. Zum Vergleich können die Beispiele aus Abbildungen 4.11a, 4.11b und 4.11c hinzugezogen werden, bei denen die Operatoren eine unterschiedliche Anzahl von Eingabe- und Ausgabeparameter besitzen. Durch diese Eigenschaft kann die Granularität der Operatoren für die operationale Beschreibung frei gewählt werden.

Für die Verdeutlichung der Granularität können die Beispiele c) und d) verglichen werden. In beiden Fällen wird das identische Endergebnis berechnet, indem für den Operanden `Dr0` die Summe der Operanden `Sr0` und `Sr1` gebildet und dabei das `carry`-Bit berücksichtigt wird. Die Berechnungsvorschrift dazu kann wie folgt angegeben werden:

$$\texttt{Dr0=add\_c(Sr0,Sr1,carry)} \iff \texttt{Dr0=c\_add(add(Sr0,Sr1),carry)}$$

Die Operatoren im Beispiel d) sind feingranularer, da die Addition durch zwei einfachere Operatoren durchgeführt wird. Die Vorteile der operationalen Beschreibung im Beispiel d) werden durch die mögliche Aufteilung der Operationen auf mehrere Taktzyklen gezeigt. Während die operationale Beschreibung aus Beispiel c) in drei Taktzyklen angegeben werden kann, dargestellt durch die blau-gestrichelte Linie, kann die Beschreibung aus Beispiel d) in vier Taktzyklen angegeben werden, dargestellt durch die rot-gestrichelte Linie. Die operationale Beschreibung aus Beispiel d) ist flexibler bei der Beschreibung der Mikroarchitektur, indem dazu mehrere verschiedene Mikroarchitekturen entworfen werden können, weil die Operatoren z. B. zu unterschiedlichen

Funktionseinheiten zugeordnet werden. Dieses Beispiel wird im nächsten Abschnitt zu der Spezifikation der Mikroarchitektur noch einmal aufgegriffen.

Das Konzept der operationalen Beschreibung erlaubt die Operatoren und Operanden mehrfach zu benutzen. In Abbildung 4.11b wird eine Instruktion gezeigt, bei der die Operanden `Dr0` und `Sr0` als Eingabeparameter und die Operanden `Dr0` und `carry` als Ausgabeparameter verwendet werden. Der Operand `Dr0` wird in diesem Beispiel gelesen und geschrieben. Dies ermöglicht die Spezifikation von Instruktionen mit einem oder zwei Operanden, wie sie z. B. in Zwei-Befehl-Architekturen verwendet werden.

Die parallele Verwendung der Operationen erlaubt es, wie in Abbildungen 4.11e und 4.11f zu sehen ist, Instruktionen für die SIMD oder MIMD Architekturen zu beschreiben. Hierbei kann auch die Verwendung der Ressourcen explizit angegeben werden, womit die Verwendung mehrerer ALUs zur Ausführung einer Instruktion angegeben wird. Für die Visualisierung der verwendeten Ressourcen können die Ressourcen des Prozessors parallel zu den Taktzyklen in einer Tabelle angegeben werden, wie in Abbildung 4.9 in Abschnitt 4.3.5.2 dargestellt.

4.3.6 Mikroarchitektur

Wie bereits beschrieben, kann aus der Spezifikation eines Instruktionssatzes nur ein Instruktionssatzsimulator abgeleitet werden. Für die Simulation einer vorgegebenen Mikroarchitektur muss die Spezifikation des Instruktionssatzsimulators durch den Prozessorentwickler erweitert werden. Die Spezifikation einer Mikroarchitektur erfolgt auf einem hohen Abstraktionsniveau, indem die Komponenten für die Beschreibung des Verhaltens einer Architektur angegeben werden. Dabei werden die Pipeline-Stufen für die überlappende Ausführung, der Datenpfad und die Ressourcen für das *Interlocking* und die Bypässe für das *Forwarding* der Werte bei der Simulation angegeben. In diesem Abschnitt werden zunächst die Sprachkonstrukte für die Beschreibung einer Mikroarchitektur in *ViCE-UPSLA* vorgestellt. Anschließend werden anhand einiger Beispiele die Ausdrucksmöglichkeiten in *ViCE-UPSLA* demonstriert.

Moderne Prozessoren benutzen Architekturen mit Pipelining-Methoden oder paralleler Ausführung, um den Instruktionsdurchsatz zu steigern, wie im Grundlagenabschnitt 2.2.1 beschrieben. In der Regel wird die Anzahl von Pipeline-Stufen oder ALUs eines Prozessors in den frühen Phasen der Entwicklung festgelegt oder gar durch die vorhandenen Fertigungsverfahren bedingt. Durch die Konzepte von *ViCE-UPSLA* wird ermöglicht, die Mikroarchitektur des Prozessors zu einem beliebigen Zeitpunkt bei der Spezifikation mit einem überschaubaren Spezifikationsaufwand zu verändern. Für die Simulation des dynamischen Verhaltens solcher Mikroarchitekturen werden im Folgenden die Sprachkonstrukte zur Modellierung der dafür relevanten Strukturen der Mikroarchitektur vorgestellt. Neben der überlappenden Ausführung, beschrieben durch die Pipeline-Stufen, wird auch die parallele Ausführung zur Leistungssteigerung der Prozessoren eingesetzt. Während bei der überlappenden Ausführung in jeder

Pipeline-Stufe maximal eine Instruktion bearbeitet wird, können bei der parallelen Ausführung wie bei MIMD Architekturen mehrere Instruktionen bearbeitet werden. Dabei werden abhängig von dem Grad der Parallelität mehrere Instruktionen im selben Takt in der gleichen Pipeline-Stufe bearbeitet, wie in Abbildung 4.12 schematisch dargestellt. Durch die Kombination dieser Konzepte können mit *ViCE-UPSLA* eine Reihe bereits existierender Mikroarchitekturen beschrieben werden.

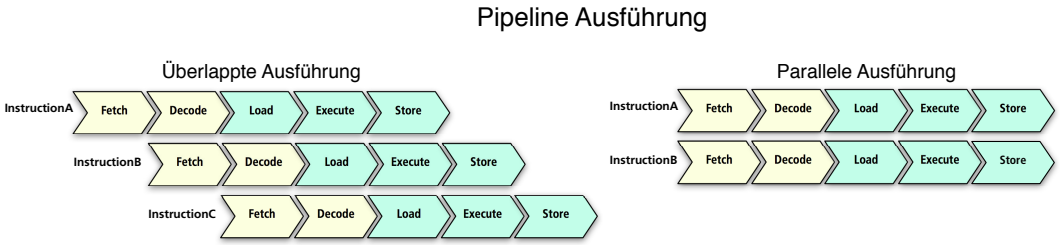


Abbildung 4.12: Überlappende und parallele Ausführung in der Pipeline.

Für die Modellierung der Mikroarchitektur werden bekannte domänenspezifische Visualisierungen aus dem Prozessorentwurf verwendet. In Abbildung 4.13 werden die visuelle Darstellung von Sprachkonstrukten und einige Beispiele für verschieden Architekturen gezeigt. Die Spezifikation der Mikroarchitektur erfolgt vergleichbar mit der operationalen Beschreibung der Instruktionen. Auf die Beispiele wird nach der Vorstellung der Sprachkonstrukte ausführlich eingegangen.

4.3.6.1 Modellierung der Mikroarchitektur

Das Konzept für die Beschreibung der Mikroarchitektur *MicroArch* eines Prozessors umfasst in *ViCE-UPSLA* den Datenpfad *DataPath*, die Pipelinestruktur *PipeStruct* und die Bypass-Struktur *ByStruct*, wie in Definition 4.12 dargestellt.

$$MicroArch := PipeStruct \times DataPath \times ByStruct \tag{4.12}$$

Die Pipelinestruktur spezifiziert die Ausführungsschritte für die überlappende Ausführung der Instruktionen. Die Pipeline-Stufen (*PipeStage*) werden untereinander angeordnet und geben die Menge und die Reihenfolge der Sequenzen an, die bei der Simulation des Prozessors befolgt werden. Dabei können in einer Mikroarchitektur beliebig viele Pipeline-Stufen angegeben werden. In der Regel werden die Aufgaben der Pipeline-Stufen durch Bezeichner gekennzeichnet, was zur Orientierung in der Spezifikation sinnvoll ist. Die tatsächlich ausgeführten Aufgaben in den Pipeline-Stufen werden anschließend durch das Angeben des Datenpfads und der Ressourcen genau spezifiziert.

Der Datenpfad *DataPath* der Mikroarchitektur wird durch einen Datenflussgraphen über die Funktionsbausteine ausgedrückt. Die einzelnen Funktionsbausteine *FunkUnit* spezifizieren die Ressourcen des Prozessors und werden durch gerichtete Kanten *Bus* verbunden. Damit kann ein Datenflussgraph wie folgt definiert werden:

$$DataPath = FunkUnit \times Bus \quad Bus \subseteq FunkUnit \times FunkUnit \tag{4.13}$$

Die operationale Beschreibung der Instruktionen spezifiziert für jeden Ausführungsschritt die Ressourcen und die Wertübergabe zwischen den Operationen einer Instruktion. Der Datenpfad spezifiziert die Struktur, die Ressourcen und die Wertübergabe in der Mikroarchitektur. In einem Mikroarchitektursimulator werden die spezifizierten Ressourcen in der Mikroarchitektur durch die Operationen der Instruktionen belegt oder freigegeben. Die Wertübergabe in der operationalen Beschreibung muss deckungsgleich mit einem Teilgraph des Datenpfads sein, womit die Ausführung der Instruktionen durch die Mikroarchitektur geleitet wird. Diese Eigenschaft wird bei der Validierung der Spezifikation ausgenutzt und in Kapitel 5 genau beschrieben. Mit diesem Konzept wird die Simulation der parallelen oder überlappenden Ausführung von Instruktionen koordiniert.

In *ViCE-UPSLA* werden drei Arten von Funktionsbausteinen *FunkUnit* verwendet. Dazu gehören die Lese- und Schreib-Ports *PORT* des Prozessors, die die Anbindung zum Registersatz beschreiben. Die Multiplexer *MUX* werden verwendet, um z. B. Ressourcen für die komplexen Adressierungsmethoden zu spezifizieren. Die arithmetisch-logischen Einheiten *ALU* beschreiben die Ressourcen für den Instruktionssatz des Prozessors. Wie in Definition 4.14 angegeben, wird jeder Funktionsbaustein einer Pipeline-Stufe *PipeStage* zugeordnet; damit kann eine Pipeline-Stufe mehrere Funktionsbausteine enthalten.

$$\begin{aligned} MUX &:= ID \times PipeStage \\ ALU &:= ID \times PipeStage \times InstructionGroup \\ PORT &:= ID \times PipeStage \times PortDest \times Ressource \end{aligned} \tag{4.14}$$

Die Lese- und Schreib-Ports *PORT* spezifizieren die Ressourcen für den Zugriff auf den Registersatz des Prozessors. Die Anzahl der Ports *Ressource* beschreibt, wie viele Werte in der entsprechenden Stufe gelesen oder gespeichert werden können. Da die Prozessoren über mehrere separate Registerbänke verfügen können, wird durch eine Verknüpfung zu den Registerbänken die Quelle *PortDest* angegeben, auf welche sich dieser Port bezieht. Damit kann die Spezifikation der Ports selektiv auf einige Bereiche des Registersatzes eingeschränkt werden. Das Konzept erlaubt die Spezifikation von z. B. MIMD Architekturen mit paralleler Verarbeitung und vollständig oder nur teilweise getrennten Registersätzen, wie in Abbildungen 4.13i und 4.13j dargestellt wird. Diese Präzisierung in *ViCE-UPSLA* kann z. B. für die Überprüfung der Konsistenz

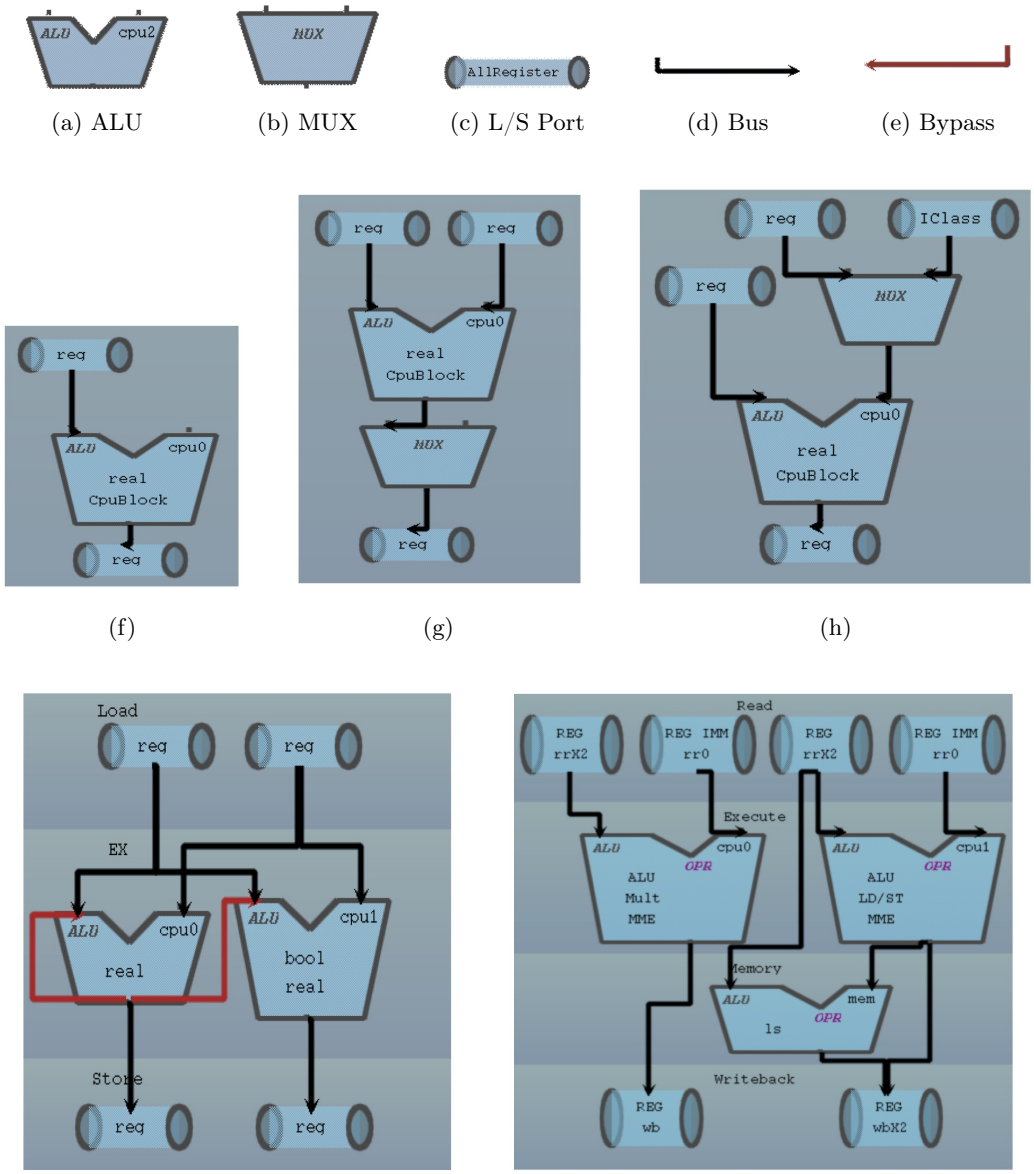
zwischen den spezifizierten Ressourcen in der Mikroarchitektur und den Adressierungsmethoden der Instruktionen verwendet werden. Die Validierung der Ressourcen wird in Kapitel 5 erläutert und eine Anwendung der Validierungsmethoden in Abschnitt 7.4 an einem Beispiel durchgeführt.

Ein Multiplexer *MUX* beschreibt Funktionseinheiten mit Berechnungsvorschriften, wie sie z. B. bei Adressierungsmethoden verwendet werden. In Abbildung 4.13h wird die Ressource für eine Adressierungsmethode dargestellt, indem der linke Operand für die ALU aus einem Registerwert und einem unmittelbaren Wert aus der Instruktion berechnet wird.

Die arithmetisch-logische Einheit *ALU* spezifiziert die Ressourcen für die Funktionseinheiten eines Prozessors, in denen die Berechnungen der Ergebnisse durch Operator-Operationen aus der operationalen Beschreibung der Instruktionen durchgeführt werden. Durch die Angabe eines Instruktionssatzes werden die Instruktionen, die in einer ALU ausgeführt werden, spezifiziert. Für die Spezifikation von Prozessoren mit paralleler Ausführung kann eine beliebige Anzahl von ALUs eingesetzt werden, um den Grad der Parallelität zu beschreiben. Ein Beispiel wird in Abbildung 4.13i gezeigt. Jeder der eingesetzten ALUs kann eine Gruppe von Instruktionen *InstructionGroup* zugeordnet werden. Dieses Konzept wird in *ViCE-UPSLA* für die Spezifikation und Simulation von SIMD oder MIMD Architekturen verwendet und ermöglicht die Validierung der Ressourcen oder der Instruktionsstruktur in der Spezifikation.

Durch die beschriebenen Komponenten für die Spezifikation der Mikroarchitektur (*ALU, PORT, MUX*) wird dem Prozessorentwickler die Möglichkeit gegeben, vorhandene Mikroarchitekturen zu beschreiben und Simulatoren dieser Architekturen zu erzeugen. In Abbildung 4.13 sind einige Kombinationsmöglichkeiten dargestellt. Zum einen ist die Beschreibung einfacher Datenpfade für gewöhnliche RISC-Prozessoren mit einer ALU gegeben, wie in Abbildungen 4.13f, 4.13g und 4.13h gezeigt. Zum anderen erlaubt *ViCE-UPSLA* die Spezifikation und Simulation von superskalaren Mikroarchitekturen für Prozessoren mit VLIW Instruktionssätzen mit mehreren ALUs, wie in den Abbildungen 4.13i und 4.13j dargestellt. In Kapitel 7 wird bei der Evaluierung der Sprache an Beispielen die Spezifikation von SISD, SIMD und VLIW Architekturen gezeigt.

Durch die Zuordnung der Funktionsbausteine zu den Pipeline-Stufen der Mikroarchitektur wird zyklengenaue, überlappende und parallele Simulation umgesetzt. Mit *ViCE-UPSLA* wird durch einen Parameter festgelegt, ob der Simulator des Prozessors mit Interlock-Mechanismen wie beim ARM-Prozessor oder wie beim MIPS-Architektur ohne Interlock-Mechanismen [Kan88, HKP+82] erzeugt werden soll. Die Spezifikation verschiedener Varianten der Ausführung kann für die Validierung der Sprachkonstrukte ausgenutzt werden.



(a) ALU (b) MUX (c) L/S Port (d) Bus (e) Bypass

(f) (g) (h)

(i) (j)

Abbildung 4.13: Visualisierung der Mikroarchitektur mit *ViCE-UPSLA*.

4.3.6.2 Erweiterung: Instruktionssatz- zum Mikroarchitektursimulator

Wie bereits beschrieben, kann von dem Prozessorentwickler zuerst ein Instruktionssatzsimulator spezifiziert werden, der dann zu einem Mikroarchitektursimulator erweitert werden kann. Für die Durchführung dieses Schrittes wird das Beispiel aus Abbildung 4.11d betrachtet und in Abbildung 4.14 vorgestellt. Dabei werden alternative Mikroarchitekturen für diese Instruktion bzw. der operationalen Beschreibung der Instruktion gezeigt.

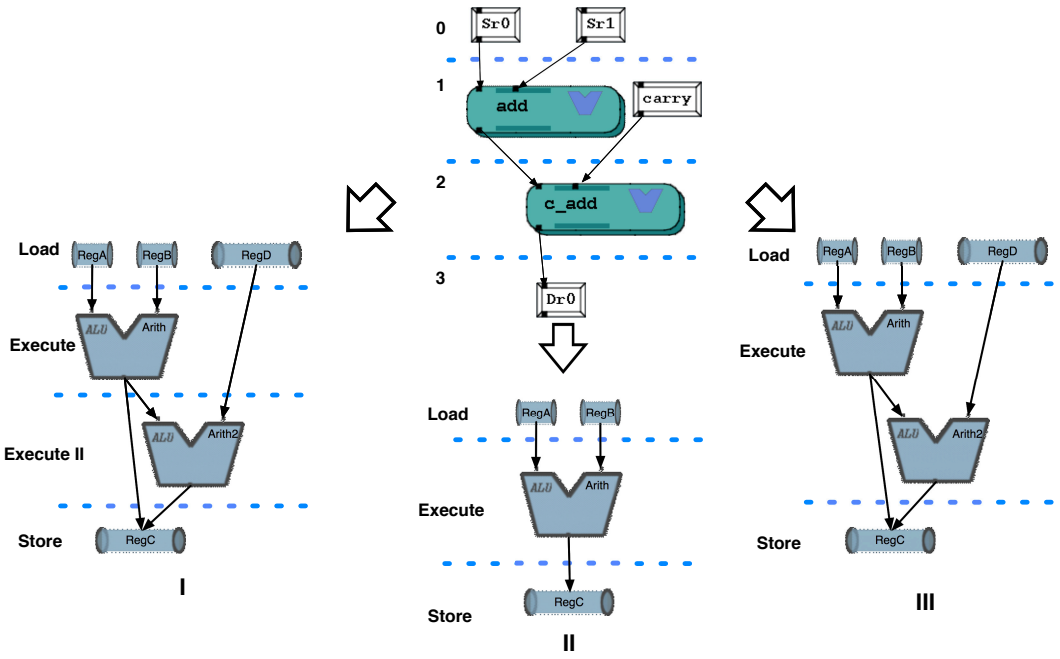


Abbildung 4.14: Mögliche Zuordnung einer Verhaltensbeschreibung zu verschiedenen Mikroarchitekturen I, II und III. Die Pipelinestufen und Ausführungszyklen werden durch blau-gestrichelte Linie dargestellt.

Die operationale Beschreibung der Instruktion beschreibt eine Addition mit Carry `add_c`. Dabei werden zunächst zwei Operanden geladen und in dem Operator `add` addiert. Das Ergebnis aus der Addition, sowie der implizite Operand `carry`-Bit, werden im Operator `c_add` als Eingabeparameter verwendet. Mit dem zweiten Operator wird das Endergebnis bestimmt und in den Zieloperanden `Dr0` geschrieben.

In Abbildung 4.14 werden zu der operationalen Beschreibung drei alternativ mögliche Mikroarchitekturen mit unterschiedlicher Anzahl von Pipelinestufen und Ressourcen gezeigt.

Die Granularität der Operatoren in der operationalen Beschreibung erlaubt die Be-

schreibung von unterschiedlich vielen Ressourcen in der Mikroarchitektur, wie in Abbildung 4.14 dargestellt. Bei der Zuordnung der operationalen Beschreibung zu der Mikroarchitektur aus der Variante I wird die Instruktion in vier Zyklen ausgeführt, wobei `add` und `c_add` den ALUs `Arith` und `Arith2` zugeordnet werden. Bei überlappender Ausführung der Instruktionen kann die Berechnung der nachfolgenden Instruktion in der ALU `Arith` begonnen werden, wenn die Instruktion aus Beispiel 4.14 die Berechnung in der `Arith2` in der Pipelinestufe `Execute II` durchführt.

In der nächsten Variante II wird nur eine ALU für die Berechnung von zwei Operationen angegeben. Die Ausführungszyklen der Instruktion werden in der Pipelinestufe `Execute` wiederholt ausgeführt. In dieser Variante muss im Simulator der Mikroarchitektur ein Interlock-Mechanismus umgesetzt werden, damit die nachfolgenden Instruktionen für die Ausführungsschritte der Instruktion aus dem Beispiel angehalten werden. Wie in Variante I werden für die Ausführung der Instruktion vier Taktzyklen benötigt, da die Stufe mit der Ausführung in der ALU für `c_add` wiederholt ausgeführt werden muss.

Die Spezifikation der Mikroarchitektur kann auch so angegeben werden, dass die Berechnung in der ALU in einem Taktzyklus durchgeführt werden kann. Hierfür werden die zwei hintereinander geschalteten ALUs in einer Pipeline-Stufe platziert, wie in Variante III in Abbildung 4.14 dargestellt.

In den Beispielen wurden verschiedene Möglichkeiten für die Spezifikation von Mikroarchitekturen zu einem gegebenen Instruktionssatz mit *ViCE-UPSLA* gezeigt. Die Sprache erlaubt durch die Kombination verschiedener Konstrukte eine Spezifikation anspruchsvoller Mikroarchitekturen und eine Nachbildung von bereits existierenden realen Prozessoren.

4.3.6.3 Bypass-Struktur

Für die Auflösung der Datenkonflikte bei der überlappenden Ausführung von Instruktionen in einem Prozessor werden Bypässe eingesetzt, wie in dem Grundlagenabschnitt 2.2.1.3 beschrieben.

In Abbildung 4.15 wird das Prinzip für die Weiterleitung (vgl. Abschnitt 2.2.1.3) schematisch dargestellt. Folgen zwei Instruktionen mit Datenabhängigkeiten aufeinander, kann die Ausführung ohne eine Lösungsroutine des Prozessors nicht fehlerfrei ausgeführt werden. Sind in der Mikroarchitektur keine Bypässe integriert, muss die Ausführung der zweiten Instruktion, wie in Abbildung 4.15 im Beispiel ohne Bypass, durch einen Pipeline-Stall angehalten werden. Durch einen Bypass wird der Wert am Ausgang der ALU zu dem Eingang der ALU weitergeleitet. Das Besondere an den Bypässen im Vergleich zu den normalen Bussen zwischen Funktionsbausteinen ist, dass die Werte nur unter der Bedingung weitergeleitet werden, dass eine Datenabhängigkeit besteht. Dieses Konzept wird durch das Sprachkonstrukt für Bypässe in *ViCE-UPSLA* vollständig umgesetzt. Ein Beispiel für die visuelle Darstellung eines Bypasses

mit *ViCE-UPSLA* in der Mikroarchitektur ist in Abbildung 4.13i dargestellt.

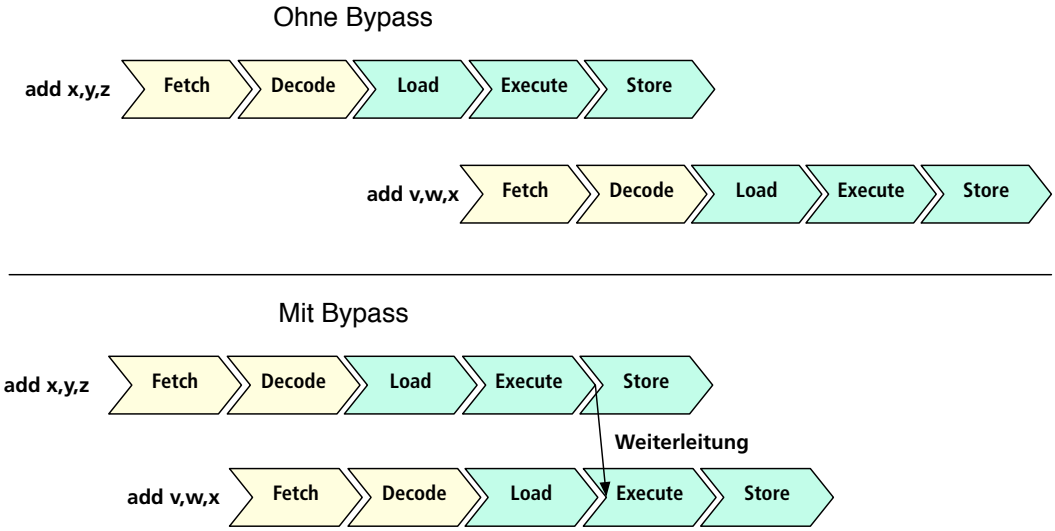


Abbildung 4.15: Datenabhängigkeit durch den gemeinsamen Operanden `x` und überlappende Ausführung mit und ohne Weiterleitung.

Die Bypass-Struktur *ByStruct* in *ViCE-UPSLA* beschreibt die bedingte Weiterleitung der Werte zwischen Funktionsbausteinen *FunkUnit* im Datenpfad, wie in Definition 4.15 angegeben. Ein Bypass wird, so wie die Datenpfade, als gerichtete Kante im Datenflussgraph verwendet und verknüpft Funktionsbausteine *FunkUnit* miteinander.

$$ByStruct \subseteq FunkUnit \times FunkUnit \tag{4.15}$$

Durch die Verknüpfung der Funktionsbausteine werden die Quelle und die Senke, zwischen denen die Werte propagiert werden, festgelegt. Der spezifizierte Bypass ist somit für alle Instruktionen, die die entsprechenden Knoten in der Mikroarchitektur durchlaufen, definiert. Aus der Sicht des Prozessorentwicklers sind die Speicherung der Werte und die Strategie, nach der der Werteaustausch durchgeführt wird, transparent.

Im Konzept von *ViCE-UPSLA* wird ein Wert nur zwischen den durch den Bypass verbundenen Punkten in der Mikroarchitektur propagiert. Für die Weiterleitung eines Wertes aus einer Quelle an mehrere Funktionsbausteine werden mehrere Bypässe benötigt. Damit können mit *ViCE-UPSLA* selektive Bypass-Strukturen beschrieben werden, was größere Freiheitsgrade für die Spezifikation realer Prozessoren ermöglicht, da eine vollständige Bypass-Überdeckung in der Regel wegen des Flächenbedarfs unerwünscht ist.

Für die Simulation spezifiziert der Bypass ein Tripel, das den aktuellen Wert, die Zieladresse des Wertes und die Priorität des Bypasses angibt. Jede Instruktion, wel-

che den Pfad mit der Quelle des Bypasses durchläuft, setzt die aktuellen Werte für die Weiterleitung, sobald die Ausführung der Instruktion die Definitionsstelle passiert. Am anderen Ende des Bypasses wird analog für jede Instruktion der Bypass auf aktuelle Werte überprüft. Die Senke des Bypasses beschreibt die Eigenschaft der bedingten Auswahl für die Werte durch eine „Zwei Phasen"-Strategie. In der ersten Phase wird überprüft, ob die Registeradresse des aktuellen Wertes mit dem Zielregister aus dem Bypass übereinstimmt. Bei einer Übereinstimmung wird die zweite Phase durchgeführt. Für den Fall, dass mehrere Bypässe in der ersten Phase eine Übereinstimmung hatten, wird der Bypass anhand der Priorität ausgewählt. In der Regel wird festgelegt, dass die Bypässe mit den jüngsten Werten die höchste Priorität besitzen. Alternativ können die Prioritäten der Bypässe durch den Benutzer in einer zusätzlichen Spezifikation angegeben werden.

Das Konzept der Mikroarchitektur in *ViCE-UPSLA* kombiniert die Spezifikation der Pipeline, des Datenpfads und der Bypässe, womit die Ressourcen und die Taktzyklen für die Ausführung der Instruktionen im Simulator vorgegeben werden. Die Spezifikation der Instruktionen, der Instruktionsformate und der Adressierungsmethoden muss mit der Spezifikation der Pipeline konsistent sein.

5 Prozessorvalidierung

In diesem Kapitel werden Konzepte und Methoden für die Validierung von Prozessoren aus Spezifikationen beschrieben. Die Validierungsmethoden müssen eine systematische Vorgehensweise für die Validierung beliebiger Prozessorspezifikationen mit der Sprache *ViCE-UPSLA* ermöglichen. Wie in dem Grundlagenabschnitt 2.5 und in den verwandten Arbeiten in Abschnitt 3.2 beschrieben, basieren die meisten Ansätze für Validierungswerkzeuge auf einem redundant erzeugten Modell, in dem das Verhalten des zu validierenden Systems beschrieben wird. Der Ansatz für die Validierung mit *ViCE-UPSLA* unterscheidet sich darin, dass die Spezifikation und Validierung eines Prozessors auf einer Spezifikation basiert, aus der sowohl die Simulatoren als auch die Testfälle für die dynamische Validierung generiert werden. Damit werden zum einen bei der Validierung keine Komponenten der Spezifikation ausgelassen, zum anderen wird der Spezifikationsaufwand für ein zweites Modell gespart.

Für die Entwicklung der Validierungsmethoden werden zunächst die Fehlerquellen für die Konstrukte aus dem Prozessorentwurf in einem Fehlermodell [Ram89] in Abschnitt 5.2 beschrieben. Damit werden die Fehler klassifiziert und so eine strukturierte Entwicklung der Methoden ermöglicht. Bei der Beschreibung des Fehlermodells werden Fehlerquellen für die Validierung der Sprachkonstrukte nach ihrer Prüfbarkeit in statische und dynamische Fehler eingeteilt. Anschließend werden in Abschnitten 5.3, 5.4 und 5.5 aus dem Fehlermodell und unter Einbeziehung der Struktur von *ViCE-UPSLA* die statischen und dynamischen Validierungsmethoden beschrieben.

Die Abschnitte 5.6 über das zusätzliche Wissen und 5.7 die Testfallspezifikation beschreiben Ergänzungen des Werkzeugsystems, um eine optimale Unterstützung des Prozessorentwicklers bei der Validierung zu erreichen.

5.1 Aufgaben der Validierung

Die Aufgabe der Validierung ist es, Inkonsistenzen in einer Spezifikation aufzudecken. Bei einer strukturierten Vorgehensweise sollten die einzelnen Validierungsmethoden immer nur einen Aspekt einer Spezifikation validieren, indem diese nur auf bestimmte Prozessorkonstrukte oder deren Eigenschaften fokussiert werden. In der Arbeit „Systematischer Entwurf digitaler Systeme: Von der System- bis zur Gatter-Ebene“ [Ram89] von F. J. Rammig werdem in diesem Zusammenhang *interne* oder *externe Konsistenz* beschrieben. Die Strategie einer Validierungsmethode sollte so beschrieben sein, dass für ein Konstrukt aus der Prozessorspezifikation die Validierungsmethode auf eine Aussage, bzw. eine Frage eingeht, z. B.:

Wurden die Sprachkonstrukte richtig verwendet?

Mit dieser Frage wird eine Anforderung an die Eigenschaften der zu validierenden Komponente formuliert. Eine so formulierte Validierungsmethode stellt fest, ob die Sprachkonstrukte in der Prozessorspezifikation im Sinne der Konventionen aus dem Prozessorentwurf oder in Relation zu anderen Sprachkonstrukten in dem Entwurf (vgl. *interne Konsistenz* [Ram89]) konsistent verwendet werden.

Desweiteren können Validierungsmethoden auf die Korrektheit des Entwurfs eingehen. Dabei soll untersucht werden, ob die Spezifikation des Prozessors den vorliegenden Entwurfsmustern und Anforderungen entspricht. Die Validierungsmethoden müssen dabei auf folgende Frage eingehen:

Wurden die richtigen Sprachkonstrukte verwendet?

In den meisten Fällen müssen für die Validierung des Entwurfs zusätzliche Angaben durch den Prozessorentwickler vorgenommen werden, womit ein zweites Modell für die Teilaspekte erzeugt wird. Bei der Validierung wird gegengeprüft, ob die spezifizierten Komponenten auch der informellen Beschreibung und den Richtlinien des Prozessorentwicklers (vgl. *externe Konsistenz* [Ram89]) entsprechen.

Wie in dem Grundlagenabschnitt 2.5.2 beschrieben, kann durch dynamische Validierung nur die Anwesenheit von Inkonsistenzen gezeigt werden, anders als bei statischer Analyse der Spezifikation. Daher ist es sinnvoll, möglichst viele Konstrukte und Eigenschaften des Prozessors durch die statische Validierung bereits in der Spezifikation zu untersuchen. Darüber hinaus besitzen die statischen Validierungsmethoden eine geringe Laufzeit und können während der Erzeugung der Spezifikation durchgeführt werden, bevor der Simulator des Prozessors erzeugt wird.

Für die Strukturierung der Validierungsmethoden wird ein Fehlermodell erstellt. Das Fehlermodell beschreibt durch die Analyse der Prozessorkonstrukte auf dem Abstraktionsniveau von *ViCE-UPSLA*, welche Fehlerquellen in den Sprachkonstrukten einer Prozessorspezifikation bei der Validierung betrachtet werden müssen. In Korrelation mit den Sprachkonstrukten von *ViCE-UPSLA* wird im Fehlermodell die Prüfbarkeit

der Sprachkonstrukte erfasst. Damit wird für die Fehlerquellen und Sprachkonstrukte oder deren Eigenschaften festgestellt, ob diese statisch oder dynamisch validiert werden können.

5.2 Fehlermodell

Wie aus der Beschreibung der Sprache in Abschnitt 4.3 hervorgeht, beschreibt die Struktur der Spezifikation eines Prozessors ein zusammenhängendes Netzwerk aus Prozessorkonstrukten, woraus ein Simulator des Prozessors generiert werden kann. Die Relationen zwischen den Sprachkonstrukten in der Spezifikation ermöglichen, Konsistenzbedingungen und Abhängigkeiten zwischen den Konstrukten des Prozessors festzulegen und zu überprüfen. Bei der Beschreibung der Sprache wurde für diesen Zweck eine formale Definition der Konstrukte angegeben. Für die Entwicklung der Validierungsmethoden werden in diesem Abschnitt die möglichen Fehlerquellen in einer Prozessorspezifikation ermittelt. Wie in Abschnitt 2.5 beschrieben, werden dabei folgende Kriterien betrachtet:

- Konsistenz und Vollständigkeit
- Struktur und Verhaltenseigenschaften
- Syntaktische und semantische Eigenschaften

In Abschnitt 5.2.1 werden zu den genannten Kriterien die Eigenschaften von Konflikten oder Inkonsistenzen beschrieben. Dabei wird die Frage geklärt, wie die Begriffe Konflikt oder Inkonsistenz im Prozessorentwurf definiert werden.

Die Fehlerquellen werden beschrieben, indem durch verschiedene Konsistenzmodelle die Fehlerquellen und Fehlerarten in der Spezifikation eines Prozessors aufgezeigt und definiert werden. Die Beschreibung der Konsistenzbedingungen wird zur Verdeutlichung zusätzlich durch formale Definitionen angegeben, welche sich auf die formalen Definitionen der Sprachkonstrukte aus der Beschreibung von *ViCE-UPSLA* beziehen. Damit können die definierten Konsistenzbedingungen aus dem Fehlermodell direkt den Sprachkonstrukten von *ViCE-UPSLA* zugeordnet werden.

Für die Beschreibung der einzelnen Komponenten des Fehlermodells werden zunächst die wesentlichen Eigenschaften und Wechselwirkungen für die untersuchten Prozessorkonstrukte definiert. Anschließend werden auf dieser Grundlage die Konsistenzbedingungen formuliert. Für die Entwicklung des Fehlermodells wird die Reihenfolge für die Betrachtung der Sprachkonstrukte aus Abschnitt 4.3 übernommen. Dabei werden zuerst die Eigenschaften des Registersatzes, der Instruktionsformate und Adressierungsmethoden, des Instruktionssatzes und abschließend die der Mikroarchitektur betrachtet werden.

5.2.1 Konflikte und Inkonsistenzen

Für eine systematische Entwicklung von Validierungsmethoden müssen zunächst die Fehlerquellen und deren Eigenschaften erfasst werden. Hierfür folgen zunächst einige Definitionen, womit der *Konflikt* im Prozessorentwurf mit *ViCE-UPSLA* eingegrenzt wird. Anschließend werden die für die Validierung relevanten Eigenschaften eines Konflikts beschrieben.

Die Validierung einer Prozessorspezifikation wird zum einen durch die Analyse der Struktur einer Spezifikation ermöglicht, zum anderen wird aus der Spezifikation ein Simulator erzeugt, welcher die Analyse des Verhaltens des Prozessors unter verschiedenen Eingaben erlaubt. Angelehnt an die Definition der Widersprüche im Buch „Konsistenzprüfungen von Domänenanforderungsspezifikationen“ von K. Lauenroth [Lau09] lassen sich zwei Arten von Konflikten im Prozessorentwurf wie folgt definieren:

- Strukturelle Konflikte — *Ein Konflikt besteht dann, wenn aus der Spezifikation hervorgeht, dass zwischen den Prozessorkonstrukten die Eigenschaften im Widerspruch zueinander verwendet werden.*
- Dynamische Konflikte — *Ein Konflikt besteht dann, wenn die Ergebnisse der Ausführung eines Programms auf dem Prozessor oder im Simulator nicht den erwarteten Ergebnissen entsprechen.*

Aus den Definitionen geht hervor, dass die Konflikte und Inkonsistenzen direkt aus der Spezifikation oder erst aus der Simulation eines Prozessors erkannt werden können. Daraus ergibt sich eine wichtige Eigenschaft, die die Zeitpunkte im Prozessorentwurfsprozess beschreibt, wann die Inkonsistenzen festgestellt werden können.

Wie aus den Grundlagen in Abschnitt 2.1 hervorgeht, wird die Spezifikation bis zur Fertigung des Prozessors fortlaufend verfeinert, sodass einige bereits spezifizierte Konstrukte erst in niedrigeren Abstraktionsebenen Inkonsistenzen aufweisen können. Die Spezifikation in *ViCE-UPSLA* kann dazu verwendet werden, die Bearbeitung auf einem niedrigeren Niveau fortzuführen. Die Fehlerquellen und die Validierungsmethoden für einige dieser Konstrukte lassen sich jedoch bereits auf den Sprachkonstrukten von *ViCE-UPSLA* formulieren.

- Konflikte auf niedrigen Abstraktionsebenen — *Ein Konflikt besteht dann, wenn bereits spezifizierte Prozessorkonstrukte auf den niedrigeren Abstraktionsebenen im Prozessorentwurf zu Konflikten führen.*

Die Konfliktquellen aus niedrigen Abstraktionsebenen werden, sofern diese durch die Sprachkonstrukte von *ViCE-UPSLA* erfasst werden können, ebenfalls in das Fehlermodell aufgenommen.

Im Fehlermodell werden die potenziellen Fehlerquellen für die Prozessorkonstrukte aus der Betrachtung der Regeln und Konventionen aus dem Prozessorentwurf erfasst.

Die Beschreibung der Konsistenzbedingungen wird durch die Relationen zwischen den Eigenschaften von Prozessorkonstrukten in einer Spezifikation angegeben. Eine Konsistenzbedingung kann z. B. zwischen dem Wertebereich eines im Instruktionsformat definierten Operationscodes und der Menge von Instruktionen beschrieben werden. Wird eine Konsistenzbedingung durch die vorhandenen Prozessorkonstrukte in einer Spezifikation angegeben, beschreiben die Fehlerquellen reguläre Konflikte.

- *Ein regulärer Konflikt besteht dann, wenn widersprüchliches Verhalten oder widersprüchliche Verwendung von Prozessorkonstrukten aus der Betrachtung der spezifizierten Prozessorkonstrukte resultiert.*

Als Gegenstück dazu werden im Fehlermodell die irregulären Konflikte ebenfalls erfasst. Wird für ein Prozessorkonstrukt eine Fehlerquelle aus den Konventionen des Prozessorentwurfs identifiziert, für die die Formulierung einer Konsistenzbedingung in der Spezifikation nicht möglich ist, wird in diesem Zusammenhang von irregulären Konflikten gesprochen. Z. B. ist für die Berechnung der Ergebnisse in einer Instruktion keine Umkehrfunktion angegeben, womit die Korrektheit der Berechnung nicht ohne zusätzliche Spezifikation durchgeführt werden kann.

- *Ein irregulärer Konflikte besteht dann, wenn für die Validierung eines Prozessorkonstrukts eine zusätzliche Spezifikation benötigt wird.*

Die zusätzliche Spezifikation wird in dieser Arbeit als zusätzliches Wissen durch den Prozessorentwickler in die Spezifikation oder bei der Beschreibung der Testfallspezifikationen eingebracht. Die Definition und Modellierung des zusätzlichen Wissens wird in Abschnitt 5.6 ausführlich behandelt. Bei der Beschreibung der Fehlerquellen in den folgenden Abschnitten werden für die Prozessorkonstrukte die regulären und irregulären Konflikte identifiziert.

5.2.2 Registersatzkonflikte

Der Registersatz beschreibt die Speicherstellen des Prozessors, die zum Halten der Werte bei der Ausführung der Programme verwendet werden. Dafür lassen sich zwei allgemeine Eigenschaften identifizieren: Wertsicherheit (die Speicherstellen müssen die dafür vorgesehenen Werte korrekt speichern und wieder zurückgeben), Erreichbarkeit (die Speicherstellen müssen über die in der Spezifikation definierten Pfade erreichbar sein). Bei der Beschreibung im Fehlermodell wird zwischen der Validierung der Registerbänke und der Validierung der Register unterschieden, da die Erreichbarkeit einer Registerbank nicht die Erreichbarkeit aller Register garantiert.

Ein Registersatz, wie in Abschnitten 4.3.3 und 2.2.4 beschrieben, wird durch die Spezifikation der physikalischen und architektonischen Register angegeben. Durch die architektonischen Register werden indirekte Strukturen für den Zugriff auf die physikalischen Registerbänke beschrieben. Damit werden durch das Aliasing verschiedene Anordnungen der physikalischen Register erzeugt.

Die Adressierungsmethoden beschreiben in *ViCE-UPSLA* die Zugriffsmöglichkeiten auf die Register. Für die Feststellung der Erreichbarkeit von Registern müssen neben der Struktur des Registersatzes auch die Adressierungsmethoden des Prozessors betrachtet werden. Die Register werden bei der Ausführung von Programmen geschrieben und gelesen. Die Validierung der Erreichbarkeit muss in beiden Benutzungsrichtungen sichergestellt werden. Durch die Adressierungsmethoden wird eine dynamische Komponente für die Beschreibung der Erreichbarkeit eingebracht, wodurch sowohl statische als auch dynamische Konflikte entstehen können. Die Komplexität der so beschriebenen Strukturen macht es notwendig, die Erreichbarkeit zu validieren, da sonst Fehlerquellen vom Prozessorentwickler leicht übersehen werden können. Im Folgenden werden die Konsistenzbedingungen für die Erreichbarkeit der Register im Rahmen von *ViCE-UPSLA* beschrieben.

In der Spezifikation eines Prozessors wird durch eine Verknüpfung die Zuordnung zwischen einer Registerbank und einer Adressierungsmethode beschrieben. Wenn eine Registerbank in der Prozessorspezifikation mit keiner Adressierungsmethode verknüpft ist, kann davon ausgegangen werden, dass die Spezifikation nicht vollständig oder inkonsistent ist. Diese Eigenschaft bildet eine notwendige Voraussetzung für die Validierung weitere Aspekte.

Diese Konsistenzbedingung für die Erreichbarkeit der Registerbänke kann wie in (5.1) beschrieben werden. Die Konsistenz der Spezifikation ist verletzt, wenn eine Registerbank p von keiner Instruktion i oder j aus dem Instruktionssatz des Prozessors zum Lesen und Schreiben verwendet werden kann. Hierfür muss die Instruktion einen interpretierten Operanden besitzen, dessen Adressierungsmethode die Registerbank mit dem Befehlsoperand *(addressing,p)* verknüpft. Dabei muss die Registerbank nicht direkt benutzt werden, es ist ausreichend, wenn z. B. die physikalischen Registerbänke indirekt über die architektonischen Registerbänke $(addressing_j, p_j)$ erreicht werden. Dabei muss gelten, dass p_j Alias von p ist. Die Bedingung für die Erreichbarkeit basiert auf strukturellen Eigenschaften, wie in Abschnitt 5.2.1 beschrieben und kann statisch durch eine Konsistenzbedingung direkt in der Spezifikation überprüft werden.

$$\begin{aligned} \forall p \in PhysReg \ : (\exists i \in instr : i = (name, ..., (adressing, p), ...)) \vee \\ (\exists j \in instr : (j = (name, ..., (adressing_j, p_j), ...) \wedge p_j \in ArchReg \ : p_j \rightarrow p) \end{aligned} \tag{5.1}$$

Die statische Überprüfung der Erreichbarkeit ist ein notwendiges jedoch nicht hinreichendes Kriterium, um die Erreichbarkeit des Registersatzes festzustellen. Ist eine Registerbank erreichbar, muss dies nicht automatisch für alle Register der Registerbank gelten. Zunächst kann z. B. die Modellierung der Adressierungsmethoden für verschiedene Instruktionen so gewählt werden, dass manche Instruktionen nur Teile einer Registerbank erreichen, damit z. B. verschiedene Prozessormodi erzeugt werden. Außerdem können durch die Verwendung der architektonischen Register Lücken in der Verlinkung der physikalischen Register entstehen. Die fehlende Überdeckung

durch die Adressierungsmethoden kann nur dynamisch festgestellt werden, da für die Adressierungsmethoden keine Umkehrfunktionen in der Spezifikation angegeben sind. Zur Validierung müssen die Instruktionen mit verschiedenen Adressierungsmethoden die Register des Registersatzes beschreiben und wieder auslesen. Werden alle Register erreicht, ist die Konsistenz der Erreichbarkeit nicht verletzt. Um die Konsistenz der Benutzung von architektonischen Registern zu zeigen, muss sichergestellt werden, dass jedes architektonische Register direkt oder indirekt verwendet wird. Hierfür wird im Unterschied zu (5.1) in (5.2) ein Registers p anstelle einer Registerbank angegeben.

$$\begin{aligned} &\forall p \in Reg \ : (\exists i \in instr : i = (name, ..., (adressing, p), ...)) \\ \vee\ &(\exists j \in instr : j = (name, ..., (adressing, a), ...) \wedge a \in aReg \ : a \rightarrow p \ ) \end{aligned} \tag{5.2}$$

Die Individualität der physikalischen Register kann durch die Spezifikation in *ViCE-UPSLA* nicht verletzt werden, die Überprüfung dieser Eigenschaften ist für die Spezifikation auf einem niedrigeren Abstraktionsniveau dennoch von Bedeutung. Die Konsistenzbedingungen können in *ViCE-UPSLA* beschrieben und die Validierungsmethoden dazu entwickelt werden, wie in Abschnitt 5.2.1 beschrieben.

Bei der Verwendung von architektonischen Registern wird das Aliasing definiert, wodurch auch indirekter Zugriff auf die Speicherstellen ermöglicht wird. Genauso wie die Erreichbarkeit der physikalischen Register kann das korrekte Aliasing durch architektonischen Register geprüft werden. Das Aliasing wird in *ViCE-UPSLA* durch vier Konstrukte unterstützt und aus der Spezifikation automatisch generiert. Die Konsistenz der Registersatz-Struktur kann auf dieser Ebene nicht verletzt werden. Die Inkonsistenzen können jedoch durch den Prozessorentwickler induziert werden, daher ist eine dynamische Überprüfung der Spezifikation sinnvoll. Damit werden irreguläre dynamische Konflikte für das Aliasing beschrieben. Die Instruktionen und Wertebereiche für die Operanden müssen von dem Prozessorentwickler als zusätzliches Wissen für das dynamische Testen angegeben werden. Die Konsistenzbedingung in (5.3) beschreibt, wenn *areg* Alias von *reg* ist, muss der gespeicherte Wert *value* in einem architektonischen Register *areg* gleich dem Wert in einem physikalischen Register *reg* sein.

$$(areg \in aReg \wedge reg \in Reg \ \wedge reg \rightarrow areg) : value(reg) = value(areg) \tag{5.3}$$

5.2.3 Adressierungsmethoden- und Instruktionsformatkonflikte

Die Adressierungsmethoden und Instruktionsformate verknüpfen in mehreren Schritten den Registersatz mit dem Instruktionssatz, wie in Abschnitt 4.3.4 durch die Konzepte von *ViCE-UPSLA* beschrieben. Der Pfad für die Verknüpfungen der Elemente in einer Prozessorspezifikation kann wie folgt dargestellt werden:

$$Instruction \rightarrow InstrForm \rightarrow intOpd \rightarrow Addr \rightarrow instrOpd \rightarrow RegSet$$

Durch die Verknüpfung zu einem Instruktionsformat (*InstrForm*) werden für eine Instruktion (*Instruction*) in erster Linie die strukturellen Eigenschaften beschrieben. Die interpretierten Operanden (*intOpd*) in einem Instruktionsformat werden mit den Adressierungsmethoden (*Addr*) verknüpft, welche aus den Befehlsoperanden (*instrOpd*) die Werte für die interpretierten Operanden bestimmen. Dazu werden die Werte aus dem Registersatz (*RegSet*) geladen. Die Berechnung der Werte in einer Adressierungsmethod (*Addr*) wird durch eine Funktion angegeben.

Die Instruktionsformate spezifizieren die Position und die Breite des Operationscodes oder der interpretierten Operanden, die für die Kodierung der Instruktionen verwendet werden. Dabei können Inkonsistenzen zwischen den Instruktionen in der Prozessorspezifikation und den Instruktionsformaten entstehen.

Die Adressierungsmethoden beschreiben den Zugriff auf die Register des Prozessors, wie bereits in Abschnitten 4.3.3 und 5.2.2 beschrieben. Die wesentliche Aufgabe der Adressierungsmethoden ist, die Werte für die interpretierten Operanden korrekt aus dem Registersatz zu laden, bzw. zu speichern. Die Konflikte in einer Adressierungsmethode können bei der Bestimmung der Registeradressen oder für die komplexen Adressierungsarten, bei der Berechnung der Werte für die interpretierten Operanden entstehen. Im Folgenden werden die Konfliktquellen für die Instruktionsformate und Adressierungsmethoden beschrieben.

Die Instruktionsformate sind in Bit-Felder eingeteilt und werden durch ein Feld für den Operationscode und durch mehrere Felder für die interpretierten Operanden beschrieben. Durch die Vorgabe der Breite werden die Wertebereiche für die Felder definiert. Die damit beschriebenen Wertebereiche müssen mit den spezifizierten Mengen der Instruktionen konsistent sein.

Jede Instruktion muss durch einen individuellen Operationscode kodiert werden können. Eine Konsistenzbedingung für diese Eigenschaft wird in (5.4) angegeben. Wird das Feld des Operationscodes o_{if} eines Instruktionsformates *if* definiert, wird damit die maximal kodierbare Menge von Instruktionen mit diesem Instruktionsformat auf $2^{|o_{if}|}$ festgelegt. Wird die maximale Menge von Instruktionen $|ownsInstrForm_{if}|$ überschritten, besteht ein Konflikt. Diese Konsistenzbedingung beschreibt einen regulären Konflikt, der auch auf niedrigeren Abstraktionsebenen zu Konflikten führt, wie in Abschnitt 5.2.1 beschrieben. Die Analyse dieser Konsistenzbedingung kann statisch direkt in der Spezifikation überprüft werden.

$$\forall if \in InstrForm \wedge \; o_{if} \in Opc \Rightarrow |ownsInstrForm_{if}| \leq 2^{|o_{if}|} \tag{5.4}$$

Analog zu der Konsistenzbedingung aus der Definition 5.4 wird auch die Anzahl der adressierbaren Register durch die Befehlsoperanden eingeschränkt. Durch die Breite eines Befehlsoperanden, z. B. durch n Bits beschrieben, können maximal 2^n Registeradressen ausgedrückt werden. Ist die Zielregisterbank dieses Operanden kleiner als sein

Wertebereich, ist das ein Hinweis auf eine Inkonsistenz, da durch die Überschreitung des Wertebereichs bei der Ausführung eine nicht vorhandene Registeradresse benutzt werden kann. Ist die Registerbank größer als der Wertebereich, ist das ebenfalls ein Hinweis auf eine mögliche Inkonsistenz, da nicht alle Register erreicht werden können. Eine solche Struktur kann jedoch auch gewollt sein.

Die Adressierungsmethoden spezifizieren den Zugriff auf die Register des Prozessors, deren Berechnungsvorschriften als Funktionen angegeben werden. Wie in Abschnitt 4.3.4 beschrieben, werden die Adressierungsmethoden in die direkten und indirekten Adressierungsmethoden eingeteilt. Die indirekten Adressierungsmethoden werden weiter in `Pre load` und `Post load` unterteilt. Abstrakt betrachtet verwendet eine Adressierungsmethode eine Registeradresse, um den Wert aus dem Register zu lesen und übergibt diesen Wert an den interpretierten Operanden. Die indirekten Adressierungsmethoden führen zusätzlich Berechnungen durch, bevor der Wert an den interpretierten Operanden übergeben wird. Abhängig von der angewendeten Adressierungsmethode, `Pre load` oder `Post load`, können die Adressierungsmethoden unterschiedliche Inkonsistenzen aufweisen, indem der Wert für den interpretierten Operanden oder die Registeradresse falsch berechnet wird.

Die direkten Adressierungsmethoden werden in *ViCE-UPSLA* vollständig automatische generiert und können in der Spezifikation oder im Simulator keine Inkonsistenzen erzeugen.

Die indirekten `Pre load` Adressierungsmethoden berechnen die Registeradressen aus den Befehlsoperanden. Ist die Implementierung der Berechnungsfunktion falsch, wird der Wert eines falschen Registers geliefert. Bei den `Post load` Adressierungsmethoden wird der Wert für den interpretierten Operanden berechnet, nachdem die Werte aus den Registern geladen wurden. Hierbei können durch fehlerhafte Implementierung der Funktion für die Adressierungsmethode falsche Werte berechnet werden.

Durch die indirekte Adressierungsmethoden verursachte Inkonsistenzen können erst aus der Simulation erkannt werden und gehören in die Kategorie der dynamischen Konflikte. Außerdem muss für die Validierung der Konflikte eine zusätzliche Angabe durch den Prozessorentwickler durchgeführt werden, womit irreguläre Konflikte beschrieben werden.

5.2.4 Instruktionssatzkonflikte

Der Instruktionssatz umfasst alle Instruktionen eines Prozessors. Für die Validierung eines Instruktionssatzes müssen seine Instruktionen validiert werden.

Die Spezifikation der Instruktionen wird durch statische Eigenschaften wie die Kodierung oder die Lese-/Schreib-Richtung der Operanden etc. angegeben. In erster Linie muss die Verwendung der Operanden und die Korrektheit der Kodierung der Instruktionen sichergestellt werden. Die dynamischen Eigenschaften einer Instruktion werden

durch eine operationale Beschreibung zyklengenau beschrieben. Dabei werden zum einen die Benutzung der Operanden und die Operationen, die bei der Ausführung der Instruktion auf die Operanden angewendet werden, beschrieben. Zum anderen wird die Benutzung der in der Mikroarchitektur definierten Ressourcen des Prozessors für jede Instruktion definiert. Damit können im Instruktionssatz eine Reihe unterschiedlicher Inkonsistenzen zu den Konstrukten wie Mikroarchitektur, Instruktionsformaten, Adressierungsmethoden usw. entstehen.

Bei der Dekodierung der Befehle werden die Instruktionen anhand ihres Operationscodes oder des Namens identifiziert. Durch die Verwendung von *ViCE-UPSLA* können identische Namen oder Operationscodes nicht spezifiziert werden. Dennoch kann eine Mehrdeutigkeit zwischen den Operationscodes verschiedener Instruktionen entstehen. Ein solcher Fehler tritt auf, wenn ein Operationscode Präfix eines längeren Operationscodes ist. Die Tabelle 5.1 zeigt ein Beispiel, in dem zu verschiedenen Instruktionen mit unterschiedlichen Parametern dasselbe Instruktionswort angegeben werden kann. Die Inkonsistenz führt erst bei der Ausführung im Simulator zu einem Fehler, kann jedoch bereits in der Spezifikation validiert werden.

Assemblerformat	instra reg, imm	instrb reg, imm
Binärkodierung	01 xx yyyy	011 xx yyy
Befehl	instra 2, 3	instrb 0, 3
Instruktionswort	**01100011**	**01100011**

Tabelle 5.1: Konflikt bei der Dekodierung von Befehlen.

Die Konsistenzbedingung für die Kodierung von Instruktionen in (5.5) beschreibt, dass kein Operationscode Präfix eines anderen Operationscodes sein darf.

$$\forall opc_x, opc_y \in Opc \wedge |opc_x| \leq |opc_y| \ : opc_x \neq prefix(opc_y) \tag{5.5}$$

Die operationale Beschreibung einer Instruktion umfasst Operanden und Operatoren, welche als Knoten in einem Datenflussgraphen *dfg* eingesetzt werden. Damit wird die Reihenfolge der auszuführenden Aktionen und die Ressourcenbelegung bei der Ausführung anhand der Pfade zyklengenau angegeben. Für Knoten in der operationalen Beschreibung werden die Ressourcen aus der Mikroarchitektur angegeben. Die operationale Beschreibung der Instruktionen und der Datenpfad der Mikroarchitektur *DataPath* müssen konsistent sein. Dabei muss für jeden Pfad zwischen den Knoten der operationalen Beschreibung *dfg* ein Pfad *datapath* zwischen den Ressourcen im Datenpfad definiert sein. Diese Konsistenzbedingung wird in (5.6) angegeben und kann statisch in der Spezifikation geprüft werden.

$$\forall dfg \in DFG : (\exists datapath \subseteq DataPath : datapath \mapsto dfg) \tag{5.6}$$

Eine Inkonsistenz wird in Abbildung 5.1 demonstriert. Wird dem Operator `add` als Ressource die ALU `Arith` und dem Operanden `Sr1` die Ressource `RegA` zugeordnet, besteht kein Konflikt. Ist dem Operanden `Sr1` die Ressource `RegB` zugeordnet, wie durch den rot gestrichelten Pfeil beschrieben, besteht kein Pfad zwischen den Ressourcen, wodurch ein Konflikt entseht.

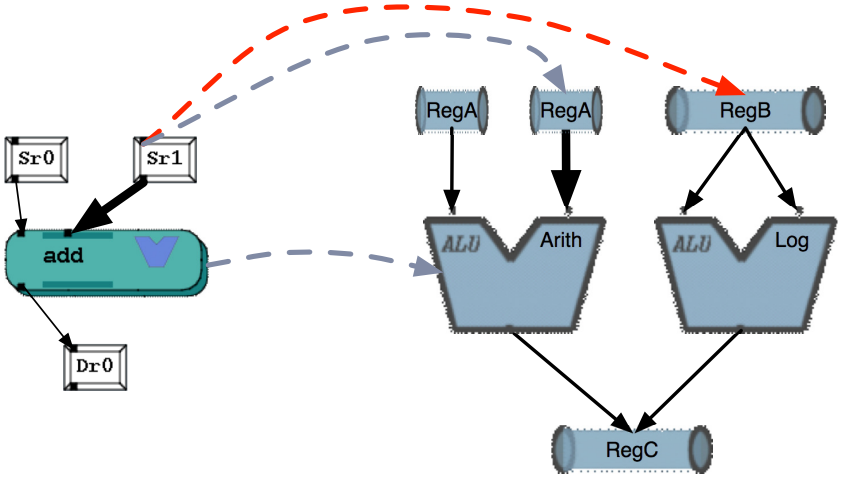


Abbildung 5.1: Inkonsistenz zwischen der operationalen Beschreibung und dem Datenpfad. Links: operationale Beschreibung; rechts: Datenpfad der Mikroarchitektur. Die Zuordnung der Knoten aus der operationalen Beschreibung zu den Ressourcen aus der Mikroarchitektur wird durch gestrichelte Pfeile beschrieben.

Die Mehrdeutigkeit von Konstrukten in der Spezifikation des Prozessors kann in den späteren Phasen des Entwurfs, z. B. bei der Entwicklung von Compilern für den Prozessor in der Codeselektion, zu Konflikten führen. Die Konsistenzbedingung wird in (5.7) angegeben. Besitzen zwei Instruktionen *i,j* identische operationale Beschreibungen *dfg* und Instruktionsformate *ownsInstrForm*, liegt eine Redundanz und unter Umständen auch eine Inkonsistenz vor.

$$\forall i, j \in instr : (dfg_i = dfg_j \wedge ownsInstrForm_i = ownsInstrForm_j \Rightarrow i = j) \quad (5.7)$$

Ein Konflikt besteht dann, wenn bei der Auswahl der Instruktionen für die Umsetzung eines Programms die Entscheidung zwischen zwei Instruktionen nicht eindeutig ist. Der Instruktionssatz kann statisch auf Redundanz geprüft werden.

Die semantische Korrektheit der Instruktionen hängt zum einen von der korrekten Implementierung der Funktionen für die Operatoren, zum anderen von der Spezifika-

tion der operationalen Beschreibung ab.

Die Inkonsistenzen in der operationalen Beschreibung können durch falsch erzeugte DFGs entstehen. Wird z. B. eine Instruktionssatzerweiterung durchgeführt, bei der eine neue Instruktion eine Folge von zwei anderen Instruktionen ersetzen soll, muss das Ergebnis der neuen Instruktion konsistent zu dem ursprünglichen Ergebnis der verwendeten Instruktionen sein. Diese Konsistenzbedingung beschreibt irreguläre Konflikte, bei denen der Prozessorentwickler die Äquivalenzen zwischen den Instruktionen und Instruktionsfolgen angeben muss.

Eine weitere Quelle für Inkonsistenzen in der operationalen Beschreibung bilden die Wertebereiche der Operanden und die Parameter von Operatoren. Die Operatoren beschreiben Funktionen, die auf die Operanden angewendet werden. Die Datentypen werden durch Bitbreiten definiert. Stimmt die Bitbreite eines Operanden mit der erwarteten Bitbreite eines Operators nicht überein, kann die Berechnung zu undefinierten Ergebnissen führen. Die Konsistenz zwischen den Wertebereichen beschreibt damit reguläre Konflikte und kann statisch validiert werden.

Die Berechnungsvorschriften der Operatoren werden wie bei den Adressierungsmethoden durch Funktionen angegeben. Für die Funktionen werden in der Spezifikation keine Umkehrfunktionen angegeben, sodass für die dynamische Validierung zusätzliche Angaben durch den Prozessorentwickler benötigt werden.

5.2.5 Pipelinekonflikte

Mit *ViCE-UPSLA* werden Prozessoren mit überlappender und paralleler Verarbeitung spezifiziert. Für die Beschreibung des Verhaltens wird eine Mikroarchitektur mit einer Pipeline angegeben, wie in Abschnitt 4.3.6 bereits beschrieben. Durch die Busse zwischen den Ressourcen wird die Übergabe der Werte bei der Verarbeitung von Instruktionen festgelegt. Die Spezifikation der Pipeline beschreibt schrittweise die Stückelung der ausführbaren Operationen für die Simulation. Damit werden die Verfügbarkeit der Werte und die Belegung der Ressourcen bei der Ausführung spezifiziert. Die Bypass-Struktur bringt eine zusätzliche dynamische Komponente, indem die Verfügbarkeit der Werte bei der Ausführung in der Pipeline zusätzlich manipuliert wird.

Die Inkonsistenzen in der Spezifikation resultieren aus den Wechselwirkungen zwischen den Instruktionen bei der Simulation. Für Prozessoren mit überlappender oder paralleler Verarbeitung sind in der Domäne des Prozessorentwurfs die Steuer-, Daten- und Ressourcenkonflikte identifiziert [HP06]. Die Eigenschaften der Konflikte werden in Bezug auf die Spezifikation mit *ViCE-UPSLA* im Folgenden erläutert.

Die Steuerkonflikte können immer dann auftreten, wenn der Befehlszähler (program counter `PC`) durch entsprechende Instruktionen verändert wird, z. B. bei der Ausführung von Sprungbefehlen. Durch Sprungbefehle verursachte Konflikte können zum einen durch fehlerhafte Berechnung der Sprungadresse oder durch unvollständige Ausführung nachfolgender Instruktionen in der Pipeline auftreten. Die Konflikte

entstehen dann, wenn die Spezifikation der Sprungbefehle keine Routine für die Berücksichtigung von nachfolgenden Instruktionen beschreibt. Steuerkonflikte sind reguläre Konflikte und können durch automatisch generierte Testfälle durch dynamisches Testen festgestellt werden.

Die Datenkonflikte werden bei der Ausführung von Instruktionen durch die Datenabhängigkeiten zwischen Operanden der aufeinanderfolgenden Instruktionen ausgelöst. Für die Auflösung der Datenkonflikte werden im Prozessorentwurf Bypässe eingesetzt, die auch mit *ViCE-UPSLA* spezifiziert werden können. Da Bypässe meist gezielt an einigen Stellen eingesetzt oder weggelassen werden, um z. B. die Fläche des Prozessors zu minimieren, werden einige Stellen, an denen Konflikte entstehen können, in Kauf genommen und anschließend durch Compiler kompensiert. Durch Individualisierung der Bypass-Struktur und mit zunehmender Größe des Instruktionssatzes erhöht sich die Anzahl der möglichen Konfliktstellen.

Die Spezifikation der Bypässe beschreibt eine bedingte Weiterleitung der Werte bei der Ausführung. Dabei werden die Quellen und die Priorität der Werte überprüft. Die Spezifikation der Bypässe kann inkonsistent sein, wenn aus der Spezifikation gezeigt werden kann, dass die Bedingungen für die Weiterleitung niemals erfüllt sein können. Diese Eigenschaft für Bypässe beschreibt einen regulären Konflikt aus dem Prozessorentwurf und kann statisch aus der Spezifikation validiert werden. Bei korrekt spezifizierten Bypässen kann die Validierung der Weiterleitung der Werte durch dynamisches Testen durchgeführt werden, wofür Testfälle aus der Spezifikation des Prozessors abgeleitet werden können.

Die Mikroarchitektur beschreibt statische Eigenschaften, indem die Instruktionen des Prozessors und deren Operationen mit den Ressourcen aus dem Datenpfad konform sein müssen. Die Konflikte der Ressourcen wurden bereits im Rahmen der Konflikte in der operationalen Beschreibung der Instruktionen behandelt. Dynamische Ressourcenkonflikte entstehen bei überlappender oder paralleler Ausführung mehrere Instruktionen in verschiedenen Pipeline-Stufen. Ein Konflikt besteht dann, wenn im selben Taktzyklus mehrere Instruktionen eine Ressource benötigen, die nicht in ausreichender Anzahl zur Verfügung steht.

Mit *ViCE-UPSLA* können Simulatoren wahlweise mit oder ohne Interlock-Mechanismen erzeugt werden. In einem Simulator mit Interlocks werden Ressourcen automatisch zugeteilt, dabei werden die Ressourcenkonflikte aufgelöst. Die möglichen Inkonsistenzen können bei der Simulation mit Interlocks auftreten, wenn die Zuteilung der Ressourcen inkonsistent ist, indem Programme mehr Taktzyklen benötigen als vorgesehen. Um solche Inkonsistenzen zu validieren, werden zusätzliche Angaben durch den Prozessorentwickler benötigt, wobei z. B. die erwartete Laufzeit für Programmfragmente angegeben wird.

Die Mikroarchitekturen ohne Interlocks werden meist so entworfen, dass per Definition keine Ressourcenkonflikte auftreten können oder durch Compilerwerkzeuge aufgelöst werden. Die Inkonsistenzen in der Spezifikation können erst durch die Simulation

aufgedeckt werden und beschreiben dynamische Konflikte.

5.3 Validierungsmethoden

Auf der Grundlage der visuellen Sprache *ViCE-UPSLA* und dem Fehlermodell werden in diesem Abschnitt die Validierungsmethoden beschrieben. Die Aufgaben der Validierung wurden bereits in Abschnitt 5.1 beschrieben. Zum Erreichen dieser Ziele werden in diesem Abschnitt die verschiedenen Validierungsmethoden vorgestellt, mit denen der Prozessorentwickler bei der Validierung unterstützt werden soll.

Wie bereits aus Abschnitt 5.2 zum Fehlermodell hervorgeht, sind die möglichen Konflikte in statische und dynamische Konflikte eingeteilt worden. Analog dazu werden die Validierungsmethoden in statische und dynamische Methoden aufgeteilt und in Abschnitten 5.4 und 5.5 beschrieben.

Die statische Konsistenz der Spezifikation ist eine notwendige Voraussetzung für die Generierung eines Prozessorsimulators und die Durchführung der dynamischen Validierung. Für die Beschreibung der statischen Validierungsmethoden in Abschnitt 5.4 werden für die Konsistenzbedingungen aus dem Fehlermodell in Abschnitt 5.2 verwendet.

Die dynamischen Validierungsmethoden in Abschnitt 5.5 müssen die statischen Validierungsmethoden ergänzen, sodass alle *ViCE-UPSLA* Sprachkonstrukte einer Prozessorspezifikation validiert werden. Eine dynamische Validierungsmethode muss beschreiben, wie die Eigenschaften eines Konstrukts in der Spezifikation aus den Ergebnissen der Simulation geprüft werden können. Dazu werden die Methoden für die Validierung sowie die Ableitung der Testfälle aus der Spezifikation beschrieben.

Neben den Konzepten für die Validierung muss für jede Validierungsmethode angegeben werden, wie die Methode angewendet wird. Außerdem muss für jede Methode angegeben werden, wie die Validierungsergebnisse interpretiert und die entdeckten Inkonsistenzen dem Prozessorentwickler präsentiert werden.

5.4 Statische Analyse

Die Sprache *ViCE-UPSLA* ist so entworfen, dass die verwendeten Sprachkonstrukte immer für sich selbst konsistent sind. Damit ist z. B. die syntaktische Analyse nicht notwendig. Durch gegebene Freiheitsgrade bei der Spezifikation kann nicht jede Konfiguration der Sprachkonstrukte konsistent sein. Die Inkonsistenzen in der Spezifikation entstehen immer dann, wenn Sprachkonstrukte oder deren Eigenschaften inkonsistent zu anderen Sprachkonstrukten verwendet werden. Außerdem können die Komponenten des spezifizierten Prozessors inkonsistent zu den allgemeinen Richtlinien aus dem Prozessorentwurf sein, die für die Erzeugung des Generators oder zur Weiterentwicklung des Prozessors befolgt werden müssen. Zuletzt können Konsistenzbedingungen aus den

zusätzlichen Anforderungen durch den Prozessorentwickler bestehen. Die Eigenschaften der Konflikte wurden in Abschnitt 5.2 im Rahmen des Fehlermodells beschrieben. In diesem Abschnitt werden zunächst die Aspekte der statischen Validierungsmethoden für die Prozessorvalidierung beschrieben.

Für die Validierung der inkonsistenten Benutzung von Komponenten eines Prozessors in einer Spezifikation wurden im Werkzeugsystem mit *ViCE-UPSLA* Konsistenzprüfungen integriert, welche die Spezifikation des Prozessors statisch überprüfen und die Inkonsistenzen aufzeigen. Für die Validierung der Inkonsistenzen durch zusätzliche Entwurfsrichtlinien wurde neben den Validierungsmethoden die Integrationsmöglichkeit des zusätzlichen Wissens in *ViCE-UPSLA* beschrieben. Die Modellierung des zusätzlichen Wissens wird in Abschnitt 5.6 beschrieben.

Die Methoden für die statische Analyse lassen sich durch drei Aspekte charakterisieren:

- Zeitpunkt der Prüfbarkeit einer Bedingung – Damit werden Szenarios für die sinnvolle Anwendung einer Methode beschrieben, gemessen an der Vollständigkeit der Prozessorspezifikation.
- Aus einer Inkonsistenz resultierende Konsequenz – Damit wird die Schwere und die Folgen durch Inkonsistenzen für eine Spezifikation bewertet.
- Typ der statischen Analyse – Durch die Typisierung einer Validierungsmethode wird die verwendete Methode bei der Validierung bestimmt.

Der Zeitpunkt für die Überprüfung einer Konsistenzbedingung ist für die Umsetzung der Konsistenzprüfungen im Werkzeugsystem wichtig. Dabei muss für die Methoden herausgestellt werden, ob diese kontinuierlich, abschließend oder angestoßen ausgeführt werden sollen. Nach der Vorstellung der statischen Validierungsmethoden werden in Abschnitt 5.4.7 verschiedene Szenarios für die Methoden beschrieben.

Eine Konsistenzprüfung liefert ein Ergebnis aus dem hervorgeht, ob eine Inkonsistenz besteht oder nicht. Das Ergebnis muss dem Prozessorentwickler präsentiert werden. Für die Präsentation der Ergebnisse lässt sich ein weiterer Aspekt für die Methoden der statischen Analyse als Konsequenz beschreiben. Die Konsequenzen beschreiben die Präsentation und die Behandlung der entdeckten Inkonsistenzen durch eine Konsistenzprüfung. In dieser Arbeit wird zwischen drei Konsequenzen differenziert:

- Hinweis – Eine vorhandene Inkonsistenz bedingt eingeschränkte Funktionalität oder Nutzung der vorhandenen Prozessorkonstrukte. Die Inkonsistenz führt in einem Simulator oder Prozessor zu keinem fehlerhaften Verhalten.

- Warnung – Eine Inkonsistenz, die zu fehlerhaftem Verhalten im Prozessor oder Simulator führt; die Generierung des Prozessorsimulators ist möglich.
- Fehler – Eine Inkonsistenz, mit der die Erzeugung eines Simulators oder Prozessors nicht sinnvoll ist.

Die entdeckten Inkonsistenzen können unterschiedlich schwere Verletzungen in der Spezifikation verursachen. Dazu müssen die Ergebnisse der Validierung und damit die Inkonsistenzen klassifiziert werden. Dabei können die unterschiedlichen Ergebnisse einer Validierungsmethode zu unterschiedlichen Konsequenzen führen. Dies soll am folgenden Beispiel verdeutlicht werden.

Bei der Analyse der Überdeckung zwischen den Befehlsoperanden `B` einer Adressierungsmethode und ihrer Zielregisterbank `R` sind drei Ergebnisse möglich. Die Anzahl der Adressen, die eine Adressierungsmethode aus den gegebenen Befehlsoperanden ableitet, kann größer, kleiner oder gleich der Anzahl der Register der Zielregisterbank sein.

Für den Fall, dass die Anzahl der Register in einer Registerbank gleich dem Adressraum einer Adressierungsmethode ist, besteht keine Inkonsistenz.

Bei der Unterschreitung der Registeranzahl `|B|<|R|` besteht eine Inkonsistenz, da nicht alle Register mit der beschriebenen Adressierungsmethode erreicht werden. Solche Konstruktionen werden in einigen Architekturen gezielt eingesetzt, wenn eine Adressierungsmethode nur auf bestimmte Bereiche eines Registersatzes zugreifen soll. Aus der Spezifikation kann nach wie vor ein Simulator oder Prozessor erzeugt werden, daher kann die Inkonsistenz als nicht schwerwiegend eingestuft werden. Hierfür ist es ausreichend, dem Prozessorentwickler einen Hinweis zu geben, dass nicht alle Register erreicht werden.

Bei der Überschreitung des Wertebereichs `|B|>|R|` besteht die Möglichkeit, für die Adressierungsmethode ungültige Adressen zu benutzen. Ungültige Adressen führen bei der Simulation oder der Ausführung im Prozessor zu Fehlern. Die Generierung des Prozessorsimulators kann jedoch durchgeführt und der Simulator benutzt werden. Bei einer solchen Inkonsistenz muss dem Benutzer eine Warnung ausgegeben werden. Die Warnungen können bei Ad hoc Entwürfen eines Prozessors in Kauf genommen werden, wenn z. B. bei einer Entwurfsraumexploration nur Teile der Prozessorspezifikation experimentell verändert werden.

Eine schwerwiegende Verletzung der Konsistenz einer Spezifikation wird dadurch definiert, dass aus der Spezifikation mit einer solchen Inkonsistenz die Erzeugung eines Simulators oder die Weiterentwicklung des Prozessors nicht sinnvoll ist. Eine solche Verletzung besteht z. B. wenn eine Instruktion mehr Ressourcen benötigt, als in der Mikroarchitektur spezifiziert sind. Damit würde mit *ViCE-UPSLA* ein ungültiger Simulator erzeugt werden. Die Erzeugung eines solchen Simulators ist nicht sinnvoll, da die Funktionalität eindeutig fehlerhaft sein wird. Die Inkonsistenz muss dem Prozessorentwickler als Fehler in der Spezifikation aufgezeigt werden.

Für die statischen Konsistenzprüfungen können allgemeine Validierungsmethoden angegeben werden, womit die Konsistenzprüfungen in dieser Arbeit klassifiziert werden. Für die Zuordnung einer allgemeinen Vorgehensweise müssen die untersuchten Eigenschaften der zu validierenden Konstrukte herausgestellt werden. Wie bereits beschrieben, kann z. B. zwischen den Befehlsoperanden der Adressierungsmethoden und den Registern einer Registerbank die Überdeckung überprüft werden. Die Register beschreiben eine Menge von Speicherstellen und die Befehlsoperanden beschreiben den Wertebereich für die Kodierung der Adressen, womit die Konsistenz der Mengenkodierung zwischen den Komponenten überprüft wird. Mit der Vorgehensweise aus dem Beispiel lassen sich verschiedene Arten von Konsistenzprüfungen für die Prozessorspezifikation in *ViCE-UPSLA* identifizieren, wie z. B. Verwendungsnachweis, Kodierung von Mengen oder Grapheinbettung usw. Die verschiedenen Arten für die Analyse werden in den jeweiligen Abschnitten ausführlich erläutert.

In diesem Abschnitt wurden die charakteristischen Eigenschaften der statischen Konsistenzprüfungen beschrieben. Dabei wurde herausgestellt, dass eine Konsistenzprüfung verschiedene Konsequenzen, abhängig von den Ergebnissen der Prüfung, beschreiben muss. In den folgenden Abschnitten 5.4.1 bis 5.4.6 werden die Methoden für die statische Analyse beschrieben. Bei der Beschreibung der Methoden wird auf die Arten der Analyse und die damit feststellbaren Inkonsistenzen eingegangen. Außerdem werden für die Methoden mögliche Konsequenzen, abhängig von den Ergebnissen der Analyse, beschrieben. In Abschnitt 5.4.7 werden Szenarios für die Anwendung der beschriebenen Validierungsmethoden beschrieben.

5.4.1 Kodierung von Mengen

In einer Prozessorspezifikation wird die Kodierung für die Menge der Instruktionen oder der Register indirekt durch die Spezifikation der Bit-Breiten der Komponenten in den Instruktionsformaten durchgeführt. Im Fehlermodell wurde bereits beschrieben, dass die Inkonsistenzen zwischen den kodierten Mengen und den Verwendungsstellen der Komponenten entstehen können. Die Bit-Breiten der Befehlsoperanden müssen z. B. konsistent zu der Menge der damit adressierbaren Register sein. Die Validierung der Mengenkodierung wird in einer Prozessorspezifikation durch die spezifizierten Komponenten und die Verknüpfungen dieser Komponenten zu den Verwendungsstellen ermöglicht. In diesem Abschnitt werden die Methoden für die Validierung der Mengenkodierung für die identifizierten Komponenten aus dem Fehlermodell beschrieben. Für die vollständige Beschreibung der Validierungsmethoden werden außerdem die Konsequenzen für die Ergebnisse festgelegt.

Als ein einleitendes Beispiel kann die Kodierung für den Operationscode in den Instruktionsformaten und die Menge der Instruktionen im Instruktionssatz betrachtet

werden. Die Instruktionsformate besitzen in der Spezifikation ein Feld für die Kodierung des Operationscodes einer Instruktion, der durch eine Bit-Breite definiert wird, z. B. 8-Bit. Mit dieser Spezifikation wird die maximale Anzahl von individuellen Operationscodes festgelegt, z. B. können mit 8-Bit maximal 256 Operationscodes beschrieben werden. Die Anzahl von Instruktionen mit diesem Instruktionsformat darf die maximal kodierbare Menge nicht überschreiten.

Für die Analyse der Mengenkodierung durch Operationscodes muss zunächst die kodierbare Menge von Instruktionen ermittelt werden. Bei der einfachen Überprüfung der Konsistenz kann festgestellt werden, ob die Bit-Breite für den Operationscode eines Instruktionsformats mit der Menge der damit erzeugten Instruktionen konsistent ist. Aus der Bedingung, dass die Operationscodes von Instruktionen individuell sein müssen folgt, dass sich auch die Instruktionen mit unterschiedlichen Instruktionsformaten in der Kodierung nicht gleichen dürfen. Dazu wurde in Abschnitt 5.2 im Fehlermodell herausgestellt, dass die Spezifikation inkonsistent ist, wenn ein kürzerer Operationscode Präfix eines längeren Operationscodes ist. Bei der Analyse muss die Menge der kodierbaren Instruktionen global über alle Instruktionsformate des Prozessors betrachtet werden und mit der Menge der tatsächlich spezifizierten Instruktionen validiert werden.

Im folgenden Beispiel wird das Prinzip für die Konsistenzbedingung verdeutlicht. Bei der Benutzung von zwei Instruktionsformaten A mit a = 3-Bit und B mit b = 2-Bit breiten Operationscodes, wie in Abbildung 5.2 dargestellt, ist die maximale Anzahl von Instruktionen im Instruktionssatz auf acht begrenzt. Werden mit dem Instruktionsformat B z. B. zwei Instruktionen erzeugt, können mit dem Instruktionsformat A nur noch vier Instruktionen eindeutig kodiert werden usw.

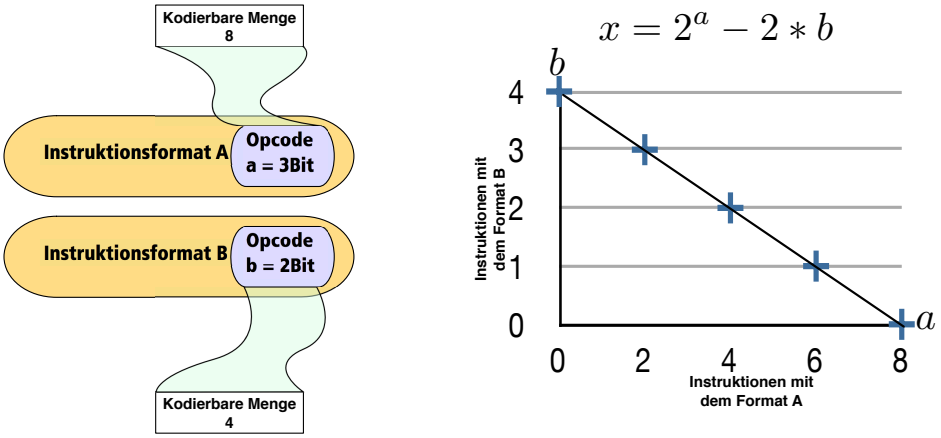


Abbildung 5.2: Berechnung der kodierbaren Menge aus der Abhängigkeit der Instruktionsformate A und B.

Die Konsistenzprüfung der Mengenkodierung kann nur positiv oder negativ als Ergebnis liefern. Solange die Anzahl der Instruktionen die kodierbare Menge nicht überschreitet, ist die Spezifikation konsistent. Ist die Obergrenze der kodierbaren Instruktionen für ein Instruktionsformat überschritten, muss der Prozessorentwickler eine Warnung erhalten, da die Realisierung eines Prozessors nicht mehr sinnvoll sein kann. Da ein Simulator aus *ViCE-UPSLA* die Operationscodes für die Dekodierung der Instruktionen nicht benötigt, kann ein korrekter Simulator erzeugt werden. Die Erzeugung solcher Simulatoren kann z. B. bei einer experimentellen Instruktionssatzerweiterung sinnvoll sein.

Die Kodierung der Registermenge durch die interpretierten Operanden kann analog zur Analyse für die Operationscodes und die Instruktionen validiert werden. Hierbei wird die Konsistenz zwischen den Wertebereichen der Befehlsoperanden aus den Adressierungsmethoden und der Menge der Register der Zielregisterbänke analysiert.

Anders als bei der Kodierung von Instruktionen, kann für die Validierung von Registermengen zwischen drei Konsequenzen für die Ergebnisse unterschieden werden:

- Die Wertebereiche stimmen überein – die Spezifikation ist konsistent.
- Der Wertebereich des Befehlsoperanden ist kleiner als die Anzahl der zu adressierenden Register – Hinweis, da nicht alle Register erreicht werden können.
- Der Wertebereich des Befehlsoperanden ist größer als die Anzahl der zu adressierenden Register – Warnung, da bei der Ausführung Registeradressen verwendet werden können, für die keine Speicherstellen definiert sind.

5.4.2 Verwendungsnachweis

Der Verwendungsnachweis soll eine Prozessorspezifikation auf Vollständigkeit prüfen. Dafür muss sichergestellt werden, dass jede in der Prozessorspezifikation eingesetzte Komponente mit einer dazu passenden Komponente in der Spezifikation verknüpft ist. So kann eine Instruktion in ein Register schreiben, wenn dessen Registerbank über die Adressierungsmethoden mit der Instruktion verknüpft ist, oder eine Instruktion kann im Simulator ausgeführt werden, wenn sie einer ALU zugeordnet ist usw.

Die Validierung der Verwendung kann am Beispiel der Registerbänke anschaulich beschrieben werden. Die Registerbänke werden durch die Befehlsoperanden in den Adressierungsmethoden verknüpft, womit der Zugriff auf die Register einer Registerbank definiert wird. Wird eine Registerbank von keiner Adressierungsmethode referenziert, ist diese Registerbank nicht direkt erreichbar. Die Bedingungen für die Erreichbarkeit von architektonischen und physikalischen Registerbänken wurden bereits im Fehlermodell diskutiert und kann für die Validierung übernommen werden.

Bei der Validierung der Erreichbarkeit kann nur festgestellt werden, ob eine Registerbank eine Verknüpfung und somit eine Verwendung findet oder nicht. Im Entwicklungsprozess können durch den Prozessorentwickler Prozessorkonstrukte aus der

Benutzung experimentell entfernt werden, ohne diese zu löschen. Wird eine Registerbank nicht verwendet, kann aus der Spezifikation sowohl der Simulator als auch der Prozessor erzeugt werden. In diesem Fall muss als Konsequenz ein Hinweis für den Prozessorentwickler ausgegeben werden.

So wie die Erreichbarkeit von Registerbänken kann auch der Verwendungsnachweis für die Instruktionen durchgeführt werden. Z. B. bei der Entwicklung von Instruktionssatzerweiterungen können in den experimentellen Phasen der Entwicklung verschiedene Konfigurationen des Instruktionssatzes oder der Instruktionen gegeneinander validiert werden. Dabei werden die nicht verwendeten Instruktionen nicht gelöscht, sondern lediglich deaktiviert, indem diese z. B. keine Verknüpfungsstellen in einer ALU erhalten. Damit kann eine Spezifikation nicht verknüpfte Instruktionen enthalten. Die Konsequenz aus solchen Inkonsistenzen sollte ein Hinweis auf die nicht verknüpften Instruktionen für den Prozessorentwickler sein.

Analog zu der beschriebenen Methode kann der Verwendungsnachweis für die Instruktionsformate, die Adressierungsmethoden usw. durchgeführt werden.

5.4.3 Grapheinbettung

Die operationale Beschreibung der Instruktionen wird durch die Verknüpfung der Operanden und Operatoren in einem Datenflussgraph verteilt auf die Zyklen der Instruktion ausgedrückt. Die operationale Beschreibung einer konkreten Instruktion beschreibt damit die Reihenfolge der Operationen und die Wertübergabe dieser Instruktion. Für die Erzeugung des Simulators werden aus einzelnen Zyklen und Operationen der Instruktion Funktionen erzeugt, die bei der Simulation ausgeführt werden. Analog dazu beschreibt die Mikroarchitektur in einem Datenflussgraph die Ressourcen und die Verknüpfungen zwischen den Ressourcen eines Prozessors. Damit wird die Koordination für die Taktzyklen und Funktionen der Instruktionen für die überlappende Ausführung in der Pipeline bei der Erzeugung des Simulators spezifiziert.

Die Operationen der operationalen Beschreibung von Instruktionen und die Ressourcen aus der Mikroarchitektur sind in der *ViCE-UPSLA* Spezifikation verknüpft. Damit wird die Validierung der Konsistenz zwischen einer Verhaltensbeschreibung und der Mikroarchitektur ermöglicht. Es reicht nicht aus, die Verknüpfungen zwischen den Elementen zu überprüfen. Die Validierung muss zeigen, dass die in der Verhaltensbeschreibung beschriebenen Pfade über die Komponenten und folglich über die Ressourcen des Prozessors auch in der Mikroarchitektur nachvollzogen werden können.

Für die Validierung der Verhaltensbeschreibung und der Mikroarchitektur muss die Überdeckung der Datenflussgraphen durch den Datenpfad gezeigt werden. Abbildung 5.3 zeigt eine Einbettung der operationalen Beschreibung einer Instruktion in eine Mikroarchitektur. Die Instruktion verwendet zwei Werte aus den Quellregistern und schreibt das berechnete Ergebnis, durch zwei Operationen, in ein Zielregister. Die Mikroarchitektur enthält jedoch nur eine ALU und einen Lese-Port, womit die direkte

Einbettung der Instruktion nicht möglich ist. Eine Lösung für die Einbettung kann durch zwei Eigenschaften in der Architektur gegeben sein. Zum einen, wenn die ALU die Operationen `neg` und `add` in einem Taktzyklus ausführen kann, zum anderen, wenn die iterative Ausführung der Instruktionen in der Mikroarchitktur möglich ist, womit die Instruktion in zwei Zyklen in der ALU ausgeführt wird.

Die Validierungsmethode für die Konsistenzprüfung zwischen der operationalen Beschreibung wird in drei Schritten durchgeführt und im Folgenden anhand des Beispiels aus Abbildung 5.3 beschrieben. In jedem der drei Schritte werden Konsistenzen der bis dahin ermittelten Eigenschaften überprüft.

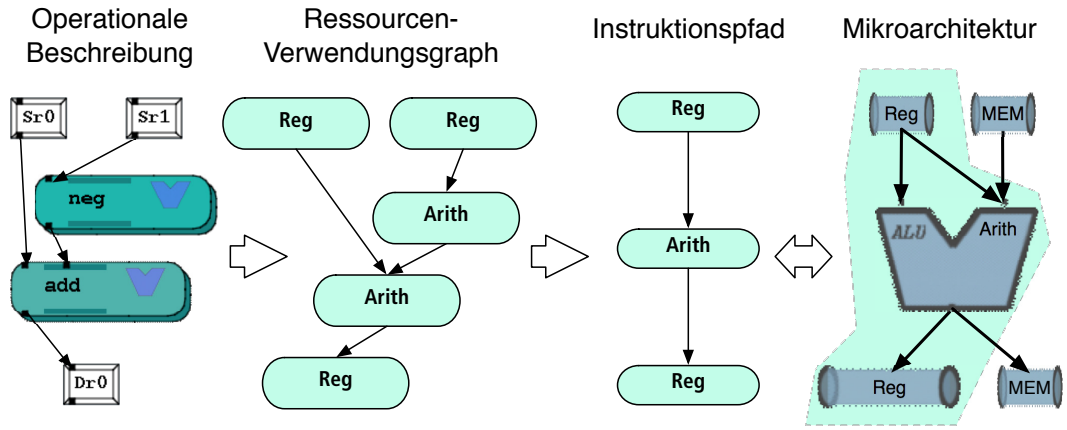


Abbildung 5.3: Schrittweise Einbettung einer operationalen Beschreibung in die Mikroarchitektur.

Im ersten Schritt werden die Ressourcen für die Knoten aus der operationalen Beschreibung ermittelt. Aus den Ressourcen und den Verknüpfungen zwischen den Knoten wird ein Ressourcen-Verwendungsgraph erzeugt. Auf dem Ressourcen-Verwendungsgraph kann bereits der erste Validierungsschritt durchgeführt werden, indem überprüft wird, ob in einem Ausführungszyklus der Instruktion Ressourcen aus unterschiedlichen Pipeline-Stufen verwendet werden.

Im zweiten Schritt muss der Ressourcen-Verwendungsgraph zu einem Instruktionspfad reduziert werden. Für die Reduzierung werden die Eigenschaften der Ressourcen oder der Mikroarchitektur berücksichtigt, z. B. ob die Ausführung in der Mikroarchitektur durch Interlocks angehalten werden kann. Für das Beispiel aus Abbildung 5.3 wird von einer Mikroarchitektur mit Interlocks ausgegangen. Bei der Reduzierung können z. B. aufgrund der Interlocks gleiche Ressourcen aus aufeinander folgenden Zyklen zusammengefasst werden. Analog können Ressourcen, die in einer größeren Anzahl in der Mikroarchitektur zur Verfügung stehen, zusammengefasst werden. Bei der Ablei-

tung des Instruktionspfads wird die Anzahl der Knoten auf ein Minimum reduziert.

Im letzten Schritt wird überprüft, ob der Instruktionspfad einem der Pfade aus der Mikroarchitektur entspricht. Die Vereinigung der Instruktionspfade für alle Instruktionen muss dabei die Mikroarchitektur vollständig überdecken. Damit kann zum einen die operationale Beschreibung der Instruktionen, zum anderen die vollständige Verwendung der Mikroarchitektur validiert werden.

Eine Inkonsistenz zwischen der Mikroarchitektur und der operationalen Beschreibung einer Instruktion führt zu einem Fehler in der Spezifikation, da aus der Spezifikation weder ein funktionierender Simulator, noch ein Prozessor erzeugt werden kann. Wird durch die Validierung nicht der vollständige Datenpfad der Mikroarchitektur überdeckt, sollte dem Prozessorentwickler ein Hinweis angezeigt werden, da die Mikroarchitektur in diesem Fall unbenutzte Ressourcen oder Pfade enthält.

5.4.4 Kontrollflussanalyse

Die Kontrollflussanalyse soll die Konsistenz der in der Mikroarchitektur spezifizierten Bypass-Struktur überprüfen. Die Weiterleitung der Werte bei der Ausführung hängt von der Spezifikation der in den Instruktionen verwendeten Operanden ab. Die Inkonsistenz eines Bypasses kann gezeigt werden, indem durch die Analyse herausgestellt wird, dass dieser niemals die Bedingungen für eine Weiterleitung eines Wertes erfüllt. Die Vorgehensweise entspricht der Methode „Dead Code Elimination“ [KRS94] und wird im Folgenden für die Anwendung bei der Prozessorvalidierung beschrieben.

In der Spezifikation von *ViCE-UPSLA* werden Bypässe zwischen den Komponenten des Datenpfads in der Mikroarchitektur eingefügt. In Abbildung 5.4 sind zwei Bypässe zwischen dem Ausgabeparameter der ALU und dem linken (grün) und dem rechten (rot) Eingabeparameter eingesetzt. Damit werden die Werte für alle Instruktionen des Prozessors beim Durchlaufen dieser Anschlussstellen propagiert. Im Beispiel wird ein Instruktionssatz aus zwei Instruktionen angegeben, die für die Beschreibung der Validierungsmethode benutzt werden. Für das Beispiel wird angenommen, dass die Operanden `Dr0` und `Sr0` der beiden Instruktionen über ihre Adressierungsmethoden jeweils auf die Registerbank `Reg` zugreifen, während der Operator `Sr1` auf die Registerbank `Srg` zugreift.

Ein eingesetzter Bypass in einer Mikroarchitektur wird im Simulator durch eine Reihe von Bedingungen umgesetzt. Bei der Ausführung im Simulator oder Prozessor wird für jede Instruktion, die einen Bypass passiert, die Gültigkeit für die Weiterleitung anhand der Bedingungen überprüft. Die Weiterleitung eines Wertes wird an Parametern wie dem Alter oder dem Ziel des Wertes festgemacht und nur im Falle einer Übereinstimmung durchgeführt.

Die Kontrollflussanalyse für die Bypässe kann in *ViCE-UPSLA* angewendet werden, indem die Bypässe in Verbindung mit allen Instruktionen des Prozessors analysiert werden. Für einen gültigen Bypass kann in der Spezifikation gezeigt werden, dass

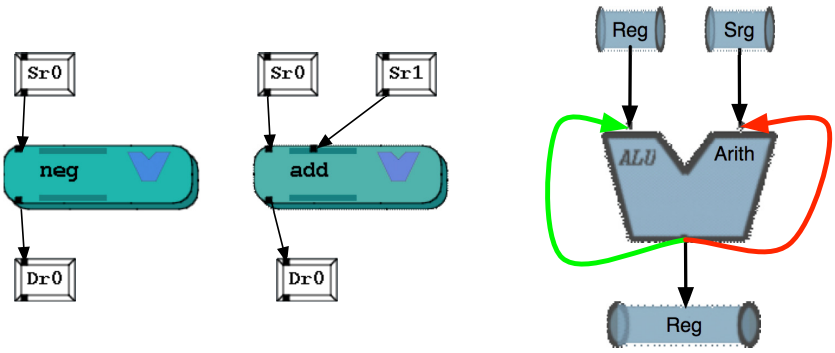


Abbildung 5.4: Zwei Instruktionen und eine Mikroarchitektur mit zwei Bypässen.

mindestens eine Instruktion den Bypass zum Weiterleiten der Werte verwenden kann.

In Beispiel aus Abbildung 5.4 ist der grün eingezeichnete Bypass gültig, da die Werte zwischen dem Ergebnis der Instruktion `neg` und dem linken Operator der `add`-Instruktion propagiert werden können. Entscheidend dafür ist die Ziel- bzw. Quelladresse des Operanden, der propagiert wird. Da die Zieladressen der Ausgabeparameter und des rechten Eingabeparameters durch die Verwendung unterschiedlicher Registerbänke nie übereinstimmen können, kann der Wert zwischen diesen beiden Punkten in der Mikroarchitektur zu keiner Zeit propagiert werden. Die Spezifikation des rot eingezeichneten Bypasses ist damit inkonsistent.

Es ist nicht ausreichend, nur die Struktur der Mikroarchitektur zu betrachten. Bei der Kontrollflussanalyse der Bypässe müssen der vollständige Instruktionssatz und die Adressierungsmethoden der einzelnen Instruktionen betrachtet werden, da die Konsistenz der Bypässe bereits dort auf die gleiche Weise verletzt werden kann. Als Konsequenz aus der Validierung muss der Prozessorentwickler Hinweise auf die unbenutzten Bypässe erhalten.

5.4.5 Validierung der Mehrdeutigkeit

Die unnötig mehrfach spezifizierten Komponenten in einer Prozessorspezifikation vergrößern eine Spezifikation und erhöhen künstlich ihre Komplexität. Z. B. können in einer Prozessorspezifikation mehrere Instruktionsformate mit der identischen Signatur spezifiziert werden, womit die Spezifikation keinen Mehrwert aus der Sicht des Simulators oder Prozessors erfährt. Mehrfach spezifizierte Instruktionsformate können jedoch vom Benutzer gewollt sein, um die Komponenten der Spezifikation zusätzlich zu sortieren. Mehrdeutigkeit in anderen Sprachkonstrukten kann jedoch fehlerhaftes Verhalten im Simulator oder Prozessor auslösen. Außerdem kann Mehrdeutigkeit in

der Prozessorspezifikation zu Problemen bei der Entwicklung von Werkzeugen für den Prozessor führen.

Ein Beispiel für mehrdeutig beschriebene Instruktionen wurde bereits in Abschnitt 5.2.4 beschrieben. Bei der Entwicklung von Werkzeugen für den Prozessor führen redundante Instruktionen zu unerwünschten Mehrdeutigkeiten. Außerdem wird der Instruktionssatz der Spezifikation unnötig vergrößert. Die redundante Spezifikation von Sprachkonstrukten kann auch auf der strukturellen Ebene entstehen. So ist z. B. eine redundante Spezifikation von architektonischen Registern in der Spezifikation nicht sinnvoll und erhöht unnötig die Komplexität des Registersatzes. Für die Validierung solcher Konstrukte muss die Konsistenzprüfung Sprachkonstrukte auf Duplikate miteinander vergleichen.

Die Benutzung des Prozessors wird, wie bereits in Abschnitt 5.2.4 beschrieben, durch mehrdeutige Kodierung der Instruktionen beeinträchtigt werden. Im Fehlermodell wurden die Konsistenzbedingungen für die Eindeutigkeit der Kodierung bereits beschrieben. Dabei wurde herausgestellt, dass nicht nur identische Operationscodes zu fehlerhaftem Verhalten führen. Die Mehrdeutigkeit kann durch Befehlsworte mit unterschiedlich langen Operationscodes entstehen.

Die Analyse kann nach dem Prinzip der Huffman-Kodierung [Sed92] durchgeführt werden, wonach sich jeder Operationscode in seinen Anfangszeichen von anderen Operationscodes unterscheiden muss.

Da in *ViCE-UPSLA* die Dekodierung von Instruktionen anhand der Bezeichner und nicht durch den Operationscode durchgeführt wird, kann aus der Prozessorspezifikation ein Simulator erzeugt und fehlerfrei benutzt werden. Diese Eigenschaft kann bei der Instruktionssatzerweiterung in den frühen Phasen der Entwicklung ausgenutzt werden und ermöglicht experimentelle Entwicklung von neuen Instruktionen. Die Fehlerbehandlung in *ViCE-UPSLA* gibt bei Entdeckung von mehrdeutigen Kodierungen eine Warnung aus. Die Entwicklung eines realen Prozessors mit einer bestehenden Inkonsistenz sollte jedoch nicht vorangetrieben werden.

5.4.6 Datentypen

Die operationale Beschreibung der Instruktionen wird durch einen Datenflussgraphen aus Operatoren und interpretierten Operanden beschrieben. Die interpretierten Operanden werden als Ein- oder Ausgabeparameter der Operatoren verwendet, die selbst wiederum aus den Adressierungsmethoden mit Hilfe der Befehlsoperanden berechnet werden. Die Verknüpfungen der Strukturen in *ViCE-UPSLA* wurden bei der Beschreibung der Sprache in Abschnitten 4.3.4 und 4.3.5 bereits vorgestellt.

Aus der Verbindung von interpretierten Operanden bis zu den Registern eines Prozessors können aus der Spezifikation die Datentypen für die interpretierten Operanden ermittelt werden. Auf dieser Grundlage wird im Folgenden, anhand von Abbildung 5.5, eine Validierungsmethode für die Validierung der Datentypen für die Operatoren

des Prozessors beschrieben.

In Abbildung 5.5 sind die Instruktionen X und Y mit den Operatoren `Operator X` und `Operator Y` und eine Instruktion Z mit der Kombination der Operatoren `X` und `Y` dargestellt. Für die Parameter der Instruktionen X und Y können die Datentypen aus den interpretierten Operanden `A,B,C` usw. ermittelt werden. Damit wird der Datentyp für den linken Parameter von `Operation X` durch den Operanden `A`, für den rechten Parameter durch den Operanden `B` usw. festgelegt.

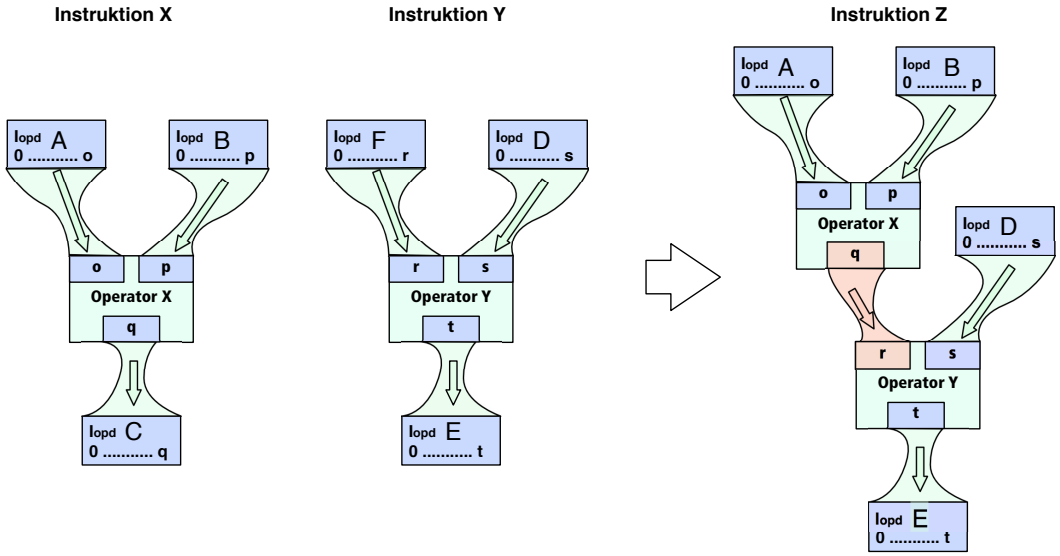


Abbildung 5.5: Operationale Beschreibung von Instruktionen X und Y und die kombinierte Instruktion Z.

Die Validierung der Datentypen in *ViCE-UPSLA* kann durchgeführt werden, indem für die gleiche Operatoren, die in verschiedenen Instruktionen verwendet werden, die ermittelten Datentypen validiert werden. Dabei muss ein Operator immer dieselben Datentypen für die Parameter besitzen. Für komplexe operationale Beschreibungen mit z. B. verketteten Operationen, wie in Abbildung 5.5 gezeigt, kann die Validierung aus den ermittelten Datentypen fortgesetzt werden, indem bereits ermittelte Datentypen übernommen werden. Im Beispiel muss nach der Analyse der Instruktion Z der Ausgabeparameter aus dem `Operator X` gleich dem linken Eingabeparameter aus dem `Operator Y` sein.

Die statische Analyse mit diesem Ansatz kann nur dann vollständig durchgeführt werden, wenn für jeden Operator die Datentypen für die Parameter ermittelt werden können. Hierfür müssen alle Instruktionen untersucht und die Ergebnisse zusammen betrachtet werden. Eine Alternative hierfür ist die Erweiterung der Spezifikation von

ViCE-UPSLA durch ein Datentypsystem, in dem für die Ein- und Ausgabeparameter der Operatoren die entsprechenden Datentypen definiert werden. Durch das zusätzliche Wissen kann die korrekte Verwendung der Operatoren in der Verhaltensbeschreibung statisch validiert werden. Bei Verletzung der Datentypen muss der Benutzer durch eine Warnung auf die Inkonsistenz hingewiesen werden.

5.4.7 Anwendungsszenarios der statischen Methoden

Die Spezifikation eines Prozessors wird durch den Prozessorentwickler in mehreren Schritten im Entwicklungsprozess erstellt. Für die Anwendung der Methoden muss die Spezifikation der Komponenten und ihrer Gegenstellen für die Validierung in der Spezifikation vorhanden sein. Alle statischen Validierungsmethoden können auf einer vollständigen Spezifikation des Prozessors durchgeführt werden. Einige der Methoden können auf einer unvollständigen Spezifikation des Prozessors durchgeführt werden. Da es sinnvoll ist, die Inkonsistenzen in der Spezifikation frühestmöglich zu erkennen, werden in diesem Abschnitt die Zeitpunkte und Szenarios für den möglichen Einsatz der Validierungsmethoden definiert und den zuvor beschriebenen Methoden zugeordnet.

Der Zeitpunkt, zu dem eine Konsistenzbedingung überprüft werden kann, wird an dem Fortschritt der Spezifikation eines Prozessors im Entwurfsprozess gemessen. In der Arbeit von C.Schmidt zur Generierung von Strukturediteren [Sch06] wird die Häufigkeit der Überprüfung beschrieben. Dabei wird zwischen der Konsistenzprüfung vor und nach einer Änderung und einer abschließenden Überprüfung einer fertigen Spezifikation unterschieden. Diese Einteilung kann für den Prozessorentwurf teilweise verwendet werden. Die Unterscheidung zwischen der Validierung vor und nach einer Änderung wird bei der Validierung im Prozessorentwurf auf die kontinuierliche Validierung nach einer Änderung reduziert. Zusätzlich wird ein Szenario für die Überprüfung beschrieben, bei dem eine statische Überprüfung durch den Prozessorentwickler angestoßen wird. Damit wird in dieser Arbeit zwischen folgenden drei Zeitpunkten unterschieden:

- Kontinuierlich – die Konsistenz der Eigenschaften zwischen den Sprachkonstrukten wird nach jeder Änderung der Sprachkonstrukte überprüft. Die Prozessorspezifikation muss dabei nicht vollständig sein.

- Angestoßen – die Konsistenzprüfung wird durch den Benutzer angestoßen, um eine bestimmte Analyse für die Eigenschaften einer Gruppe von Konstrukten durchzuführen. Die Prozessorspezifikation muss nicht vollständig sein, die Spezifikation der Prozessorkonstrukte in der Gruppe für die Konsistenzprüfung muss abgeschlossen sein.

- Abschließend – die Konsistenzprüfung wird am Ende des Entwurfs durchgeführt, bevor die Spezifikation in eine weitere Entwicklungsstufe überführt wird, z. B.

wenn der Prozessorsimulator generiert wird. Die Prozessorspezifikation muss vollständig beschrieben sein.

Die kontinuierliche Überprüfung von Konsistenzbedingungen ist eine sehr effektive Art, den Prozessorentwickler zu unterstützen, da die Inkonsistenzen sofort entdeckt werden. Dieses Szenario kann für die Methoden angewendet werden, bei denen aus der Spezifikation entschieden werden kann, ob das Sprachkonstrukt vollständig beschrieben ist. Wird z. B. ein Bezeichner oder Operationscode für eine Instruktion festgelegt, ist die Spezifikation dieser Eigenschaft vollständig abgeschlossen und kann auf Mehrdeutigkeit zu anderen bereits spezifizierten Instruktionen validiert werden. Analog dazu kann die Mengenkodierung oder die Validierung der Datentypen kontinuierlich durchgeführt werden.

Als Gegenbeispiel kann Validierung für die operationale Beschreibung einer Instruktion betrachtet werden. Dabei besteht ein Entscheidungsproblem, da die Spezifikation in mehreren Bearbeitungsschritten erzeugt wird und während der Bearbeitung inkonsistente Zustände besitzt, bei denen z. B. die Verknüpfungen zwischen den Operationen nicht vollständig oder die Ressourcen gar nicht angegeben sind.

Durch das Entscheidungsproblem in der Spezifikation können einige Methoden erst auf einer vollständigen Spezifikation des Prozessors, z. B. vor der Generierung des Simulators, durchgeführt werden. Für die Möglichkeit der Validierung einer unvollständigen Spezifikation wird das Szenario der angestoßenen Validierung eingeführt. Die angestoßene Validierung wird benötigt, um dem Prozessorentwickler die Möglichkeit zu geben, Teile der Spezifikation zu validieren, deren Sprachkonstrukte aus seiner Sicht vollständig angegeben sind. Die angestoßene Validierung kann z. B. für die Validierung der Grapheinbettung zwischen der operationalen Beschreibung und der Mikroarchitektur durchgeführt werden. Bei diesem Szenario kann der Prozessorentwickler die Validierung nach jeder neu erzeugten abstrakten Instruktion durchführen, die als Schablone für die konkreten Instruktionen verwendet wird, bevor die Spezifikation der konkreten Instruktionen durchgeführt wird. So wie die Grapheinbettung können weitere Methoden wie z.B. die Kontrollflussanalyse oder der Verwendungsnachweises angestoßen werden. Die angestoßene Überprüfung ist besonders wichtig für Sprachkonstrukte, auf denen aufbauend weitere Komponenten der Prozessorspezifikation beschrieben werden.

5.5 Dynamische Validierung

Die dynamische Validierung wird im Prozessorentwurf nahezu immer durchgeführt, unabhängig davon, ob ein neuer Prozessor entwickelt oder eine Instruktionssatzerweiterung durchgeführt wird. In der Regel wird der Entwurf überprüft, indem für den Prozessor entworfene Testprogramme oder Test-Suites auf dem Prozessor oder im Simulator ausgeführt werden. Die Testprogramme werden von den Prozessorentwicklern

oder Testentwicklern entworfen, um die Funktionalität, Korrektheit oder Geschwindigkeit des entworfenen Prozessors zu überprüfen. Die Validierung wird durchgeführt, indem bei der Ausführung der Testprogramme gesammelten Ergebnisse mit den durch die Entwickler erwarteten Ergebnissen verglichen werden. Die Qualität dieser Validierung hängt stark von den verwendeten Testprogrammen ab. Eine unpassende Abstimmung der Test-Suite kann z. B. dazu führen, dass manche Teile der Spezifikation nicht validiert werden, wenn diese nicht im Fokus des Prozessorentwicklers liegen.

Ziel dieses Abschnittes ist die Beschreibung dynamischer Validierungsmethoden, mit denen der Prozessorentwickler bei der dynamischen Validierung des Prozessors unterstützt wird und die Qualität der Validierung gesteigert wird. Die dynamischen Validierungsmethoden sollen dabei beschreiben, wie zu einer konkreten Prozessorspezifikation in *ViCE-UPSLA* Testprogramme automatisch erzeugt werden können und in der Werkzeugkette mit *ViCE-UPSLA* umgesetzt werden. Zum einen kann durch die Sprache *ViCE-UPSLA* eine systematische Vorgehensweise für die Betrachtung der Sprachkonstrukte einer Prozessorspezifikation beschrieben werden. Zum anderen werden aus der Spezifikation für den Entwurf eines Prozessors optimierte Testprogramme abgeleitet. Die Entwicklung von Generatoren für die automatische Erzeugung der Testprogramme wird durch die in *ViCE-UPSLA* verwendeten Sprachkonstrukte und die Struktur der Sprache ermöglicht. Durch die Integration der Generatoren in das Werkzeugsystem kann die dynamische Validierung teilweise oder vollständig automatisiert werden.

Im Folgenden werden die dynamischen Validierungsmethoden beschrieben, welche aus der Analyse der Prozessoreigenschaften aus dem Fehlermodell aus Abschnitt 5.2 und der Struktur der Sprache *ViCE-UPSLA* abgeleitet werden. Dazu werden in Abschnitt 5.5.1 die Beschreibung für die Vorgehensweise und die Taxonomie bei der Entwicklung und der Beschreibung von dynamischen Validierungsmethoden vorgestellt. Die Umsetzung der Methoden im Werkzeugsystem wird in Kapitel 6 behandelt.

5.5.1 Konzepte und Methoden

Das Prinzip der dynamischen Validierung wurde in Abschnitt 2.5.2 beschrieben und zeichnet sich dadurch aus, dass aus dem Verhalten eines Systems Schlussfolgerungen über die Inkonsistenzen in der Spezifikation gemacht werden. Wird die dynamische Validierung im Prozessorentwurf mit *ViCE-UPSLA* durchgeführt, so wird eine Prozessorspezifikation in *ViCE-UPSLA* validiert. Dazu werden die Simulationsergebnisse aus der Ausführung eines Testprogramms im Simulator für Rückschlüsse auf Inkonsistenzen in der Spezifikation des Prozessors verwendet.

Grundsätzlich können Inkonsistenzen in der Spezifikation durch statische Analyse überprüft werden. In einer *ViCE-UPSLA* Spezifikation werden für die Operatoren in der operationalen Beschreibung oder in den Adressierungsmethoden der Instruktionen Funktionen in der Sprache C angegeben. Der so eingebrachte Fremdcode verhindert

die statische Validierung einiger Eigenschaften des Prozessors. Die fehlerhafte Implementierung der Funktionen oder die inkonsistente Benutzung der Komponenten in der Spezifikation kann somit nicht statisch überprüft werden. Wie bereits im Fehlermodell in Abschnitt 5.2 beschrieben, charakterisieren die dynamischen Konflikte Quellen für Inkonsistenzen in der Spezifikation, deren Anwesenheit erst aus der Analyse der Simulationsergebnisse gezeigt werden kann. Weiterhin können die dynamischen Konflikte, wie bereits auch die statischen Konflikte, in reguläre und irreguläre Konflikte unterteilt werden. Für die regulären Konflikte werden die Testfälle aus der Spezifikation automatisch generiert. Bei den irregulären Konflikten wird das zusätzliche Wissen des Prozessorentwicklers benötigt.

Die natürliche Vorgehensweise beim Testen, bei dem ein System durch den Vergleich der gesammelten Ergebnissen aus der Simulation mit den erwarteten Ergebnissen validiert wird, wird auch beim modellbasierten Testen verwendet. In Abschnitt 2.5.3 wurden verschiedene Szenarios des modellbasierten Testens vorgestellt. Die Begriffe des modellbasierten Testens können, wie bereits beschrieben, für die Domäne des Prozessorentwurfs übertragen werden. Für die dynamische Validierung mit *ViCE-UPSLA* kann das grundlegende Konzept des modellbasierten Testens als eine Strategie übernommen werden. Zusätzlich werden zwei weitere Strategien für die dynamische Validierung vorgestellt, welche bei der Analyse der Domäne für den Prozessorentwurf erarbeitet wurden.

Anhand von Abbildung 5.6 werden zunächst die Komponenten der dynamischen Validierung vorgestellt. Diese Komponenten werden bei der Beschreibung der dynamischen Validierungsmethoden in weiteren Abschnitten verwendet. Die Komponenten beschreiben die Spezifikation und Automatisierung für die Generierung der Testfälle. Anschließend werden drei Strategien bei der Validierung vorgestellt. Die Strategien beschreiben die Vorgehensweise, wie die Validierung durchgeführt wird und wie die Ergebnisse aus den dynamischen Validierungsmethoden verwendet werden.

Für die Beschreibung der Validierungsmethoden mit *ViCE-UPSLA* werden die Komponenten zur Generierung der Testfälle und der Simulatoren in Abbildungen 5.6 dargestellt. An erster Stelle steht der Prozessorentwickler, der aus den Anforderungen an den Prozessor, bestehend aus einer informellen Beschreibung oder einem Datenblatt, eine formale Spezifikation des Prozessors in *ViCE-UPSLA* erzeugt. Aus der Prozessorspezifikation kann mit einem Generator ein Prozessorsimulator erzeugt werden. Dieser wird bei der dynamischen Validierung verwendet. Wie bereits beschrieben, sollen die Inkonsistenzen in der Spezifikation des Prozessors aufgedeckt werden. Damit entspricht die Prozessorspezifikation in *ViCE-UPSLA* aus Abbildung 5.6 dem Systemmodell. Der Prozessorsimulator entspricht dem SUT (System unter Test) des modellbasierten Testens und wird aus der Prozessorspezifikation generiert.

Unabhängig von den verwendeten Strategien, welche in diesem Abschnitt später ausführlich beschrieben werden, werden bei der dynamischen Validierung neben dem

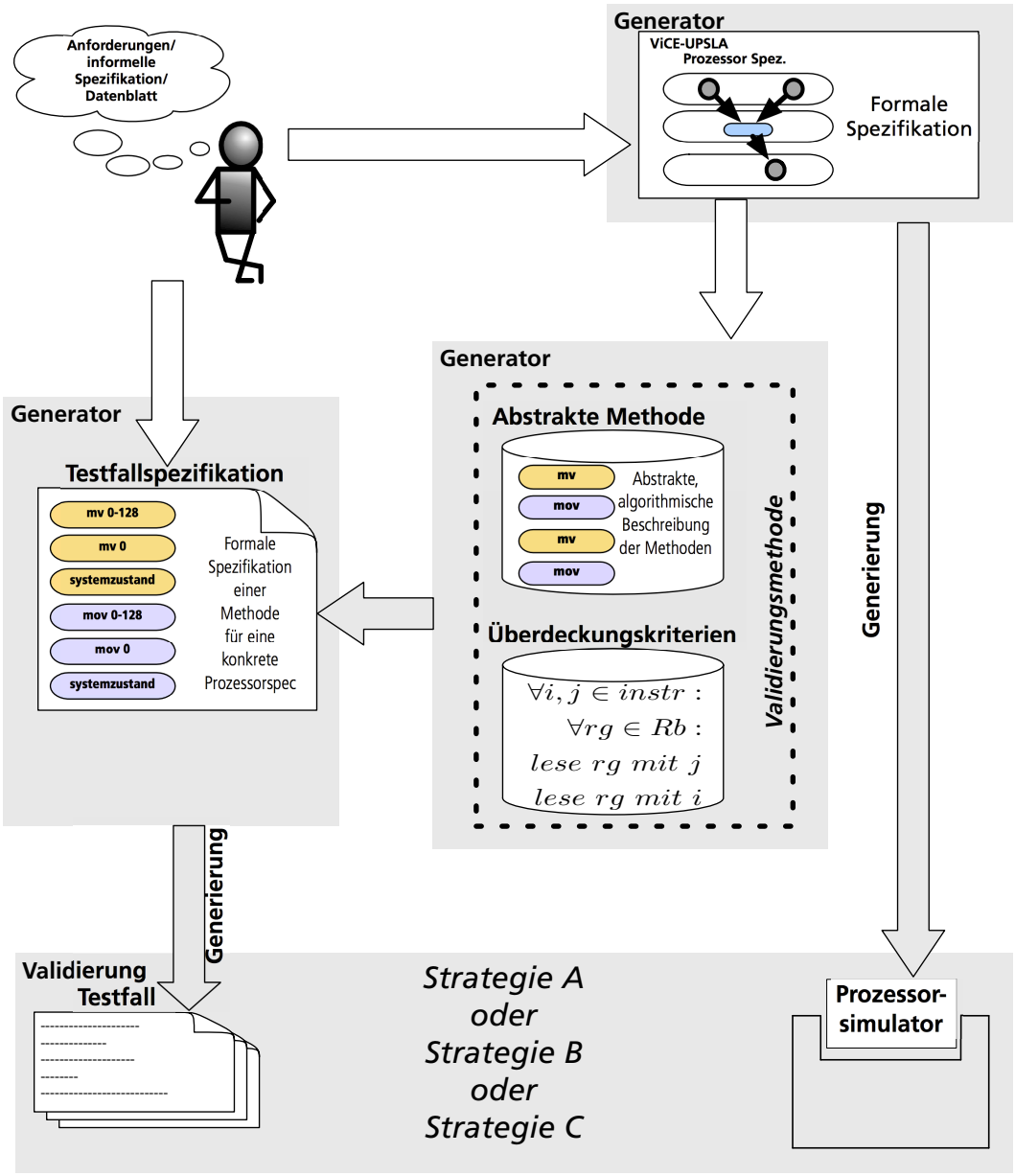


Abbildung 5.6: Übersicht der dynamischen Validierung.

Simulator die Testfälle benötigt. In der Regel enthält ein Testfall im Prozessorentwurf eine große Anzahl von Instruktionen und Befehlen. Die Ausführung eines Testfalls im Simulator soll Systemzustände erzeugen, aus denen die Inkonsistenzen in der Spezifikation aufgedeckt werden. Für die Beschreibung der Testfälle im Werkzeugsystem mit *ViCE-UPSLA* wird eine Testfallspezifikation verwendet, aus der die Testfälle automatisch generiert werden. Für diesen Zweck wird die Testfallspezifikation in einer visuellen Form für den Prozessorentwickler zugänglich gemacht. Die Spezifikationssprache und die Eigenschaften für die Beschreibung einer Testfallspezifikation werden in Abschnitt 5.7 ausführlich erläutert. Wie in den verwandten Arbeiten in Abschnitt 3.2 beschrieben, bieten einige Werkzeuge aus der Domäne des MBT vergleichbare Möglichkeiten.

Aus der Sicht der Validierungsmethoden beschreibt eine Testfallspezifikation die Konstruktion eines Testfalls einer Validierungsmethode für eine konkrete Prozessorspezifikation. Damit wird eine formale Spezifikation eines Testfalls für die Validierung einer Eigenschaft für einen bestimmten Prozessor angegeben. Die Testfallspezifikation beschreibt konkret welche Instruktionsfolgen bei der Validierung ausgeführt werden und welche Instruktionen in den Instruktionsfolgen eingesetzt werden. Außerdem werden in einer Testfallspezifikation konkrete Werte für die Operanden der Instruktionen angegeben. Eine Testfallspezifikation wird durch einen Generator aus einer Prozessorspezifikation erzeugt und kann vollständig oder teilweise generiert werden. Wird eine Testfallspezifikation nicht vollständig generiert, wie es für manche Methoden der Fall ist, werden zusätzliche Eingaben durch den Prozessorentwickler benötigt. Die Testfallspezifikation ist mit der Prozessorspezifikation verknüpft. Diese Integration ermöglicht eine automatische Anpassung bereits beschriebener Testfallspezifikationen an die Änderungen der Prozessorspezifikation.

Wie bereits beschrieben, wird die Testfallspezifikation teilweise oder vollständig aus der Prozessorspezifikation generiert. Die Generierung wird durch die Struktur der Sprache *ViCE-UPSLA* ermöglicht. Für die Erzeugung der Testfallspezifikationen werden die Validierungsmethoden durch Generatoren im Werkzeugsystem integriert. Eine Validierungsmethode wird (vgl. Abbildung 5.6) durch eine *abstrakte Methode* und die *Überdeckungskriterien* ausgedrückt. Zunächst wird der Begriff der Überdeckungskriterien für die dynamische Validierung mit *ViCE-UPSLA* erläutert. Anschließend wird auf die abstrakte Methode eingegangen.

In Abschnitt 5.2 zum Fehlermodell wurden die Konsistenzbedingungen für die Prozessorkonstrukte in einer Spezifikation angegeben. Die Überdeckungskriterien werden aus dem Fehlermodell abgeleitet und beschreiben, für welche Prozessorkonstrukte oder Wertebereiche aus der Spezifikation eines Prozessors die Konsistenzbedingungen überprüft werden müssen. Z. B. beschreibt die Überdeckungsmenge für die Erreichbarkeit von Registern, welche Register der Spezifikation validiert werden müssen. Die Überdeckungskriterien geben dazu an, dass alle Register des Registersatzes erreichbar und damit validiert werden müssen. Umgesetzt in einem Generator beschreiben die Über-

deckungskriterien, wie in einer gegebenen Prozessorspezifikation die Sprachkonstrukte für die Validierung ermittelt werden. Durch die Überdeckung einer Prozessorspezifikation für eine Validierungsmethode wird für eine Testfallspezifikation die Überdeckungsmenge der Elemente für die Validierung definiert.

Die Überdeckungskriterien beschreiben, wie die Sprachkonstrukte einer Prozessorspezifikation für die Konsistenzprüfung einer Validierungsmethode ermittelt werden. Das Ergebnis daraus wird in der Testfallspezifikation als die Überdeckungsmenge angegeben.

Als nächstes wird bei der Beschreibung einer Validierungsmethode die Vorgehensweise für die Überprüfung einer Eigenschaft benötigt. Die Vorgehensweise kann algorithmisch ausgedrückt und durch eine abstrakte Methode angegeben werden. Die abstrakten Methoden wurden in dieser Arbeit aus dem Domänenwissen des Prozessorentwurfs entwickelt und beschreiben Teilaufgaben, mit denen eine Eigenschaft für ein Prozessorkonstrukt überprüft wird. Z. B. beschreibt die Konsistenzbedingung für die Erreichbarkeit der Register, dass ein Register genau dann erreichbar ist, wenn mindestens eine Instruktion des Prozessors dieses Register schreiben und lesen kann. Die abstrakte Methode für die Konsistenzbedingung wird durch die Instruktionsschritte beschrieben, die bei der Validierung eines Registers mit einem bestimmten Wert ausgeführt werden:

- Systemzustand initialisieren
- Instruktion X ausführen
- Systemzustand aufzeichnen
- ...

Die Konsistenz der Erreichbarkeit eines Registers ist dann sichergestellt, wenn zwischen dem initialen und dem aufgezeichneten Systemzustand eine Änderung des gespeicherten Wertes im Register festgestellt werden kann. Damit kann eine abstrakte Validierungsmethode als eine algorithmische Beschreibung definiert werden, welche schrittweise die Anweisungen und das Auslesen oder Initialisieren der Systemzustände angibt.

Eine abstrakte Validierungsmethode beschreibt eine Vorgehensweise, ausgedrückt durch Instruktionsfolgen, mit der die Gültigkeit einer Eigenschaft für eine Komponente des Prozessors überprüft wird.

Die Inkonsistenzen in einer Prozessorspezifikation werden durch ungültige Relationen zwischen den Komponenten ausgelöst. Damit werden für die Überprüfung einer

Konsistenzbedingung immer zwei in Relation zueinander stehende Prozessorkonstrukte betrachtet. Dadurch, dass eine abstrakte Methode eine Inkonsistenz immer zwischen zwei Komponenten des Prozessors aufzeigt, können einige *abstrakte Methoden* für die Validierung verschiedener Prozessorkonstrukte verwendet werden. So besitzen z. B. die Validierung der Erreichbarkeit von Registern und die Validierung des Adressraums einer konkreten Adressierungsmethode identische abstrakte Methoden. Die Validierungsmethoden unterscheiden sich in den Überdeckungskriterien, also in der Menge der zur Prüfung festgelegten Konstrukte oder Wertebereiche. Umgekehrt können bei gleich bleibender Überdeckung und unterschiedlichen abstrakten Methoden unterschiedliche Eigenschaften der Konstrukte validiert werden.

Eine *Validierungsmethode* wird, wie bereits beschrieben durch, die Kombination aus einer *abstrakten Methode* und den Überdeckungskriterien vollständig angegeben. Umgesetzt in einem Generator, wie in Abbildung 5.6 dargestellt, wird eine Prozessorspezifikation in *ViCE-UPSLA* anhand der Validierungsmethode analysiert. Das Ergebnis ist eine generierte Testfallspezifikation.

Die automatische Generierung von Testfallspezifikationen für die Validierung einiger Sprachkonstrukte, wie Adressierungsmethoden oder Operationen der operationalen Beschreibung, kann durch die vielfältige Ausprägung der Instruktionssätze in verschiedenen Prozessoren nicht durchgeführt werden. Z. B. beschreibt die abstrakte Methode für die Validierung der Adressierungsmethoden, dass `Move_Immediate` Instruktionen in den Instruktionsfolgen verwendet werden sollen. Wenn der Instruktionssatz des Prozessors die benötigten Instruktionen nicht enthält, kann die Testfallspezifikation nicht vollständig generiert werden. In diesem Fall muss der Prozessorentwickler die Testfallspezifikation ergänzen, indem er alternative Instruktionen für die Validierung bestimmt.

Die Verwendung der Sprache *ViCE-UPSLA* und das Werkzeugsystem ermöglichen als Abhilfe die Erzeugung von *Testrahmen-Instruktionen* [TKPD11]. Testrahmen-Instruktionen werden automatisch aus der Spezifikation des Prozessors generiert, ohne den spezifizierten Instruktionssatz und den Simulator des Prozessors zu beeinflussen. Die Testrahmen-Instruktionen werden so konstruiert, dass diese anstelle der Instruktionen des Prozessors in den Instruktionsfolgen bei der Validierung eingesetzt werden können. Ein Beispiel für Testrahmen-Instruktionen sind z. B. die Adressierungsinstruktionen. Hierfür wird für jede Adressierungsmethode des Prozessors eine Instruktion automatisch erzeugt, die lediglich einen unmittelbaren Wert aus einem Befehlswort mit der Adressierungsmethode in ein Register schreibt.

Für die dynamischen Validierungsmethoden wurden bisher die Komponenten für die Erzeugung des Simulators und der Testfälle für die Validierung angegeben. Der Validierungsprozess wird durchgeführt, indem die erzeugten Testfälle in einem Simulator ausgeführt werden und Daten für die Auswertung der Validierung erzeugen. Die Vorgehensweise für die Durchführung der Validierung kann sich auf Grund der Konstruktion

der Testfälle unterscheiden. Wie bereits am Anfang des Abschnittes beschrieben, werden hierfür drei Strategien vorgestellt.

Die Strategie *A* beschreibt die aus dem modellbasierten Testen bekannte Vorgehensweise und wird in Abbildung 5.7 dargestellt. Diese Strategie kann für Validierungsmethoden angewendet werden, deren Testfälle die Eingabe für den Simulator und die erwartete Ausgabe des Simulators enthalten. Bei der Validierung werden die erwarteten Systemzustände mit den bei der Simulation aufgezeichneten Systemzuständen verglichen.

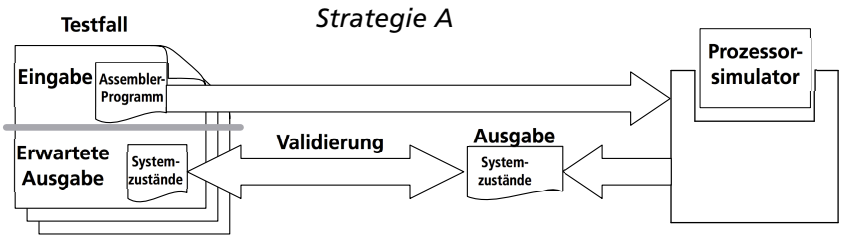


Abbildung 5.7: Strategie *A* mit der erwarteten Ausgabe des Simulators.

Die Strategie *B* wurde in dieser Arbeit entwickelt und wird durch die Eigenschaften einiger Sprachkonstrukte aus dem Prozessorentwurf ermöglicht. Abbildung 5.8 zeigt die Vorgehensweise bei der Validierung. In einem Testfall wird anstelle der erwarteten Ausgabe für die Validierung eine Referenzeingabe angegeben. Die Referenzeingabe zeichnet sich dadurch aus, dass die Konfliktquellen durch Konstruktion der Eingabe z. B. auf Kosten der Geschwindigkeit eliminiert werden. Damit wird ein Testfall durch zwei redundanten Eingaben für den Simulator beschrieben. Für die Validierung werden beide Eingaben im Simulator ausgeführt. Bei der Auswertung wird die reguläre Ausgabe des Systems mit der Referenzausgabe verglichen.

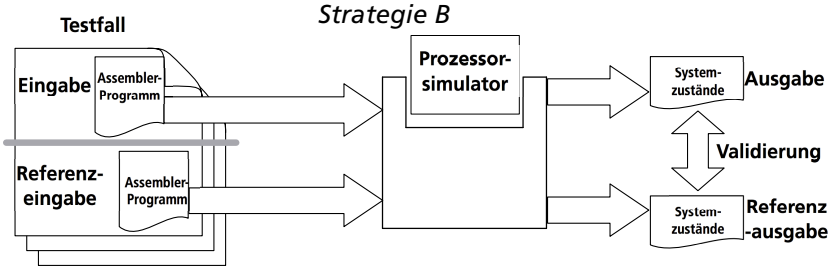


Abbildung 5.8: Strategie *B* mit Referenzeingabe für den Simulator.

Die Strategie *C* wurde ebenfalls in dieser Arbeit entwickelt und wird durch die Er-

zeugung verschiedener Simulatoren aus einer Spezifikation ermöglicht. Aus einer Prozessorspezifikation können z. B. ein Instruktionssatz- und Mikroarchitektursimulator erzeugt werden. In einem Instruktionssatzsimulator werden die Effekte der Mikroarchitektur nicht berücksichtigt. Wie in Abbildung 5.9 dargestellt, wird ein Testfall bei dieser Strategie durch eine Eingabe beschrieben. Die erzeugte Ausgabe aus zwei verschiedenen Simulatoren können miteinander verglichen werden, um auf Inkonsistenzen in der Spezifikation zu schließen.

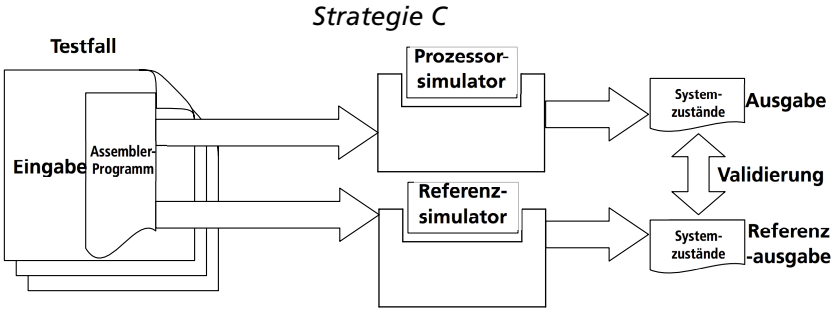


Abbildung 5.9: Strategie C mit einem Referenzsimulator.

Im Folgenden werden zu den im Fehlermodell beschriebenen Inkonsistenzen die abstrakten Methoden und die Überdeckungskriterien angegeben und damit die Validierungsmethoden beschrieben. Für die Validierungsmethoden werden zusätzlich die möglichen Strategien bei der Validierung angegeben. Außerdem wird untersucht, inwieweit das zusätzliche Wissen benötigt wird.

5.5.2 Registererreichbarkeit

In einer Prozessorspezifikation wird der Registersatz durch Registerbänke und den darin enthaltenen Registern spezifiziert. Die Erreichbarkeit der Registerbänke kann, wie bereits in Abschnitt 5.4.2 durch den Verwendungsnachweis gezeigt, statisch validiert werden. In der Prozessorspezifikation beschreiben die Adressierungsmethoden die Verknüpfung zwischen den interpretierten Operanden der Instruktionen und den Registern. Durch die verwendeten Funktionen in den Adressierungsmethoden kann die Erreichbarkeit der einzelnen Register nicht durch statische Analyse überprüft werden. Somit muss die Erreichbarkeit von Registern dynamisch validiert werden.

Für die Beschreibung der dynamischen Validierungsmethode für die Registererreichbarkeit werden im Folgenden, wie bereits in den Konzepten zur dynamischen Validierung in Abschnitt 5.5.1 beschrieben, die Überdeckungskriterien und die abstrakte Methode beschrieben. Abschließend wird die Strategie für die Durchführung der Validierung beschrieben.

Für die Beschreibung der Überdeckungskriterien werden Regeln aufgestellt, nach welchen die Menge der Register für die Validierung ermittelt wird. In *ViCE-UPSLA* wird das Konzept der architektonischen Register verwendet, wodurch eine triviale Überdeckung aller Register für die Validierung nicht aussagekräftig ist. Bei der Beschreibung der Überdeckungskriterien wird berücksichtigt, dass architektonische Register für die Strukturierung eines Registersatzes benutzt werden und nicht zwangsweise auf einem direkten Weg durch eine Adressierungsmethode erreichbar sein müssen. Für eine systematische Erzeugung der Überdeckungsmenge müssen die Regeln für beliebige Strukturen des Registersatzes anwendbar sein.

Die Register des Prozessors werden geschrieben und gelesen. Bei der Validierung der Registererreichbarkeit müssen beide Benutzungsrichtungen separat betrachtet werden. Die Überdeckungskriterien sind in beiden Fällen gleich, die Überdeckungsmenge muss jedoch aufgrund der Möglichkeit, unterschiedliche Adressierungsmethoden zu benutzen, individuell ermittelt werden. Im Folgenden wird die Validierung für die Schreibrichtung der Register erläutert.

Überdeckungskriterium In Abschnitt 5.2 zum Fehlermodell wurde die Bedingung aufgestellt, dass die Erreichbarkeit für alle Register des Registersatzes gezeigt werden muss. Die architektonischen Register werden dazu eingesetzt, um alternative Sichten auf die physikalischen Register zu beschreiben. Dabei können Registerbänke zur Strukturierung verwendet werden. Aus diesem Grund kann die einfache Annahme nicht für alle Konstruktionen eines Registersatzes erfüllt und sinnvoll sein. Für die Beschreibung der Überdeckungskriterien wird in diesem Abschnitt zunächst von der Menge aller Register ausgegangen. Diese wird anschließend auf eine sinnvolle und notwendige Überdeckungsmenge, bedingt durch die Konstruktion des Registersatzes, reduziert.

Für die Erklärung der strukturellen Abhängigkeiten zwischen architektonischen und physikalischen Registern wird Abbildung 5.10 verwendet. Abbildung 5.10 zeigt schematisch eine mögliche Registerkonfiguration für die Konstrukte der architektonischen Register in *ViCE-UPSLA*.

Die architektonischen Register, die zur Strukturierung als Zwischenbausteine angelegt und in übergeordneten Registerbänken zusammengefasst werden, besitzen in der Regel keine Adressierungsmethoden, in denen sie verwendet werden. Diese Register können von der Validierung ausgeschlossen werden. In dem Beispiel in Abbildung 5.10 können die Registerbänke A und B nur zur Strukturierung verwendet werden. Damit ist die Überdeckungsmenge des Registersatzes in diesem Beispiel auf die Registerbänke R, D und C reduziert.

Die beschriebene Festlegung der Überdeckungsmenge wird aus der Struktur einer Prozessorspezifikation in *ViCE-UPSLA* automatisch abgeleitet und für die Validierung in der Werkzeugkette in einem Generator integriert.

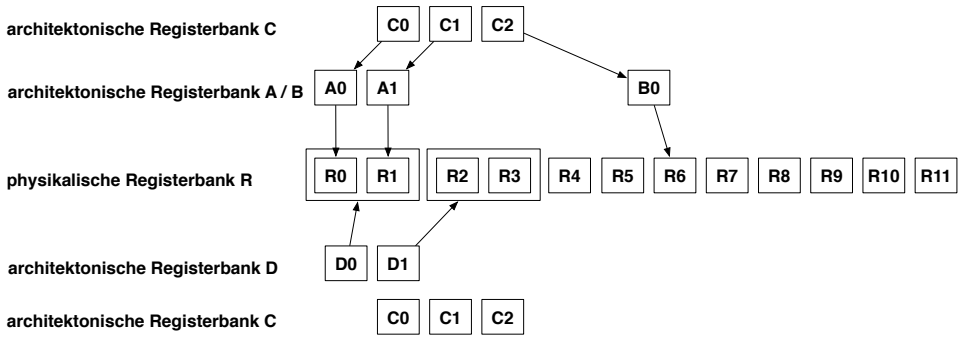


Abbildung 5.10: Registersatz-Struktur. Die R-Register repräsentieren die physikalischen Register und die A/B/C/D-Register die architektonische Register. Das Aliasing wird durch die Pfeile dargestellt.

Abstrakte Methode Bei der Validierung der Schreibrichtung muss für ein Register gezeigt werden, dass das Ergebnis einer Instruktion mit der entsprechenden Adressierungsmethode in dieses Register geschrieben wird. Damit das Schreiben festgestellt wird, werden die entsprechenden Systemzustände erzeugt und verglichen. Im ersten Schritt der Testfallspezifikation muss der neutrale Systemzustand initialisiert werden. Anschließend wird die Instruktion mit der entsprechenden Adressierungsmethode mit verschiedenen Werten aus der Überdeckungsmenge instanziiert und wiederholt ausgeführt. Der Befehlsoperand der Adressierungsmethode, der für die Berechnung der Registeradresse verwendet wird, muss dabei den vollständigen Wertebereich durchlaufen. Damit wird sichergestellt, dass die Adressierungsmethode auch alle mit ihr erreichbaren Register verwendet. Nach jeder Instruktion, bei der ein Wert in die Registerbank geschrieben wird, wird der Systemzustand des Prozessors aufgezeichnet. Wurde nach der Ausführung einer Instruktion eine Systemzustandsänderung für ein Register festgestellt, ist das Register in der Schreibrichtung erreichbar.

Strategie Die Testfallspezifikation wird aus den Überdeckungskriterien und durch die beschriebenen abstrakten Methoden erzeugt. Für die Erzeugung der Testfälle können aus der Überdeckungsmenge der Register die erwarteten Ergebnisse abgeleitet werden. Damit kann für diese Validierungsmethode die Strategie A angewendet werden. Nach der Ausführung der Testfälle kann aus den initialen und den finalen Systemzuständen die Benutzung der Register rekonstruiert und dem Prozessorentwickler angezeigt werden. Durch die automatische Generierung der Testfallspezifikation und der erwarteten Ergebnisse kann die Validierung automatisch durchgeführt werden.

5.5.3 Register - Aliasing

Es ist sinnvoll, die Erreichbarkeit der Register zu validieren, bevor das Aliasing überprüft wird. Der Registersatz wird durch die Sprachkonstrukte für physikalische Register und architektonische Register gebildet. Alle Sprachkonstrukte für die Register in *ViCE-UPSLA* besitzen eine Spezifikation auf einem hohen Abstraktionsniveau, in dem verschiedene Parameter wie z. B. die Größe, die Anzahl oder die Zielregisterbank angegeben werden. Z. B. werden bei der Erzeugung einer physikalischen Registerbank lediglich die Parameter wie Breite, Anzahl oder Offset der Register angegeben. Aus dieser Spezifikation werden die einzelnen Register als Komponenten der Spezifikation automatisch erzeugt.

In Abschnitt 5.2.2 wurden bereits Konsistenzbedingungen für das Aliasing aufgestellt, wobei festgestellt wurde, dass der Registersatz für die Simulation vollständig generiert wird und die Konsistenz des Registersatzes auf dieser Ebene nicht verletzt werden kann. Die Inkonsistenzen können lediglich zwischen der informellen Beschreibung und der Spezifikation des Prozessors durch den Entwickler induziert werden.

Die Überdeckungskriterien für die Generierung der Testfallspezifikation sowie die abstrakte Methode können aus der Validierung der Erreichbarkeit von Registern übernommen werden. Wobei die Überdeckungsmenge für die Registerbänke auf die architektonischen Register beschränkt werden kann.

Überdeckungskriterium Bei der Validierung der Register-Alias müssen alle architektonische Registerbänke validiert werden, für die eine Adressierungsmethode existiert. Nach der Validierung der Erreichbarkeit kann die Überdeckung dieser Methode übernommen werden. Diese Überdeckung kann wie bereits gezeigt vollständig aus der Spezifikation des Prozessors abgeleitet werden.
Die Testfallspezifikation für die Validierung der Alias kann also automatisch generiert werden. Aus der Konsistenzbedingung und der abstrakten Methode wird für die Validierung nur die Eingabe für das System erzeugt. Die erwartete Ausgabe des Systems kann aus der Spezifikation nicht abgeleitet werden und muss durch den Prozessorentwickler erzeugt werden.

Strategie Aus den aufgezeichneten Systemzuständen bei der Validierung wird eine Zuordnung zwischen den architektonischen Registern und den physikalischen Registern nachgebildet. Die Validierung der Registerbank C, aus Abbildung 5.10 in Abschnitt 5.5.2, erzeugt eine Zuordnung wie in Abbildung 5.11 dargestellt. Bei der Validierung kann der Prozessorentwickler entscheiden, ob die Ergebnisse aus der Simulation mit der informellen Beschreibung des Prozessors übereinstimmen.

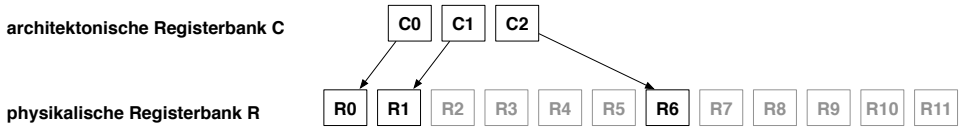


Abbildung 5.11: Ermittelte Zuordnung der Alias zu den physikalischen Registern aus der Simulation.

5.5.4 Adressierung

Die Adressierungsmethoden sind das Bindeglied in der Spezifikation zwischen den interpretierten Operanden der Instruktionen und dem Registersatz. In Abschnitt 4.3.4.2 wurden die unterschiedlichen Adressierungsmethoden vorgestellt. Für die Validierung sind die Art der Adressierung, direkt oder indirekt, und die verwendeten Befehlsoperanden, Register oder immediate Werte, relevant. Daraus ergeben sich sechs verschiedene Arten der Adressierung, die bei der Validierung unterschiedliche behandelt werden. Zusätzlich müssen Adressierungsmethoden für das Laden und Speichern von Werten getrennt betrachtet werden. Für die Validierung der Benutzungsrichtungen werden unterschiedliche abstrakte Methoden benötigt.

Wie bereits im Fehlermodell festgestellt, wird die Konsistenz der direkten Adressierungsmethoden in *ViCE-UPSLA* statisch validiert. Die `Pre load` und `Post load` Adressierungsmethoden verwenden beim Laden oder Speichern eines Wertes eine Funktion. Durch die verwendeten Funktionen wird entweder die Zieladresse oder der Inhalt des Registers berechnet. Um die Konsistenz ihrer Eigenschaften zu sichern, muss die Validierung der indirekten Adressierungsmethoden dynamisch durchgeführt werden. Im Folgenden wird die Validierung der `Pre load` Adressierungsmethoden beschrieben.

Die `Pre load` Adressierungsmethoden berechnen für die Leserichtung die Adresse des Registers, aus dem der Wert für den interpretierten Operanden geladen wird. Für die Schreibrichtung wird analog aus den Befehlsoperanden die Adresse des Registers berechnet, in das der Wert des interpretierten Operanden gespeichert wird. In Abbildung 5.12 wird schematisch rot dargestellt, dass aus dem Befehlsoperanden $\mathtt{B}_{opd}$ mit Hilfe der Funktion `f` eine Adresse eines Registers $\mathtt{R}_y$ aus der Registerbank berechnet wird. Bei `Pre load` Adressierungsmethoden werden die Werte selbst, in Abbildung 5.12 grün dargestellt, nicht verändert und müssen nicht dynamisch validiert werden.

Die Konsistenz der `Pre load` Adressierungsmethoden kann in zwei Punkte aufgeteilt werden:

- Konsistenz zwischen dem Wertebereich des Befehlsoperanden B_{opd} und dem Adressraum.
- Korrektheit der berechneten Adresse.

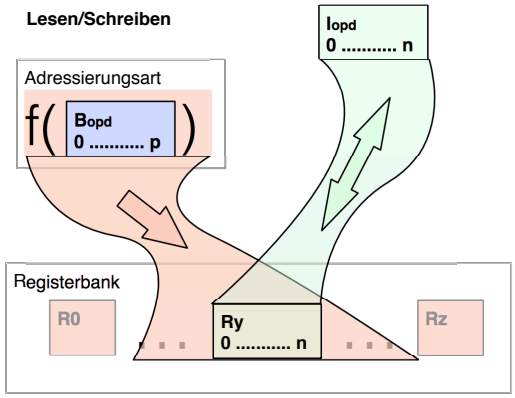


Abbildung 5.12: Wertebereiche der `Pre load` Adressierungsmethoden; grün: statische Validierung; rot: dynamische Validierung; blau: Überdeckungsmenge.

Die Überdeckungskriterien für die Schwerpunkte sind gleich. Es werden jeweils unterschiedliche abstrakte Methoden angegeben, wodurch unterschiedliche Validierungsmethoden beschrieben werden.

Überdeckungskriterium Wie aus Abbildung 5.12 hervorgeht, wird für jede `Pre load` Adressierungsmethode die Konsistenz zwischen den Ergebnissen der Funktion und den Registeradressen überprüft. Die Adressen werden aus den Befehlsoperanden, die für die Adressierungsmethoden verwendet werden, berechnet. Damit lassen sich zwei Wertebereiche festlegen. Zum einen wird ein Wertebereich durch die Befehlsoperanden beschrieben. Dieser Wertebereich ist durch die Mengenkodierung in der Prozessorspezifikation in *ViCE-UPSLA* angegeben. Zum anderen ist der zweite Wertebereich für die Registeradressen durch die Spezifikation der Registerbank eindeutig vorgegeben. Bei der Validierung einer `Pre load` Adressierungsmethode wird der durch die Befehlsoperanden vorgegebene Wertebereich überdeckt. Der Wertebereich der Registerbank wird für die Beschreibung der erwarteten Ergebnisse benutzt und darf bei der Validierung nicht verletzt werden.

Abstrakte Methode – Konsistenz der Wertebereiche Mit dieser abstrakten Methode soll die Konsistenz zwischen der Adressierungsmethode und der von ihr verwendeten Registerbank sichergestellt werden. Dabei darf die Adressierungsmethode für die möglichen Eingaben durch die verwendeten Befehlsoperanden den Wertebereich der Registerbank nicht verletzen. Dazu wird, analog zu der Validierung der Erreichbarkeit von Registern in Abschnitt 5.5.2, die Erreichbarkeit von Adressen durch eine Adres-

sierungsmethode mit derselben abstrakten Methode validiert. Nach der Initialisierung des Systemzustandes wird die Instruktion mit der zu validierenden Adressierungsmethode und mit den Werten aus dem Wertebereich der Befehlsoperanden ausgeführt. Nach jedem Schreibvorgang wird der Systemzustand aufgezeichnet, womit später die Belegung der Adressen durch die Adressierungsmethode ermittelt wird.

Strategie Die Testfallspezifikation und das erwartete Verhalten des Systems lassen sich automatisch aus der Spezifikation des Prozessors generieren. Für die Validierungsmethoden wird die Strategie A angewendet. Dabei werden die simulierten Ergebnisse mit den erwarteten Ergebnissen verglichen. Die simulierten Ergebnisse werden aus den initialen und den finalen Systemzuständen gewonnen. Die Systemzustände aus der Simulation beschreiben die Zuordnung zwischen den Werten für die Befehlsoperanden und den Registeradressen und können schematisch wie in Abbildung 5.13 dargestellt werden. Die erwarteten Ergebnisse werden nach den Konventionen des Prozessorentwurfs durch die Überdeckungsmenge der Registerbank beschrieben und können aus der Prozessorspezifikation abgeleitet werden.

Die Validierung der Leserichtung von `Pre load` Adressierungsmethoden wird analog durchgeführt, dabei muss die Vorgehensweise für die abstrakte Methode aus der Registererreichbarkeit aus Abschnitt 5.5.2 adaptiert werden.

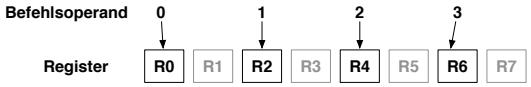


Abbildung 5.13: Simulierte Ergebnisse; Zuordnung zwischen Eingabewerten der Befehlsoperanden und den Registeradressen.

Abstrakte Methode – Korrektheit der Adressberechnung Durch eine weitere Methode kann die Validierungsmethode für die korrekte Adressberechnung angegeben werden. Dabei wird die Berechnung der Adressen in einer Adressierungsmethode durch eine alternative Folge von Instruktionen nachgebildet. Die Berechnung einer konkreten Adresse wird gegen die Berechnung durch die Instruktionen validiert.

Bei dieser abstrakten Methode wird nach der Initialisierung des Systems eine Instruktion mit der zu validierenden Adressierungsmethode instanziiert und ausgeführt, anschließend wird der Systemzustand aufgezeichnet. Nach der erneuten Initialisierung des Systems wird eine durch den Prozessorentwickler angegebene Instruktionsfolge für die alternative Berechnung der Adresse mit denselben Werten instanziiert und ausgeführt. Die Folge von Instruktionen simuliert die Berechnung der Adresse in der Funktion `f` und beschreibt bei der Verwendung gleicher Parameter dasselbe Zielregister.

Strategie Die Validierungsmethode beschreibt zwei Eingaben für den Simulator. Anschließend werden die aufgezeichneten Systemzustände validiert. Damit wird für die Validierungsmethode die Strategie B verwendet, bei der das simulierte und das erwartete Ergebnis aus zwei Eingaben für den Simulator erzeugt werden. Die Ergebnisse der Simulation können automatisch validiert werden.

5.5.5 Instruktionssemantik

Die meisten Eigenschaften der Instruktionsbeschreibung, wie z. B. Ressourcenbelegung oder Kodierung der Instruktionen, werden in *ViCE-UPSLA* statisch validiert. In der operationalen Beschreibung werden Operatoren verwendet, die neben einer strukturellen Beschreibung durch Funktionen spezifiziert sind. Eingesetzt in den Datenflussgraphen der operationalen Beschreibung, definieren die Operatoren das Verhalten der Instruktionen. Wie aus dem Fehlermodell in Abschnitt 5.2 hervorgeht, muss die operationale Beschreibung der Instruktionen dynamisch validiert werden. Die dynamische Validierung wird durch die Verwendung der Funktionen in den Operatoren benötigt. Wie schon für die Validierung der Adressierungsmethoden, müssen für die Operatoren die Wertebereiche und die korrekte Berechnung der Ergebnisse validiert werden. In diesem Abschnitt wird zunächst die Validierung der Operatoren vorgestellt. Anschließend wird eine Methode für die Validierung von komplexen Instruktionen beschrieben, die z. B. bei einer Instruktionssatzerweiterung erzeugt werden. Dabei wird beschrieben, wie die operationale Beschreibung neuer Instruktionen auf ihre Korrektheit überprüft werden kann.

Operatoren Die Validierung der Operatoren kann mit der bereits beschriebenen Validierung der indirekten Adressierungsmethoden verglichen werden. Bei der Validierung von Operatoren müssen zwei Eigenschaften sichergestellt werden. Zum einen können die Operatoren auf Konsistenz bezüglich der Wertebereiche für die Ein- und Ausgabeparameter validiert werden, zum anderen kann die Korrektheit der Ergebnisse validiert werden.

In Abbildung 5.14 wird schematisch ein Operator dargestellt, welcher für die Eingabeparameter `A` und `B` die Bitbreiten `n` und `q` besitzt und für den Ausgabeparameter `C` die Bitbreite von `p` nicht verletzen darf. Der Wertebereich des Ausgabeparameters kann verletzt werden, indem bei gültigen Eingabeparametern `A` und `B` die Bitbreite das Ergebnis aus der Funktion des Operators den Wertebereich von `C` überschreitet. Die Validierung soll zeigen, dass der Operator für beliebige gültige Eingaben den Wertebereich des Ausgabeparameters nicht verletzt.

Überdeckungskriterium Für die Validierung der Operatoren wird die Überdeckungsmenge aus der Prozessorspezifikation ermittelt. Aus der Spezifikation der operationalen

Beschreibung, wie in Abbildung 5.14 dargestellt, werden die Wertebereiche der Operatoren (blau) automatisch festgestellt.

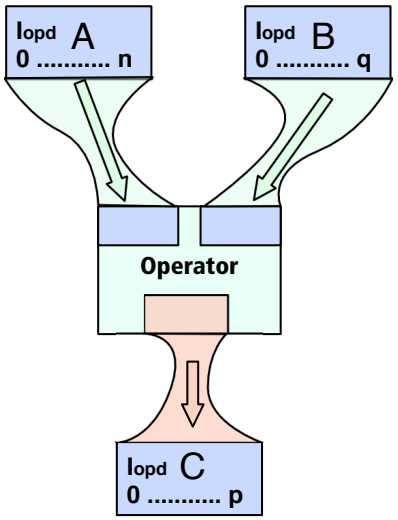


Abbildung 5.14: Wertebereiche Operatoren; grün: statische Validierung; rot: dynamische Validierung; blau: Überdeckung.

Abstrakte Methode – Konsistenz der Wertebereiche Die abstrakte Methode für die Validierung der Wertebereiche sieht vor, dass nach der Initialisierung des Systems die Instruktionen mit den Eingabeparametern ausgeführt und resultierende Systemzustände aufgezeichnet werden.

Die Erzeugung der Testfallspezifikation kann automatisiert und vollständig aus der Spezifikation abgeleitet werden. Für die Validierung werden Instruktionen benötigt, deren Verhaltensbeschreibung einen Operator enthält. Für den Fall, dass nicht alle Operatoren individuell in einer Instruktion verwendet werden, können die Testrahmen-Instruktionen (vgl. Abschnitt 5.5.1) eingesetzt werden.

Strategie Aus den aufgezeichneten Systemzuständen können als Ergebnis die Wertebereiche ermittelt werden. Die erwarteten Ergebnisse werden aus der Spezifikation des Prozessors generiert, womit die Strategie A angewendet werden kann.

Für die Beschreibung der Validierungsmethode der Korrektheit der Berechnung werden dieselben Überdeckungskriterien verwendet, wie für die Validierung der Wertebereiche. Die abstrakte Methode wird analog zu den Methoden für die Validierung der

indirekten Adressierungsmethoden beschrieben, indem der Prozessorentwickler alternative Berechnungen für das Ergebnis oder die erwarteten Systemzustände vorgibt.

Operationale Beschreibung In der operationalen Beschreibung werden die Operatoren eingesetzt und spezifizieren die Berechnung der Werte für die Instruktionen. Die Validierung der Instruktionen mit nur einem Operator wurde bereits beschrieben. Eine operationale Beschreibung kann durch Verkettung von Operatoren ausgedrückt werden, wie in Abschnitt 4.3.5.4 beschrieben. Dies kann zum einen durch die Spezifikation der Operatoren vorgegeben werden oder aus z. B. einer Instruktionssatzerweiterung resultieren, indem eine neue Instruktion zwei ursprünglich vorhandene Instruktionen in einem Prozessor ersetzt. Dabei können z. B. „excessed precision“ oder andere Seiteneffekte durch die Kombinationen von Operanden auftreten. Die Validierung von komplexen operationalen Beschreibungen auf Seiteneffekte wird dadurch ermöglicht, dass das Verhalten einer Instruktion des Prozessors gegen das Verhalten von Instruktionsfolgen validiert werden kann.

Überdeckungskriterium Die Überdeckungskriterien erfassen die Instruktionen des Prozessors, deren operationale Beschreibung aus mehreren Operatoren besteht. Die Wertebereiche der Operanden werden anhand der Operatoren der Instruktionen aus der Spezifikation ermittelt. Die Überdeckungsmenge für die Validierung kann damit aus der Prozessorspezifikation automatisch abgeleitet werden. Abbildung 5.15 zeigt schematisch die Parameter (blau), die aus der Spezifikation abgeleitet werden. Der damit beschriebene Wertebereich wird in die Überdeckungsmenge übernommen.

Abstrakte Methode Bei der Validierung wird für jede Instruktion eine alternative Folge von Instruktionen mit der Berechnung der Referenzergebnisse angegeben. Abbildung 5.15 stellt schematisch eine Instruktion mit zwei Operatoren und eine Instruktionsfolge für die Berechnung der Referenzergebnisse dar. Die Instruktionsfolge für die Berechnung der Referenzergebnisse kann durch die Analyse des Instruktionssatzes automatisch erzeugt werden, falls die entsprechenden Instruktionen mit einem Operator im Instruktionssatz spezifiziert sind. Alternativ können die benötigten Instruktionen als Testrahmen-Instruktionen generiert und verwendet werden. Nach der Initialisierung des Systemzustandes werden die Instruktionen, so wie die Instruktionsfolge für die Referenzberechnung, ausgeführt und die finalen Systemzustände aufgezeichnet.

Strategie Die Validierung wird nach der Strategie B automatisch durchgeführt, indem die erzielten Ergebnisse gegeneinander überprüft werden. Damit wird z. B. herausgestellt, ob Seiteneffekte zwischen den Operatoren bestehen oder ob eine Instruktionssatzerweiterung korrekt die ursprünglichen Instruktionen ersetzt.

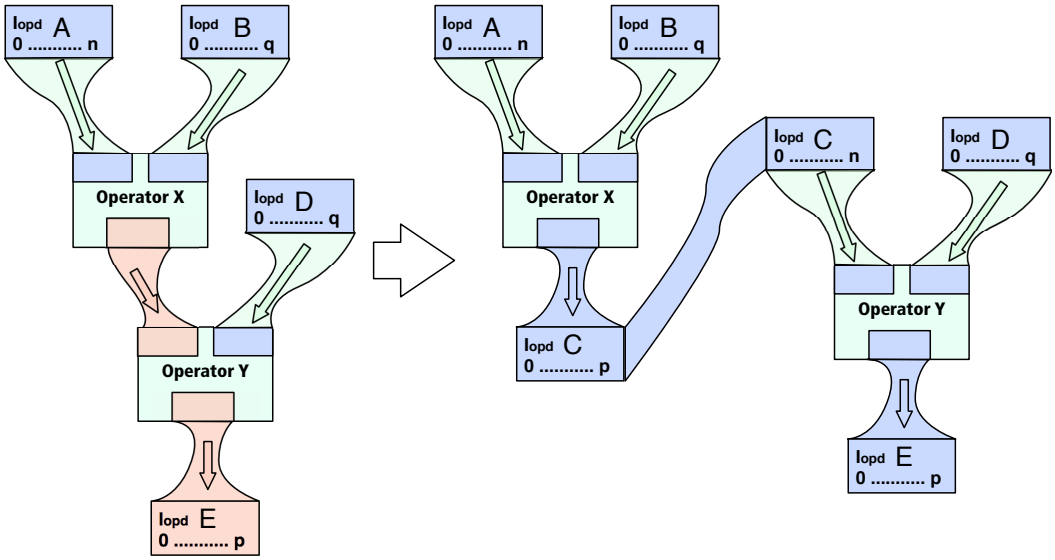


Abbildung 5.15: Validierung der Verhaltensbeschreibung von Instruktionen; links: komplexe Instruktion; rechts: Referenzberechnung durch eine Instruktionsfolge.

5.5.6 Pipelinekonflikte

Die Pipelineverarbeitung wird in den Prozessoren eingesetzt, um den Instruktionsdurchsatz zu steigern, wobei die Anzahl der Instruktionen pro Taktzyklus (Instructions per Cycle, *IPC*) durch die überlappende Verarbeitung gesteigert wird. Die Ausführungsdauer der Instruktionen wird nicht verändert, die Leistungssteigerung wird durch mehrere gleichzeitig ausgeführte Instruktionen erzielt. In Abbildung 5.16 werden an einem abstrakten Beispiel die überlappende und die serielle Ausführung der Instruktionen `MUL` und `ADD` dargestellt.

Für die Pipeline sind Daten-, Ressourcen- und Steuerkonflikte [HP06] in der Domäne des Prozessorentwurfs bekannt. Im Folgenden werden Methoden für die dynamischen Validierung und automatische Generierung der Testfälle beschrieben.

Ressourcenkonflikte Wie im Fehlermodell bereits erläutert, wird der Ressourcenbedarf für die einzelnen Instruktionen statisch validiert. Damit wird sichergestellt, dass jede Instruktion für sich alleine in dem Prozessor ausgeführt werden kann. Die Ressourcenkonflikte können dennoch entstehen, wenn mehrere Instruktionen in der Pipeline zum selben Zeitpunkt dieselbe Ressource benötigen. Mit *ViCE-UPSLA* erzeugte Si-

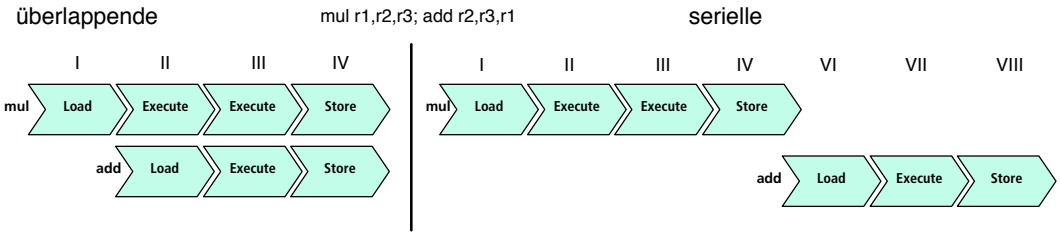


Abbildung 5.16: Überlappende und serielle Verarbeitung von zwei Instruktionen mit unterschiedlicher Ausführungsdauer.

mulatoren verzögern die Instruktionen bei nicht ausreichenden Ressourcen durch einen Pipeline-Interlock-Mechanismus, wodurch Ressourcenkonflikte nicht auftreten können. Es ist dennoch sinnvoll eine Validierung der Ressourcenkonflikte durchzuführen. Damit kann z. B. herausgestellt werden, ob bestimmte Sequenzen von Instruktionen längere Ausführungszeiten durch den Interlock-Mechanismus haben als in der informellen Beschreibung des Prozessors vorgesehen.

Die dynamische Validierung der Ressourcenkonflikte wird besonders dann benötigt, wenn der spezifizierte Zielprozessor über keinen Pipeline-Interlock-Mechanismus verfügen soll, wie z. B. bei MIPS-Prozessoren [HKP+82, Kan88]. Hierfür muss im Prozessorsimulator der Pipeline-Interlock-Mechanismus deaktiviert werden. In Abbildung 5.16 wird am Beispiel der Instruktionen `mul` und `add` bei überlappender Ausführung eine Konfliktsituation gezeigt. Die Ressource `Execute` wird im dritten Taktzyklus sowohl für die `ADD` als auch für die `MUL` Instruktion benötigt. Ist die Ressource nur einmal vorhanden, kann die Instruktionssequenz nicht korrekt ausgeführt werden.

Überdeckungskriterium Bei der Validierung der Ressourcenkonflikte müssen alle Kombinationen von Instruktionspaaren des Prozessors validiert werden. Außerdem müssen die Wertebereiche der Befehlsoperanden in die Überdeckungsmenge aufgenommen werden, falls die Ausführungsdauer der Instruktionen von diesen Parametern abhängt. In *ViCE-UPSLA* kann die Überdeckungsmenge reduziert werden, indem aus der Spezifikation die Instruktionen mit gleichen Ausführungszyklen und verwendeten Ressourcen zusammengefasst werden. Z. B. würden die arithmetischen Instruktionen wie Addition und Subtraktion mit gleichen Ausführungszyklen und gleichem Ressourcenbedarf in eine Gruppe eingeordnet. Die Gruppierung kann in *ViCE-UPSLA* durch abstrakte Instruktionen durchgeführt werden. Die Überdeckungsmenge wird aus den Repräsentanten jeder Gruppen erzeugt und aus der Spezifikation des Prozessors automatisch abgeleitet.

Abstrakte Methode Für die Validierung werden die Instruktionen paarweise mit unterschiedlichen Abständen in der Pipeline ausgeführt. Die Ausführung der Instruktionspaare wird wiederholt und dabei die Befehlsoperanden aus dem Wertebereich der Operanden instanziiert, damit der Wertebereich der Parameter überdeckt wird.

Strategie Bei der Validierung wird untersucht, ob Kombinationen von Instruktionen fehlerfrei ausgeführt werden. Diese Bedingung beschreibt das erwartete Ergebnis für die Validierung. Das Validierungsergebnis ist positiv, wenn der generierte Testfall vollständig ausgeführt wurde. Bei bestehenden Ressourcenkonflikten wird die Ausführung im Simulator unterbrochen und die Instruktionen, die den Konflikt ausgelöst haben, ausgegeben. Die Vorgehensweise entspricht der Strategie A, bei der das erwartete Ergebnis aus der Spezifikation abgeleitet wird.

Datenkonflikte Die Datenkonflikte entstehen bei der überlappenden Ausführung von Instruktionen in der Pipeline. In Abbildung 5.16 wird die Ausführung der Instruktionen `mul r1,r2,r3` und `add r2,r3,r1` dargestellt. In dem Beispiel besteht zwischen den Instruktionen eine Datenabhängigkeit, da die Instruktion `add` das Ergebnis `r1` der Instruktion `mul` als zweiten Operanden benötigt.

Für die Auflösung der Datenkonflikte werden in Prozessoren Bypässe eingesetzt. Diese stellen die benötigten Werte zur Verfügung, bevor diese in die Registerbank zurückgeschrieben wurden. Bypässe haben im Prozessorlayout hohen Flächenbedarf bei der Chipentwicklung, deshalb werden Bypässe meistens nur partiell eingesetzt, sodass eine für jeden Prozessor individuelle Konfiguration entsteht.

Die dynamische Validierung der Datenkonflikte wird in *ViCE-UPSLA* durchgeführt, wenn die Spezifikation des Prozessors über eine Bypass-Struktur verfügt. Durch die dynamische Validierung der Datenkonflikte kann die spezifizierte Konfiguration der Bypass-Struktur ermittelt und mit der informellen Beschreibung für den Entwurf des Prozessors verglichen werden.

Überdeckungskriterium Wie bereits bei der Validierung der Ressourcenkonflikte, muss die Validierung der Datenkonflikte ein fehlerfreie Verarbeitung für alle Kombinationen von Instruktionen des Prozessors zeigen. Alternativ kann bei einem strukturierten Instruktionssatz die Überdeckungsmenge durch abstrakte Instruktionen für die Überdeckung reduziert werden. Dafür werden die Instruktionen mit gleicher Ausführungsdauer in der Pipeline zusammengefasst und jeweils ein Repräsentant aus jeder abstrakten Instruktion verwendet. Eine Überdeckungsmenge für die Validierung kann aus der Prozessorbeschreibung in *ViCE-UPSLA* automatisch generiert werden.

Abstrakte Methode Bei der Validierung muss jedes bei der Überdeckung ermittelte Instruktionspaar in der Pipeline ausgeführt werden. Dabei müssen alle Abstände, bis

auf die Länge der Pipeline, zwischen den Instruktionen überprüft werden. Für die Validierung der Bypässe muss die Ausführung der Instruktionsfolgen in der Pipeline mit und ohne Datenkonflikte überprüft werden. Diese Bedingung ist notwendig, um die korrekte Erkennung der Datenkonflikten sicherzustellen.

Für die Erzeugung der Referenzergebnisse können zwei Vorgehensweisen angewendet werden. Zum einen können die Datenkonflikte aufgelöst werden, wenn der Abstand der Instruktionen über die Länge der Pipeline erhöht wird, wie in Abbildung 5.16 dargestellt. Zum anderen kann ein Instruktionssatzsimulator eingesetzt werden, der keine überlappende Ausführung in der Pipeline enthält und frei von Datenkonflikten ist. Die Spezifikation in *ViCE-UPSLA* enthält sowohl die Spezifikation aller Instruktionen, als auch die Spezifikation der Mikroarchitektur, womit die Testfallspezifikation automatisch generiert werden kann.

Strategie Die Validierung der Datenkonflikte kann automatisch durchgeführt werden. Dabei kann sowohl die Strategie B als auch die Strategie C angewendet werden. Enthält eine Mikroarchitektur eine vollständige Bypass-Struktur, kann auch die Auswertung der Ergebnisse automatisch durchgeführt werden. Bei selektiv eingesetzten Bypässen wird für die Auswertung der Ergebnisse das Wissen des Prozessorentwicklers benötigt. Bei selektiver Bypass-Struktur kann dem Prozessorentwickler die bei der Simulation ermittelte Konfiguration angezeigt werden.

Steuerkonflikte Die Steuerkonflikte entstehen, wie alle Pipelinekonflikte, aus der Wechselwirkung zwischen den Instruktionen bei der Ausführung in der Pipeline. Die Steuerkonflikte können durch Instruktionen ausgelöst werden, die den Programmzähler verändern. Z. B. können die Sprungbefehle einen Konflikt auslösen, wenn unmittelbar hinter einem Sprungbefehl eine Instruktion nicht vollständig ausgeführt wurde.

In *ViCE-UPSLA* werden bedingte Sprungbefehle ohne Sprungvorhersage spezifiziert. Die Steuerkonflikte werden mit Hilfe von `Branch Delay Slots` (Warteplätze) oder durch Ressourcenbelegung behandelt. Die Spezifikation dieser Instruktionen kann dynamisch validiert werden. Bei der Validierung der bedingten Sprünge müssen beide Fälle, mit und ohne Sprung, betrachtet werden.

Überdeckungskriterium Für die Validierung müssen alle Instruktionen, die den Programmzähler verändern, gepaart mit den restlichen Instruktionen des Prozessors validiert werden. Die Überdeckungsmenge kann automatisch erzeugt werden, indem in der *ViCE-UPSLA* Spezifikation des Prozessors die Instruktionen mit dem Zugriff auf den Programmzähler ermittelt werden. Die Instruktionspaare für die Validierung können damit, wie schon für andere Pipelinekonflikte, automatisch erzeugt werden.

Abstrakte Methode Die abstrakte Methode ist eine Vereinfachung der Vorgehensweise für die Validierung von Datenkonflikten. Dabei werden Instruktionspaare, z. B. Sprungbefehl gefolgt von Addition, in unterschiedlichen Abständen in der Pipeline ausgeführt. Für die Referenzergebnisse werden Instruktionspaare über die Länge der Pipeline durch `NOP` Instruktionen auseinander gezogen. Alternativ kann die Erzeugung der Referenzergebnisse durch einen Instruktionssatzsimulator übernommen werden.

Strategie Die Validierung erfolgt, wie schon bei der Validierung der Datenkonflikte, indem zwei verschiedene Eingaben für einen Mikroarchitektursimulator verwendet werden oder zwei Simulatoren bei der Validierung eingesetzt werden. Damit entspricht die Vorgehensweise bei der Validierung der Strategie B oder `C`.

5.6 Zusätzliches Wissen

Aus der Beschreibung der statischen und dynamischen Validierungsmethoden geht hervor, dass für die Validierung einiger Sprachkonstrukte zusätzliche Angaben durch den Prozessorentwickler erforderlich sind, da z. B. die Redundanz für die Validierung fehlt. Hierfür wird in den meisten Validierungswerkzeugen, wie in den verwandten Arbeiten in Abschnitt 3.2 beschrieben, ein redundant erzeugtes Modell eingesetzt. Mit *ViCE-UPSLA* werden vollständige Prozessorspezifikationen erzeugt, aus denen Simulatoren automatisch generiert werden können. Als Erweiterung der Spezifikationssprache für die Validierung können die vorhandenen Prozessorkonstrukte durch zusätzliche Eigenschaften aus dem Prozessorentwurf erweitert werden. Dieser Ansatz besitzt den Vorteil, dass die Prozessorentwickler kein zusätzliches Modell für die Validierung erzeugen müssen und die zusätzlich eingebrachte Spezifikation für die Weiterentwicklung des Prozessors auf niedrigeren Abstraktionsebenen genutzt werden kann.

Bei der Beschreibung der Validierungsmethoden wurde ermittelt, welches zusätzliche Wissen in der Spezifikation für die Realisierung der Validierungsmethoden benötigt wird. Das entwickelte Werkzeugsystem soll den Benutzer auch bei der Beschreibung des zusätzlichen Wissens unterstützen, indem die Spezifikation des zusätzlichen Wissens in das Werkzeug integriert wird. Hierzu werden in diesem Abschnitt die verschiedenen Arten und die Modellierungsmöglichkeiten des zusätzlichen Wissens beschrieben. In den folgenden Abschnitten wird zunächst beschrieben, wie das zusätzliche Wissen verwendet werden kann und in welche Klassen das zusätzliche Wissen eingeteilt ist. Anschließend werden verschiedene Modelle im Zusammenhang mit der Sprache *ViCE-UPSLA* vorgestellt.

5.6.1 Modellierung des zusätzlichen Wissens

Für die Beschreibung des zusätzlichen Wissens wird zunächst das Prinzip für die Modellierung vorgestellt. Hierfür werden anhand von Abbildung 5.17 die Komponenten

aus der Prozessorspezifikation und der Validierung im Zusammenhang mit dem zusätzlichen Wissen abstrakt erläutert.

Für die Spezifikation von Prozessoren wurden in *ViCE-UPSLA* für die Prozessorkonstrukte *A,B* die Sprachkonstrukte `A` und `B`, wie in Abbildung 5.17, umgesetzt. Für die Generierung eines Simulators aus der Spezifikation werden mit *ViCE-UPSLA* verschiedene Eigenschaften aus der informellen Spezifikation des Prozessors durch die Sprachkonstrukte ausgedrückt. In Abbildung 5.17 werden die Prozessorkonstrukte durch Sprachkonstrukt `A` mit den Eigenschaften `V,W,X` und Sprachkonstrukt `B` mit den Eigenschaften `Y,Z` beschrieben. Damit ist die Prozessorspezifikation für die Generierung eines Simulators vollständig angegeben.

Für die Beschreibung der Validierungsmethoden wurde ein Fehlermodell in Abschnitt 5.2 beschrieben, indem die Fehlerquellen in einer Prozessorspezifikation lokalisiert wurden. Dabei wurden die Eigenschaften der Sprachkonstrukte und der Prozessorkonstrukte betrachtet. In dem Beispiel aus der Abbildung 5.17 wird angenommen, dass für die Eigenschaften `V,X,Z` der Sprachkonstrukte jeweils eine Fehlerquelle im Fehlermodell lokalisiert wurde.

Bei der Beschreibung der Validierungsmethoden in Kapitel 5 wurden die Methoden für alle Sprachkonstrukte und deren Eigenschaften aus dem Fehlermodell betrachtet. Im Beispiel aus Abbildung 5.17 wird dargestellt, dass die Eigenschaften `X` und `Z` aus der Prozessorspezifikation automatisch validiert werden können, indem durch die Validierungsmethoden `I,II` die Konsistenz zwischen `X` und `Y` sowie `W` und `Z` überprüft wird. Wenn die Eigenschaft `V` des Sprachkonstrukts `A` validiert werden soll, wird zusätzliches Wissen des Prozessorentwicklers benötigt. Wenn z.B., wie in Abbildung 5.17 gezeigt, das Prozessorkonstrukt `B` eine Eigenschaft besitzt, mit der eine Konsistenzbedingung formuliert werden kann, ermöglicht die Modellierung des zusätzlichen Wissens die Umsetzung einer entsprechenden Validierungsmethode `III`.

Die Modellierung des zusätzlichen Wissens für die Umsetzung der Validierungsmethode `III` kann durchgeführt werden, wenn das Prozessorkonstrukt `B` eine Eigenschaft besitzt, mit der eine Konsistenzbedingung für die Eigenschaft `V` formuliert werden kann. Damit wird *ViCE-UPSLA* um eine zusätzliche Prozessoreigenschaft erweitert. Die Erweiterung der Spezifikation wird für die Erzeugung eines Simulators nicht benötigt und nur für die Validierung verwendet. Die neu eingebrachte Spezifikation kann unter Umständen für die Entwicklung des Prozessors auf niedrigeren Abstraktionsebenen verwendet werden.

Die Beschreibung solcher Konstrukte wird in die Sprache *ViCE-UPSLA* integriert. Damit kann die Spezifikation des Prozessors mit einem geringen Spezifikationsaufwand auf einem hohen Abstraktionsniveau ergänzt werden.

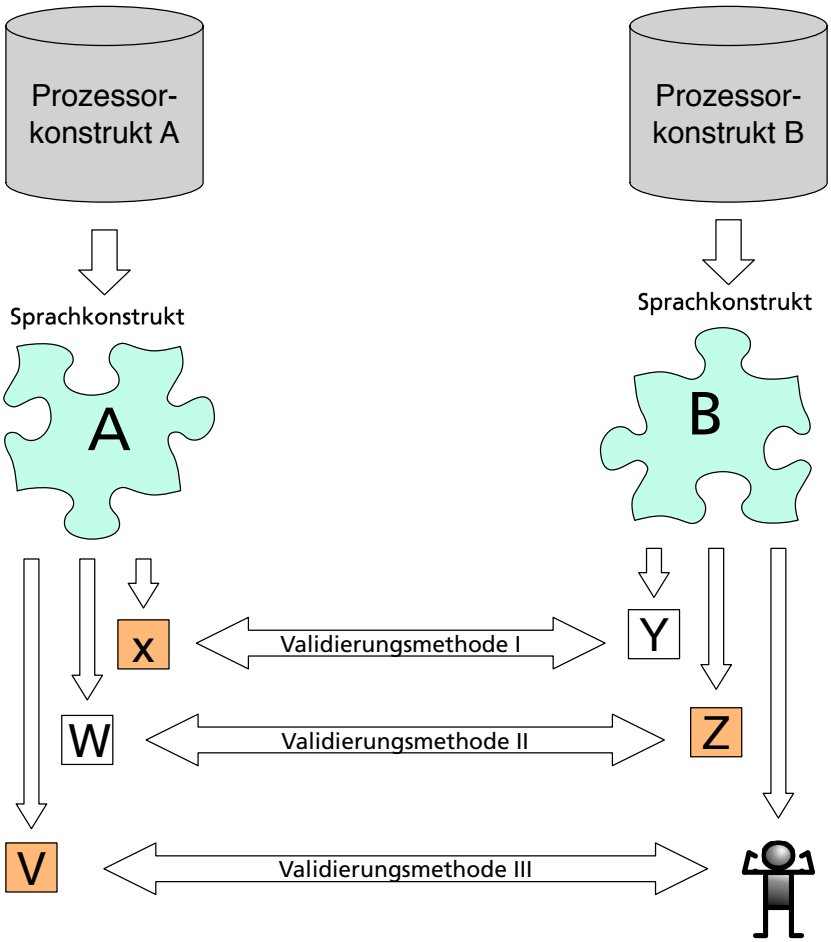


Abbildung 5.17: Modellierung des zusätzlichen Wissens.

5.6.2 Klassifikation des zusätzlichen Wissens

Das zusätzliche Wissen wird durch die Erweiterungen in der Spezifikation beschrieben, die für die Formulierung der Validierungsmethoden benötigt werden. Die zusätzlich modellierten Eigenschaften können z. B. aus der informellen Beschreibung in die Spezifikation einfließen oder bedingt durch die Vorgehensweise bei der Spezifikation aus dem Wissen des Prozessorentwicklers formuliert werden. Damit kann zwischen dem impliziten und expliziten Wissen unterschieden werden. Im Folgenden werden die Unterschiede genau erläutert.

Das explizite Wissen wird durch die Eigenschaften des Prozessors, die explizit in der

informellen Beschreibung des Prozessors angegeben sind, charakterisiert. Z. B. werden in *ViCE-UPSLA* für die Spezifikation der Operatoren keine Datentypen angegeben, da diese Eigenschaft aus den angegebenen C-Funktionen der Operatoren für die Erzeugung eines Simulators folgt. In einer informellen Beschreibung wird diese Eigenschaft für die Komponenten des Prozessors konkret angegeben. Die Erweiterung der Spezifikation durch Datentypen kann zum einen für die statische Validierung verwendet werden, zum anderen können die spezifizierten Datentypen für die weitere Entwicklung des Prozessors auf niedrigeren Abstraktionsstufen genutzt werden.

Eine andere Form, zusätzliches Wissen in die Spezifikation einzufügen ist das implizite Wissen. Die Adressierungsmethoden z. B. beschreiben in einer Spezifikation den Zugriff auf die Registerbänke des Prozessors und die Berechnung der interpretierten Operanden. Dabei können die Adressierungsmethoden sowohl für die Lese- als auch für die Schreibrichtung benutzt werden. In einem Prozessor können Adressierungsmethoden angegeben werden, deren Benutzung nur in einer Richtung sinnvoll ist. Durch die Angabe der Benutzungsrichtung von Adressierungsmethoden kann die konsistente Verwendung dieser Sprachkonstrukte statisch geprüft werden. Die so angegebene Eigenschaft für die Sprachkonstrukte folgt aus dem impliziten Wissen des Prozessorentwicklers.

In den folgenden Abschnitten werden die Möglichkeiten für die Modellierung des zusätzlichen Wissens an Beispielen gezeigt. Dabei wird zunächst die Kategorie für das explizite Wissen und anschließend die Spezifikation des impliziten Wissens beschrieben.

5.6.3 Explizites Wissen

Durch das explizite Wissen wird die Prozessorspezifikation verfeinert, indem weitere Eigenschaften aus der informellen Beschreibung in die Spezifikation übernommen werden.

5.6.3.1 Datentypsystem

Für die Validierung der operationalen Beschreibung von Instruktionen wurden in Abschnitten 5.4.6 und 5.5.5 statische und dynamische Methoden angegeben. Bei der statischen Validierung wurde herausgestellt, dass für die Operanden der operationalen Beschreibung Datentypen aus der Spezifikation ermittelt werden können. Damit kann die konsistente Benutzung der Operatoren in Abhängigkeit zu den Operanden sichergestellt werden.

Die Datentypen der Operatoren für die statische Validierung werden in der Spezifikation aus den Datentypen der Operanden ermittelt. Abhängig von der Konfiguration des Prozessors kann die statische Analyse der Operatoren nur automatisch angewendet werden, wenn die Datentypen für die Operatoren aus der Spezifikation ermittelt werden können. Diese Bedingung kann umgangen werden, indem die Spezifikation der

Operatoren durch Datentypen ergänzt wird. Die Datentypen der Operatoren können aus der informellen Beschreibung des Prozessors entnommen werden.

Im Detail werden in der Spezifikation des Prozessors für die Ein- und Ausgabeparameter der Operatoren die Datentypen angegeben. Mit den spezifizierten Datentypen für Operatoren kann die Validierung der Konsistenz in der operationalen Beschreibung der Instruktionen kontinuierlich durchgeführt werden, indem neu erstellte Verknüpfungen zwischen den Operationen die Konsistenzprüfungen zwischen den Verknüpfungsknoten anstoßen. Abbildung 5.18 zeigt ein Modell für die Vorgabe der Datentypen für zwei Operatoren und die Validierung in der operationalen Beschreibung einer Instruktion. Damit wird die statische Validierungsmethode, wie bereits in Abschnitt 5.4.6 beschrieben, für beliebige Konfigurationen der Operatoren in der operationalen Beschreibung ermöglicht.

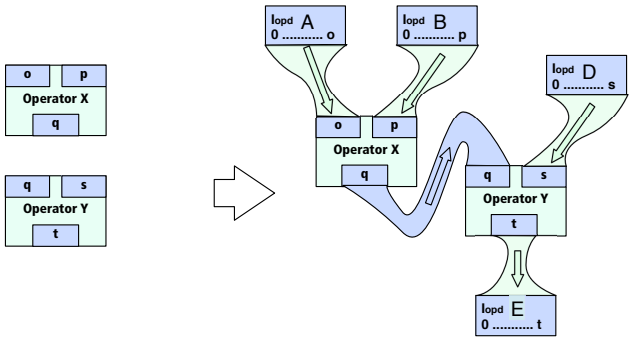


Abbildung 5.18: Spezifikation und Validierung von Datentypen.

Analog zu der Spezifikation der Datentypen für die Operatoren können auch die Adressierungsmethoden mit einem Datentypsystem erweitert werden. Damit wird die Validierung der Adressierungsmethoden und des Registersatzes unterstützt. Dem Prozessorentwickler wird damit ermöglicht, die Prozessorspezifikation vollständig mit einem Datentypsystem zu beschreiben und das vorgegeben System automatisch zu validieren.

Das somit beschriebene zusätzliche Wissen erweitert die Spezifikation mit zusätzlichen Eigenschaften des Prozessors aus der informellen Beschreibung. Die Beschreibung der Datentypen kann für die Weiterentwicklung des Prozessors und für die Beschreibung der Funktionen von Operatoren genutzt werden.

5.6.3.2 Ressourcen

Bei der Beschreibung der statischen Validierungsmethoden in Abschnitt 5.4 wurde die Methode zur Grapheinbettung vorgestellt. Diese Methode wird durch die Zuordnung

der Ressourcen für die Komponenten der operationalen Beschreibung und der Zuordnung der Instruktionen zu den ALUs des Prozessors ermöglicht. Bei der Validierung wird geprüft, ob der Datenflussgraph aus der Verhaltensbeschreibung einer Instruktion zu dem Datenpfad der Mikroarchitektur konsistent ist.

Durch die detaillierte Beschreibung der Ressourcen einer ALU kann die Spezifikation erweitert werden, indem eine Zuordnung zwischen ALUs und den Operatoren für die operationale Beschreibung erzeugt wird. Die Position für die Spezifikation der Operatoren kann bei der Entwicklung eines Mikroarchitektursimulators in die ALU verlegt werden. Diese Spezifikation entspricht der Entwicklung der Hardware in fortgeschrittenen Entwicklungsstufen und kann für die Weiterentwicklung des Prozessors verwendet werden.

Analog zu der Zuordnung der Operatoren zu den ALUs kann die Spezifikation erweitert werden, indem die Adressierungsmethoden als Ressourcen für die Konstrukte des Datenpfads angegeben werden, welche die entsprechenden Lese- und Schreib-Ports bzw. die Multiplexer der Mikroarchitektur beschreiben.

Die Definition der Ressourcen erfordert geringen Spezifikationsaufwand, da eine überschaubare Menge von Komponenten spezifiziert wird, und ermöglicht die automatische Durchführung der statischen Validierung.

5.6.4 Implizites Wissen

Wie bereits erläutert, beschreibt das implizite Wissen die Eigenschaften für die Prozessorkonstrukte, die nicht unmittelbar aus der informellen Beschreibung folgen. Die Spezifikation von implizitem Wissen wird am folgenden Beispiel beschrieben.

5.6.4.1 Verwendungsnachweis

In Abschnitt 5.4.2 wurde eine Validierungsmethode zum Verwendungsnachweis der Sprachkonstrukte vorgestellt. Bei der Validierung soll z. B. die Benutzung von Registerbänken des Registersatzes in der Spezifikation sichergestellt werden. Wie bereits in Abschnitt 4.3.3 erläutert, werden in *ViCE-UPSLA* architektonische Register verwendet, wobei einige Registerbänke zur Strukturierung des Registersatzes eingesetzt und nur indirekt in der Spezifikation verwendet werden.

Bei der Validierung der Benutzung werden alle Registerbänke auf die gleiche Weise geprüft und das Ergebnis dem Prozessorentwickler präsentiert. Dabei bekommt der Prozessorentwickler alle zur Strukturierung eingesetzten Registerbänke als nicht benutzte Registerbänke angezeigt. Für die exakte Validierung des Registersatzes kann die Spezifikation einer Registerbank mit der Eigenschaft für die Benutzung `direkt` oder `indirekt` erweitert werden. Durch diese Spezifikation wird die Konsistenzbedingung der Erreichbarkeit für einige Registerbänke außer Kraft gesetzt. Die zusätzlichen Eigenschaften ermöglichen bereits während der Bearbeitung, die Spezifikation auf In-

konsistenzen zu prüfen. Damit können Hinweise für den Prozessorentwickler angezeigt werden, falls z. B. eine indirekte Registerbank durch eine Adressierungsmethode referenziert wird.

In der Regel wird in der informellen Spezifikation keine Hilfsbeschreibung zur Strukturierung, sondern nur die Struktur des Registersatzes angegeben. Die Verwendung der architektonischen Registerbänke zur Strukturierung wird durch den Prozessorentwickler durchgeführt, womit die Eigenschaften aus dem impliziten Wissen des Benutzers folgen und bei der automatischen Validierung sinnvoll eingesetzt werden. Diese Erweiterung kann neben der statischen Validierung bei der Generierung der Testfälle für die Validierung der Registererreichbarkeit, wie in Abschnitt 5.5.2 beschrieben, verwendet werden. Dabei kann die Information über die Verwendung der Registerbänke für eine zusätzliche Präzisierung der Überdeckungskriterien verwendet werden.

5.7 Testfallspezifikation

In den Werkzeugsystemen aus MBT, wie in Abschnitt 3.2 vorgestellt, werden Modelle für die Spezifikation der Testfälle für die dynamische Validierung der Systeme verwendet. Damit wird Testentwicklern die Möglichkeit gegeben, die Struktur der Testfälle in einem Modell zu bearbeiten. Der Ansatz von *ViCE-UPSLA* verfolgt das Ziel, Testfälle und Simulatoren aus einer Prozessorspezifikation zu generieren. Die Testfallspezifikation wird bei der Erzeugung der Testfälle in einem Zwischenschritt eingesetzt, indem aus der Spezifikation des Prozessors eine Testfallspezifikation generiert wird und anschließend aus der Testfallspezifikation die Testfälle generiert werden. Damit wird die Möglichkeit geschaffen, die Testfälle in einer angemessenen Darstellung dem Prozessorentwickler zur Kontrolle oder Bearbeitung anzubieten.

Bereits bei der Vorstellung der Konzepte und der Methoden für die dynamische Validierung wurde das Konzept der Testfallspezifikation eingeführt. In Abbildung 5.6 und 5.19 wird die Testfallspezifikation so eingeordnet, dass diese zum einen aus der Prozessorspezifikation durch die Validierungsmethoden abgeleitet oder durch den Benutzer vorgegeben oder bearbeitet werden kann. Damit die Testfallspezifikation sowie die Sprache *ViCE-UPSLA* in einer visuellen Repräsentation auf einem hohen Abstraktionsniveau dem Prozessorentwickler zu Verfügung stehen, wird in diesem Abschnitt eine Spezifikationssprache *TFS* für die Beschreibung solcher Testfallspezifikationen eingeführt. Die Anforderungen an die Sprache für die Testfallspezifikation werden anhand von Benutzungsszenarios veranschaulicht, indem die automatische Generierung einer Testfallspezifikation und die Bearbeitung durch den Prozessorentwickler betrachtet werden.

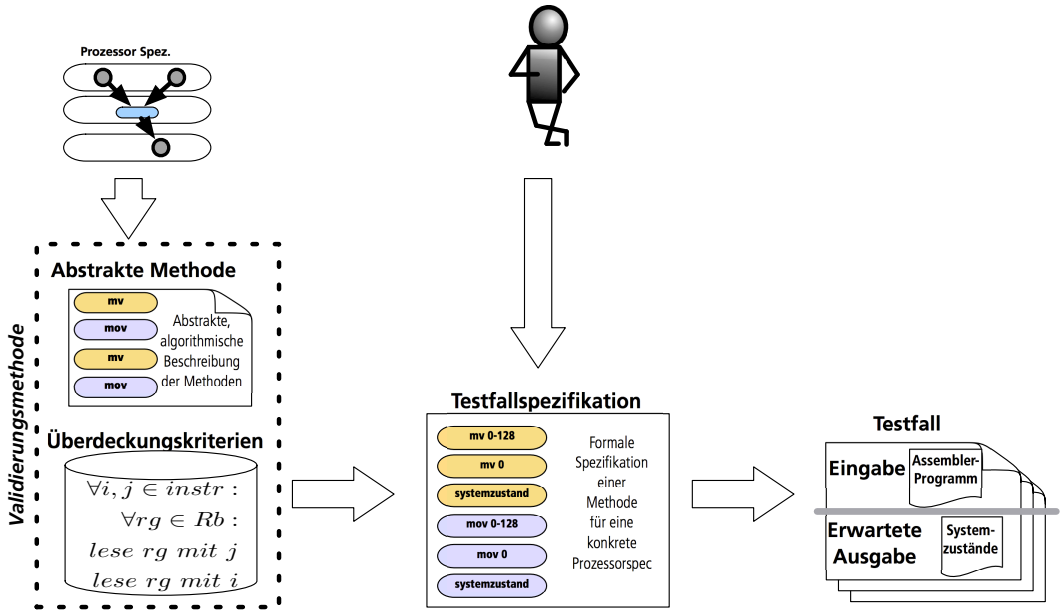


Abbildung 5.19: Einordnung der Testfallspezifikation.

5.7.1 Aufgaben der Testfallspezifikation

Wie aus dem Konzept für die dynamische Validierung hervorgeht, wird eine Testfallspezifikation dazu verwendet, die Testfälle für die dynamische Validierung zu erzeugen. Zur Erzeugung einer Testfallspezifikation wird eine Validierungsmethode auf eine Prozessorspezifikation angewendet, welche in einem Generator in dem Werkzeugsystem umgesetzt ist. Die Komponenten für die Beschreibung einer dynamischen Validierungsmethoden wurden in Abschnitt 5.5 eingeführt. Dabei beschreiben die Überdeckungskriterien die Strategie für die Ermittlung der Menge der Prozessorkonstrukte für die Anwendung der Validierungsmethode. Die abstrakte Methode beschreibt eine Teilaufgabe, mit der eine Eigenschaft eines Prozessorkonstruktes validiert werden kann. Damit ist eine Testfallspezifikation eine Instanz, aus einer abstrakten Methode und den entsprechenden Überdeckungsmengen. Aus einer Testfallspezifikation wird ein Testfall für einen konkreten Prozessor erzeugt.

Zur Erläuterung der Konzepte wird die Sprache *TFS* zunächst aus der Sicht der automatischen Erzeugung einer Testfallspezifikation aus der Prozessorspezifikation betrachtet. Anschließend werden die visuellen Konzepte der Sprache vorgestellt, womit die Bearbeitung einer Testfallspezifikation dem Prozessorentwickler ermöglicht wird.

Für die Identifizierung der Struktur, die eine Testfallspezifikation beschreiben soll,

wird zunächst betrachtet, wie ein Testfall aus einer Prozessorspezifikation durch die Überdeckungskriterien und die abstrakte Methode einer Validierungsmethode abgeleitet wird. In Abbildung 5.20 wird dargestellt, dass ein Testfall durch die Überdeckungskriterien und die abstrakte Methode der Validierungsmethode in drei Schritten aus einer Prozessorspezifikation abgeleitet wird.

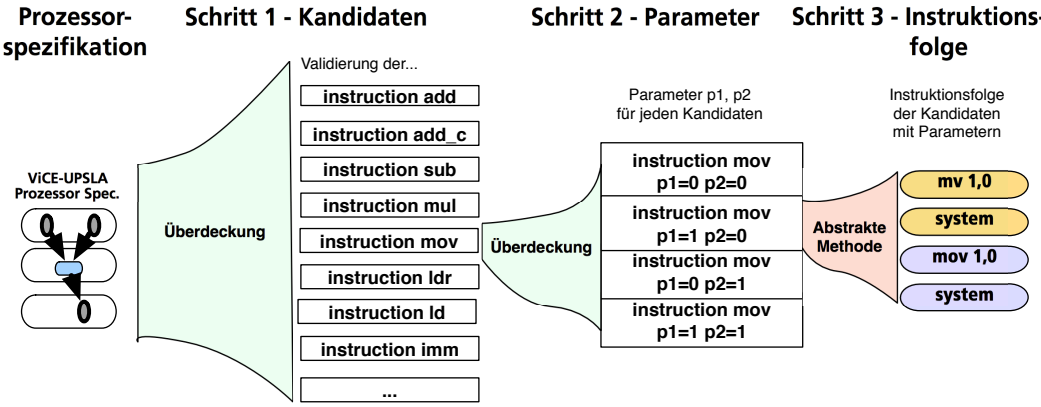


Abbildung 5.20: Ableitung eines Testfalls aus einer Prozessorspezifikation. Grün: Überdeckung, Rot: abstrakte Methode.

Im Detail wird in Abbildung 5.20 die Ableitung aus einer Prozessorspezifikation zu einem Testfall ohne den Einsatz einer Testfallspezifikation dargestellt. Die Prozessorspezifikation wird als Eingabe benutzt, aus der die Überdeckung (grün dargestellt) und die abstrakten Methoden (rot dargestellt) einen Testfall erzeugen. In Beispiel wird aus der Prozessorspezifikation durch die Überdeckungskriterien im ersten Schritt die Menge der zu validierenden Instruktionen ermittelt. Im zweiten Schritt werden ebenfalls anhand der Überdeckung die Parameter und die Wertebereiche der Parameter aus der Prozessorspezifikation bestimmt. Im dritten Schritt werden mit den Codemustern der abstrakten Methode die Instruktionsfolgen eines Testfalls aus der ermittelten Menge der Instruktionen und Parameter erzeugt.

Die Instruktionsfolgen werden bei der Generierung des Testfalls erzeugt. Die Spezifikation einer konkreten Überdeckung sowie einer konkreten Instanz aus der abstrakten Methode kann zunächst getrennt voneinander betrachtet werden. Im Folgenden werden die Sprachkonstrukte für die Spezifikation der algorithmischen Beschreibung aus der abstrakten Methode und anschließend die Modellierung der Überdeckungskriterien vorgestellt.

5.7.2 Modellierung der abstrakten Methoden

In Abschnitt 5.5 wurden für die dynamischen Validierungsmethoden stets die abstrakten Methoden angegeben. In einer Testfallspezifikation muss eine abstrakte Methode durch einen Ablaufplan aus den Instruktionen des spezifizierten Prozessors und entsprechend der Mikroarchitektur beschrieben werden. Für die verwendeten Instruktionen muss die Signatur der Instruktionen, z. B. die Anzahl der Operatoren, die Benutzungsrichtung usw., berücksichtigt werden. Für die Berücksichtigung der Mikroarchitektur muss der Ablaufplan der Testfallspezifikation an die Anzahl der Pipelinestufe angepasst werden. Im Folgenden wird die Vorgehensweise zur Bildung eines solchen Ablaufplans am Beispiel der Validierung von Datenkonflikten beschrieben.

In Abschnitt 5.5.6 wurde bei der Beschreibung der Validierungsmethoden für die Pipelinekonflikte die abstrakte Methode zur Validierung von Datenkonflikten beschrieben. Die abstrakte Methode für die Validierung der Datenkonflikte sieht vor, dass bei der Validierung die Instruktionen des Prozessors paarweise ausgeführt werden. Die Operatoren der Instruktionen müssen mit Datenabhängigkeiten instanziiert werden. Damit werden Datenkonflikte zwischen Instruktionspaaren provoziert. Für die Erzeugung der Referenzergebnisse werden die gleichen Instruktionspaare mit denselben Parametern im Abstand der Pipelinelänge angeordnet. Für die Validierung der Bypass-Struktur werden auch die Instruktionspaare ohne Datenkonflikte in der abstrakten Methode berücksichtigt. Damit soll die unerwünschte Weiterleitung der Werte validiert werden. Die abstrakte Methode beschreibt, dass die Validierung auf alle Instruktionspaare des Prozessors angewendet werden muss. Für die Erfüllung dieser Bedingung wird in der Testfallspezifikation, z. B. für Instruktionen mit unterschiedlichen Instruktionsformaten, der Ablaufplan für die Validierung separat erzeugt. Analog wird je nach Länge der Pipeline eine entsprechende Anzahl von Testfällen erzeugt, um Datenkonflikte bei verschiedenen Abständen zwischen den Instruktionen in der Pipeline zu überprüfen.

Der Ablaufplan wird beschrieben, indem für die einzelnen Instruktionen Schritte angegeben werden. In Abbildung 5.21 werden die Instruktionspaare für die Validierung durch die Schritte mit den Bezeichnungen `first` und `second` angegeben. Die Schritte werden bei der Generierung der Testfälle durch die Instruktionen aus der Überdeckungsmenge ersetzt.

Die Struktur für die Schritte wird aus den Instruktionsformaten der Instruktionen aus der Prozessorspezifikation abgeleitet. Die Schritte werden für die Übersichtlichkeit der Spezifikation benannt und enthalten Platzhalter für die Verknüpfungen zu den Instruktionsmengen oder den Mengen von Parametern. Damit kann die abstrakte Methode für die Validierung unabhängig von den Überdeckungskriterien beschrieben werden. In Abbildung 5.21 wird beispielhaft dargestellt, wie der Ablaufplan für die Validierung der Datenkonflikte für zwei Instruktionsgruppen beschrieben wird.

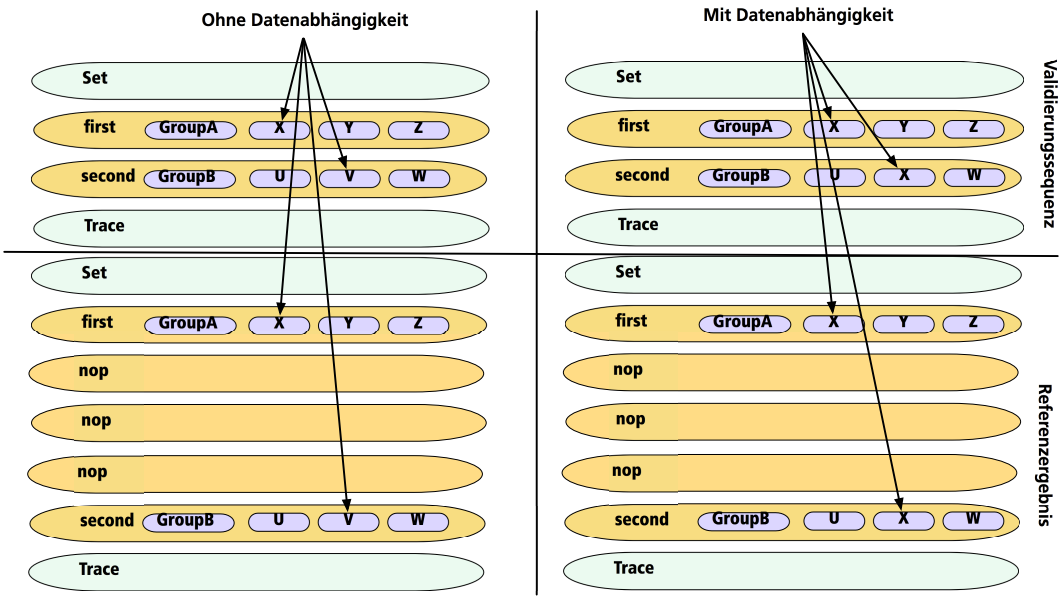


Abbildung 5.21: Ablaufplan einer Testfallspezifikation.

Neben der Spezifikation der Schritte für die Instruktionen wird in der Testfallspezifikation das Setzen oder Initialisieren des Prozessorzustandes durch `Set` und das Aufzeichnen des Systemzustandes durch `Trace` beschrieben. Diese Befehle gehören nicht zu dem Instruktionssatz des Prozessors und werden in *ViCE-UPSLA* automatisch generiert. Die aufgezeichneten Systemzustände durch `Trace` Befehle werden für die Auswertung der Validierungsergebnisse verwendet.

5.7.3 Modellierung der Überdeckung

Für die Erzeugung einer konkreten Testfallspezifikation wird neben der Ablaufstruktur die Überdeckung der Prozessorkonstrukte benötigt. Die Überdeckung der Konstrukte wird in Mengen ausgedrückt. Bei der automatischen Generierung werden die Mengen der Prozessorkonstrukte oder die Wertebereiche für die Operanden für die Validierung aus der Prozessorspezifikation ermittelt.

In Abbildung 5.22 werden zwei Schritte aus der Ablaufstruktur und die Spezifikation der Mengen angegeben. Dabei werden für den Schritt `first` die arithmetischen Instruktionen des Prozessors vorgegeben, während für den Schritt `second` die Lade- und Speicher-Instruktionen angegeben werden. Diese Testfallspezifikation gibt an, dass bei der Generierung eines Testfalls mit dem vorgegebenen Ablaufplan alle Kombinationen aus den Instruktionen erzeugt werden: `add-ldr`, `sub-ldr`, `add-ld` usw. Für die

Parameter der Instruktionen werden `X` und `Y` durch Wertebereiche definiert, während die Parameter `U` und `Z` durch Einermengen beschrieben sind. Damit wird spezifiziert, dass die Parameter `U` und `Z` für alle Wiederholungen des Ablaufplans nicht verändert werden, während für Parameter `X` und `Y` alle Kombinationen aus den Elementen der Menge in dem Testfall erzeugt werden.

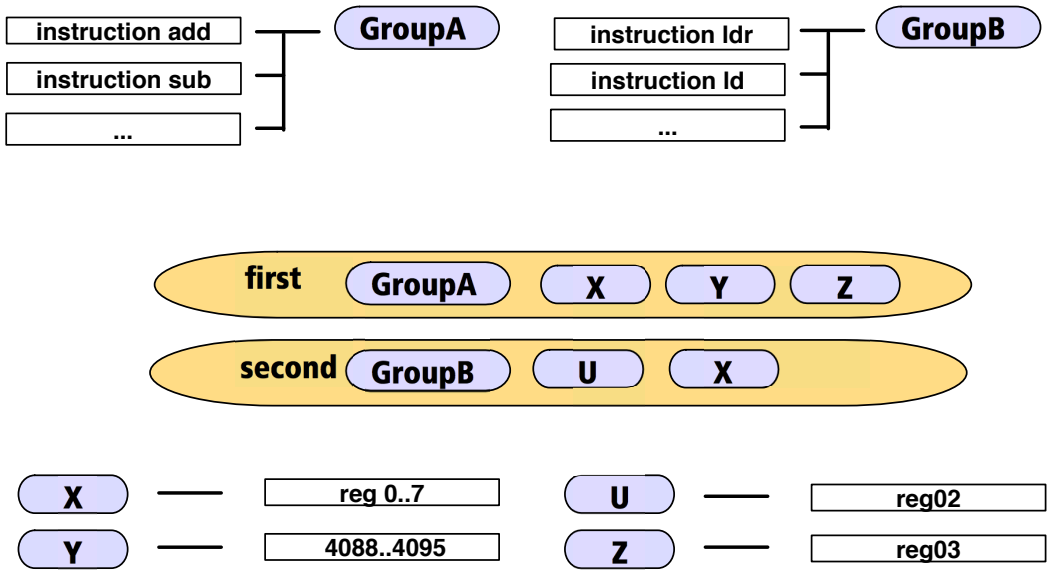


Abbildung 5.22: Überdeckung für die Repräsentanten der algorithmischen Beschreibung.

Damit ist eine Testfallspezifikation vollständig angegeben und kann zur Generierung von Testfällen benutzt werden. Die Funktionsweise des Generators wird in Kapitel 6 beschrieben.

5.7.4 Bearbeitung der Testfallspezifikation

Die Komponenten der Testfallspezifikation wurden in den vorangegangenen Abschnitten beschrieben. Die Spezifikation für die Testfälle wird in einer visuellen Darstellung auf einem hohen Abstraktionsniveau beschrieben. Damit die Testfallspezifikation automatisch aus der Prozessorspezifikation abgeleitet werden kann, ist eine enge Kopplung zu den Konstrukten von *ViCE-UPSLA* notwendig. Hierfür wird die Sprache für die Testfallspezifikation in das Werkzeugsystem integriert. Damit kann der Benutzer für die Beschreibung oder Bearbeitung einer Testfallspezifikation direkt auf die Konstrukte aus der Prozessorspezifikation zugreifen.

Für die Spezifikation eines Schrittes in *TFS* können die Instruktionsformate verwen-

det werden, womit die Anzahl der Operanden automatisch übernommen wird. Damit werden bereits bekannte Komponenten aus dem Prozessorentwurf wiederverwendet.

Die Spezifikation der Überdeckungsmengen wird durch eine Menge von Automatismen dem Prozessorentwickler komfortabel angeboten, indem z. B. die Spezifikation einer Überdeckungsmenge durch abstrakte Instruktionen oder Instruktionsformate ausgedrückt wird. Sind z. B. die arithmetischen Instruktionen eines Prozessors in einer abstrakten Instruktion zusammengefasst und wird diese als eine Gruppe in der Testfallspezifikation verwendet, werden alle darin enthaltenen Instruktionen in der Überdeckungsmenge aufgenommen. Analog können Wertebereiche oder Register zu den Mengen der Operanden hinzugefügt werden.

Basierend auf den vorgestellten Werkzeugen kann der Prozessorentwickler die automatisch erzeugten Testfallspezifikationen bearbeiten. Zum einen kann der Benutzer die Ablaufstruktur der Testfälle nach seinen Vorstellungen ändern. Zum anderen können die Überdeckungsmengen nach den Vorstellungen des Prozessorentwicklers angepasst werden. Darüber hinaus ist die Spezifikation eigener Testfälle für die Validierung und Evaluierung von Prozessoren mit der Spezifikationssprache *TFS* für die Testfallspezifikation möglich.

6 Generatoren

In diesem Kapitel werden die Generatoren des Werkzeugsystems vorgestellt. Der Simulatorgenerator wird dazu eingesetzt, aus einer Prozessorspezifikation in *ViCE-UPSLA* einen Simulator zu erzeugen. Der Testfallgenerator erzeugt aus derselben Prozessorspezifikation geeignete Testfälle für die dynamische Validierung der Prozessorspezifikation.

In Abschnitt 2.4.1 wurden verschiedene Simulationsebenen beschrieben. Bei der Beschreibung der Sprache *ViCE-UPSLA* wurden damit zwei Szenarios für die Spezifikation eines Instruktionssatz- und Mikroarchitektursimulators in Abschnitt 4.3 vorgestellt. Mit dem Simulatorgenerator können die benannten Simulatoren, wie in Abschnitt 2.4.1 zu den verschiedenen Ebenen bei der Simulation beschrieben, automatisch generiert werden. Für die Beschreibung des Generators in Abschnitt 6.1 wird zunächst die Vorgehensweise in *ViCE-UPSLA* für die Erzeugung eines kompilierenden Simulators erklärt. Die verschiedenen Arten der Umsetzung von Simulatoren wurden in Abschnitt 2.4.2 beschrieben. Anschließend wird auf die Generierung eines Instruktionssatzsimulators und eines Mikroarchitektursimulators in Abschnitt 6.1 eingegangen.

In den Abschnitten 5.5 und 5.7 wurden die dynamischen Validierungsmethoden und die visuelle Sprache zur Beschreibung von Testfallspezifikationen vorgestellt. Die Testfallspezifikation selbst wird aus der Prozessorspezifikation generiert oder durch den Prozessorentwickler erzeugt. Die Generierung von Testfallspezifikationen wird in Abschnitt 6.2 beschrieben.

6.1 Simulatorgenerator

Mit *ViCE-UPSLA* werden aus der Prozessorspezifikation kompilierende Simulatoren [ZTM95] generiert. Bei der Erzeugung eines kompilierenden Simulators wird ein dedizierter Simulator für ein spezielles Programm erzeugt, wie in Abschnitt 2.4.2 beschrieben. Dieser Ansatz zeichnet sich durch hohe Simulationsgeschwindigkeit und flexible Ausdrucksmöglichkeit für die Genauigkeit der Simulation aus.

In Abbildung 6.1 wird das Prinzip für die Erzeugung eines Simulators mit *ViCE-UPSLA* dargestellt. Aus der Prozessorspezifikation wird eine Bibliothek des Prozessors generiert. Die Bibliothek enthält die Definitionen für die Konstrukte des Prozessors, wie Instruktionen oder Adressierungsmethoden, in denen die Ergebnisse berechnet werden. Dabei werden die Instruktionen in Fragmente entsprechend der Prozessorspezifikation zerlegt und enthalten die jeweiligen Makrodefinitionen für die Ausführung. Für die Beschreibung der Prozessorzustände werden Datenstrukturen für den Registersatz des

Prozessors in der Bibliothek erzeugt. Darüber hinaus enthält die Bibliothek das Pipelineverhalten und die Abbildung der Ressourcen des Zielprozessors. Die ausführliche Beschreibung der Prozessorbibliothek wird in Abschnitt 6.1.1 vorgestellt.

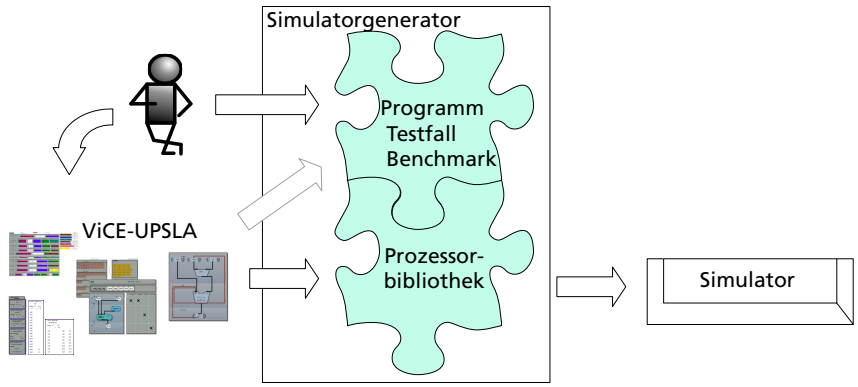


Abbildung 6.1: Simulatorgenerator.

Der Simulatorgenerator wird mit der Bibliothek des Prozessors parametrisiert und erhält als Eingabe den Quellcode eines Programms für den spezifizierten Prozessor. Damit wird ein dedizierter Simulator für das Programm erzeugt. Bei der Generierung des Simulators werden die Befehle des Programms durch die Anweisungen aus den Makrodefinitionen ersetzt, welche die Instruktionen des Prozessors repräsentieren. Für die Simulation von Prozessoren mit einer vorgegebenen Mikroarchitektur wird die Ausführung von Instruktionen überlappt. Die Fragmente aufeinander folgender Instruktionen werden verschränkt angeordnet und simulieren damit die Ausführung von Instruktionen in einer Pipeline. Die Vorgehensweise des Generators bei der Verteilung der Ressourcen einer Mikroarchitektur und für die Überlappung der Instruktionen wird in Abschnitt 6.1.2 ausführlich erläutert. Der damit erzeugte Simulator imitiert das Verhalten des Prozessors bei der Ausführung eines bestimmten Programms.

In Abschnitt 6.1.1 wird zunächst die Erzeugung der Prozessorbibliothek beschrieben. Dabei wird an einem Beispiel erklärt, wie die Prozessorkonstrukte aus der Spezifikation in der Bibliothek umgesetzt werden. Anschließend wird die Generierung eines Simulators und die Umsetzung der Simulation für das Verhalten einer vorgegebenen Mikroarchitektur erklärt.

6.1.1 Prozessorbibliotheksgenerator

Die Prozessorbibliothek wird als zentrale Komponente für die Erzeugung eines Prozessorsimulators eingesetzt. Ein Generator übersetzt die Komponenten der visuellen Spezifikation des Prozessors in eine für den Simulatorgenerator benötigte Repräsentation,

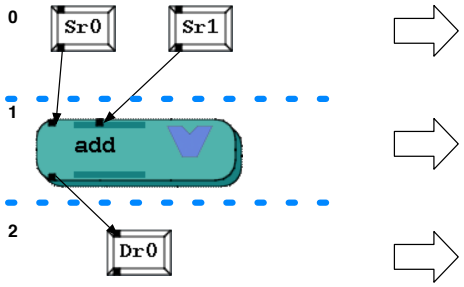
die bei der Erzeugung von Simulatoren verwendet wird. Dabei werden unterschiedliche Dokumente aus der Prozessorspezifikation erzeugt, die in unterschiedlichen Phasen des Simulatorgenerators eingesetzt werden.

Der Generator erzeugt eine Bibliothek des Prozessors, die in drei Teilen betrachtet werden kann, welche für die Generierung eines Simulators in Abschnitt 6.1.2 in drei Schritten nacheinander verwendet werden. Aus der Prozessorspezifikation werden die strukturellen, die funktionalen und die Verhaltenskomponenten extrahiert. Die strukturellen Komponenten definieren z. B. die Struktur und die Ableitung der Instruktionen aus dem Assemblercode des Programms für den Simulator. Die funktionalen Komponenten definieren die Makrodefinitionen und die Funktionen, die bei der Umsetzung der Instruktionen im Simulator verwendet werden. Die Verhaltenskomponenten beschreiben das Verhalten der Mikroarchitektur, z.B wie die Instruktionen überlappt werden und welche Ressourcen bei der Simulation belegt oder benötigt werden. Bei der Erzeugung des Simulators wird anhand dieser Komponenten das Verhalten der Mikroarchitektur aus der Prozessorspezifikation nachgebildet. Im Folgenden wird die Generierung der einzelnen Komponenten erläutert.

Bei der Generierung der strukturellen Bibliothek werden die Instruktionen in der Prozessorspezifikation betrachtet. Dabei werden für jede Instruktion anhand der Verknüpfungen aus der Spezifikation ihre strukturellen Eigenschaften ermittelt und in einer Definition zusammengetragen. Z. B. werden für jede Instruktion aus der Zuordnung des Instruktionsformats der Assemblerbefehl und die Adressierungsmethoden der Operanden in die Bibliothek übernommen. Damit beschreibt die strukturelle Bibliothek durch Definitionen, wie die Befehle des zu simulierenden Programms in die Instruktionen des Prozessors übersetzt werden.

Die operationalen Beschreibungen der Instruktionen spezifizieren in der Prozessorspezifikation das Verhalten der Instruktionen. Hierfür werden in der funktionalen Bibliothek des Prozessors für jede Instruktion die Makrodefinitionen und Funktionen aus der operationalen Beschreibung abgeleitet. Dabei werden z. B. aus den interpretierten Operanden der Instruktionen die Anweisungen in den Makrodefinitionen und die Ein- und Ausgabeparameter für die Funktionen festgelegt. Für die Instruktionen mit mehreren Ausführungsschritten wird die entsprechende Anzahl von Makrodefinitionen für die jeweiligen Ausführungsschritte erzeugt. Abbildung 6.2 zeigt links die operationale Beschreibung einer Instruktion in *ViCE-UPSLA* und rechts die daraus erzeugten Makrodefinitionen in der Bibliothek des Prozessors. Die operationale Beschreibung der Instruktion in dem vorliegenden Beispiel wird dabei den Ausführungszyklen der Instruktion entsprechend in drei Fragmente aufgeteilt. Die Adressierungsmethoden werden analog zu den Instruktionen als Makrodefinitionen und Funktionen in der Bibliothek des Prozessors umgesetzt. Bei der Erzeugung des Simulators werden die Funktionen und die Makrodefinitionen durch die entsprechenden Funktionsaufrufe oder Makros

verwendet.




```
#define simi_add_0(name, cycle, closureid,Cond , Dr0 , Sr0 , Sr1)\
  sim_mach_pre_instr_0(name); \
  sim_mach_state->b_c##closureid = Sr0;  \
  sim_mach_state->k_c##closureid = sim_mach_state->b_c##closureid;\
  sim_mach_state->c_c##closureid = Sr1; \
  sim_mach_state->j_c##closureid = sim_mach_state->c_c##closureid;  \
  sim_mach_post_instr_0(name)
 ----------------------------------------------------------------------
#define simi_add_1(name, cycle, closureid,Cond , Dr0 , Sr0 , Sr1)\
  sim_mach_pre_instr_1(name); \
 ( sim_mach_state->e_c##closureid ) = \
  add( sim_mach_state->j_c##closureid , sim_mach_state->k_c##closureid );\
  sim_mach_post_instr_1(name)
 ----------------------------------------------------------------------
#define simi_add_2(name, cycle, closureid,Cond , Dr0 , Sr0 , Sr1)\
  sim_mach_pre_instr_2(name); \
  sim_mach_state->a_c##closureid = sim_mach_state->e_c##closureid;\
  Dr0 = sim_mach_state->a_c##closureid;   \
  sim_mach_post_instr_2(name)
```

Abbildung 6.2: Makrodefinitionen der operationalen Beschreibung einer Instruktion in der Prozessorbibliothek.

Der Registersatz einer Prozessorspezifikation wird dazu genutzt, die Werte bei der Ausführung der Programme zu speichern oder zur Verfügung zu stellen. Dazu wird der Registersatz in der Prozessorbibliothek in einer Datenstruktur beschrieben, in der alle Registerbänke und Speicher definiert werden. Die Datenstruktur definiert die Struktur für die Systemzustände des Prozessors bei der Ausführung des Simulators. In der Prozessorbibliothek werden Funktionen abgelegt, welche das Setzen der Struktur oder das Aufzeichnen des Strukturinhalts ermöglichen. Diese Funktionen werden z. B. bei der dynamischen Validierung verwendet.

Der dritte Teil der Prozessorbibliothek beschreibt durch Datenstrukturen die Definition für das Verhalten der Mikroarchitektur. Für die Simulation des Verhaltens wird aus der Prozessorspezifikation ein Ressourcenplan für die einzelnen Pipeline-Stufen der Mikroarchitektur erzeugt. Für die Instruktionen des Prozessors werden aus der Prozessorspezifikation die Ressourcennutzung und die Pipeline-Stufen für die jeweiligen Ausführungszyklen erzeugt. Bei der Erzeugung eines Simulators beschreibt der Ressourcenplan die vorhandenen Ressourcen und die Ressourcen-Nutzung der für die Ausführung einer Instruktion benötigten Ressourcen. Dabei wird durch die entsprechenden Funktionen im Generator die Ressourcen-Nutzung für die Fragmente der Instruktionen ermittelt und in dem Ressourcenplan belegt und wieder freigegeben. Der Simulatorgenerator verwendet diese Komponenten, um z. B. Interlocks oder Überlappungen von Instruktionen zu simulieren.

In Abbildung 6.3 sind ein Ressourcenplan für eine einfache Mikroarchitektur mit einer vierstufigen Pipeline und die Ressourcenbelegung von zwei Instruktionen `ADD` und `STR` dargestellt. Beide Instruktionen benötigen drei Taktzyklen bei der Ausführung und belegen dabei zum Teil unterschiedliche Ressourcen und Pipeline-Stufen der

Mikroarchitektur.

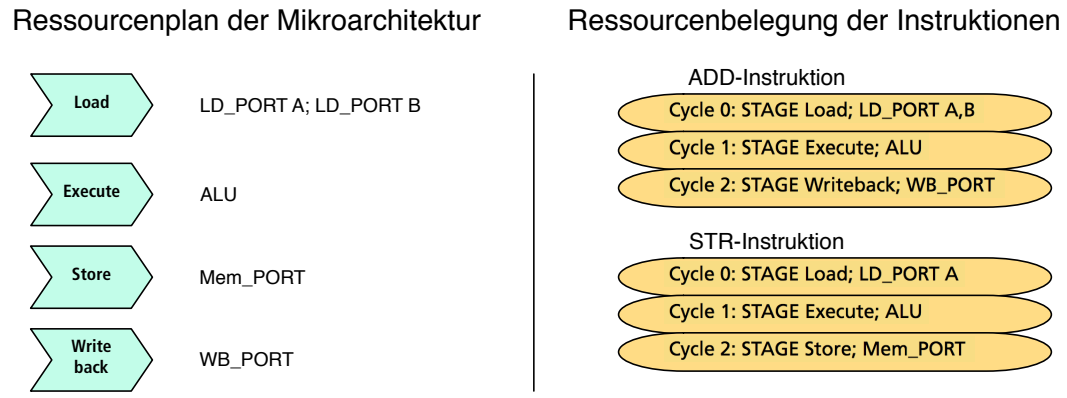


Abbildung 6.3: Ressourcenplan der Mikroarchitektur und Ressourcenbelegung zweier Instruktionen.

6.1.2 Generierungsprozess

Die Sprache *ViCE-UPSLA* wurde so entworfen, dass der Prozessorentwickler Entwürfe mit und ohne Beschreibung einer Mikroarchitektur erzeugen kann. Wird z. B. in einer Entwicklungsphase nur ein Instruktionssatzsimulator benötigt, wie in Abschnitt 2.4 beschrieben, kann aus der Prozessorspezifikation ohne Angabe einer Mikroarchitektur ein Simulator erzeugt werden. Aus einer Prozessorspezifikation mit einer vorgegebenen Mikroarchitektur können wahlweise beide Arten von Simulatoren generiert werden. Bei der Validierung der Prozessorspezifikation können beide Simulatoren eingesetzt werden, um z. B. die Pipelinekonflikte mit dem Mikroarchitektursimulator zu untersuchen, indem der Instruktionssatzsimulator die Referenzergebnisse erzeugt. Bei der Erzeugung der Mikroarchitektursimulatoren können Eigenschaften wie Interlocks oder Bypässe optional spezifiziert werden.

In diesem Abschnitt werden die Schritte für die Erzeugung eines Mikroarchitektursimulators beschrieben. Dazu werden die in Abschnitt 6.1.1 beschriebenen Bibliotheken verwendet. Abbildung 6.4 zeigt schematisch eine Übersicht der Transformationsschritte des Generators. Bei der Erzeugung eines Simulators wird, wie bereits beschrieben, ein konkretes Programm verwendet, dessen Verhalten bei der Ausführung im Prozessor simuliert wird. Das Programm muss als Assemblercode für den Zielprozessor vorliegen.

Im ersten Schritt wird der Assemblercode in den strukturierten Quellcode des Prozessors überführt. Hierfür wird die strukturelle Bibliothek eingesetzt, die die Kodierung der Instruktionen, die Benutzungsrichtung der Operanden und deren Adressierungs-

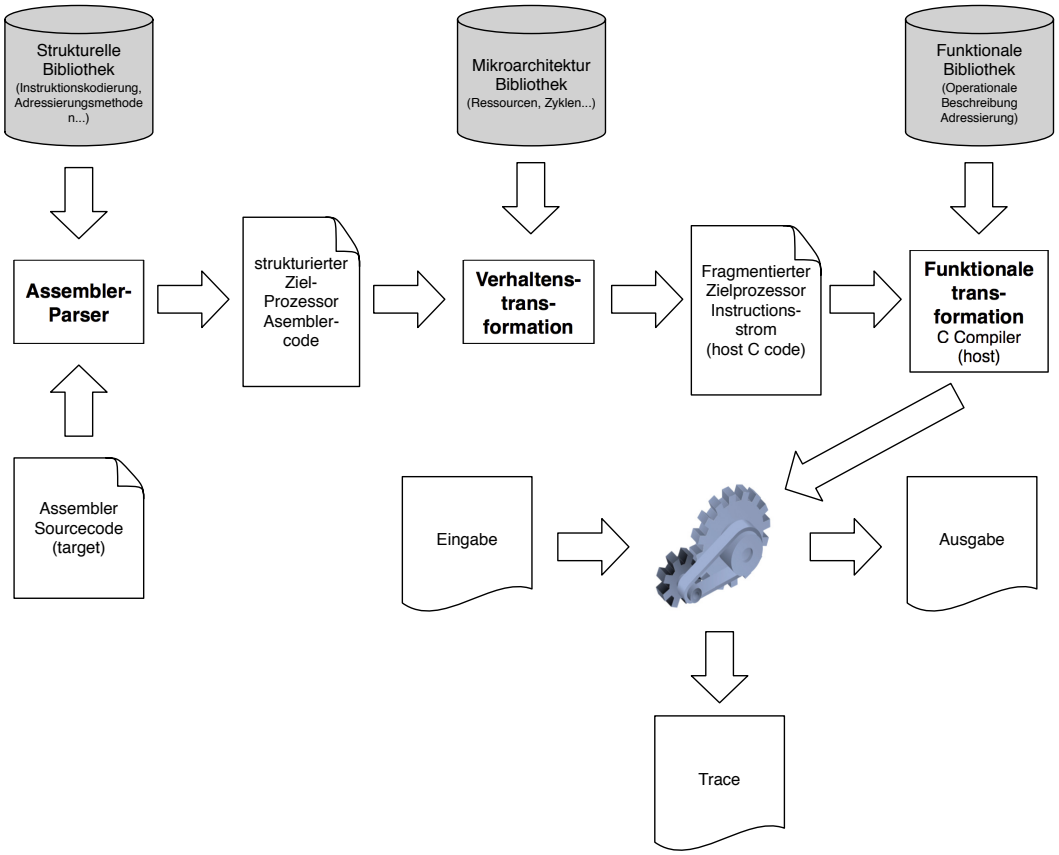


Abbildung 6.4: Generierungsschritte eines Mikroarchitektursimulator.

methoden enthält. Bei der Transformation werden die Assemblerbefehle durch die Instruktionen des Prozessors ausgedrückt. Dabei werden z. B. für die Operanden der Instruktionen die Adressierungsmethoden bestimmt. Die mit *ViCE-UPSLA* erzeugten Simulatoren werden mit zusätzlichen Funktionen ausgestattet, die für die Validierung verwendet werden. Dazu zählen z. B. das Setzen und das Auslesen der Prozessorzustände. Die Aufrufe dieser Funktionen können durch nicht prozessorspezifische Befehle im Assemblercode integriert werden. Bei der Erzeugung der Testfälle werden die Befehle aus der Testfallspezifikation automatisch in den Assemblercode integriert. Im Assembler-Parser werden die Befehle für die Validierung in Funktionsaufrufe mit den entsprechenden Parametern für die Datenstruktur des Registersatzes übersetzt.

Die Assemblerbefehle wurden im ersten Transformationsschritt in eine Instruktions-

folge übersetzt, die als Eingabe für den zweiten Transformationsschritt verwendet wird. Bei der Erzeugung eines Instruktionssatzsimulators wird dieser Transformationsschritt übersprungen. Der zweite Transformationsschritt beschreibt die Verhaltenstransformation. In der Bibliothek für die zweite Phase sind die Komponenten für das Verhalten der Mikroarchitektur enthalten, wie in Abschnitt 6.1.1 bereits beschrieben. In Abbildung 6.5 wird die Struktur der Bibliothek und die Ein- und Ausgabe des zweiten Transformationsschritts dargestellt. Dabei werden als Eingabe die Fragmente der Instruktionen `add`, `mul` und `or` dargestellt. In der Bibliothek werden die Ressourcen-Nutzung dieser Instruktionen und der Ressourcenplan der Mikroarchitektur des Prozessors abgebildet. Als Ausgabe wird die Anordnung der Instruktionsfragmente für die Simulation des Verhaltens einer Mikroarchitektur mit Interlock-Mechanismus angegeben.

In der Spezifikation der Mikroarchitektur werden in *ViCE-UPSLA* die Ressourcen angegeben und mit den Knoten der operationalen Beschreibung der Instruktionen verknüpft. Diese Spezifikation ermöglicht es, Mikroarchitekturen mit Interlocks zu simulieren. Dabei wird bei der Anordnung der Instruktionsfragmente neben der Pipelinestruktur auch die Verfügbarkeit der Ressourcen geprüft. Hierfür wird die Datenstruktur für den Ressourcenplan aus der Verhaltensbibliothek verwendet, in der die Ressourcen quantitativ angegeben sind. Bei der Anordnung der Fragmente wird stets geprüft, ob die Ressourcen für ein Fragment einer Instruktion verfügbar sind. Außerdem wird geprüft, welche Fragmente der Instruktionen in einem Taktzyklus ausgeführt werden können. In Abbildung 6.5 wird in der Ausgabe aus dem Transformationsschritt im Zyklus 3 das Verhalten des Interlock-Mechanismus umgesetzt. Dabei werden die vorletzten Ausführungsschritte der Instruktionen `mul` und `or` parallel angeordnet. Beide Instruktionen können jedoch nicht im nächsten Taktzyklus ihren letzten Schritt ausführen, da die Ressource `wb-Port` nur einmal zur Verfügung steht. Die Instruktion `or` wird für einen Takt angehalten.

Im letzten Transformationsschritt werden die Fragmenten der Instruktionen durch die Anweisungen aus den Makrodefinitionen ersetzt. Die Makrodefinitionen für die einzelnen Fragmente der Instruktionen sind in der funktionalen Bibliothek, erzeugt aus der Prozessorspezifikation durch den Prozessorbibliotheksgenerator, enthalten.

Mit den beschriebenen Transformationsschritten wird somit für ein gegebenes Programm ein ausführbarer Simulator für einen spezifizierten Prozessor erzeugt. Mit der entsprechenden Eingabe für das Programm wird die Ausgabe des Prozessors erzeugt. Außerdem können bei der Ausführung die Systemzustände des Prozessors aufgezeichnet werden. Die Aufzeichnung der Systemzustände ermöglicht das Nachvollziehen der Ausführung eines Programms im Prozessor während der Simulation und kann für die Validierung der Programme oder der Prozessorspezifikation genutzt werden.

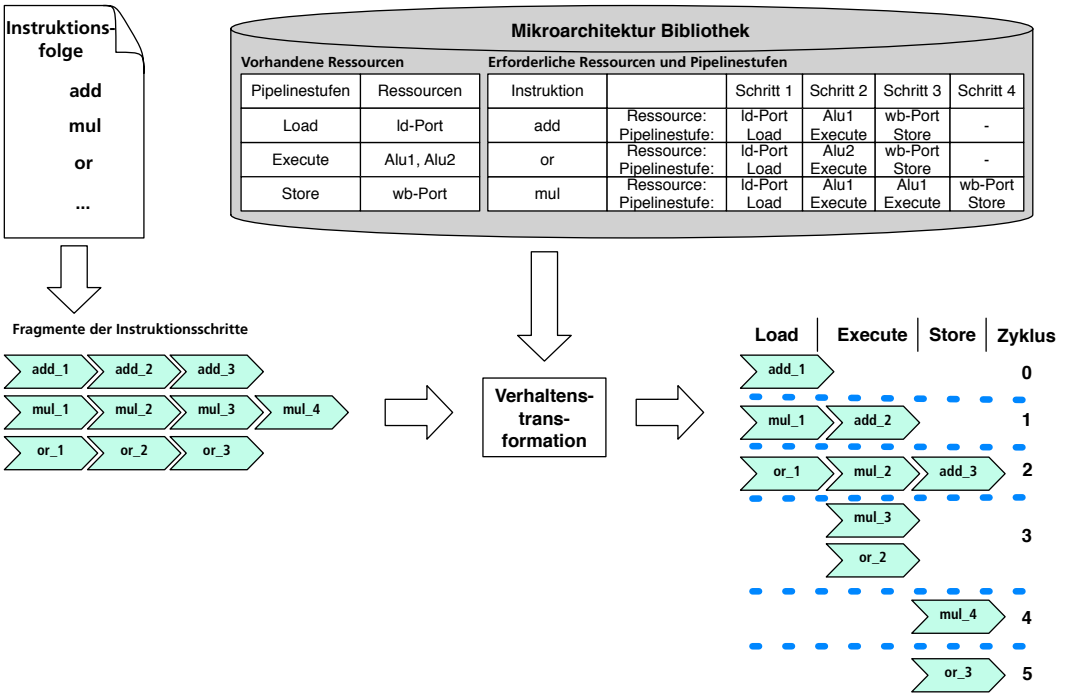


Abbildung 6.5: Überlappung der Instruktionsausführung für die Simulation des Verhaltens einer Mikroarchitektur.

6.2 Testfallgenerator

Eine dynamische Validierungsmethode beschreibt durch die abstrakte Methode und die Überdeckungskriterien die allgemeine Vorgehensweise für die Validierung einer Eigenschaft des Prozessors und die Strategie für die Ermittlung der Komponenten für die Validierung. Die Aufgaben der Testfallspezifikation im Validierungsprozess wurden in Abschnitt 5.7 erläutert. Dabei wurde herausgestellt, dass die Testfallspezifikation durch einen Ablaufplan und eine Mengenbeschreibung angegeben wird. Die Aufgabe des Testfallgenerators ist es, die beschriebenen Validierungsmethoden aus Abschnitt 5.5 umzusetzen und die Testfallspezifikationen aus der Analyse der Prozessorstruktur zu erzeugen. Für die Beschreibung der Vorgehensweise bei der Generierung einer Testfallspezifikation werden zwei Validierungsmethoden als Beispiele verwendet. Dazu werden die Validierung der Erreichbarkeit von Registern und die Validierung der Datenkonflikte betrachtet und im Folgenden kurz beschrieben.

Die Validierungsmethode für die Erreichbarkeit der Register wurde in Abschnitt 5.5.2 vorgestellt. Die Methode gibt an, dass für die Validierung der Register die Adressierungsmethoden des Prozessors überdeckt werden müssen. Damit wird ermittelt, ob alle Register durch die spezifizierten Komponenten im Prozessor erreicht werden.

Für die Validierung der Register muss im ersten Schritt aus der Prozessorspezifikation anhand der Adressierungsmethoden des Prozessors die konkrete Menge der Instruktionen für die Validierung ermittelt werden. Im zweiten Schritt müssen ausgehend von der Menge der Instruktionen die Ablaufschritte der abstrakten Methode entsprechend für die verwendeten Instruktionen erzeugt werden. Dabei müssen die `Set` und `Trace` Befehle in den Ablaufplan integriert werden. Damit erzeugte Ablaufpläne bestehen aus Instruktionen und Befehlen zur Aufzeichnung oder Initialisierung der Systemzustände. Im letzten Schritt müssen die Wertebereiche für Operanden der Instruktionen aus der Prozessorspezifikation ermittelt und übernommen werden. Damit ist die Testfallspezifikation vollständig beschrieben und kann für die Generierung eines Testfalls verwendet werden.

Für die Validierung der Datenkonflikte in der Pipeline müssen die Instruktionen des Prozessors paarweise und mit unterschiedlichen Abständen in der Pipeline ausgeführt werden. Wie in Abschnitt 5.5.6 beschrieben, muss im ersten Schritt die Menge der Instruktionspaare aus der Spezifikation ermittelt werden, zwischen denen Datenkonflikte entstehen können. Für die Erzeugung des Ablaufplans muss die Tiefe der Pipeline aus der Spezifikation des Prozessors berücksichtigt werden. Für die Instruktionspaare müssen die entsprechenden Ablaufschritte mit unterschiedlichen Abständen in der Pipeline erzeugt werden. Außerdem müssen bei der Beschreibung des Ablaufplans die Befehle zur Aufzeichnung der Systemzustände eingefügt werden. Im letzten Schritt werden die Operanden der Instruktionen mit Wertebereichen versehen, wobei diese für die Validierung der Datenkonflikte generisch angegeben werden, womit Testfälle mit und ohne Datenkonflikte beschrieben werden.

Die Generierung der Testfallspezifikationen für die unterschiedlichen Validierungsmethoden aus Abschnitt 5.5 wird nach demselben Schema, wie in Abbildung 6.6 dargestellt, durchgeführt. Die Erzeugung einer Testfallspezifikation erfolgt dabei in drei Schritten.

Im ersten Schritt wird die Prozessorspezifikation analysiert und anhand der Überdeckungskriterien die Menge der Instruktionen für die Validierung festgelegt. Die bei der Strukturanalyse ermittelte Überdeckungsmenge wird als Eingabe für den zweiten Schritt verwendet. Die Strukturanalyse der Spezifikation ist für die verschiedenen Validierungsmethoden unterschiedlich, da jeweils unterschiedliche Konstrukte der Spezifikation validiert werden.

Im zweiten Schritt wird der Ablaufplan für die Instruktionen der Überdeckungsmenge aus dem ersten Schritt erzeugt. Aus der Beschreibung der abstrakten Methoden

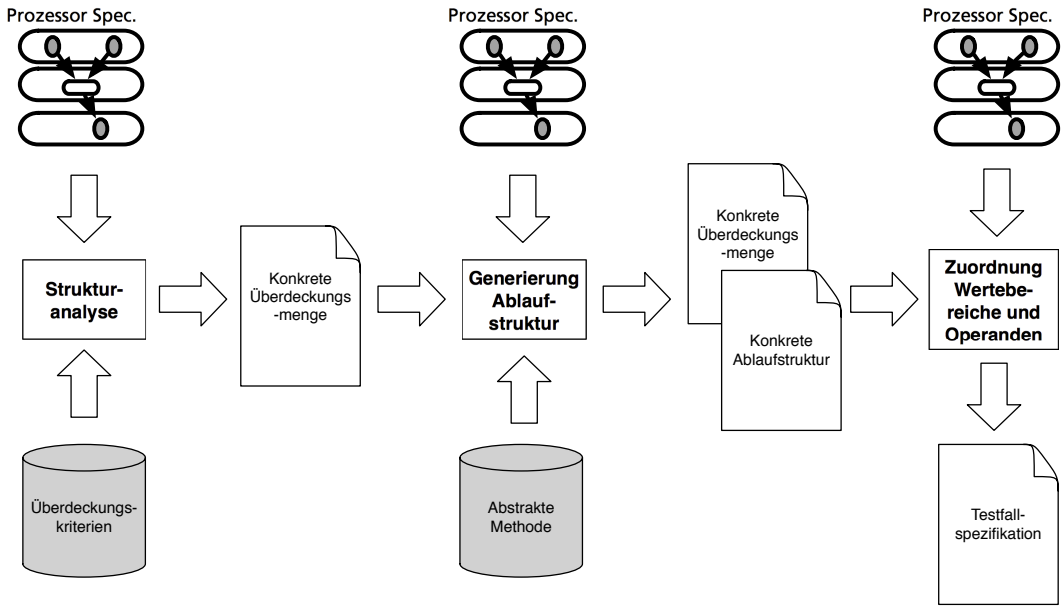


Abbildung 6.6: Testfallspezifikationsgenerator.

wird ein Generator implementiert, der eine Ablaufstruktur aus der Überdeckungsmenge erzeugt. So wie die Überdeckungskriterien sind auch die abstrakten Methoden für die verschiedenen Validierungsmethoden unterschiedlich. Die Parser für die Erzeugung verschiedener Ablaufpläne werden für die verschiedenen Validierungsmethoden separat implementiert.

Nach dem zweiten Schritt des Generators sind die Überdeckungsmengen für die Instruktionen und die Ablaufstrukturen für die Validierungsmethoden festgelegt. Im letzten Schritt werden die Wertebereiche für die Operanden der Instruktionen aus der Ablaufstruktur ermittelt und eingesetzt.

In Abschnitt 5.7 wurde die Sprache für die Beschreibung von Testfallspezifikationen bereits vorgestellt. Damit wird dem Prozessorentwickler ermöglicht, die Testfallspezifikation einzusehen oder zu bearbeiten. Die Generierung eines Testfalls aus einer Testfallspezifikation wird an dieser Stelle kurz erläutert. Die Konstrukte einer Testfallspezifikation können durch Sprachkonstrukte der funktionalen Sprache XQuery [LS04] abgebildet werden. Damit werden die Überdeckungsmengen durch Namensräume und die Ablaufstruktur durch Funktionen beschrieben. Die Testfallspezifikation wird in ein XQuery Programm übersetzt, wie in Abbildung 6.7 dargestellt. Abschlie-

ßend erzeugt der XQuery Interpreter aus dem XQuery Quellcode einen Testfall als Assembler-Quellcode für den Zielprozessor.

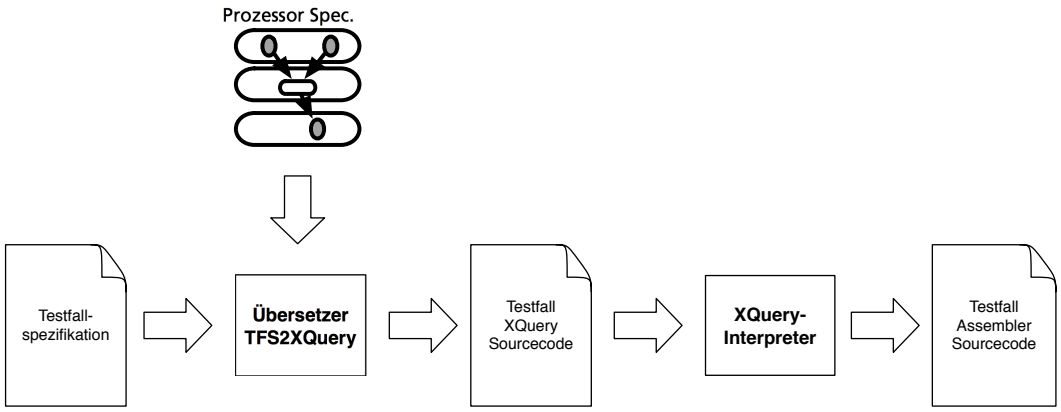


Abbildung 6.7: Testfallgenerator.

7 Evaluierung

Die Sprache *ViCE-UPSLA* wurde für die Unterstützung der Prozessorentwickler bei dem Entwurf neuer Prozessoren oder für die Szenarios der Entwurfsraumexploration oder Instruktionssatzerweiterung entworfen. Das Konzept der Sprache hat das Ziel, bei der Beschreibung von Prozessorkonstrukten durch domänenspezifische Visualisierung ein großes Spektrum von Prozessorarchitekturen zu erfassen. Für die Überprüfung der Entwürfe werden aus den Prozessorspezifikationen durch den Einsatz von Generatoren automatisch Simulatoren der spezifizierten Prozessoren erzeugt. Für die systematische Validierung der Entwürfe wurden statische und dynamische Validierungsmethoden in dem Werkzeugsystem integriert. Damit wird der Prozessorentwickler bei der Validierung der Entwürfe unterstützt.

Durch die Entwurfsziele und Anforderungen an das Werkzeugsystem erschließen sich drei Bereiche für die Evaluierung, bei denen die Sprache, die Simulation und die Validierung untersucht werden. Im Folgenden werden zuerst die Aufgaben und die Vorgehensweise bei der Evaluierung beschrieben. In Abschnitt 7.2 werden die bei der Evaluierung verwendeten Prozessorarchitekturen vorgestellt. Die Evaluierung der Sprache wird in Abschnitt 7.3 beschrieben. Anschließend folgen die Evaluierung der Validierungsmethoden in Abschnitt 7.4 und der Simulation 7.5.

7.1 Ziele der Evaluierung

Bei der Evaluierung müssen die visuelle Sprache, die erzeugten Simulatoren und die Validierungsmethoden betrachtet werden. Für die Durchführung der Evaluierung der einzelnen Bereiche wurden zwei reale Prozessoren spezifiziert und Simulatoren der Prozessoren erzeugt. Hierfür wurde der ARM RISC-Prozessor [Lim95, Lim87] und der CoreVA VLIW Prozessor [Jun11, JSPR10] in Abschnitt 2.3 bereits vorgestellt. In Abschnitt 7.2 werden die betrachteten Architekturen und Instruktionssätze der Prozessoren zusammengefasst. Damit soll gezeigt werden, dass die Sprache *ViCE-UPSLA* für die Spezifikation von realen RISC-Prozessoren geeignet ist. Durch die Spezifikation des CoreVA-Prozessors wird gezeigt, dass mit der Sprache spezifische Aspekte von SIMD oder VLIW Architekturen ebenfalls umgesetzt werden können. Für die Durchführung der Evaluierung der Sprache wird die Vorgehensweise bei der Spezifikation der CoreVA-Architektur in Abschnitt 7.2.2 schrittweise beschrieben. Dabei werden konkrete Stellen für die Anwendung der Validierungsmethoden lokalisiert und bei der Evaluierung des Validierungsprozesses erläutert. Durch diese Vorgehensweise kann der

Spezifikationsaufwand für die Entwurfsszenarios von der Beschreibung der Instruktionssatzerweiterungen bis zu der Entwurfsraumexploration bei der Evaluierung der Sprache aufgegriffen werden.

Das erste Ziel der Evaluierung ist die Analyse der Ausdrucksfähigkeit der Sprache *ViCE-UPSLA*. Hierfür wird die Evaluierung entlang der kognitiven Dimensionen aus der Arbeit von Thomas R.G.Green und Marian Petre „Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework“ [GP96] durchgeführt. Der Begriff der kognitiven Dimension sowie die verschiedenen Dimensionen von Green und Petre werden in Abschnitt 7.3 erläutert und auf die Sprachkonstrukte von *ViCE-UPSLA* angewendet.

Die Evaluierung der Validierungsmethoden in Abschnitt 7.4 soll die Effektivität und die Anwendbarkeit der Methoden aufzeigen. Für die Bewertung der Validierungsmethoden und der Umsetzung im Werkzeugsystem werden die erarbeiteten Ansätze von A.C.Dias Neto und G.H.Travassos [NT08b, NT08a] für die Bewertung von MBT-Werkzeugen verwendet. Dabei wird auf die Validierung der Spezifikationen des CoreVA- und ARM-Prozessors eingegangen. Abschließend werden beispielhaft realitätsnahe Fehler, wie in Abschnitt 5.2 beschrieben, in die Spezifikation eingestreut und die Validierung dieser Fehler beschrieben. Damit soll die Funktionsfähigkeit der Validierungsmethoden gezeigt werden.

Aus der Spezifikation von Prozessoren werden mit *ViCE-UPSLA* Mikroarchitektur- und Instruktionssatzsimulatoren generiert. Die Evaluierung der Simulatoren soll die Unterschiede in der Simulationsgeschwindigkeit für unterschiedliche Architekturen und Instruktionssätze zeigen. Das Augenmerk der Evaluierung wird auf die Simulationsgeschwindigkeit in Abhängigkeit von der Komplexität der Prozessorarchitektur und die Leistungsfähigkeit der generierten Simulatoren gerichtet. Für die Evaluierung werden für die Simulation einige Anwendungen, wie die Berechnung der Fibonacci-Zahlen oder die Multiplikation komplexer Zahlen, verwendet. Die ausgewählten Anwendungen spiegeln die typischen Berechnungsmuster für die Verarbeitung von verschiedenen Algorithmen in realer Software.

7.2 Spezifikation von Prozessoren

Die Mächtigkeit der Spezifikationssprache wird anhand der Implementierungen von realen Prozessoren gezeigt. Hierfür wurden zwei RISC-Architekturen, CoreVA und ARM, ausgewählt. Damit wird gezeigt, dass mit *ViCE-UPSLA* die verschiedenen Design-Philosophien wie SISD, SIMD und VLIW (vgl. Abschnitt 2.3) umgesetzt werden können. Für die Spezifikation der Prozessoren wurden die Beschreibungen der Prozessorarchitektur *Architekture Manual* [JD11] des CoreVA-Prozessors und die Instruktionssatzbeschreibung *ARM Data Sheet, Open Access* [Lim95] verwendet. Die Spezifikation der Prozessoren wird in Abschnitten 7.2.1 und 7.2.2 beschrieben.

Bei der Spezifikation der Prozessoren wurden einige Konstrukte der Architekturen nicht betrachtet, bzw. nur teilweise umgesetzt. Prozessorspezifische Einschränkungen, wie die Auswahl der Instruktionen für die Spezifikation des Instruktionssatzes, werden unmittelbar in Abschnitten zu den Prozessoren beschrieben. Die Einschränkungen durch die Sprache *ViCE-UPSLA*, welche für beide Architekturen im gleichen Maß gemacht wurden, werden im Folgenden zusammengefasst und begründet.

Eine der Eigenschaften von ARM und CoreVA ist die bedingte Ausführung von Instruktionen. Die Spezifikation der bedingten Ausführung wird in *ViCE-UPSLA* unterstützt. Dabei muss die bedingte Ausführung für jede Instruktion in der operationalen Beschreibung einzeln spezifiziert werden. Für die Umsetzung der bedingten Ausführung für alle Instruktionen eines Instruktionssatzes wird ein neues Sprachkonstrukt benötigt. Die Spezifikation der bedingten Ausführung wird nur für die Sprungbefehle der beiden Architekturen umgesetzt.

Des weiteren kann das zyklengenaue Verhalten von iterativ ausgeführten Instruktionen, wie z. B. `LDM` (Load Multiple) oder `STM` (Store Multiple), in *ViCE-UPSLA* noch nicht ausgedrückt werden. Die Ausführung solcher Instruktionen wird ohne Iterationen in der ALU ausgeführt.

Bei der Spezifikation der Sprunginstruktionen wurde nur auf bestimmte und absolute Sprunginstruktionen eingegangen. Die Spezifikation von relativen Sprüngen wird von den Sprachkonstrukten in *ViCE-UPSLA* nicht unterstützt. Die Implementierung von relativen Sprüngen ist durch die Art der kompilierenden Simulation nicht möglich.

ARM-Prozessoren unterstützen durch das `THUMB`-Konzept die Rekonfiguration des Instruktionssatzes. Diese Eigenschaft wird zur Codereduzierung genutzt, dabei wird zwischen dem `User`-Modus mit 32-Bit Instruktionssatz und dem `THUMB`-Modus mit 16-Bit Instruktionssatz gewechselt. Die Spezifikation der Rekonfiguration von Instruktionssätzen liegt nicht im Fokus von *ViCE-UPSLA* und wird nicht unterstützt. Für die Umsetzung der unterschiedlichen Instruktionssatzkonfigurationen müssen die einzelnen Instruktionssätze nebeneinander spezifiziert werden.

7.2.1 ARM-Spezifikation

Für die Spezifikation des ARM-Prozessors wurden der Registersatz und die Instruktionen für den `User`-Modus aus dem *ARM Data Sheet* verwendet. Dabei wurden datenverarbeitende Instruktionen, Speicherzugriffinstruktionen und bedingte Sprunginstruktionen umgesetzt. In Abbildung 7.1 werden die umgesetzten Instruktionen aufgelistet.

In dem Datenblatt des ARM-Instruktionssatzes werden für den zweiten Quell-Operanden der arithmetischen Instruktionen, wie z. B. ADC, ADD, AND usw., zwei Adressierungsmethoden angegeben. Die Auswahl der Adressierungsmethoden wird durch die Steuerbits im Instruktionsformat festgelegt. Für die Spezifikation der Instruktionsformate für den ausgewählten Instruktionssatz wurden sieben *ViCE-UPSLA*-Instruktionsformate und sechs Adressierungsmethoden spezifiziert. Für die Strukturierung

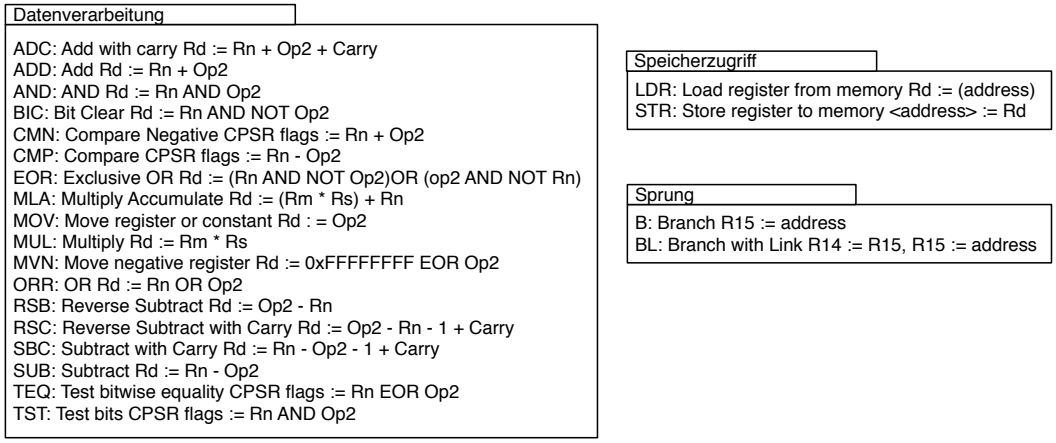


Abbildung 7.1: Spezifizierte ARM-Instruktionen mit *ViCE-UPSLA*

des Instruktionssatzes wurden acht abstrakte Instruktionen, wie in Abschnitt 4.3.5.1 beschrieben, für die unterschiedlichen Instruktionsarten spezifiziert.

Für die Beschreibung des Instruktionssatzes mit den verschiedenen Varianten der Adressierungsmethoden wurden in der Spezifikation 52 *ViCE-UPSLA*-Instruktionen umgesetzt. Für die Spezifikation der operationalen Beschreibung der Instruktionen sind folgende Operatoren verwendet worden: `bl, cmp, cmm, b, bl, st, not, eor, mov, and, orr, neg, add, add_c, add_s, add_cs, sub, sub_c, sub_s, sub_cs, mul, mla, ldr, ldrb, str und strb`.

Für den beschriebenen Instruktionssatz wurden zwei Mikroarchitekturen spezifiziert. Die erste Mikroarchitektur beschreibt die folgenden Pipelinestufen:

`Load`→`Execute`→`Memory`→`Store`

Dabei werden die Speicherzugriffe und die Berechnung der Instruktionsergebnisse in zwei unterschiedlichen Pipelinestufen durchgeführt. Die zweite Mikroarchitektur fasst die Ausführung der Instruktionen und den Speicherzugriff in eine Pipelinestufe, sodass die Mikroarchitektur durch folgende Pipelinestufen beschrieben wird:

`Load`→`(Execute/Memory)`→`Store`

Für beide Mikroarchitekturen des ARM-Prozessors werden je drei Lese-Ports und zwei Schreib-Ports spezifiziert. Die Spezifikation von Bypässen oder Interlocks stellt in *ViCE-UPSLA* einen vernachlässigbaren Spezifikationsaufwand dar und kann ad hoc durchgeführt werden.

Damit wurden zwei Prozessorspezifikationen mit einem identischen Instruktionssatz und unterschiedlichen Mikroarchitekturen spezifiziert. Die Generierung von Simulatoren kann automatisch durchgeführt werden.

7.2.2 CoreVA-Spezifikation

In diesem Abschnitt wird die Beschreibung der CoreVA Spezifikation durch den vollständigen Spezifikationsprozess der Architektur durchgeführt und die Möglichkeiten für die Validierung der Spezifikation aufgezeigt. Zunächst werden die betrachteten Randinformationen der Architektur und die Planung für Spezifikationsschritte vorgestellt. Die Mikroarchitektur des CoreVA-Prozessors wird in der Dokumentation durch die folgenden sechs Pipelinestufen beschrieben:

`Fetch`→`Decode`→`Read`→`Execute`→`Memory`→`Writeback`

In der `Execute`-Pipelinestufe werden vier inhomogene Funktionseinheiten mit unterschiedlichen Instruktionssätzen verwendet. Für die Spezifikation des Registersatzes wird eine Registerbank mit 32 32-Bit breiten Registern `R0-R31` betrachtet. Außerdem wurden Spezialregister wie `carry`, `PC` und `cond` verwendet und ein 128kByte großer Memory-Speicher spezifiziert. Der Instruktionssatz wird durch die folgenden 32-Bit Instruktionsarten angegeben:

- *Data processing* (arithmetische und logische Instruktionen).
- *Branch* (Sprunginstruktionen).
- *Condition Logic* (logische Instruktionen für die bedingte Ausführung).
- *Load/Store* (Lade- und Speicherinstruktionen).
- *Vektorinstruktionen* (16-Bit SIMD Instruktionen).

Für den Instruktionssatz wird eine SIMD-Erweiterung durch Vektorinstruktionen beschrieben. Die Vektorinstruktionen beschreiben arithmetische und logische Instruktionen, welche nach dem SIMD-Prinzip parallel 16-Bit Operationen auf zwei Datenströmen ausführen. In Abbildung 7.2 wird eine Übersicht der umgesetzten Instruktionen in der Spezifikation gegeben.

Die Spezifikation der CoreVA-Architektur wird mit der Beschreibung der Mikroarchitektur begonnen, damit die statische Validierung der Ressourcen und die Grapheinbettung zu einem frühestmöglichen Zeitpunkt durchgeführt werden können. Für die Spezifikation in *ViCE-UPSLA* werden die Pipelinestufen `Fetch` und `Decode` nicht benötigt und bei der Spezifikation ausgelassen. Außerdem wird für die Vereinfachung der Spezifikation zunächst die Spezifikation einer einfachen SIMD-Architektur mit einer universellen Funktionseinheit angegeben, welche die Instruktionssätze aus den vier

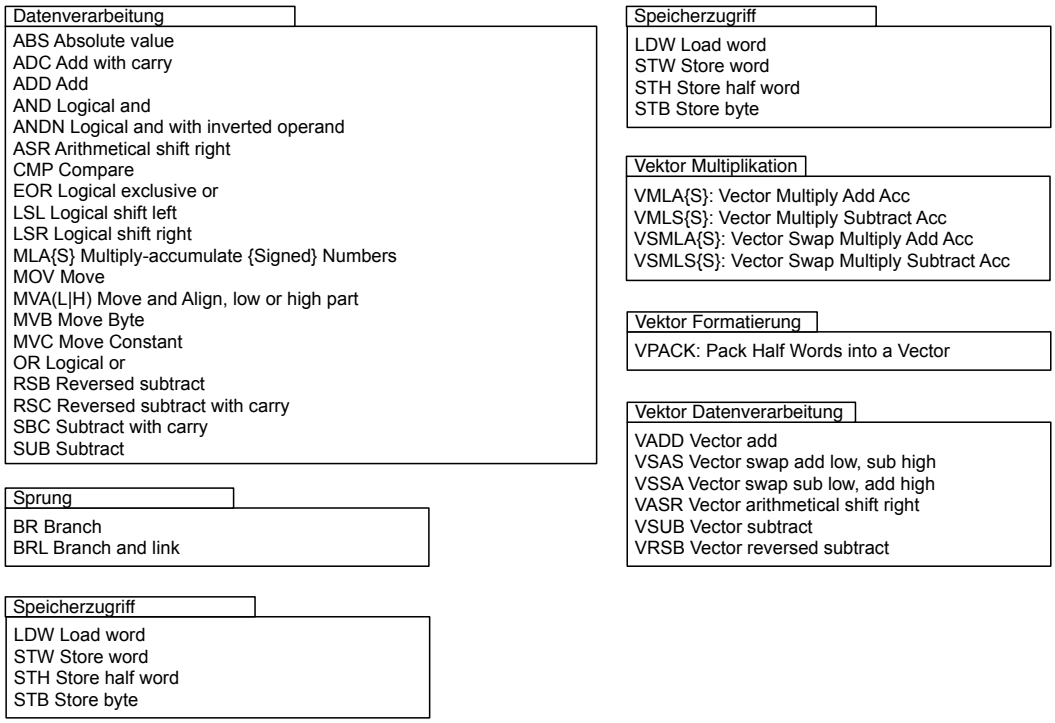


Abbildung 7.2: Spezifizierte CoreVA Instruktionen mit *ViCE-UPSLA*

inhomogenen Funktionseinheiten vereint. In Abschnitt 7.2.3 wird im Rahmen einer Entwurfsraumexploration die Erweiterung der spezifizierten Mikroarchitektur zu einer vierfach parallelen Architektur beschrieben.

Nach der Spezifikation der Mikroarchitektur wurde mit der Spezifikation der Registerbänke, der Adressierungsmethoden und der Instruktionsformate für die Beschreibung des Instruktionssatzes fortgefahren. Dabei wurden die bereits erwähnten 32 32-Bit Register und die Spezialregister der Architektur spezifiziert.

Für die Beschreibung des Instruktionssatzes wurden acht Instruktionsformate mit insgesamt acht Adressierungsmethoden spezifiziert. Für die Operanden in den Instruktionsformaten wurden in dieser Phase der Spezifikation die Lese- und Schreib-Ports aus der Spezifikation der Mikroarchitektur zugeordnet.

Mit dieser Spezifikation können bereits die ersten statischen Validierungsmethoden angewendet werden. Für die beschriebenen Komponenten können der Verwendungsnachweis für die Registerbänke und die Kodierung der Wertebereiche für die Operanden in den Instruktionsformaten validiert werden.

Die Spezifikation des Instruktionssatzes wurde in mehreren Iterationen durchgeführt, indem zuerst die arithmetischen Instruktionen mit den Adressierungsmethoden `Reg, Reg, Immediate` und anschließend die Instruktionen mit den Adressierungsmethoden `Reg, Reg, Reg` spezifiziert wurden.

In jeder Iteration wurden die abstrakten Instruktionen inklusive der operationalen Beschreibung mit abstrakten Operatoren erzeugt. Bei der Erzeugung von konkreten Instruktionen wurde die Struktur für die operationale Beschreibung der Instruktion aus den abstrakten Instruktionen übernommen. Für die konkrete Beschreibung des Verhaltens einer Instruktion muss anschließend lediglich der abstrakte Operator in der operationalen Beschreibung durch einen konkreten Operator ersetzt werden. Damit wird der Spezifikationsaufwand auf ein Minimum reduziert.

Bis auf Vektorinstruktionen unterscheidet sich die Komplexität der Instruktionsbeschreibung des CoreVA-Instruktionssatzes nicht von dem ARM-Instruktionssatz. Vektorinstruktionen führen parallel zwei Berechnungen in einem Ausführungszyklus in der ALU durch. Dabei werden die geladenen Werte aus den Registern in zwei Datenströme aufgeteilt und die Operationen auf diesen Datenströmen durchgeführt. Für die anschauliche Spezifikation der parallelen Verarbeitung wurden bei der Beschreibung der Vektorinstruktionen zwei zusätzliche Arten von Operatoren eingeführt. Dabei wurden zwei Operatoren `dem0` und `dem1` zum Extrahieren der einzelnen Datenströme aus einem geladenen Wert und ein Operator `mo` für das Zusammenführen zweier Datenströme zu einem Wert spezifiziert. In Abbildung 7.3 wird die operationale Beschreibung für die `VADD` Vektoraddition in *ViCE-UPSLA* dargestellt. In den Taktzyklen `0` und `2` werden die Operanden der Instruktion gelesen und geschrieben. Die Wertberechnung wird im Taktzyklus `1` durchgeführt. Dabei werden zwei 16-Bit Additionen durch die Operatoren `vadd16` durchgeführt. Die Operatoren `dem0`, bzw. `dem1` extrahieren die oberen bzw. unteren Bits des geladenen Wertes aus dem Register in zwei Vektorwerte für die Addition. Der Operator `mo` fügt die berechneten Vektorwerte aus den Additionen zu einem Wert zusammen.

So wie die `VADD`-Instruktion wurden die restlichen Vektorinstruktionen der SIMD CoreVA Erweiterung umgesetzt, indem das Zusammenfügen und Extrahieren der Datenströme in der operationalen Beschreibung spezifiziert wurden.

7.2.3 Entwurfsraumexploration der CoreVA-Architektur

Bei der Entwicklung der CoreVA-Architektur wurde im ersten Schritt eine SIMD-Architektur umgesetzt, indem nur eine universale ALU mit dem Instruktionssatz des Prozessors erzeugt wurde. Für die Beschreibung der VLIW-Architektur des CoreVA-Prozessors bestehen zwei Möglichkeiten. Zum einen kann die Kapazität der Ressourcen in der bereits beschriebenen Architektur aus Abbildung 7.6 vervierfacht werden. Zum anderen kann die konkrete Nachbildung der CoreVA-Architektur durchgeführt werden. Die visuelle Darstellung der konkreten CoreVA-Architektur ist in Abbildung 7.5

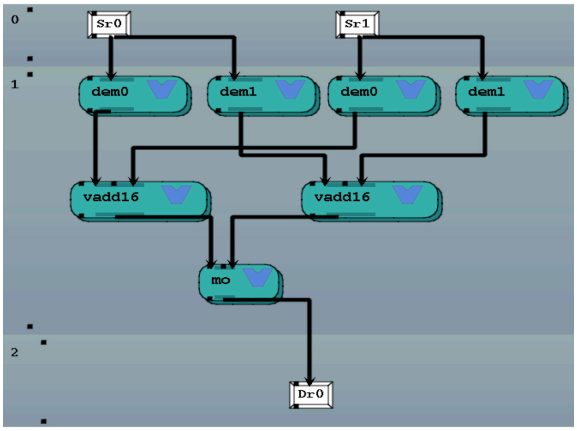


Abbildung 7.3: Vektoraddition aus dem CoreVA Instruktionssatz in *ViCE-UPSLA* Visualisierung.

dargestellt.

Die Vervielfachung der Kapazität der Ressourcen ist eine schnelle und einfache Methode, um die Simulation paralleler Ausführung in einem Prozessor zu testen. Der Vorteil bei dieser Vorgehensweise liegt darin, dass die bis dahin durchgeführte Validierung bis auf die Validierung der Ressourcen nach wie vor gültig ist. Außerdem ist der Spezifikationsaufwand verschwindend gering.

Für die Nachbildung der vierfach parallelen CoreVA-Architektur müssen, neben der Beschreibung der Architektur, neue Instruktionsgruppen für jede ALU der Architektur spezifiziert werden. Dabei müssen Komponenten der Instruktionen aus den neuen Gruppen exakt den entsprechenden Ressourcen aus der Mikroarchitektur zugeordnet werden.

7.3 Evaluierung der Sprache ViCE-UPSLA

Die visuelle Sprache *ViCE-UPSLA* wurde in einer systematischen Vorgehensweise (vgl. [CKK08, Kla10]) entwickelt und orientiert sich an domänenspezifischen Darstellungen für Prozessorkonstrukte. Für die Umsetzung der Sprache wurde das Generatorsystem DEViL [Sch06] verwendet, womit die Entwicklung einer anspruchsvollen , visuellen Sprache ermöglicht wurde. Für die Bewertung der Qualität der Sprache werden in diesem Abschnitt die Sprachkonstrukte der Sprache qualitativ aus der Sicht des Sprachentwurfs für visuelle Sprachen betrachtet. Die Bewertung der Sprachkonstrukte wird anhand des *Cognitive Dimension Framworks* (*CDF*) von T. R. G. Green und M. Petre

[GP96] durchgeführt.

Das *Cognitive Dimension Framwork* umfasst 13 Dimensionen, mit denen verschiedene Aspekte einer visuellen Sprache, wie z. B. die Benutzerfreundlichkeit oder die Ausdrucksfähigkeit, bewertet werden können. Eine Dimension beschreibt immer die Charakteristik der Sprachkonstrukte bezüglich einer Eigenschaft. Die verschiedenen Dimensionen schließen sich teilweise gegenseitig aus, sodass die Entwicklung einer „perfekten“ visuellen Sprache nach dem *CDF* nicht möglich ist. Im Folgenden werden einige Dimensionen aus dem *CDF* vorgestellt und auf ausgewählte Sprachkonstrukte aus *ViCE-UPSLA* angewendet. Damit soll gezeigt werden, welche Aspekte von *ViCE-UPSLA* nach dem *CDF* die Dimensionen erfüllen oder sich gegenseitig ausschließen.

Für die Bewertung der Sprachkonstrukte werden folgende kognitive Dimensionen betrachtet:

Abbildungsnähe beschreibt die Qualität der Abbildung einer Problembeschreibung durch Komponenten der Sprache. Dabei steigt die Qualität der Abbildung, je exakter die Darstellung der Problembeschreibung durch die Konstrukte der Sprache nachgebildet wird.

Konsistenz der Sprachkonstrukte beschreibt die Möglichkeiten der Schlussfolgerung aus den Eigenschaften bekannter Sprachkonstrukte auf die Eigenschaften vergleichbarer Sprachkonstrukte. Die konsistente Darstellung der Sprachkonstrukte begünstigt die Wiedererkennung und vereinfacht damit die kognitive Distanz zum Verstehen oder Erlernen neuer Konstrukte in einer Sprache.

Abstraktionsgrad beschreibt die Möglichkeiten für die Strukturierung der Konstrukte der Sprache in verschiedenen Ausprägungen. Durch eine Sprache kann die Abstraktion gefordert, erlaubt oder unterbunden werden. Durch die Möglichkeit der Abstraktion wird die Strukturierung der Programme und bessere Übersichtlichkeit für den Programmierer gefördert.

Schrittweises Überprüfen beschreibt die Möglichkeit der Überprüfung von Teilentwürfen in einem nicht fertigen Programm. Die Möglichkeit einer frühen Überprüfung fördert das Vertrauen und das Verständnis des Benutzers in den bis dahin beschriebenen Entwurf.

Verborgene Abhängigkeit beschreibt die Qualität für das Erkennen der Verknüpfungen oder Abhängigkeit zwischen den Konstrukten in einem Programm. Damit wird die Sichtbarkeit der Verknüpfungen zwischen den Sprachkonstrukten bewertet.

Ausdrucksfähigkeit beschreibt die Verständlichkeit der beschriebenen Komponenten bezüglich ihrer Aufgaben bei der Lösung eines Problems. Damit wird

bewertet, wie gut der Leser eines Programms die Struktur und die Aufgaben der Sprachkonstrukte aus dem Gelesenen nachvollziehen kann.

Im Folgenden werden die beschriebenen Dimensionen aus *CDF* auf die Sprachkonstrukte von *ViCE-UPSLA* angewendet. Damit werden die Sprachkonstrukte bezüglich ihrer Qualität in der jeweiligen Dimension bewertet.

Für die Spezifikation der CoreVA-Architektur müssen die Pipelinestufen und die Ressourcen des Prozessors aus der informellen Beschreibung, wie in Abbildung 7.4 dargestellt, in *ViCE-UPSLA* ausgedrückt werden. In Abbildung 7.5 ist die Spezifikation der Mikroarchitektur in *ViCE-UPSLA* dargestellt.

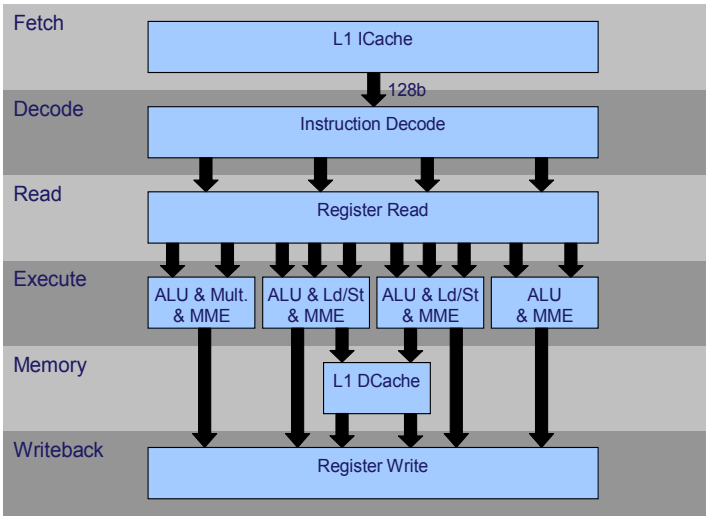


Abbildung 7.4: Darstellung der CoreVA Mikroarchitektur in der informellen Beschreibung [JD11].

Wie den Abbildungen 7.4 und 7.5 entnommen werden kann, ist die Spezifikation sehr nah an der Problembeschreibung aus der informellen Beschreibung dargestellt. Damit wird gezeigt, dass die Abbildungsnähe der Mikroarchitektur mit *ViCE-UPSLA* sehr hoch ist. Diese Aussage trifft in *ViCE-UPSLA* auf eine Reihe von Sprachkonstrukten zu, wie z. B. die Darstellung der Registerbänke oder der Instruktionsformate.

Als nächste Dimension wird die Konsistenz der Darstellung betrachtet. Dazu wird die visuelle Darstellung der vereinfachten CoreVA-Architektur und der `LDW` Instruktion in Abbildung 7.6 nebeneinander gestellt. Die Spezifikation der Pipelinestufen und der Ausführungszyklen der Instruktionen durch untereinander angeordnete Felder besitzen identische Bedeutung. Damit wird beschrieben, dass die Verarbeitung der dar-

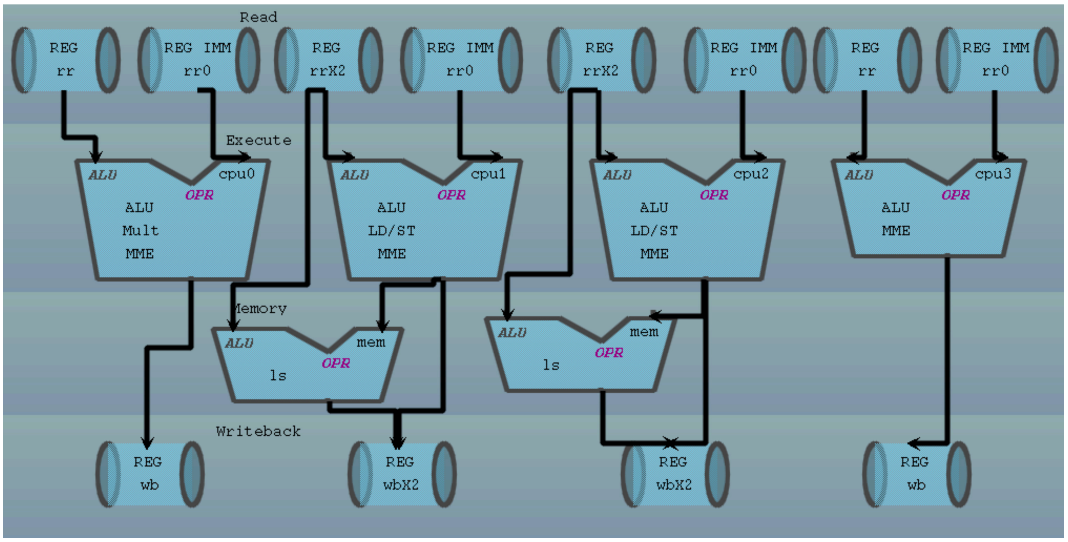


Abbildung 7.5: Darstellung der CoreVA Mikroarchitektur in *ViCE-UPSLA*.

auf verteilten Komponenten nacheinander durchgeführt wird. Analog zu den Ausführungszyklen wird die Wertübergabe zwischen den Knoten im Datenpfad und in der operationalen Beschreibung auf dieselbe Art dargestellt. Die kognitive Distanz für das Verstehen der operationalen Beschreibung ist für einen Prozessorentwickler, der mit der Darstellung einer Mikroarchitektur vertraut ist, praktisch nicht vorhanden. Damit wird eine einheitliche Darstellung der verschiedenen Sprachkonstrukte von den Prozessorentwicklern positiv aufgenommen.

Die Dimensionen der Abbildungsnähe und der Konsistenz korrelieren mit der Dimension der Ausdrucksfähigkeit aus *CDF*. Je besser die Zusammenhänge und Darstellungen zu den Erwartungen der Prozessorentwickler passen, desto besser werden die Aufgaben der Sprachkonstrukte beim Betrachten von komplexen Strukturen erkannt. Die Sprachkonstrukte in *ViCE-UPSLA* erfüllen die Dimensionen der Abbildungsnähe und Konsistenz, sodass von einer hohen Ausdrucksfähigkeit der Sprachkonstrukte gesprochen werden kann.

In Abschnitt 4.3.5.1 wurde das Konzept der abstrakten Instruktionen beschrieben. Damit werden Instruktionen mit gleichen Eigenschaften zusammengefasst. Durch die abstrakten Instruktionen werden die Instruktionen des Prozessors in zusammenhängende Gruppen eingeordnet. Damit erfüllt das Sprachkonstrukt der abstrakten Instruktionen in *ViCE-UPSLA* die Dimension der Abstraktion, womit eine bessere Übersichtlichkeit der Struktur des Instruktionssatzes beschrieben wird.

Wie bereits in Abschnitt 7.2.2 beschrieben, wurden die abstrakten Instruktionen

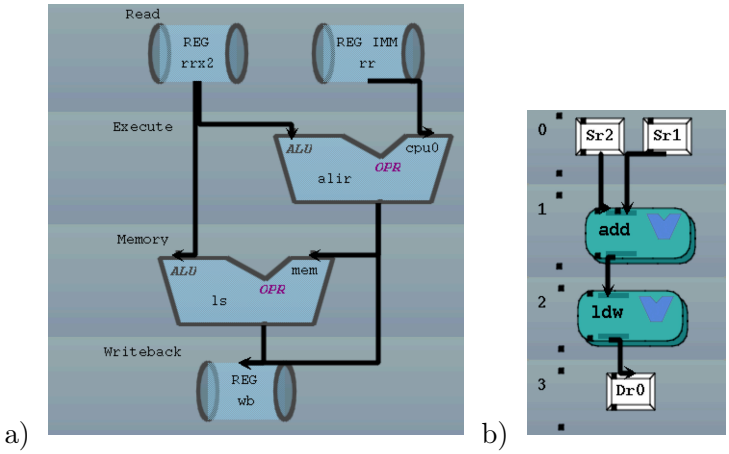


Abbildung 7.6: a) Die vereinfachte CoreVA-Mikroarchitektur bestehend aus vier Pipelinestufen, drei Lese-Ports, einem Schreib-Port, einer Funktionseinheit für arithmetische und logische Instruktionen und einer Funktionseinheit für den Speicherzugriff. b) Operationale Beschreibung der LDW-Instruktion bestehend aus zwei Quelloperanden, aus denen die Speicheradresse berechnet wird. Mit der Adresse wird der Wert für den Zieloperanden aus dem Speicher durch die Operation `ldw` geladen.

bei der Spezifikation des CoreVA-Instruktionssatzes für die Reduzierung des Spezifikationsaufwands und für die Anwendung der statischen Validierungsmethoden der Spezifikation eingesetzt. Die Eigenschaften der abstrakten Instruktionen können in der Spezifikation geprüft werden, bevor mit der Spezifikation der konkreten Instruktion des Prozessors fortgefahren wird. Damit erfüllen die abstrakten Instruktionen die Dimension der schrittweisen Überprüfung von Teilentwürfen.

Eine visuelle Sprache kann nicht alle Dimensionen aus *CDF* im gleichen Maß erfüllen. Die Operanden in der operationalen Beschreibung der Instruktionen besitzen eine Verknüpfung zu den Instruktionsformaten und damit zu den Adressierungsmethoden der Spezifikation. Während die Verknüpfung zu den Instruktionsformaten offensichtlich dargestellt wird, ist die Verknüpfung zu den Adressierungsmethoden direkt nicht nachvollziehbar. Zum Nachvollziehen der Verknüpfungen muss der Prozessorentwickler die Spezifikation der Instruktionsformate in einer weiteren Darstellung betrachten. In der anderen Richtung ist es aus der visuellen Darstellung einer Adressierungsmethode nicht nachvollziehbar, welche Instruktionen diese Adressierungsmethode referenzieren. Die Darstellung der Verknüpfung würde die einfache Darstellung der Adressierungsmethoden unverhältnismäßig kompliziert gestalten. Die Anwendung der Dimension für die verborgenen Abhängigkeiten zeigt, dass die einfache Visualisierung von Sprachkon-

strukten und die Darstellung von Abhängigkeiten sich in diesem Fall gegenseitig ausschließen. Die verborgenen Abhängigkeiten können in diesem Zusammenhang durch andere Darstellungsparadigmen ohne Verlust der einfachen Visualisierung gelöst werden, indem z. B. dreidimensionale Sprachkonstrukte verwendet werden.

Durch Anwendung der kognitiven Dimensionen auf die Sprachkonstrukte von *ViCE-UPSLA* wurde gezeigt, dass die Darstellungen und Strukturen der Sprache eine hohe Qualität aufweisen. Eine vollständige Anwendung des *CDF* von T. R. G. Green und M. Petre [GP96] kann analog zu den beschriebenen Anwendungen der Dimensionen für Abbildungsnähe, Konsistenz, Abstraktionsgrad, schrittweises Überprüfen, verborgene Abhängigkeit und Ausdrucksfähigkeit durchgeführt werden.

Zusammenfassend besitzt die Sprache durch die visuellen Darstellungen eine hohe Akzeptanz seitens der Entwickler aus der Domäne des Prozessorentwurfs. Außerdem kann die Mächtigkeit der Sprache durch die Spezifikation der *ARM*- und *CoreVA*-Prozessoren einwandfrei gezeigt werden.

7.4 Evaluierung der Validierungsmethoden

Die Evaluierung der Validierungsmethoden kann aus zwei Gesichtspunkten betrachtet werden. Zum einen kann die Vorgehensweise bei der Validierung bewertet werden. Zum anderen können die Ergebnisse der Validierungsmethoden untersucht werden. Im Folgenden werden zuerst die Validierungsmethoden bewertet. Anschließend werden die Ergebnisse der Validierung der ARM- und der CoreVA-Architektur und die Validierung der induzierten Fehler vorgestellt.

7.4.1 Bewertung der Vorgehensweise bei der Validierung

Die Bewertung der statischen Validierungsmethoden wurde im Rahmen der Evaluierung für ViCE-UPSLA bereits durchgeführt, indem in Abschnitt 7.3 die Dimension “Schrittweise Überprüfen“ auf die Sprache angewendet wurde. Der frühe Einsatz der statischen Validierungsmethoden steigert damit die Qualität des Entwurfs. Bei der Entwicklung der CoreVA-Architektur konnte die Validierung der Mehrdeutigkeit von Instruktionsformaten, der Kodierung von Wertebereichen und der Verwendungsnachweis für die Registerbänke zu einem Zeitpunkt automatisch durchgeführt werden, bevor der Instruktionssatz des Prozessors spezifiziert wurde. Durch die strukturelle Vorgehensweise bei dem Entwurf wurden in der Spezifikation der CoreVA-Architektur keine Inkonsistenzen festgestellt. Bei der Durchführung der statischen Validierung für die Grapheinbettung wurde eine Inkonsistenz in der CoreVA- und ARM-Spezifikation entdeckt. Dabei wurde festgestellt, dass die operationale Beschreibung der Speicher-Instruktionen inkonsistent zu dem Datenpfad der Mikroarchitekturen sind. Die Anwendung der Validierungsmethoden und entdeckte Inkonsistenz werden in Abschnitt

7.4.2 ausführlich beschrieben.

Für die Bewertung der Validierungswerkzeuge für modellbasiertes Testen werden von A. C. Dias Neto und G. H. Travassos in mehreren Arbeiten [NT08b, NT08a] die Vorgehensweise für die Bewertung und die Auswahl von MBT Werkzeugen beschrieben. Die dynamischen Validierungsmethoden in *ViCE-UPSLA* basieren zum Teil auf den Konzepten des MBT, sodass eine Bewertung der Eigenschaften für die dynamischen Validierungsmethoden anhand der Bewertungskriterien aus dem MBT durchgeführt werden kann. Dabei werden folgende charakteristische Eigenschaften von Dias Neto und Travassos betrachtet:

- Automatisierungsgrad: Damit wird der Aufwand für die Erzeugung eines Testfalls, gemessen an dem Anteil der automatisch und manuell durchgeführten Arbeitsschritten bewertet.
- Test-Überdeckungskriterien: Bewertung der Regeln für die Überdeckung der Spezifikation für die Erzeugung der Testfälle.
- Werkzeugunterstützung: Bewertet die Existenz und die Qualität der Werkzeuge, die für die Erzeugung oder Bearbeitung der Testfälle zur Verfügung gestellt werden.

Für die Bewertung der dynamischen Validierungsmethoden werden die Methoden für die Validierung der Registererreichbarkeit (vgl. Abschnitt 5.5.2) und die Validierung der Daten- und Ressourcenkonflikte in der Pipeline (vgl. Abschnitt 5.5.6) betrachtet.

Für die Validierung der Registererreichbarkeit wurden die Testfallspezifikationen aus den Spezifikationen von ARM und CoreVA automatisch abgeleitet. Dabei wurden durch den integrierten Generator der Ablaufplan und die Überdeckungsmengen für die Validierung automatisch erzeugt. Für die Generierung der Testfälle aus der Testfallspezifikation mussten für beide Architekturen die `mov`-Instruktionen manuell bestimmt werden, da die Generatoren die Semantik der spezifizierten Instruktionen nicht ableiten können. Aus der vollständigen Testfallspezifikation wurden die Testfälle vollständig für die Erzeugung der Simulation generiert. Der Automatisierungslevel dieser Methoden ist nach Dias Neto und Travassos sehr hoch, da eine Erzeugung der anwendbaren Testfälle zum großen Teil automatisch durchgeführt wird. Die Werkzeugunterstützung wird durch die Sprache *TFS* (vgl. Abschnitt 5.7) ermöglicht, damit wurde, wie bereits beschrieben, die Konkretisierung der Testfälle manuell durchgeführt. Aufgrund der kleinen Registersatz-Struktur liegt die Größe der Testfälle im kByte Bereich. Die Validierung mit den erzeugten Testfällen wurde einwandfrei durchgeführt.

Für die Validierung der Daten- und Ressourcenkonflikte konnten die Ablaufstrukturen für einzelne Instruktionsgruppen automatisch aus der Spezifikation der Prozessoren abgeleitet werden. Für die Validierung müssen die Ablaufstrukturen an die

Pipelinestrukturen der Prozessoren angepasst werden. Diese Struktur wurde ebenfalls automatisch durch den Generator für die Validierungsmethoden übernommen. Die Überdeckungsmengen für die Instruktionen der verschiedenen Gruppen werden aus den Instruktionssätzen und die Wertebereiche für die Parameter automatisch ermittelt. Der Automatisierungsgrad und die Werkzeugunterstützung sind auch für diese Methoden auf einem hohen Niveau. Bei der Generierung der Testfälle aus der Spezifikation und die Durchführung der Validierung im Simulator wurden Schwächen bei Regeln für Überdeckungskriterien festgestellt. Durch eine optimistische Strategie für die Ermittlung der Wertebereiche wurden für die Testfälle Dateien in einer Größe von 500 bis 700Mbyte erzeugt. Damit wurden bei der Durchführung der Validierung Trace-Daten im Bereich von 700 bis 1500Mbyte generiert. Die so entstandenen Datenmengen konnten mit vorhandenen Werkzeugen nicht untersucht werden. Für die Durchführung der Validierung mussten die Wertebereiche für die Parameter der Instruktionen manuell angepasst werden. Damit konnte die Testfälle auf eine Größe von 1 bis 5MByte sinnvoll reduziert werden. Dabei wurden ca. eine Million Kombinationen von Instruktionen getestet.

Analog zu der Bewertung der Validierungsmethoden für die Registererreichbarkeit, Daten- und Ressourcenkonflikte können weitere Validierungsmethoden aus Abschnitt 5.5 betrachtet werden.

7.4.2 Effektivität der Validierungsmethoden

Bei der Validierung der ARM- und CoreVA-Architekturen wurde eine systematische Inkonsistenz in der Spezifikation aufgedeckt, die bei der Erzeugung der Spezifikationen aus den Datenblättern unbeabsichtigt entstanden ist. Im Folgenden wird die gefundene Inkonsistenz beschrieben und die Quelle aus der informellen Beschreibung sowie die Validierung des Fehlers genau benannt. Anschließend wird die Evaluierung der Methoden durchgeführt, indem typische Fehler in der Spezifikation erzeugt und mittels der Validierung überprüft werden.

In Abbildung 7.7 wird die visuelle Spezifikation der `stw` Instruktionen des CoreVA-Prozessors, die Spezifikation des Instruktionsformats für diese Instruktion und die CoreVA-Mikroarchitektur dargestellt. Die Validierung der Grapheinbettung meldet bei einer solchen Spezifikation einen Fehler, da die operationale Beschreibung inkonsistent zu dem Datenpfad der Mikroarchitektur ist. Wird die Inkonsistenz ignoriert und ein Simulator aus der fehlerhaften Spezifikation erzeugt, führt dieser Fehler bei der Simulation zu Ressourcenkonflikten, die durch den Simulator ebenfalls angezeigt werden.

Die Inkonsistenz hat ihren Ursprung in der informellen Beschreibung der Architektur. In der Dokumentation werden Lade- und Speicher-Instruktionen `ldw` und `stw` für den CoreVA-Prozessor und `ldr`, `str`, `ldrb` und `strb` für den ARM-Prozessor beschrieben. Dabei wird in der Beschreibung des Prozessors für die Instruktionen `ldw` und `stw` ein Instruktionsformat angegeben. Bei der systematischen Spezifikation wurde zuerst

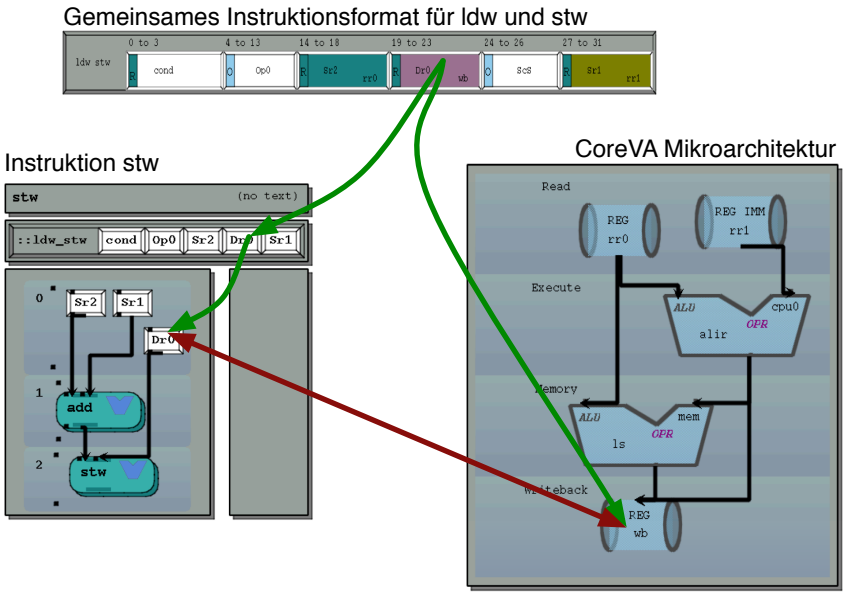


Abbildung 7.7: CoreVA VLIW Mikroarchitektur. Spezifikation der `stw`-Instruktion und des Instruktionsformats `ldw_stw`. Die grünen Pfeile in Abbildung stellen die Verknüpfungen zwischen den Komponenten in der Spezifikation dar. Durch den roten Pfeil wird die Konfliktstelle aufgezeigt.

ein Instruktionsformat für die Instruktionen spezifiziert und die Ressourcen für die einzelnen Operanden festgelegt. Für die Spezifikation der operationalen Beschreibung der Instruktionen wurde das bereits spezifizierte Instruktionsformat sowohl für `ldw`- als auch für die `stw`-Instruktion eingesetzt und die operationale Beschreibung anhand der Dokumentation erzeugt. Die Spezifikation der Instruktionen in *ViCE-UPSLA* wird in Abbildung 7.8 dargestellt. Beide Instruktionen verwenden die Operanden `Sr1` und `Sr2` für die Berechnung der Adresse. Der Operand `Dr0` wird in den Instruktionen in unterschiedlichen Ausführungsschritten verwendet und benötigt unterschiedliche Ressourcen im Datenpfad der Mikroarchitektur. Da diese Information nicht explizit in der Dokumentation vermerkt ist, wurde diese Eigenschaft bei der Erzeugung der Spezifikation übersehen. Für die Auflösung der Inkonsistenz müssen in ViCE-UPSLA für Lade- und Speicher-Instruktionen zwei unterschiedliche Instruktionsformate spezifiziert werden, damit die Ressourcen für die Knoten in der operationalen Beschreibung korrekt angegeben werden können.

Für die Evaluierung der Effektivität der Validierungsmethoden wurden verschiedene Fehler in die Spezifikation der Prozessoren induziert und mit den Validerungsme-

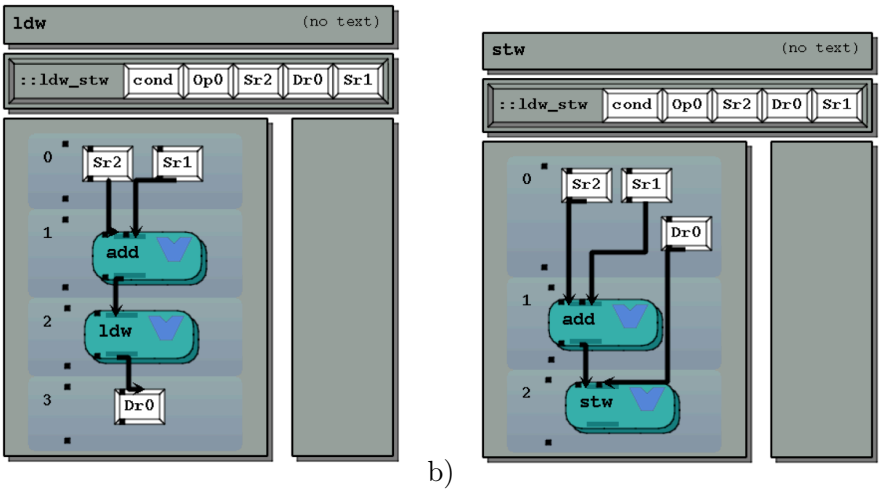


Abbildung 7.8: Spezifikation der a) `ldw` Lade-Instruktion und b) `stw` Speicher-Instruktion.

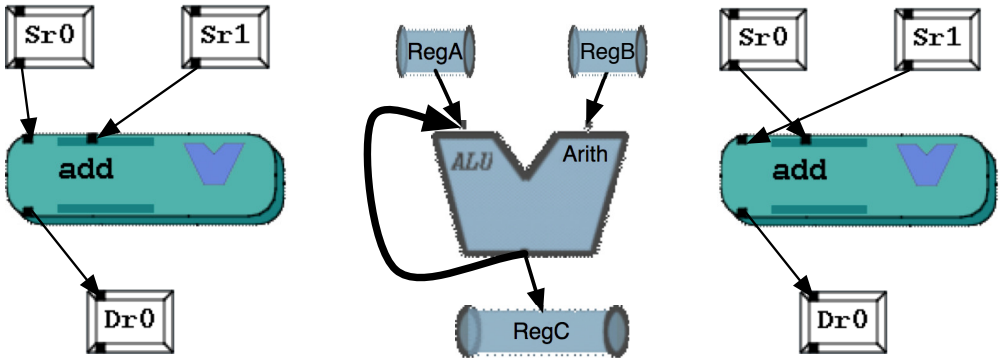
thoden untersucht. Hierfür werden zwei Beispiele aus der Evaluierung vorgestellt. Im ersten Beispiel werden in die Funktionen der bereits beschriebenen Lade- und Speicher-Instruktionen ein Fehler induziert, indem falsche Adressen im Speicher verwendet werden. Im zweiten Beispiel wird ein Fehler in die operationale Beschreibung einer Instruktion induziert, wodurch Datenkonflikte bei der Ausführung (vgl. Abschnitt 5.5.6) entstehen.

Für die Beschreibung der Beispiele wird die Indizierung eines Fehlers und die entsprechende Validierung durch statische und dynamische Validierungsmethoden beschrieben. Im ersten Schritt wird der induzierte Fehler beschrieben und anschließend die Anwendung Validierung für dessen Fehlerquelle.

Die Instruktionen `stw` bzw. `ldw` lesen und schreiben 4Byte Worte aus dem Speicher. Wie in Abbildung 7.8 in der operationalen Beschreibung der Instruktion dargestellt wird die Adresse für das Speichern des Wertes aus zwei Operanden `Sr1` und `Sr2` mittels 32-Bit Addition berechnet. Für das Speichern eines Wortes muss die Adresse des ersten Bytes eines Wortes bestimmt werden. Damit werden die Adressen auf ein vielfaches von vier normiert. In der *ViCE-UPSLA* Spezifikation wird die Normierung der Adresse in dem Operator `stw` bzw. `ldw` in der operationalen Beschreibung der Instruktion durchgeführt, indem z. B. die Adresse durch die Berechnung `adresse &= (~3)` normiert wird. Der Fehler wurde induziert, indem die beschriebene Normierung der Adresse ausgelassen wurde. Die Validierungsmethode für Lade- und Speicher-Instruktionen sieht

vor, dass bei der Validierung die Grenzwerte des Speicherbereichs validiert werden und dabei aufeinander folgende Adressen in Viererblöcken identische Werte liefern müssen. Bei der Validierung wurden die spezifizierten Lade- und Speicher-Instruktionen mit der Überdeckung des signifikanten Teils der Wertebereiche für die Operanden durchgeführt. Damit wurden bei der Validierung ca. 128.000 Instruktionen ausgeführt. Aus den aufgezeichneten Systemzuständen konnte die falsche Berechnung der Adressen einwandfrei festgestellt werden, da die geladenen Worte ohne Normierung der Adresse für alle Eingaben unterschiedlich sind.

Bei einem weiteren Szenario wurden Datenkonflikte in einer ARM Spezifikation mit einem Bypass zwischen dem Ausgabe-Port und dem linken Eingabe-Port der ALU validiert. In Abbildung 7.9 werden die Spezifikation des Datenpfades mit einem Bypass und die korrekte und fehlerhafte Spezifikation der `add`-Instruktion dargestellt. Ein Fehler wurde induziert, indem bei einer `add_i dr, sr, imm` Add-Immediate-Instruktion des Prozessors die Eingabeoperanden vertauscht wurden, sodass die Instruktion anstelle `dr=sr+imm` die Berechnung `dr=imm+sr` durchführt. Die statische Validierung der Kontrollflussanalyse meldet bei diesem Szenario keine Inkonsistenz, da der spezifizierte Bypass für eine Reihe anderer Instruktionen aus dem Instruktionssatz korrekt spezifiziert ist. Außerdem wird das Ergebnis der Instruktion mit dem induzierten Fehler nach wie vor korrekt berechnet.



(a) Korrekte `add`-Instruktion (b) Datenpfad (c) Fehlerhafte `add`-Instruktion

Abbildung 7.9: Spezifikation der korrekten und inkonsistenzen `add`-Instruktion für den gegebenen Datenpfad.

Für die Validierung der Datenkonflikte werden die Testfälle, wie in Abschnitt 7.4.1 beschrieben, für die Validierung der Datenkonflikte aus der Spezifikation automatisch generiert. Bei der Erzeugung der Testfälle wurden Paare von Instruktionen aus einer Menge von 28 Instruktionen gebildet, sodass alle Instruktionspaare mit und ohne Datenabhängigkeit und in unterschiedlichen Abständen in der Pipeline validiert wur-

den. Damit wurden ca. 15.000 Kombinationen überprüft. Der induzierte Fehler wurde bei der Validierung gleich durch mehrere Stellen, in denen Add-Immediate verwendet wurden, gezeigt. Dabei wurde in den entsprechenden Kombinationen ein Datenkonflikt durch den Bypass nicht aufgelöst, sodass die Simulationsergebnisse nicht mit den Referenzergebnissen übereinstimmten.

7.4.3 Zusammenfassung und Ergebnisse der Evaluierung von Validierungsmethoden

Für die Validierungsmethoden konnten die Anwendbarkeit erfolgreich gezeigt werden. Dabei wurden bei der Spezifikation des Prozessors echte Inkonsistenzen entdeckt, die bei der strikten Verfolgung der Dokumentationen der Prozessoren gemacht wurden. Außerdem wurden die Validierungsmethoden nach den Kriterien aus dem MBT bewertet. Dabei wurde gezeigt, dass die Erzeugung der Testfälle für die dynamische Validierung auf einem hohen Automatisierungslevel liegt und mit einer angemessenen Werkzeugunterstützung durchgeführt wird. Die Testfälle für die Validierung der Registererreichbarkeit sind im Vergleich zu Testfällen, wie z. B. der Validierung der Adressierungsmethoden oder der Pipelinekonflikte, klein. Die Größe der Testfälle für die Validierung der Adressierungsmethoden resultiert aus den Wertebereichen der Operanden. Für die Optimierung der dynamischen Validierung können die Strategien zur Ermittlung der Wertebereiche für die Operanden verbessert werden.

7.5 Evaluierung der Simulation

Bei der Evaluierung der Simulation werden Simulatoren für unterschiedliche Programme erzeugt und die Simulationsgeschwindigkeit gemessen. Die ausgewählten Programme werden mit verschiedenen Konfigurationen des ARM- und CoreVA-Prozessors simuliert, sodass aussagekräftige Ergebnisse zur Simulation verschiedener Aspekte gemacht werden können. Die Evaluierung soll die Effizienz automatisch generierter Simulatoren zeigen. Die Ergebnisse der Evaluation werden aufzeigen, welche Konfigurationen der Prozessoren wie effizient funktionieren. Im Folgenden werden zuerst die verwendeten Konfigurationen der Prozessoren und die implementierten Programme vorgestellt. Für die Präsentation der Ergebnisse aus der Simulation wird anschließend die Vorgehensweise bei der Ermittlung der Daten beschrieben.

Für die Evaluierung der Simulation wurden folgende Programme implementiert:

- Berechnung der Fibonacci-Zahlen
- Multiplikation komplexer Zahlen
- Multiplikation von 4×4-Matrizen

- x.264 Video-Codec Hotspot Funktion `pixel_satd_wxh16` [Bun10]

Die Berechnung von Fibonacci-Zahlen wird iterativ durchgeführt. Der Algorithmus wird mit den ersten zwei Fibonacci-Zahlen initiiert. Anschließend wird eine Schleife wiederholt, wobei in jedem Schleifendurchlauf jeweils die nächste Fibonacci-Zahl berechnet wird. Durch die Datenabhängigkeit bei der Berechnung der Zahlen wird sichergestellt, dass ein Compiler die einzelnen Schleifeniterationen nicht wegoptimieren kann. Damit wird die quantitative Messung der Simulationsgeschwindigkeit einfacher arithmetischer Instruktionen ermöglicht.

Mit der Multiplikation komplexer Zahlen soll die Datenstrom basierte Verarbeitung mit begrenzter Parallelisierung simuliert werden. Damit wird ein Paradigma für die Netzwerk-Datenverarbeitung in eingebetteten Systemen nachempfunden. Für die Multiplikation der komplexen Zahlen wird ebenfalls eine Schleife verwendet. Dabei werden in jedem Schleifendurchlauf zwei komplexe Zahlen aus dem Speicher des Prozessors geladen, das Produkt berechnet und in den Speicher zurück geschrieben.

Mit der Matrix-Multiplikation soll die Simulationsgeschwindigkeit in verschiedenen Architekturen verglichen werden. Insbesondere sollen die Parallelisierung der Datenverarbeitung und die Verdoppelung des Durchsatzes durch die 16-Bit SIMD Instruktionen gezeigt werden. Bei der Matrix-Multiplikation wurden in einer Schleife, wie schon bei der Multiplikation der komplexen Zahlen, Zeilen bzw. Spalten von zwei Matrizen aus dem Speicher geladen und das Ergebnis der Multiplikation in eine neue Matrix im Speicher abgelegt. Bei der Ausführung wurde durch eine zweite Schleife die Multiplikation mehrerer Matrizen aus dem Speicher durchgeführt.

Für die Evaluierung der Simulation für die VLIW-Architektur des CoreVA-Prozessors wurde eine für die Architektur optimierte Version der Funktion `pixel_satd_wxh16` des `x.264` Video-Codecs verwendet. Für die Ausführung der Funktion wird eine Schleife wiederholt ausgeführt. Dazu wurden der typische Datenstrom für die Funktion aus dem Speicher geladen und die Ergebnisse der Funktion zurückgeschrieben. Mit der Simulation der Funktion `pixel_satd_wxh16` soll die Parallelisierung durch die VLIW-Architektur im Vergleich zu einer einfachen SIMD-Architektur gezeigt werden.

Bei der Validierung wurden zwei Konfigurationen des ARM-Prozessors und zwei Konfigurationen des CoreVA-Prozessors eingesetzt.

Für die Beispiele der ARM-Architektur wurden die Konfigurationen (im Folgenden `ARM` und `ARM_By` genannt) des Prozessors, wie in Abschnitt 7.2.1 beschrieben, verwendet. Die Konfiguration des `ARM_By` enthält, im Gegensatz zu `ARM`, einen Bypass zwischen dem Ausgabe-Port und dem linken Eingabe-Port der ALU.

Als CoreVA-Architektur wurden eine SIMD-Variante (im Folgenden `CoreVA_SIMD` genannt), wie in Abschnitt 7.2.2 beschrieben, mit einer Funktionseinheit und die VLIW-Variante (Im Folgenden `CoreVA_VLIW` genannt), wie in Abschnitt 7.2.3 beschrieben, mit vierfach vorhandenen Ressourcen verwendet.

Für die Ausführung der Testprogramme wurden die Simulatoren mit den entspre-

chenden Architekturen generiert. Für die generierten Simulatoren wurden die Anzahl der Instruktionen im Rumpf der Schleifen und die dafür benötigte Anzahl der Zyklen der simulierten Architektur erfasst. Als Simulationssystem wurde ein Host-System mit einem Intel Core i5 2.4 GHz Prozessor verwendet. Für die Erfassung der Simulationsgeschwindigkeit wurde das Profiling-Werkzeug *callgrind* [Dev10] verwendet. Mit dem Werkzeug wird die Anzahl der Instruktionen auf dem Simulationssystem (im Folgenden Host-Instruktionen genannt) gezählt. Für die Ermittlung von vergleichbaren Daten wurde die Differenz zwischen verschiedenen Mengen von Schleifeniterationen der vorgestellten Programme berechnet, sodass für einen Schleifendurchlauf gemittelt die Anzahl der Host-Instruktionen für eine Instruktion des Simulators angegeben wird.

Die Berechnung der Fibonacci-Zahlen wurde in Verbindung mit den Architekturen `ARM`, `ARM_By` und `CoreVA_SIMD` simuliert. Die Ergebnisse werden in Tabelle 7.1 dargestellt. Bei der Berechnung der Fibonacci-Zahlen bestehen im Rumpf der Schlei-

Tabelle 7.1: Simulation der Fibonacci-Zahlen mit `ARM`, `ARM_By` und `CoreVA_SIMD`.

Architektur	Instruktionen im Rumpf	Ausführungszyklen für eine Iteration	Host-Instruktionen pro Architektur-Instruktion
`ARM`	6	8	7,2
`ARM_By`	6	8	7,5
`CoreVA_SIMD`	6	8	17,5

fe keine Datenabhängigkeiten zwischen den Instruktionen. Dadurch sind die Anzahl der Instruktionen und die Anzahl der Taktzyklen für alle drei Instruktionen gleich. Der Unterschied in der Anzahl der Host-Instruktionen für die verschiedenen Architekturen kann auf die Komplexität der Instruktionsformate und andere Konstrukte der Instruktionen zurückgeführt werden. Z. B. unterscheiden sich die Architekturen `ARM` und `ARM_By` lediglich durch einen Bypass in der Mikroarchitektur. Damit kann der Unterschied in der Anzahl der Host-Instruktionen erklärt werden, da der Bypass, auch wenn nicht verwendet, für jede ausgeführte Instruktion überprüft wird. Die höhere Anzahl von Host-Instruktionen für die Simulation der `CoreVA_SIMD` Instruktionen resultiert aus den komplexeren Adressierungsmethoden der Architektur.

Die Multiplikation komplexer Zahlen wurde mit den Architekturen `ARM` und `ARM_By` durchgeführt. Die gemessenen Ergebnisse sind in Tabelle 7.2 angegeben. Bei diesem Beispiel kann sehr gut beobachtet werden, dass die Simulationsgeschwindigkeit durch den verwendeten Bypass nur minimal verschlechtert wird. Durch die hohe Datenabhängigkeit zwischen den Instruktionen verringert sich die Anzahl der Taktzyklen für die Multiplikation von zwei komplexen Zahlen durch den eingesetzten Bypass von 45 auf 24 Zyklen.

Im nächsten Beispiel in Tabelle 7.3 werden die Ergebnisse aus der Multiplikation der 4×4-Matrizen mit den Architekturen `ARM` und `CoreVA_SIMD` beschrieben. Für

Tabelle 7.2: Multiplikation komplexer Zahlen mit `ARM`, `ARM_By`.

Architektur	Instruktionen im Rumpf	Ausführungszyklen für eine Iteration	Host-Instruktionen pro Architektur-Instruktion
`ARM`	17	45	20,9
`ARM_By`	17	24	23

die `CoreVA_SIMD`-Architektur wurde die Multiplikation von Matrizen mit 32-Bit und 16-Bit Werten implementiert.

Tabelle 7.3: Matrix Multiplikation mit `ARM`, `CoreVA_SIMD`.

Architektur	Instruktionen im Rumpf	Ausführungszyklen für eine Iteration	Host-Instruktionen pro Architektur-Instruktion
`ARM` 32-Bit	22	32	18,9
`CoreVA_SIMD` 32-Bit	22	34	30,1
`CoreVA_SIMD` 16-Bit	15	27	27

Wie bereits an den vorangegangenen Beispielen gezeigt, wird die `ARM`-Architektur etwas effizienter simuliert als die `CoreVA` Architektur. Bemerkenswert ist der Unterschied zwischen der 16- und 32-Bit Matrix Multiplikation mit der `CoreVA_SIMD` Architektur. Bei der Multiplikation von 16-Bit Matrizen werden SIMD Instruktionen verwendet, womit die Simulation einen Geschwindigkeitsvorteil gegenüber der 32-Bit Multiplikation bekommt.

Im letzten Beispiel wird die Simulation der Hotspot Funktion `pixel_satd_wxh16` des Video-Codecs x.264 zwischen den Architekturen `CoreVA_SIMD` und `CoreVA_VLIW` verglichen. Die Ergebnisse der Simulation werden in Tabelle 7.4 gezeigt. In diesem Bei-

Tabelle 7.4: Funktion `pixel_satd_wxh16` mit `CoreVA_SIMD` und `CoreVA_VLIW`.

Architektur	Instruktionen im Rumpf	Ausführungszyklen für eine Iteration	Host-Instruktionen pro Architektur-Instruktion
`CoreVA_SIMD`	39	50	27
`CoreVA_VLIW`	39	16	21

spiel wird verdeutlicht, dass die Ausnutzung der SIMD-Instruktionen einen deutlichen Geschwindigkeitsvorteil bei der Simulation mit sich bringt. Dabei benötigt die vierfach

parallele VLIW-Architektur nur 1/3 der Taktzyklen der einfach parallelen Architektur für die Ausführung einer Iteration.
Durch die Evaluierung der Simulation wird deutlich, dass mit *ViCE-UPSLA* generierte Simulatoren sehr effektiv ausgeführt werden. Für die Ziele der weiterführenden Entwicklung ist es sinnvoll, die Effizienz der Prozessorkonstrukte im Simulator genau zu ermitteln, um die Effektivität der Simulation weiter zu steigern.

7.5.1 Zusammenfassung der Evaluierung

Durch die Evaluierung des Werkzeugsystems wurde gezeigt, dass mit der Sprache *ViCE-UPSLA* reale Prozessoren auf einem hohen Abstraktionsniveau spezifiziert und validiert werden können. Dabei sind die visuellen Sprachkonstrukte für die Spezifikation von Prozessoren angemessen modelliert.

Bei der Evaluierung der Validierungsmethoden wurde gezeigt, dass durch die Integration in das Werkzeugsystem ein hoher Automatisierungssgrad erreicht wird, womit die Validierungsmethoden auch unter realen Bedingungen eingesetzt werden können.

Die automatisch generierten Simulatoren aus der Spezifikation werden effizient und realitätsnah simuliert, sodass das entwickelte Werkzeugsystem mit *ViCE-UPSLA* für die Entwicklung von Prozessoren, für die Durchführung von Instruktionssatzerweiterungen oder für die Entwurfsraumexploration eingesetzt werden kann.

8 Resümee

Das Ziel dieser Arbeit war es, den systematischen Entwurf von Prozessoren, ihre Evaluierung und die Validierung zentraler Aspekte der intendierten Mikroarchitektur zu unterstützen. Hierzu wurden die bekannten Konzepte und das Umfeld des Prozessorentwurfs untersucht und die Anforderungen für die Entwicklung einer domänenspezifischen visuellen Sprache und eines zugehörigen Werkzeugsystems definiert. Das realisierte Werkzeugsystem ermöglicht die Spezifikation, Simulation und Validierung von Prozessorentwürfen. Die Umsetzung der Konzepte in einem Werkzeugsystem hat den Vorteil, dass der Prozessorentwickler bereits bei der Beschreibung eines Prozessors unterstützt wird und in derselben Umgebung die Validierung des Prozessors durchführen kann.

Bei der Entwicklung der Sprache wurde besonderes Augenmerk auf die domänenspezifische Visualisierung der zentralen Prozessorkonstrukte gelegt. Dabei wurden in einer systematischen Vorgehensweise, wie in [Kla10] beschrieben, die Aufgaben der Prozessorentwickler und das Umfeld dieser Aufgaben untersucht. Das gesammelte Wissen floss in die Struktur der Sprache ein, sodass der strukturierte Entwurf von Instruktionssätzen/Prozessoren durch geeignete Sprachkonstrukte wie z.B. Adressierungsarten und Instruktionsformate ermöglicht wird. Neben den offensichtlichen Prozessorkonstrukten wurden strukturierende Konstrukte definiert. Die abstrakten Instruktionen z. B. fördern die Strukturierung der Entwürfe und dienen dazu, Redundanzen zu vermeiden. Die so entwickelte Sprache entspricht den hohen Anforderungen im realen Umfeld des Prozessorentwurfs. Das Verständnis der visuellen Beschreibung erfordert keine explizite Schulung der Prozessorentwickler. Mit der Sprache können die Prozessoren entlang ihres Instruktionssatzes und der Mikroarchitektur auf einem hohen Abstraktionsniveau spezifiziert werden.

Um die Simulation der spezifizierten Prozessoren zu ermöglichen, wurden Generatoren in das Werkzeugsystem integriert, mit denen Prozessorsimulatoren vollständig aus der Spezifikation generiert werden. Es handelt sich um zyklengenaue Mikroarchitektur- oder Instruktionssatzsimulatoren mit beliebig spezifizierten Interlock-Mechanismen oder Bypass-Strukturen. Damit kann das Werkzeugsystem bereits in frühen Phasen des Prozessorentwurfs eingesetzt werden und ermöglicht die Durchführung von Entwurfsszenarios wie Instruktionssatzerweiterung oder Entwurfsraumexploration.

Ein wichtiger Beitrag dieser Arbeit ist die Validierung von Prozessoren auf einem hohen Abstraktionsniveau ausgehend von ihrer Spezifikation. Hierfür wurden die typischen Fehlerquellen im Prozessorentwurf lokalisiert, klassifiziert und beschrieben. Damit wurden systematisch die Konzepte für die Validierungsmethoden erarbeitet und in dem Werkzeugsystem umgesetzt. Das Werkzeugsystem wurde durch die Konzepte des modellbasierten Testens zu einem Validierungswerkzeug mit der Möglichkeit der statischen und dynamischen Validierung der Entwürfe erweitert. Damit wird für typische Fehler in einer Prozessorspezifikation eine Sammlung von Validierungsmethoden bereitgestellt, womit z. B. die Testfälle automatisch oder mit geringem Spezifikationsaufwand für den jeweiligen Prozessor spezialisiert und mit passenden Wertebereichen instanziiert werden müssen. Außerdem wurde ein Konzept umgesetzt, das den Entwurf eigener Testfälle erlaubt.

Im Rahmen der Evaluierung wurden die Erwartungen an die entwickelte Sprache und die Validierungsmethoden überprüft und bestätigt. Dabei wurden die Konzepte und Konstrukte der Sprache untersucht. Ihre Evaluierung zeigt, dass die Sprache den Anforderungen für die Spezifikation realer Prozessoren auf einem hohen Abstraktionsniveau entspricht. Der Spezifikationsaufwand für die Erzeugung eines Simulators mit einer Teilmenge von Instruktionen, um z. B. die frühe Simulation bestimmter Anwendungen zu ermöglichen, ist sehr gering. Ausgehend von einer solchen Teilspezifikation kann die Prozessorspezifikation sukzessive vervollständigt werden.

Neben der Evaluierung der Sprache und der Simulation wurden auch die Validierungsmethoden untersucht. Dabei wurde ermittelt, dass die Erzeugung von Testfällen aus der Spezifikation automatisch durchgeführt werden kann. Auch die Anwendung der Validierungsmethoden ist praktikabel, wenn die Wertebereiche sinnvoll eingeschränkt werden. Hierfür wurde eine weitere Spezifikationssprache vorgestellt, mit der die Bearbeitung der Testfallspezifikationen komfortabel durchgeführt werden kann.

Ausblick Das entwickelte Werkzeugsystem und insbesondere die visuelle Sprache sind hinreichend vollständig und können real eingesetzt werden. Für die Weiterentwicklung der Sprache liegt das Potenzial in der Definition weiterer Sprachkonstrukte mit denen die Spezifikation zusätzlicher Prozessoreigenschaften ermöglicht wird. Z. B. könnte ein Konzept für die Spezifikation der bedingten Ausführung entwickelt werden, mit dem diese Eigenschaft global für alle Instruktionen eines Instruktionssatzes festgelegt werden kann. Außerdem kann im Rahmen einer zukünftigen Spracherweiterung untersucht werden, wie die Spezifikation und die Simulation des zyklengenauen Verhaltens iterativ ausgeführter Instruktionen, wie z. B. `LDM` (Load Multiple) oder `STM` (Store Multiple), mit *ViCE-UPSLA* umgesetzt werden kann.

Eine andere Entwicklungsrichtung wäre, das Potenzial und die Eignung dreidimensionaler visueller Sprachen zu betrachten. Damit kann wahrscheinlich die Qualität der Sprache bezüglich Ausdrucksfähigkeit und Visualisierung der Prozessorkonstrukte gesteigert werden.

Bei der Evaluierung des Werkzeugsystems wurden Unterschiede in der Simulationsgeschwindigkeit für SISD, SIMD und VLIW Architekturen festgestellt. Die Geschwindigkeit der Simulation kann optimiert werden, indem die verschiedenen Möglichkeiten für die Beschreibung der Prozessorstrukturen analysiert und die effizientesten Spezifikationskonstrukte übernommen werden. Damit können Richtlinien für eine effektive Modellierung von Prozessoren aufgestellt werden.

Die in dieser Arbeit entwickelten Validierungsmethoden überdecken allgemein die Prozessorkonstrukte, die in einer Spezifikation benutzt werden können. Durch die Fokussierung der Validierung auf bestimmte Eigenschaften von Prozessoren mit unterschiedlichen Design-Philosophien, wie SISD oder SIMD, kann die Effektivität und die Anwendbarkeit der Validierungsmethoden gesteigert werden. Damit können aus den vorhandenen Methoden durch die Analyse verschiedener Konzepte im Prozessorentwurf weitere spezialisierte Methoden abgeleitet werden.

Als eine Erweiterung des Werkzeugsystems könnte eine angepasste Darstellung der Ergebnisse für die dynamischen Validierungsmethoden entwickelt werden. Eine domänenspezifische Visualisierung der Validierungsergebnisse für konkrete Prozessorspezifikationen kann den Validierungsprozess unterstützen und damit die Qualität der Validierung steigern.

Abbildungsverzeichnis

Literaturverzeichnis

[ACE13] ACE. SuperTest compiler test and validation suite, 2013. http://www.ace.nl/compiler/supertest.html.

[Bab13] Boris Babajan. Основные Принципы Архитектуры e2k, 2013. http://www.mcst.ru/e2k_arch.shtml. [Online; Zugriff am 23. März 2013].

[BPK13] Florian Brandner, Viktor Pavlu, and Andreas Krall. Automatic generation of compiler backends. *Software: Practice and Experience*, 43(2):207—240, 2013.

[Bun10] Bundesministerium für Bildung und Forschung. Enablers of Ambiente Services and Systems Part C - Wide Area Courage, 2010. http://www.easy-c.de.

[CKK08] Bastian Cramer, Dennis Klassen, and Uwe Kastens. Entwicklung und Evaluierung einer domänenspezifischen Sprache für SPS-Schrittketten. *In Proceedings of Domain-Specific Modeling Languages DSML'08*, 21:59–73, 2008.

[CoW08] CoWare Inc. *LISA Language Reference Manual*, 2008.

[Dev10] V Developers. Valgrind user manual–callgrind, 2010.

[DHTK07] Ralf Dreesen, Michael Hußmann, Michael Thies, and Uwe Kastens. Register Allocation for Processors with Dynamically Reconfigurable Register Banks. *In Proceedings of the 5rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 5rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2007)*, 2007.

[DMK+06] Ajit Dingankar, Deepak A. Mathaikutty, Sreekumar V. Kodakura, Sandeep Shukla, and David Lilja. MMV: Metamodeling Based Microprocessor Validation Environment. *2006 IEEE International High Level Design ant Test Workshop*, 2006.

[Dre12] Ralf Dreesen. ViDL: A Versatile ISA Description Language. *In 19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS-19)*, 2012.

[Fle97] Wolfgang Fleisch. Validierung von entwurfsspezifikationen komponentenbasierter software für verteilte, eingebettete automatisierungssysteme mittels simulation. 4. Berichtskolloquium des GK PVS, 1997.

[FPF95] A. Furth, J. Van Preat, and M. Freerick. Describing Instruction Set Processors Using nML. *IEEE. The European Design and Test Conference(EDTC'95)*, pages 503–507, 1995.

[FR96] Michael J. Flynn and Kevin W. Rudd. Parallel Architectures. *ACM Computing Surveys*, 28(1), 1996.

[Git08] Claus Gittinger. Modellbasierte testentwicklung - verwendung von aktivitätsdiagrammen zur grafischen entwicklung von testfällen. In *GI Jahrestagung (1)'08*, pages 215–218, 2008.

[GK83] Daniel D. Gajski and Robert H. Kuhn. Guest Editors' Introduction: New VLSI Tools. *Computer*, 16(12):11–14, December 1983.

[GNRS09] Helmut Götz, Markus Nickolaus, Thomas Roßner, and Knut Salomon. *Modellbasiertes Testen - Modellierung und Generierung von Tests - Grundlagen und Kriterien für Werkzeugeinsatz, Werkzeuge in der Übersicht.* Heise Zeitschriften Verlag, 2009.

[GP96] T. R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[HKP+82] John Hennessy, Norman Kouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. MIPS: A Microprocessor Architecture. *IEEE*, 1982.

[HL10] Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer, 2010.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach.* Academic Press, iv edition, September 2006.

[Inc06] Inc Tensilica. *Tensilica Instruction Extension (TIE) Language*, 2006.

[Jam95] Tariq Jamil. RISC versus CISC. *Potentials, IEEE*, 14(3):13–16, August/-September 1995.

[JD11] Thorsten Jungeblut and Ralf Dreesen. *CoreVA Architekture Manual*, 2011.

[JDP+10] Thorsten Jungeblut, Ralf Dreesen, Mario Porrmann, Michael Thies, Ulrich Rückert, and Uwe Kastens. A Framework for the Design Space Exploration of Software-Dened Radio Applications. *2nd International ICST Conference on Mobile Lightweight Wireless Systems*, 2010.

[Jer00] Tor E. Jeremiassen. Sleipnir - An Instruction-Level Simulator Generator. *IEEE*, 2000.

[JSPR10] Thorsten Jungeblut, Gregor Sievers, Mario Porrmann, and Ulrich Rückert. Design Space Exploration for Memory Subsystems of VLIW Architectures. *In Proceedings of IEEE International Conference on Networking, Architecture, and Storage*, 2010.

[Jun11] Thorsten Jungeblut. *Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren.* PhD thesis, Universität Bielefeld, 2011.

[Kan88] Gerry Kane. *MIPS R2000 RISC Architecture.* Longman Higher Education, 1988.

[Kla10] Dennis Klassen. *Entwicklung einer visuellen domänenspezifischen Sprache: Konzeption, Implementation, Evaluation.* Vdm Verlag Dr. Müller, 2010.

[Kla13] Dennis Klassen. ViCE-UPSLA: A Visual High Level Language for Accurate Simulation of Interlocked Pipelined Processors. *In Proceedings of ATPS Workshop on SE13 Software Engineering*, 215:59–74, 2013.

[KLST04] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback Driven Instruction-Set Extension. In *Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, D.C., USA, June 2004.

[KRS94] Jens Knoop, Oliver Ruething, and Bernhard Steffen. Partial Dead Code Elimination. *PLDI '94 Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 29(6):147–158, 1994.

[Lau09] Kim Lauenroth. *Konsistenzprüfungen von Domänenanforderungsspezifikationen.* Logos Verlag Berlin, 2009.

[LEL99] Rainer Leupers, Johann Elster, and Birger Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators. *in: Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 339–342, 1999.

[Lim87] ARM Limited. *ARM Instruction Set. Final - Open Access.* Acorn Computers Limited, 1987.

[Lim95] ARM Limited. *ARM 7TDMI Data Sheet, Open Access.* Copyright Advanced RISC Machines Ltd (ARM) 1995, arm ddi 0029e edition, August 1995.

[LS04] Wolfgang Lehner and Harald Schöning. *XQuery - Grundlagen und fortgeschrittene Methoden.* Dpunkt Verlag, 2004.

[LWS94] Gunther Lehmann, Bernhard Wunder, and Manfred Selz. *Schaltungsdesign mit VHDL.* © G. Lehmann/B. Wunder/M. Selz, 1994.

[MCS13] MCST.ru. Микропроцессор нового поколениа Эльбрус (Mikroprozessor der neuen Geneartion Elbrus), 2013. http://www.mcst.ru/b_4-5.shtml. [Online; Zugriff am 23. März 2013].

[MD03] Prabhat Mishra and Nikil Dutt. A Methodology for Validation of Microprocessors using Equivalence Checking. *Proceedings of the Fourth International Workshop on Microprocessor Test and Verification Common Challenges and Solutions (MTV'03)*, page 83, 88 2003.

[Mic94] IBM Microelectronics. *PowerPC 604: RISC Microprozessor User's Manaul.* IBM Microelectronics, 1994.

[Mor06] Dmitrij Moroz. Суровая правда, скрытая за розовыми очками: история компании Transmeta (Bittere Wahrheit hinter der rosa Brille: Geschichte der Firma Transmeta). *Системный Администратор (Systemadministrator)*, (9), September 2006.

[Mot92] Motorola Inc. *MOTOROLA M68000 FAMILY Programmer's Reference Manual*, 1992.

[NMEH81] Robert N. Noyce and Jr Marcian E. Hoff. A History of Microprocessor Development at Intel. *IEEE MICRO*, February 1981.

[NT08a] Arilo Claudio Dias Neto and Guilherme Horta Travassos. A Surveying Model Based Testing Approaches Characterization Attributes. *ESEM '08 Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 324–326, 2008.

[NT08b] Arilo Claudio Dias Neto and Guilherme Horta Travassos. Supporting the Selection of Model-based Testing Approaches for Software Projects. *AST '08 Proceedings of the 3rd international workshop on Automation of software test*, pages 21–24, 2008.

[PC81] B. Sc P. Corcoran. Simulator generator system. *In Proceedings of IEEE Computers and Digital Techniques*, 128:61–63, 1981.

[PD08] Mishra Prabhat and Nikil Dutt, editors. *Processor Description Languages (Morgan Kaufmann Series in Systems on Silicon)*, volume 1. Morgan Kaufmann Publishers/Elsevier, 2008.

[Pie95] Georg Piepenbrock. *Methoden des Software-Pipelining für Prozessoren mit Instruktionsparallelität.* PhD thesis, Universität-Gesamthochschule Paderborn, 1995.

[PP05] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Methodological Issues in Model-Based Testing. *Model-Based Testing of Reactive Systems*, 2005.

[Ram89] Franz J. Rammig. *Systematischer Entwurf digitaler Systeme: Von der System- bis zur Gatter-Ebene.* B.G. Teubner Verlag Stuttgart, 1989.

[RBGW10] Thomas Roßner, Christian Brandes, Helmut Götz, and Mario Winter. *Basiswissen Modellbasierter Test.* Dpunkt Verlag, August 30 2010.

[Sch06] Carsten Schmidt. *Generierung von Strukturediteren für anspruchsvolle visuellen Spezifikationen.* PhD thesis, Universität-Paderborn, 2006.

[Sch07] I. Schieferdecker. Modellbasiertes Testen. *OBJEKTspektrum*, pages 39–45, Mai/Juni 2007.

[Sed92] Robert Sedgewick. *Algorithmen.* Addison-Wesley, Januar 15, 1992.

[SG79] Edward Stritter and Tom Gunter. Microsystems a Microprocessor Architecture for a Changing World: The Motorola 68000, February 1979.

[Sie12] Christian Siemers. Rechnerarchitektur I/II. Technical report, TU Clausthal, 2012.

[SL05] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest.* Dpunkt, Juli 8 2005.

[SÖ10] Robin Sving and Peter Öman. Pilot project for model based testing using conformiq qtronic. Master's thesis, UPPSALA Universitet, 2010.

[Tah94] Dipl.-Ing. Sofiène Tahar. *Eine Methode zur formalen Verifikation von RISC-Prozessoren.* VDI Verlag, 1994.

[TKPD11] G. Theodorou, N. Kranitis, A. Paschalis, and D.Gizopoulos. A Software-Based Self-Test Methodology for On-Line Testing of Processor Caches. *IEEE*, 2011.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, March 2007.

[Wec08] Dieter Wecker. *Einführung in den Prozessorentwurf: Von der Planung bis zum Prototyp.* Expert-Verlag, August 2008.

[ZTM95] Vojin Zivojnovic, Steven Tjiang, and Heinrich Meyr. Compiled Simulation of Programmable DSP Architectures. *IEEE Workshop on VLSI Signal Proccessing Sakai, Japan*, pages 187–196, 1995.