Eine Methode zur formalen Verifikation von RISC-Prozessoren

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Fakultät für Informatik
der Universität Karlsruhe (Technische Hochschule)

vorgelegte

Dissertation

von

Dipl.-Ing. Sofiène Tahar
aus Sousse, Tunesien

Tag der mündlichen Prüfung: 20. Dezember 1994

Erster Gutachter: Prof. Dr.-Ing. D. Schmid
Institut für Rechnerentwurf und Fehlertoleranz

Zweiter Gutachter: Prof. Dr. P. Deussen
Institut für Logik, Komplexität und Deduktionssysteme

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe.

An dieser Stelle möchte ich allen danken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. D. Schmid für die Betreuung dieser Arbeit, die Übernahme des Referates, sowie für seine großzügige Unterstützung und sein stetes Interesse.

Herrn Prof. Dr. P. Deussen (Institut für Logik, Komplexität und Deduktionssysteme der Universität Karlsruhe) danke ich für die Übernahme des Korreferates und die sorgfältige Begutachtung der Arbeit.

Weiterhin danke ich allen Mitarbeiterinnen und Mitarbeitern am Institut für Rechnerentwurf und Fehlertoleranz für die kooperative Zusammenarbeit. Besonderen Dank schulde ich Herrn Dr. Rammaya Kumar, Herrn Dr. Thomas Kropf, Herrn Klaus Schneider und Herrn Ralf Reetz für die initiale Anregung zu dieser Arbeit, die ständige Bereitschaft zu klärenden Diskussionen sowie für die vielen konstruktiven Vorschläge. Ferner möchte ich Herrn Dirk Eisenbiegler meinen Dank für die orthographische Korrektur der Abhandlung aussprechen.

Schließlich möchte ich den Studenten Ghassen Ben Amor und Mathias Dehof für ihre effiziente und zuverlässige Unterstützung bei den vielen Implementierungsarbeiten danken.

Karlsruhe, im Dezember 1994 *Sofiène Tahar*

Für Rym und Mariam

Inhaltsverzeichnis

Liste der Symbole

Abkürzungen

ALU	Arithmetic Logic Unit
BDD	Binary Decision Diagram
BTA	Branch Target Address
CISC	Complex Instruction Set Processor
CMOS	Complementary Metal Oxide Semiconductor
EBM	Electronic Block Model
EDIF	Electronic Design Interchange Format
FIFO	First In First Out
FPU	Floating Point Unit
HDL	Hardware Description Language
HOL	Higher Order Logic
I/O	Input/Output
IR	Instruction Register
MMU	Memory Management Unit
NOP	No Operation
PC	Program Counter
RF	Register File
RISC	Reduced Instruction Set Processor
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

Logik

$\neg$	Negation (NICHT)
$\wedge$	Konjunktion (UND)
$\vee$	Disjunktion (ODER)
$\Rightarrow$	Implikation
$\Leftrightarrow$	Äquivalenz
$=$	Gleichheit
$\neq$	Ungleichheit
$\bigwedge x_i$	Konjunktion der x_i: $\bigwedge x_i = x_1 \wedge x_2 \wedge \ldots$
$\bigvee x_i$	Disjunktion der x_i: $\bigvee x_i = x_1 \vee x_2 \vee \ldots$
$:=$	wird definiert als (Gleichheit per Definition)

$\forall x .$	für alle *x* gilt
$\exists x .$	es existiert ein *x* für das gilt
x / P	*x* so daß *P*
$f\colon A \mapsto B$	Funktion *f* hat Definitionsbereich *A* und Wertebereich *B*
$f \circ g$	Komposition der Funktionen *f* und *g*
$a \in A$	*a* ist in der Menge *A* enthalten
$a \notin A$	*a* ist nicht in der Menge *A* enthalten
$A \# B$	*A* ist Teilmenge von *B*
$A \cap B$	Schnitt der Mengen *A* und *B*
$A \cup B$	Vereinigung der Mengen *A* und *B*
$\mathrm{I}t \rightarrow t_1 \mid t_2$	Wenn *t* wahr ist dann t_1 sonst t_2 (bedingter Term)
$x{:}\sigma$	Variable *x* hat den Typ σ
$\sigma' \rightarrow \sigma''$	Funktionstyp der Argumenten- und Werte-Typen σ' bzw. σ''
$t[a/x]$	Substitution von *x* durch *a* in *t*
$\lambda x.\ t$	λ-Term
$\varepsilon x . Px$	ε-Term (Auswahl-Term)
let $x = t_1$ in t_2	Variable *x* in t_2 durch t_1 ersetzen (Let-Term)
$\vdash_{def}$	Definition in Logik
$\vdash$	Theorem in Logik
T, F	Konstante “wahr” (*True*) und “falsch” (*False*)

Pipelining

IF	Instruction Fetch (Befehl holen)
ID	Instruction Decode (Befehl dekodieren)
EX	Instruction Execute (Befehl ausführen)
MEM	Memory access (Speicherzugriff)
WB	Write Back (Resultat schreiben)
ϕ_1	Taktphase 1
ϕ_2	Taktphase 2
n_s	Pipelinetiefe
n_p	Anzahl der Phasen des Taktes
u	Befehlszykluseinheit
t	Taktzykluseinheit
τ	Taktphaseneinheit
C	Kontextparameter

Funktionen

f_A,f_L,f_S,f_C,	Klassenabstraktionsfunktionen
f_z	Zeitabstraktionsfunktion
ψ_i	Pipelinestufenbezeichner ($\psi_1 = IF$, $\psi_2 = ID$, $\psi_3 = EX$, usw.)
φ_k	Taktphasenbezeichner ($\varphi_1 = \phi_1$, $\varphi_2 = \phi_2$, usw.)
θ	Ordnungswert einer Pipelinestufe
ξ	Ordnungswert einer Taktphase
ϑ	Typ eines Stufenbefehls
$\mathcal{S}_k$, $\mathcal{S}_s$, $\mathcal{S}_p$	Beweiszielsetzfunktion der Klassen-, Stufen bzw. Phasenebene
$\mathcal{T}$	parametrisierbare Taktik der semantischen Korrektheit

Befehlsklassen

ALU	Arithmetisch-logische Befehle
CONTROL	Kontrollbefehle
LOAD	Ladebefehle
STORE	Speicherbefehle

Registerbezeichner

A	ALU-Eingangsregister
ALUout	ALU-Ausgangsregister
ALUout1	Pipelinepuffer für ALUout
B	ALU-Eingangsregister
BTA	Phasenpuffer für PC
DMAR	Adressregister für Speicherdaten
D-MEM	Datenspeicher
I-MEM	Befehlsspeicher
IR	Befehlsregister
IR1…IR3	Pipelinepuffer für IR
LMDR	Datenregister für Ladebefehle
PC	Befehlszähler
RF	Registersatz
SMDR	Datenregister für Speicherbefehle

Notation bei Speicherelementen

$R(t)$	Inhalt von Register R zum Zeitpukt t
$R' \leftarrow R''$	Übertragen des Inhalts von R' nach R'' (Register-Transfer)
$R_{n\ldots p}$	Teilwort von R mit den Bitstellen n bis p ($n < p$)
$M\ [ADR]$	Wort im Speichersatz M mit der Adresse im Adresswort ADR
$M(t)\ [ADR\ (t')]$	wie oben aber mit zeitbehafteter Zustandsdarstellung

Kurzfassung

Die Hardware-Verifikation mit formalen Systemen gewinnt aus dem Blickwinkel der Hardware-Entwerfer immer mehr an Interesse. Dies liegt daran, daß mit einer steigenden Integrationsdichte bei immer komplexeren Systemen konventionelle Validierungsverfahren wie die Simulation für den Nachweis der Korrektheit eines Entwurfes ausscheiden. Bei der Anwendung formaler Methoden zur Validierung realistischer Hardware-Entwürfe ist die Automatisierung der Verifikationsverfahren unabdingbar. In der vorliegenden Arbeit wird Logik höherer Ordnung wegen deren Sprachmächtigkeit verwendet, jedoch ist dieser Formalismus nur schwach automatisierbar. Man ist deshalb bestrebt, eine Automatisierung des Verifikationsprozesses durch systematische Verwendung von schaltungsspezifischen Heuristiken und Methoden zu realisieren, welche die formale Spezifikation und Verifikation für eine Klasse von Schaltungen weitgehend unterstützen.

Im Rahmen dieser Arbeit werden in dieser Hinsicht RISC-Prozessoren als eine Klasse von realen Entwürfen bzw. Schaltungen betrachtet. Sie sind vielfältig anwendbar und finden in den letzten Jahren eine immer breitere Anwendung. Zur Steigerung der Rechenleistung verwenden RISC-Prozessoren eine Pipelineverarbeitung, die zu komplexeren Hardware-Entwürfen führt, deren Korrektheit aber nicht mehr ohne weiteres gesichert ist. Darüberhinaus werden RISC-Prozessoren zunehmend bei sicherheitskritischen Anwendungen eingesetzt, die eine korrekte Implementierung erfordern.

Der Stand der Technik in der formalen Verifikation von Mikroprozessoren ist gekennzeichnet durch die Verifikation von einzelnen mikroprogrammierten Prozessoren, die meist stark vereinfachte, unrealistische Modelle von CISC-Prozessoren darstellen. In der vorliegenden Arbeit wird erstmals eine Methode entwickelt, welche, basierend auf einem hierarchischen Modell, einen allgemeinen Rahmen zur formalen Spezifikation und Verifikation von herkömmlichen RISC-Prozessorkernen beschreibt. Das entwickelte hierarchische Interpreter-Modell entspricht der Natur eines RISC-Prozessorentwurfs und berücksichtigt vor allem die parallele Pipelineverarbeitung der RISC-Prozessoren. Zur Beherrschung der Verifikationskomplexität werden Befehle mit ähnlicher Semantik in Klassen zusammengefaßt, wonach die verschiedenen Abstraktionsebenen parametrisiert werden. Die formalen Spezifikationen der verschiedenen Befehle können dann leicht aus informalen Beschreibungen der Architektur bzw. der Pipelinestruktur abgeleitet werden.

Wegen der Pipelineverarbeitung der Befehle spielen temporale Aspekte bei der Formalisierung der Modellebenen eine große Rolle. Deshalb wurde ein Zeitabstraktionsmechanismus entwickelt, der die temporalen Gegebenheiten einer Befehlsausführung in einer Pipeline zu beschreiben gestattet. Anhand dieses Zeitabstraktionskonzeptes werden die Befehle aller Abstraktionsebenen mit verschiedenen Zeitgranularitäten spezifiziert und beim Korrektheitsbeweis in einem Zusammenhang gebracht. Das hierarchische Modell und die entsprechende Formalisierung können nicht nur zur Verifikation, sondern auch zur Simulation oder in der formalen Synthese verwendet werden.

Basierend auf diesem hierarchischen Modell wird der Korrektheitsbeweis eines RISC-Prozessors in zwei unabhängigen Schritten realisiert. Im ersten Schritt wird die Korrektheit der Pipeline-Verarbeitung bewiesen, indem unter Annahme von Software-Randbedingungen einer gegebenen Architektur gezeigt wird, daß die Hardware keine Pipeline-Konflikte (Ressourcen-, Daten- und Kontrollkonflikte) zuläßt. Im zweiten Schritt wird die semantische Korrektheit der Befehle verifiziert, indem hierarchisch gezeigt wird, daß die Hardware-Implementierung die Semantik jedes Maschinenbefehls erfüllt.

Zur Automatisierung des Verifikationsprozesses wurde eine Reihe von parametrisierten Taktiken entwickelt, die unabhängig von der gegebenen Architektur allgemein anwendbar sind. Darüberhinaus ist das implementierte Verifikationsverfahren konstruktiv, d.h. bei einem Fehlschlag des Korrektheitsbeweises werden nützliche Hinweise auf das Fehlverhalten ausgegeben, um entsprechende Veränderungen am Entwurf vornehmen zu können. Die gesamte Verifikationsmethode kann sowohl in einem verifikationsbasierten Entwurf als auch zur nachträglichen Verifikation verwendet werden.

Die entwickelte Spezifikations- und Verifikationsmethode wird schließlich anhand des RISC-Prozessors DLX validiert, der in einer kommerziellen Entwurfsumgebung realisiert wurde und der die Komplexität herkömmlicher RISC-Kerne besitzt. Der DLX-Prozessor umfaßt die wesentlichen RISC-Konzepte der Prozessoren Intel i860, Sun SPARC, Motorola M88000 und MIPS R3000.

Kapitel 1: Einleitung

1.1 Motivation

Heutige VLSI Schaltungen haben, bedingt durch die steigende Integrationsdichte und neue Technologien, einen sehr hohen Komplexitätsgrad erreicht. Durch neue Anwendungsbereiche, insbesondere in sicherheitskritischen Systemen, stehen VLSI Schaltungen unter dem Zwang, daß sie fehlerfrei sein müssen. Die steigende Entwurfskomplexität führt jedoch dazu, daß Korrektheitsaussagen über eine Komponente eines großen und komplexen Systems immer schwieriger zu treffen sind.

Hinsichtlich der auftretenden Fehler unterscheidet man bei VLSI Schaltungen üblicherweise zwischen *Entwurfs-* und *Fertigungsfehlern*. Fertigungsfehler entstehen bei der Produktion von integrierten Schaltungen, indem physikalische Defekte zum Fehlverhalten einer (korrekt) entworfenen Schaltung führen. Das Problem der Fertigungsfehler ist heutzutage reichlich erforscht und man findet eine Vielzahl von *Testverfahren* [Wund91] zur Aufdeckung von Fertigungsfehlern, die auch in der Praxis erfolgreich eingesetzt werden. Hohe Maßstäbe wurden damit bei industriellen Testverfahren erreicht.

Entwurfsfehler entstehen, wenn die beabsichtigte Funktion einer Schaltung nicht mehr erfüllt ist. Diese Fehler können während der manuellen oder automatischen Umsetzung einer Schaltungsspezifikation in eine Implementierung oder auch bei nachträglichen Schaltungsoptimierungen (z.B. zum Einsparen der Silizium-Fläche oder zum Erhöhen der Arbeitsgeschwindigkeit) zustande kommen.

Simulation ist zur Zeit die *Validierung*smethode, die üblicherweise in VLSI CAD-Systemen zur Überprüfung der Funktionskorrektheit von Hardware-Systemen verwendet wird [HöMS87, Cade91]. Sie ist automatisch ausführbar und wird von Hardware-Entwerfern als Hauptwerkzeug eingesetzt, um Entwurfsfehler zu entdecken. Wegen der großen Rechenzeiten, der mühsamen Bestimmung der notwendigen Stimuli sowie der langwierigen Auswertung der Simulationsergebnisse ist es jedoch bei komplexen Schaltungen und Systemen praktisch nicht möglich, eine erschöpfende Simulation in angemessener Zeit durchzuführen [CaPr88, Ramm89]. Üblicherweise wird nach dem Entwurf eine reduzierte Untermenge der Testmuster in der Hoffnung simuliert, in diesem Prozeß keine Fehler zu übersehen [Gupt92]. Jedoch bleiben häufig Fehler unentdeckt [Yoel90], und es existiert eine Menge von Beispielen, bei denen Fehler zu spät im Entwurfsprozeß oder sogar erst nach der kommerziellen Vermarktung eines Produkts entdeckt wurden [Gupt92]. Erfahrungen aus der Industrie besagen, daß aufgrund mangelnder Validierung mittels Simulation bis zur Hälfte aller neuen kundenspezifischen Schaltungsentwürfe (sog. “ASICs”) bei ihrem Einsatz nicht das gewünschte Verhalten aufweisen [HuRo91], was zu kostspieligen Neuentwurfszyklen führen kann [Ande91]. Auch Verfahren der *symbolischen* Simulation [CaJB79] brachten nicht die erhoffte Lösung des Problems. Mit diesem neuen Ansatz erzielt man zwar eine bemerkenswerte Verbesserung gegenüber der konventionellen Simulation [CaPr88], indem statt binärer Eingangsvektoren Symbole verwendet werden, welche ganze Klassen von Stimuliwerten repräsentieren. Trotz neuerer Methoden [Brya90] stößt man jedoch aus Komplexitätsgründen auch hier recht schnell auf Grenzen.

Eine Alternative könnte der Einsatz von Techniken der automatischen *Synthese* sein, die auf dem sogenannten “korrekten Konstruieren” (engl. “correctness-by-construction”) beruhen. Synthesesysteme garantieren aber keine vollständige Korrektheit der Schaltung, da es sich bei ihnen im allgemeinen um sehr komplexe Softwaresysteme handelt, die schwer beherrschbar und somit fehleranfällig sind [McPC88]. Solange Synthese-Algorithmen noch nicht formal verifiziert sind, kann eine automatische Synthese deshalb keinen korrekten Entwurf garantieren [Chin90, BuVe91]. Außerdem ist der heutige Stand der Technik so, daß in der Praxis Synthese-Verfahren in der Automatisierung des Entwurfs vorwiegend auf niedrigen (Gatter-) Ebenen erfolgreich eingesetzt werden. Auf höheren Ebenen hingegen liefert die automatische Synthese oft noch nicht die Qualität eines Handentwurfs.

Weil es unmöglich ist, eine vollständige Korrektheit durch Simulation zu erreichen und heutige Synthesesysteme noch nicht in der Lage sind, einen korrekten Entwurf zu garantieren, werden schon seit längerem neue Methoden für eine kostengünstige und zuverlässige Validierung von Schaltungsentwürfen gesucht [JLRS91]. Eine relativ neue und zugleich attraktive Möglichkeit stellen in dieser Hinsicht Verfahren der *Hardware-Verifikation* dar [CaPr88, Yoel90, Gupt92]. Korrektheit wird hier in ihrer exakten mathematischen Bedeutung verstanden, indem durch einen formalen Beweis gezeigt wird, daß der Entwurf frei von Fehlern ist [Evek91b]. Die Betrachtung aller Fälle ist implizit in der Methodik der formalen Verifikation enthalten [CaPr88], da Korrektheitsaussagen ohne Rücksicht auf Eingangswerte gelten [Gupt92]. Die Hardware-

Verifikation ermöglicht es somit, die komplette Korrektheit zu überprüfen. Sie kann außerdem bereits während des Entwurfsprozesses eingesetzt werden und nicht erst zur nachträglichen Überprüfung des Entwurfs [Gupt92].

Verifikation und Synthese schließen sich gegenseitig nicht aus [Evek87, Ueha87, Chin90], denn Techniken der formalen Verifikation werden weiterhin benötigt, um Prozeduren der Syntheseprogramme auf Korrektheit zu prüfen [BrLe90, Gupt92]. In dieser Hinsicht gab es in der letzten Zeit interessante Konzepte des verifikationsfreundlichen Entwurfs (engl. “design for verifiability”) [Miln90, BuVe91] und der formalen Synthese [FoMa89, HaLD89], die Theorembeweiser zugrundelegen, um korrektheitserhaltende Transformationen der Spezifikation in eine Implementierung zu erhalten [DeMa89, Camp90, Basi91].

Die formale Verifikation wird vermutlich auch in Zukunft die Simulation nicht vollständig ersetzen [Yoel90]. Ein Hybridansatz [Miln85, Cami88], in dem die Vorteile der Simulation mit der Genauigkeit und Beweismächtigkeit der formalen Methoden kombiniert werden, ist in vieler Hinsicht vorteilhafter. Hierbei gewinnt ein Entwerfer mehr Vertrauen in die Korrektheit seines Entwurfs, indem zuerst eine Simulation durchgeführt wird, um die abstrakte Spezifikation des Entwurfs zu überprüfen und dann eine formale Verifikation des Entwurfs gegenüber seiner Implementierung folgt [CaPr88]. Ein wichtiger Punkt solcher Hybridansätze ist die Möglichkeit der Verwendung derselben formalen Beschreibung sowohl für das Simulationsprogramm als auch für das Verifikationswerkzeug [Miln85]. Das verlangt aber, daß die formalen Beschreibungen dafür geeignet sind [Joyc88, Cami88].

Es gibt verschiedene Ansätze zur formalen Hardware-Verifikation, wie z.B. Tautologieprüfung (engl. “tautology checking”), Theorembeweiser oder Modellüberprüfung (engl. “model checking”). Diese Methoden weisen unterschiedliche Stärken und Schwächen auf und werden in Kapitel 1.2.2 näher erläutert.

1.2 Hardware-Verifikation

Im Rahmen der Hardware-Verifikation soll zunächst geklärt werden, was unter *Korrektheit* eines Entwurfs zu verstehen ist.

1.2.1 Korrektheit

Für die Korrektheit gibt es zwei konventionelle Definitionen [CaPr88, StBE88]:

1. Korrektheit wird im Bezug auf eine *Eigenschaft* einer Hardware definiert. Demnach ist eine Hardware korrekt, wenn bewiesen werden kann, daß eine Schaltung diese Eigenschaft erfüllt. Solche Eigenschaften gruppieren sich in zwei Klassen [CaPr88]: *Sicherheit*, welche bedeutet, daß ein System sich niemals in einem undefinierten Zustand befinden darf, und *Lebendigkeit*, die besagt, daß es für jeden Zustand, in dem sich das System befindet, einen Nachfolgezustand gibt, der auch erreicht werden kann.
2. Korrektheit wird im Zusammenhang mit zwei Beschreibungen der Schaltung auf unterschiedlichen Abstraktionsebenen definiert, nämlich einer Spezifikation und einer Implementierung. Demnach heißt eine Schaltung korrekt, wenn bewiesen werden kann, daß die Implementierung die Spezifikation erfüllt [CaPr88, Yoel90, Gupt92]. Unter *Spezifikation* wird dabei die formale Beschreibung des beabsichtigtem Verhaltens der Schaltung verstanden, während *Implementierung* die formale Beschreibung der konkreten Realisierung der Schaltung bezeichnet.

Die erste Definition der Korrektheit wird meist im Zusammenhang mit Programmverifikation verwendet. Die zweite hingegen ist an den Entwurfsprozeß angelehnt und wird in dieser Arbeit zugrundegelegt. Spezifikation und Implementierung bezeichnen dabei Beschreibungsformen auf irgendeiner Ebene der Entwurfshierarchie, die von der höchsten Abstraktionsebene (Systemebene) bis zur niedrigsten (Realisierungs) Ebene, üblicherweise dem physikalischen Layout reichen kann [GaKu83, WaTh85, Ramm89]. In einer Entwurfshierarchie kann die Implementierung einer Ebene als Spezifikation der nächst niedrigeren Ebene dienen. Der effektivste Einsatz der Hardware-Verifikation liegt auf höheren Ebenen des VLSI-Entwurfs [JLRS91] und wird deshalb entsprechend in dieser Arbeit verfolgt.

1.2.2 Stand der Technik

In den letzten Jahren gab es viele Versuche, die formale Verifikation auf VLSI-Schaltungen anzuwenden. Es existiert jedoch ein Lücke zwischen den erzielten Forschungsfortschritten und deren Anwendung in der realen Welt [Sriv91]. Diese Situation ist zum Teil durch skeptische und konservative Einstellungen gegenüber formalen Verfahren und deren Einsatz verursacht [Hall90, Cook92]. Jedoch gibt es auch berechtigte Gründe dafür, denn heutige Verifikationswerkzeuge können entweder nur eingeschränkt bei bestimmten Problemklassen eingesetzt werden oder erfordern grundlegendes Wissen über mathematische Logik, das oft bei Entwerfern nicht vorhanden ist. Damit formale Methoden ihren Forschungscharakter verlieren und von Hardware-Entwerfern akzeptiert werden, muß eine Reihe von Bedingungen erfüllt sein [CaPr88, Levy91, KrKS93]:

- Hoher Automatisierungsgrad,
- Anwendbarkeit auf *realistische* komplexe Schaltungen,
- mächtige Spezifikationsformalismen, die eine natürliche Beschreibung erlauben und Anforderungen der heutigen Technik genügen,
- Möglichkeit der hierarchischen und modularen Verifikation und
- Integration der Verifikationswerkzeuge in kommerzielle VLSI CAD-Systeme.

Der theoretische Hintergrund der formalen Verifikation ist gegeben durch formale Systeme wie Aussagenlogik, temporale Aussagenlogik, Prädikatenlogik oder Prozeßalgebra [Evek91a]. Die herangezogene Logik bestimmt die Ausdrucksstärke und den Grad der Automatisierung des verwendeten Verifikationsverfahrens. In der Literatur wurde in diesem Zusammenhang eine Fülle von Formalismen vorgeschlagen [CaPr88, StBE88, Yoel90, Gupt92]. Die Auswahl zwischen den Alternativen der vielen Verifikationsansätze ist bestimmt durch die Betrachtung der verschiedenen Eigenschaften der Verifikationssysteme [Levy91, Gupt92, AVCM93]: Korrektheit und Vollständigkeit der Beweismethoden, Automatisierungsgrad (automatisch, halbautomatisch oder manuell) sowie Modellierungsmächtigkeit und Werkzeugunterstützung.

In den letzten Jahren haben sich dabei vor allem zwei Gruppen von Ansätzen herauskristallisiert [Keut91, Gupt92, KuSK93]: *automatische,* auf *entscheidbare* Verfahren basierte Ansätze und *interaktive,* auf *nicht entscheidbare* Verfahren basierte Ansätze. Legt man Aussagenlogik oder temporale Aussagenlogik zugrunde, so lassen sich effizient implementierbare vollautomatische Verfahren angeben. Die exponentiell wachsende Komplexität führt jedoch dazu, daß nur Probleme bis zu einer bestimmten Größe handhabbar sind. Verwendet man dagegen Prädikatenlogik erster oder höherer Ordnung, so können aufgrund der Sprachmächtigkeit komplexe Sachverhalte auf höheren Abstraktionsebenen beschrieben werden. Da jedoch für Prädikatenlogik keine Entscheidungsverfahren mehr existieren [Chur36], ist eine Verifikation hier auf eine starke Interaktion mit dem Entwerfer angewiesen. In den nächsten Abschnitten werden Ansätze beider Gruppen ausführlicher beschrieben.

1.2.2.1 Automatische Ansätze

Die *automatischen* Ansätze basieren auf Aussagenlogik, temporaler Aussagenlogik und auf Verfahren zum Vergleich von Automaten. Durch die Existenz von Entscheidungsverfahren sind vollautomatisierte Werkzeuge möglich [BiMa88]. Aussagenlogik ist *vollständig* und *entscheidbar* [Bläs87, Schö89]. Die Verfahren, die auf Aussagenlogik basieren, sind daher gut automatisierbar, jedoch sind ihre Ausdrucksmittel für die Verifikation von Schaltungen sehr begrenzt. Für spezielle Anwendungen, wie die Verifikation von Schaltnetzen auf Logikebene, reicht aber die Aussagenlogik aus.

In den letzten Jahren wurden insbesondere bei der Entwicklung von automatischen Beweisern der Aussagenlogik, den *Tautologieprüfern*, große Fortschritte mit Hilfe von Entscheidungsgraphen (engl. “BDD’s” für “binary decision diagrams”) erzielt. Diese Graphen wurden zuerst in [Aker78] zur Repräsentation von Booleschen Funktionen vorgeschlagen und von Bryant [Brya86] derart ergänzt, daß daraus eine Normalform-Darstellung entstand. Diese Darstellung wird erfolgreich zur Verifikation komplexer Schaltnetze eingesetzt [MWBS88, MaBi88] und ständig in ihrer Effizienz verbessert [MiLY90], jedoch steigt die Komplexität der Verifikation mit Tautologieprüfern exponentiell an, so daß der Anwendung dieses Verfahrens Grenzen gesetzt sind.

Für die Verifikation von Schaltnetzen ist die klassische Aussagenlogik einsetzbar, jedoch reicht sie für die Verifikation von Schaltwerken nicht mehr aus. Hierzu wird auf die Automatentheorie zurückgegriffen, indem sequentielle Schaltungen durch endliche Automaten [Koha78] dargestellt werden, so daß sich die Schaltungsäquivalenz einfach auf die wohldefinierte Automatenäquivalenz zurückführen läßt. Der Vergleich zweier endlicher Automaten geschieht durch Konstruktion des Produktautomaten, dessen Ausgang bei Gleichheit den Wahrheitswert “wahr” annimmt [SuFr86]. Obwohl es effiziente Verfahren zur Bildung des Produktautomaten gibt [CoMa91], ist ein expliziter Durchlauf aller erreichbaren Zustände im allgemeinen wegen des großen Speicherplatz- und Rechenzeitbedarfs nicht möglich [Evek91b].

Der Einsatz temporaler Logik zur Hardware-Spezifikation und Verifikation wurde zuerst von Pnueli [Pnue77] eingeführt. Bei dieser Logik handelte es sich um eine Erweiterung der klassischen Aussagenlogik um die temporalen Operatoren “Next”, “Always”, “Sometimes” und “Until”. Abhängig von der Ordnung der Zeitpunkte oder Intervalle der Zukunft unterscheidet man zwischen Ansätzen mit linearer [BaVe86, FuFu89] oder “verzweigender” [ClES83, ClLM89] temporaler Logik. Die lineare Zeitdarstellung ermöglicht bessere Ausdrucksmöglichkeiten, während verzweigende temporale Aussagenlogik bessere Entscheidungsmöglichkeiten bietet. Eine weitere, in der Verifikation erfolgreich eingesetzte Variante temporaler Logik beruht auf sog. “Zeitintervallen”. Sie wurde erstmals in [Mosz83] zur Hardware-Beschreibung eingeführt; später wurde durch eine Erweiterung der Programmiersprache PROLOG ermöglicht, über Zeitintervalle zu schlußfolgern [FKTM86].

Basierend auf einem Modell mit “verzweigender Zeit” [ClES83] wurde ein verwandter Ansatz entwickelt, der sowohl temporale Aussagenlogik als auch die Automatentheorie zugrundelegt und auf dem Prinzip der Modellüberprüfung [BrCD85] beruht. Hierbei besteht die Modellüberprüfung aus dem Beweis, daß eine Spezifikation, gegeben als temporallogische Formel, bezüglich des semantischen Modells der Implementierung, gegeben als endlicher Automat, wahr ist [Gupt92]. Dank der Entscheidbarkeit des zugrundeliegenden Kalküls ist eine komplette automatische Beweisausführung möglich, und es läßt sich durch Implementierung von effizienten Algorithmen [BCMD91] ein hoher Automatisierungsgrad erreichen. Wegen der Ähnlichkeit des prinzipiellen Ansatzes mit den Verfahren der Automatenäquivalenz trifft hier die gleiche Begrenzung zu: die Laufzeitkomplexität steigt exponentiell mit der Größe des Problems an [Gupt92]. Trotz erheblicher Fortschritte bei der Behandlung von umfangreichen Problemen durch neue Beweisverfahren, insbesondere durch Verwendung von BDD’s, [BCMD91] reichen diese Techniken nicht aus, um komplexe Systeme, bestehend aus mehreren Abstraktionsebenen, zu verifizieren.

Allen Verfahren des automatischen Ansatzes ist gemeinsam, daß die Verifikationskomplexität exponentiell wächst. Außerdem schließt deren Ausdrucksschwäche die Spezifikation von komplexen Datentypen (wie natürliche Zahlen, Listen, usw.) sowie von generischen, parametrisierbaren Komponenten aus. Ein wesentlicher zusätzlicher Nachteil dieser Ansätze ist die Schwierigkeit der Einbeziehung von Konzepten der Hierarchie und der Modularisierung [Four90], die in der Behandlung von komplexen Systemen unverzichtbar sind [WaTh85]. Verfahren der automatischen Ansätze konnten deshalb bisher nur eingeschränkt für bestimmte Klassen von Problemen wie z.B. die Schaltnetz- oder Steuerwerk-

Verifikation eingesetzt werden. Existierende automatische Ansätze können daher nicht den Kern eines Verifikationssystems bilden, sondern werden effizient innerhalb einer Verifikationsumgebung eingebettet werden müssen, wobei sie mit anderen Ansätzen gekoppelt werden [ClLM89, Chin90, Alle93, JoSe94].

1.2.2.2 Interaktive Ansätze

Die *interaktive* Ansätze basieren auf der Logik erster und höherer Ordnung. Spezifikation und Implementierung werden durch Prädikate in der gegebenen Logik repräsentiert und die Beziehung zwischen ihnen wird wie ein Theorem in Logik betrachtet, das mit Hilfe des deduktiven Systems abgeleitet wird [Gupt92]. Es können zwei Klassen von interaktiven Ansätzen unterschieden werden, nämlich solche, die auf Logik erster Ordnung und solche, die auf Logik höherer Ordnung beruhen. Abgesehen von der Klassifizierung in Prädikatenlogik erster und höherer Ordnung existieren weitere spezielle Kalküle, die anschließend noch erläutert werden.

Die Prädikatenlogik erster Ordnung wurde zur Hardware-Verifikation erstmals von Wagner [Wagn77] eingesetzt. Die Prädikatenlogik erster Ordnung ist zwar vollständig [Göde30, Fitt90], aber nicht mehr entscheidbar [Chur36, Fitt90]. Zur Automatisierung der Beweise wird deshalb meist eine eingeschränkte Sprache der Logik erster Ordnung zugrundegelegt. Beispiele dazu sind die quantorenfreie Boyer-Moore-Logik [BoMo83], die im Boyer-Moore-Theorembeweiser mechanisiert wird [BoMo88], die Gleichungslogik [HoKu89], die im REVE-System [Lesc83] bzw. im RRL-System [KaZh89] verwendet wird, PROLOG, das bei den Beweisern PROVE [SrAg86] und VERIFY [Barr84] benutzt wird, oder State-Delta-Logik[1] [Croc77], die im Verifikationssystem SVSD [Marc91] verwendet wird. Im automatischen Resolutionsbeweiser OTTER [McCU90] wird die gesamte Sprache der Logik erster Ordnung zur Hardware-Verifikation verwendet. Außer diesen Systemen gibt es weitere, auf Prädikatenlogik erster Ordnung basierende Theorembeweiser, die in der Hardware-Verifikation verwendet werden, wie z.B. das Verifikationssystem CLIO [BiMS89], das auf funktionaler Sprache und Logik erster Ordnung basiert, und der LP-Beweiser [GaGu91], der auf einer Teilmenge der mehrsortigen Logik erster Ordnung basiert. Der Nachteil der Anwendung der Prädikatenlogik erster Ordnung ist, daß Verhaltensbeschreibungen einer Hardware sich nur sehr umständlich formalisieren lassen und somit groß und unübersichtlich werden. Obwohl diese

---

[1] State-Delta-Logik ist eine Logik erster Ordnung, ergänzt mit temporallogischen Formeln zur Beschreibung der Zustandsänderungen von Automaten.

Logik eine größere Ausdruckskraft als Aussagenlogik aufweist [Levy91, AVCM93] und für die Verifikation ausreichend mächtig ist, stößt man jedoch auch hier sehr schnell an die Grenzen der Ausdrucksmächtigkeit dieser Logik, weshalb man sich oft der Prädikatenlogik höherer Ordnung zuwendet [Gord86, Joyc91].
Prädikatenlogik höherer Ordnung ist aber weder vollständig, noch entscheidbar [Göde31, Andr86]. Theorembeweiser dieser Logik, die zur Hardware-Verifikation eingesetzt werden, wie VERITAS [HaDa92], HOL [GoMe93], EHDM [HCLR88], LAMBDA [AbHL90], NUPRL [Cons86], PVS [OwSR93] oder ISABELLE [Paul94], sind daher alle interaktiv. Der Nachteil der hier verwendeten Logik ist offensichtlich: Beweise werden komplexer, und wegen des Mangels an Automatisierung entsteht beim Benutzer ein Bedarf an grundlegenden Kenntnissen in der Logik und in den Techniken des Theorembeweisens [Loew91]. Ein erheblicher Vorteil dieser Logik gegenüber Aussagenlogik ist jedoch deren Sprachmächtigkeit [HaDa85, CaGM87]. Sie erlaubt unter Verwendung von komplexen Datentypen, von Abstraktionsmechanismen und von generischen modularen Beschreibungen eine den Hardware-Beschreibungssprachen (sog. HDLs) ähnliche natürliche Spezifikation [Gord86, Joyc91]. Damit ist es möglich, komplexe Systeme mit verschiedenen Abstraktionshierarchien, wie sie in einem realen Entwurfsprozeß verwendet werden, formal zu beschreiben und zu verifizieren.

Bisher gibt es für die Logik höherer Ordnung keine allgemein anwendbaren Verifikationsmethoden, die von Hardware-Entwerfern verwendet werden können. Ein erfolgreicher, aber eingeschränkter Ansatz der Integration der Techniken der Logik höherer Ordnung innerhalb einer Schaltungsentwurfsumgebung wurde mit dem LAMBDA-System [AbHL90] vorgestellt, in dem eine interaktive formale Synthese durchgeführt wird [MaFo91]. Ähnliche Ansätze der formalen Synthese sind in [HaLD89, DeMa89, Camp90] gegeben.

Zu den interaktiven Ansätzen gehören auch Arbeiten, die spezifische Kalküle speziell für die Verifikation von digitalen Schaltungen eingesetzt haben, um dadurch einen größeren Grad der Automatisierung der Beweise zu erreichen [Gord86]. Sie verwenden verschiedene Logiken, die auf Prädikatenlogik oder funktionale Sprachen zurückgeführt werden können. Stellvertretend für die erste Gruppe sei die PPLAMBDA-Logik [Paul85] erwähnt, die vom LCF-Beweiser [GoMW79] verwendet wird. Durch eine Erweiterung dieser Logik mit CCS [Miln80] wurde später das LCF_LSM-System [Gord83a] entwickelt. Zur zweiten Gruppe gehören das CIRCAL-System [Miln83] und die Arbeiten in [BCPP88], die auf Prozeßalgebra [BaWe90] bzw. funktionaler Programmierung beruhen.

Systeme, die spezifische Kalküle zugrundelegen, konnten sich bisher nicht durchsetzen, da sich der Aufwand zur Erstellung solcher speziellen Theorembeweiser als zu groß herausgestellt hat [CaPr88].

1.2.3 Schlußfolgerung

Allen interaktiven Ansätze ist gemeinsam, daß der Entwerfer sich in formaler Logik und in der komplexen Bedienung von Theorembeweisern auskennen muß. Im Gegensatz zu nicht interaktiven Ansätzen ist hier theoretisch keine Einschränkung der Systemgröße vorhanden, so daß ganze Systeme auf allen Abstraktionsebenen (von der Systemebene bis zur Transistorebene) spezifiziert und verifiziert werden können. Aus diesem wesentlichen Grund wird in dieser Arbeit, trotz ihrer Unentscheidbarkeit, Logik höherer Ordnung als Basis der Hardware-Verifikation verwendet.

Die grundlegende Einschränkung der Theorembeweiser ist, daß sie sich für Logik erster und höherer Ordnung nicht völlig automatisieren lassen [BuVe91]. Um eine direkte systematische Anwendung der Verifikation auf komplexe Schaltungen in Entwurfsumgebungen zu ermöglichen, muß aber der Verifikationsprozeß weitgehend *automatisiert* werden [Pygo88, Loew91, Sriv91, ScKK91, Gupt92]. Neuerdings wurde jedoch gezeigt, daß durch eine effiziente Integration von automatischen Beweistechniken der Logik erster Ordnung in eine Umgebung der Logik höherer Ordnung und durch entsprechende Strukturierung und Transformation des Beweisziels in Logik erster Ordnung erste Fortschritte in der Automatisierung des Verifikationsprozesses erzielt werden können [ScKK91, KuSK93]. Aufgrund der großen Lücke zwischen einer Spezifikation auf hoher Abstraktionsebene und einer Hardware-Implementierung auf niedriger Ebene [Joyc89, HuZh93] ist eine Automatisierung für beliebige Schaltungsarten mit dem heutigen Stand der Technik nicht erreichbar. Man ist deshalb bestrebt, eine Reihe von angepaßten Spezifikationsformalismen sowie entwurfsspezifische Heuristiken [NaSt88] und Strategien [KaAL91] zur Planung und Strukturierung des Korrektheitsbeweises einzusetzen. Diese sind aber praktisch nur für eine bestimmte *Klasse* von Schaltungen zu verallgemeinern und so zu automatisieren. Eine Automatisierung des Verifikationsprozesses kann also durch systematische Verwendung von *schaltungsspezifischen* Methoden und Heuristiken, welche die formale Spezifikation und Verifikation weitgehend unterstützen [ALAD92] und im Rahmen eines Frameworks zur Verifikation integriert werden [Chin90, Sriv91, Alle93], erreicht werden. Bei einer passenden Einschränkung des Anwendungsbereiches wird die formale Verifikation durch entwurfsspezifische Methoden und Entscheidungen unterstützt [Evek91a, Sriv91, KuSK93], indem man sich den natürlichen Entwurfsvorgang einer speziellen Schaltungsklasse zunutze macht, um die Verifikation z.B. durch Einfügung von Hierarchien so zu strukturieren, daß sie handhabbar wird. Einige Schaltungsklassen, die in etwa dem Spektrum digitaler Systeme entsprechen, sind die folgenden [Keut91, Gupt92, KuSK93]:

- Arithmetische Rechenwerke (z.B. Gleitkommaprozessoren),
- Endliche Automaten,
- Systolische Architekturen (Array-Prozessoren),
- E/A-Protokollschaltungen,
- Signalprozessoren,
- Mikroprozessoren, usw.

Mikroprozessoren sind vielfältig anwendbare komplexe Schaltungen, die eine besonders gut abgegrenzte Klasse von realen Entwürfen bilden. Seit den Anfängen der Forschung im Bereich der Hardware-Verifikation dienten Mikroprozessoren als Maßstab für die Leistung und Güte von Verifikationssystemen [Gord83, Hunt85, SrBi90]. Dies liegt vor allem daran, daß die vielseitige Struktur und das Verhalten eines typischen Mikroprozessors einen großen Bereich der Hardware-Verifikationsproblematik abdecken. Außerdem werden Mikroprozessoren zunehmend in sicherheitskritischen Anwendungen eingesetzt, die Menschenleben gefährden können [Thom93]. Daher muß eine korrekte Implementierung des Prozessors gefordert werden. Wegen dieses ansteigenden Einsatzes von Mikroprozessoren in solchen Bereichen ist es unabdingbar, daß sich die eingesetzten Prozessoren in *allen Fällen* korrekt verhalten [Cull88]. Um die Zuverlässigkeit dieser Systeme zu erhöhen, stellt die formale Verifikation einen wichtigen Ansatz dar, mit dem eine systematische Analyse aller Fälle gesichert ist [Cohn89b].

Heutige Verifikationsverfahren sind entweder auf datenbasierte Schaltungen oder auf kontrollbasierte Schaltungen effizient anwendbar (siehe Kapitel 1.2.2). Die größte Zahl der Entwurfsfehler ist jedoch meist nicht isoliert im Datenpfad oder im Steuerwerk, sondern durch deren Wechselwirkung verursacht [Keut91]. Im Gegensatz zu anderen Schaltungsklassen besitzen Mikroprozessoren eine Wechselwirkung zwischen komplexen arithmetischen Operationen und Steuerungen, was deren Verifikation nicht einfach macht [Keut91].

In jüngerer Zeit wurde eine Reihe von Arbeiten über die Mikroprozessor-Verifikation unter Verwendung von verschiedenen Verifikationssystemen durchgeführt [Gord83b, Hunt85, BCPP88, Cohn88, Joyc89, SeSr89, SrBi90, Wind90a]. Verglichen mit modernen realistischen Schaltungen, die heutzutage kommerziell angeboten werden, wurden aus technischen wie auch aus methodischen Gründen drastisch vereinfachte Entwürfe (sog. “toy-machines”) verifiziert, die nur einen Bruchteil der modernen Fähigkeiten realer — kommerzieller — Schaltungen aufweisen. Außerdem behandeln diese Arbeiten, mit Ausnahme von Windley [Wind90a], der eine strukturierte Methode zur Verifikation von mikroprogrammierten Prozessoren innerhalb HOL vorstellte, jeweils nur einen speziellen Prozessor und stellen keine allgemeinen Verfahren oder Methoden für die Verifikation von Mikroprozessoren dar.

Mit dem Ziel der Anwendung von formalen Methoden zur Verifikation von realistischen, kommerziellen Prozessoren wendet man sich an den heutigen Mikroprozessorenmarkt. Dieser wird von zwei Prozessorklassen geprägt, die in die Kategorien CISC[2] und RISC[3] eingeordnet werden können. Überwiegend werden zwar noch CISC-Prozessoren entworfen und produziert, aber in den letzten Jahren gewinnen RISC-Prozessoren immer mehr Marktanteile. Gegenüber CISCs besitzen RISC-Prozessoren spezifische Eigenschaften, die im Hinblick auf eine formale Verifikation von großem Vorteil sind. Dennoch ergeben sich durch Berücksichtigung der parallelen Pipeline-Verarbeitung eines RISC-Prozessors auch zusätzliche Schwierigkeiten bei der Verifikation, z.B. durch die Behandlung von Pipeline-Konflikten.

1.3 Ziele der Arbeit

Unter Berücksichtigung des beschriebenen Standes der Technik verfolgt die Arbeit folgende Ziele:

- Entwicklung eines allgemeinen Konzepts zur formalen Verifikation von *RISC*-Prozessoren.
- Verifikation neuer Aspekte eines Mikroprozessorentwurfs, die in bisherigen Arbeiten noch nicht ausreichend behandelt worden sind, wie *Pipelining*.
- Systematisierung des Verifikationsprozesses durch weitgehende *Automatisierung*.
- Das Verifikationsverfahren soll *konstruktiv* sein: Bei einem Fehlschlag des Korrektheitsbeweises sollen nützliche Hinweise auf die Fehlerursache ausgegeben werden.

---

[2] Complex Instruction Set Computer [Patt82].

[3] Reduced Instruction Set Computer [Patt82].

- Verwendung dieses Rahmens zur formalen Verifikation einer großen Anzahl *realer* RISC-Prozessoren, die nicht nur zum Zwecke der Verifikation entworfen worden sind.
- *Implementierung* dieses Rahmens in HOL [GoMe93] und *Integration* in ein umfassendes Verifikationswerkzeug, genannt MEPHISTO [KrKS93].

1.4 Aufbau der Arbeit

Nachdem in den bisherigen Abschnitten die Verifikation von Mikroprozessoren, welche eine spezifische, aber sehr wichtige Klasse von Schaltungen darstellen motiviert wurde, wird im zweiten Kapitel ein Überblick über den Stand der Technik der Verifikation von Mikroprozessoren gegeben. Es werden der Entwurfsprozeß und die gängigen Validierungsmethoden für Mikroprozessoren erläutert und es wird ausführlich auf andere existierende Arbeiten eingegangen. Nach einer Diskussion über CISC- und RISC-Prozessoren wird anschließend die genaue Abgrenzung der Arbeit festgelegt. Im dritten Kapitel werden theoretische Grundlagen für die weiteren Kapitel gelegt. Insbesondere wird auf Prädikatenlogik höherer Ordnung eingegangen, da alle Spezifikations- und Verifikationsansätze auf diese Logik zurückgeführt werden. Danach wird die formale Beschreibung von Hardware-Schaltungen mit Hilfe dieser Logik eingeführt. In Kapitel 4 werden die wesentlichen Eigenschaften eines RISC-Prozessors sowie das Pipelineprinzip beschrieben. Das Pipelining stellt bei RISC-Prozessoren eine charakteristische Besonderheit dar und wird daher sowohl im allgemeinen als auch im Zusammenhang mit RISC-Prozessoren geschildert. In Kapitel 5 wird ein Modell vorgestellt, das aus mehreren Hierarchien besteht, die den Entwurfsebenen eines RISC-Prozessors entsprechen. Beruhend auf einem hierarchischen Verifikationsmodell für mikroprogrammierte Prozessoren werden der Aufbau des RISC-Modells und die Eigenschaften der verschiedenen Hierarchieebenen angegeben. Mit dem in Kapitel 5 beschriebenen Modell kann nun die formale Spezifikation und Verifikation erfolgen. In Kapitel 6 werden zunächst Spezifikationsaspekte wie Abstraktionsmechanismen und die temporale Verhaltensbeschreibung erläutert. Dann folgt die formale Spezifikation der Modellebenen. Die formale Verifikation wird im nachfolgenden Kapitel 7 behandelt. Sie besteht aus zwei unabhängigen Teilen, deren erster sich auf die Korrektheit der Pipelineverarbeitung bezieht, während der zweite die semantische Korrektheit der Befehle behandelt. Eine Implementierung der vorgestellten Methode in dem Beweissystem HOL wird in Kapitel 8 beschrieben, wobei die zusätzliche Einbettung des Verifikationsverfahrens in einer umfassenden Entwurf- und Verifikationsumgebung dargestellt wird. Anschließend folgt eine Validierung des gesamten Verifikationskonzepts anhand der Verifikation eines implementierten RISC-Prozessors. Es wurde dazu der RISC-Prozessor DLX ausgewählt, der einen allgemeinen RISC-Prozessor darstellt und die Eigenschaften bekannter RISCs, wie Intel i860, Motorola M88000, SPARC und MIPS R3000 umfaßt [HePa90]. Die mit diesem Verifikationsbeispiel gewonnenen experimentellen Ergebnisse werden zur Bewertung des Verfahrens herangezogen. Im letzten Kapitel werden die Ergebnisse der Arbeit zusammengefaßt und es wird untersucht, inwieweit die in den Kapiteln 1.3 und 2.5 genannten Ziele erreicht werden konnten. Ein Ausblick geht dann insbesondere auf die Ausbaumöglichkeiten der Verifikationsmethode ein.

Kapitel 2: Formale Verifikation von Mikroprozessoren

Mikroprozessoren werden in großen Stückzahlen produziert und erst nach dem Beginn der Fertigung entdeckte Entwurfsfehler sind deshalb mit sehr hohen Kosten verbunden. Abgesehen von den ökonomischen Aspekten verlangt aber auch der Einsatz von Mikroprozessoren in sicherheitskritischen Anwendungen, daß eine korrekte Funktion unabdingbar ist. Mit der derzeitigen Intergrationsdichte, die über 3 Millionen Transistoren auf einem Chip erreicht [Sain93, Bemm93, Moor93], gehören Mikroprozessor-Chips zu den aufwendigsten integrierten Schaltungen heutiger VLSI-Technologie, wobei die Tendenz zu immer komplexeren Schaltungen geht. In den nächsten beiden Abschnitten wird auf die üblichen Entwurfs- und Validierungsprozesse eingegangen. Dann wird eine ausführliche Beschreibung des Standes der Technik in der formalen Verifikation von Mikroprozessoren gegeben. Nach einer Diskussion über CISC und RISC-Mikroprozessorkonzepte werden schließlich die Zielsetzung und Abgrenzung dieser Arbeit näher dargestellt.

2.1 Der Mikroprozessorentwurfsprozeß

Der Entwurf eines Mikroprozessors ist ein komplizierter und langwieriger Prozeß. Im allgemeinen besteht der Entwurfsprozeß von kommerziellen VLSI-Mikroprozessoren aus folgenden Schritten [WSEB87, Tred88, Taba90]:

- *Definition der Architektur*: Festlegung des Befehlssatzes, des Programmiermodells, der Systemschnittstellen und der Ausnahmebehandlung unter Betrachtung von Nebenbedingungen, wie des verwendeten Technologieprozesses, des angestrebten Preis-Leistungsverhältnisses, der Möglichkeiten der Testbarkeit, usw.

- *Definition der Mikroarchitektur* (*Organisation*): Abbildung des Verhaltens der Prozessorarchitektur auf eine Blockstruktur, Partitionierung von Funktionseinheiten des Prozessors und Festlegung von deren Verbindungen.

- *Funktioneller Entwurf*: Abbildung der Funktionsblöcke in Strukturelemente (Schaltungsblöcke) wie z.B. ALU, PLA, Register oder Multiplexer, und Generierung einer kompletten funktionalen Beschreibung.

- *Schaltungsentwurf*: technologieabhängiger Logikentwurf von Schaltungsblöcken auf Gatterebene sowie Minimierung der Steuer- und Dateneinheiten.

- *Physikalischer Entwurf*: Abbildung der Schaltungsblöcke in ein Layout und Layout-Integration durch Festlegung der Kanäle, Plazieren und Verdrahten.

Der kommerzielle Entwurf von VLSI-Hochleistungsschaltungen wie Mikroprozessoren besteht aus einer Reihe von manuellen und automatischen Syntheseschritten. Die Verwendung von Synthesewerkzeugen ist für den ganzen Entwurfsprozeß nicht möglich, da viele Entscheidungsbedingungen auf verschiedenen Ebenen zu treffen sind, die eine Übersicht über die Operationen des Prozessors erfordern [Tred88]. Bis zur Mikroarchitektur geschieht der Entwurf üblicherweise manuell, auf Gatter- bzw. Layout-Ebene hingegen werden heute fast ausnahmslos automatische Werkzeuge eingesetzt [Spru89]. Manuelle Schritte bleiben jedoch erforderlich, um einen optimalen Entwurf mit hoher Leistung und effizientem Flächenbedarf zu erzielen [Tred88].

Die Validierung (auch oft fälschlicherweise Verifikation genannt) eines Mikroprozessors ist an die oben erwähnten Entwurfsschritte angelehnt. Man unterscheidet lediglich zwischen Architektur-, Funktions- und Implementierungsvalidierung [Tred88, VNMP89, JYKH91, BrSW93, NAUK93, Sain93]. Die Architekturvalidierung besteht aus einem “black-box”-Verhaltenstest aus der Programmierersicht unter Verwendung von deterministischen als auch von Zufallstestprogrammen [ABDG91]. Hierbei werden, unabhängig von der Implementierung, pathologische Fälle der Architektur und der Systemumgebung geprüft, indem Pfade von Befehlssequenzen auf der Mikroarchitekturebene untersucht werden [Tred88]. Die Funktionsvalidierung besteht aus einem “white-box” Strukturtest auf der Mikroarchitektur- bzw. Logikebene, wobei die Funktionalität von verschiedenen Teilkomponenten des Hardware-Entwurfs, wie Automaten oder Rechenwerke, geprüft wird [Sain93]. Schließlich umfaßt die Implementierungsvalidierung verschiedene “Post-silicon” Herstellungstestverfahren zum Testen des Layout-Entwurfs [Ance86, Spru89, TaAS89, JYKH91].

Die Entwurfsvalidierung (insbesondere auf höheren Ebenen) bildet bei heutigen kommerziellen Mikroprozessoren den Flaschenhals des Entwurfsprozesses und kann bis zu 50% des gesamten Aufwands betragen [Tred88, NAUK93]. Im nächsten Abschnitt wird auf die verschiedenen Verfahren der Entwurfsvalidierung näher eingegangen.

2.2 Mikroprozessor-Entwurfsvalidierungsverfahren

Wie jede Hardware kann die Korrektheit eines Mikroprozessors auf verschiedene Weisen untersucht werden. Lange Zeit hat man diskret aufgebaute Prototypen eingesetzt, um Mikroprozessorentwürfe zu validieren [Tred88]. Mit hoher Integrationsdichte und steigender Systemkomplexität von VLSI-Schaltungen schwindet jedoch diese Methode wegen ihrer Mängel infolge der parasitären, thermischen und elektrischen Eigenschaften [CaPr88]. Außerdem machen steigende Entwurfs- und Herstellungskosten sowie drängende Markteinführungsfristen die Verwendung von Prototypen für eine extensive Korrektheitsprüfung fast unmöglich [CaPr88]. Heute wird üblicherweise die Simulation als Hauptwerkzeug verwendet, um Prozessoren auf verschiedenen Detaillierungsebenen zu testen [Spru89, VNMP89, ABDG91, NAUK93, BrSW93]. Im allgemeinen werden Aspekte auf höheren (Architektur- und Mikroarchitektur-) Ebenen unter Verwendung von allgemeinen Programmiersprachen (und zunehmend auch von Hardware-Beschreibungssprachen) beschrieben und simuliert. Auf der Entwurfs- bzw. Logikebene nutzt man Hardware-Beschreibungen oder Netzlistenbeschreibungen aus CAD-Graphikeingaben, um die Struktur zu simulieren. Auf den unteren Entwurfsebenen (Gatter- und Transistorebene) verwendet man spezielle Simulationswerkzeuge wie MOSSIM [Brya81] oder SPICE [Nage75].

Wie bereits in Kapitel 1 angedeutet, tragen Verfahren der Simulation nur zu einer Teil-Korrektheitsprüfung bei, in der gezeigt wird, daß sich der Mikroprozessor bei einem gegebenen Satz von Stimulidaten, die einen größtmöglichen Teil des Schaltungsverhaltens abdecken sollen, korrekt verhält. Eine erschöpfende Simulation ist wegen der exponentiellen Anzahl der Fälle in angemessener Zeit praktisch unmöglich [Pygo88]. Zum Beispiel sei der vergleichsweise kleine 8-Bit Z80-Prozessor [Kham82] genannt, der mit 208 internen Speicherelementen und 13 Eingangssignalen 2^{221} verschiedene Zustände einnehmen kann. Wenn man annimmt, daß in jeder Mikrosekunde ein Zustandsübergang simuliert werden kann, dann würde eine extensive Simulation des ganzen Prozessors 10^{53} Jahre dauern [Pygo88]. Tatsache ist also, daß in einer akzeptablen Zeit nur ein Bruchteil der möglichen Zustände eines Mikroprozessors überhaupt getestet werden kann [Pygo88]. Es kamen daher eine Fülle von Mikroprozessoren auf dem Markt, bei denen sich später gezeigt hat, daß sie sich unter gewissen ungewöhnlichen Umständen (z.B. wenn eine Unterbrechung inmitten eines Block-Transfer-Befehls auftritt) fehlerhaft verhalten [Pygo88]. Aus dieser unbefriedigenden Situation heraus hat man sich in letzter Zeit Methoden der formalen Verifikation als Alternative zur Validierung von Mikroprozessoren zugewandt [Tred88, Bode90], die auf formalen Sprachen zur Spezifikation von Verhalten und Struktur basieren und die formale Beweiser zur Korrektheitsprüfung auf verschiedenen Ebenen des Entwurfsprozesses einsetzen. Bisher hat man jedoch meist nur drastisch vereinfachte, nicht kommerzielle Modelle mit wenigen Befehlen und Fähigkeiten formal spezifiziert und verifiziert. Im nächsten Abschnitt wird der Stand der Technik der Mikroprozessor-Verifikation ausführlicher beschrieben.

2.3 Stand der Technik der Mikroprozessorverifikation

Die Verwendung formaler Methoden zur Schaltungsüberprüfung beim Mikroprozessorentwurf ist keine neue Idee, Boolesche Algebra wurde beispielsweise schon lange für den Beweis der Funktionalität von logischen Gatternetzwerken eingesetzt [SmBH82]. Formale Methoden wurden beispielsweise in der Spezifikation, dem Entwurf und der Teil-Verifikation von kommerziell verfügbaren Mikroprozessorsystemen wie dem VIPER-Mikroprozessor [Cohn89a], dem C/30-Schaltprozessor [Cook90], dem Mikrokode der Gleitkommaeinheit des INMOS T800 Transputers [MaSh87] oder dem Befehlssatz des Motorola 6800 Prozessors [Bowe87] verwendet. Diese Anwendungen halten sich aber in Grenzen und liefern außerdem keine vollständige Verifikation (s. Kapitel 2.3.13 unten). Korrektheitsbeweise für Mikroprozessoren wurden in einigen Beispielen auch mit Papier und Bleistift durchgeführt. In den meisten Fällen jedoch ist der Beweis so komplex, daß ein mechanisches Beweissystem notwendig ist.

Seit den Anfängen der Forschung im Bereich der Hardware-Verifikation hat man immer danach gestrebt, Mikroprozessoren als komplexe Schaltungen mit vielfältigen Anwendungen zu verifizieren. Es wurde daher eine Fülle von prototyp-artigen Entwürfen mit unterschiedlichen Verifikationssystemen spezifiziert und auf Korrektheit formal geprüft. Manche Arbeiten behandeln denselben Mikroprozessor zum Vergleich der Leistung und der Grenzen verschiedener Systeme (s. Tabelle 2-1). Einige Arbeiten beschäftigten sich zwar mit kommerziell verfügbaren einfachen Mikroprozessoren [Cohn89b, Cook90], konnten aber keine vollständigen Ergebnisse liefern, da entweder die Korrektheitsbeweise unbeendet blieben oder nur Teilaspekte eines Mikroprozessors behandelt wurden. Die folgende Tabelle gibt eine erste Übersicht über eine Auswahl vorangegangener Mikroprozessorverifikationen, die dann in den nächsten Abschnitten näher erläutert werden:

Tabelle 2-1: Mikroprozessorverifikationen

Mikroprozessor	**Verifikationssystem**	**Referenz**
Gordon's Computer	LCF_LSM	[Gord83b]
	VERIFY	[Barr84]
	Silica Pithecus	[Weis86]
	HOL	[JoBG86]
(Tamarack-1)	HOL	[Joyc88a]
FM8501	Boyer-Moore SDVS	[Hunt85] [CCLO88]
FM8502	Boyer-Moore	[Hunt89]
VIPER	*Papier-und-Bleistift* LCF_LSM HOL HOL	[Cull86] [Cull88] [Cohn89a] [Aror90]
Tamarack-3	HOL	[Joyc89]
C/30	SDVS	[Cook90]
SECD	HOL	[GrBi90]
AVM-1	HOL	[Wind90a]
MTI	FP, OBJ2 TACHE, Boyer-Moore, OBJ3	[BCPP88] [Alle93]
KUE-CHIP2	*Model Checking*	[HaHY92]
Mini Cayuga	CLIO	[SrBi90]
SPARC	LAMBDA	[Buck92]

2.3.1 Gordon's Computer

Gordon's Computer ist ein primitiver 8-Bit PDP-8-ähnlicher mikroprogrammierter Mikroprozessor mit einem 8 Befehle umfassenden Befehlssatz [Gord83b]. Durch die Verifikation dieses Modells in LCF_LSM [Gord83a] hat Gordon erstmalig gezeigt, wie die Korrektheit von Mikroprozessoren durch die Verwendung mechanischer Beweissysteme formal geprüft werden kann. Die Verifikation besteht aus einer Fallanalyse von Zustandsänderungen durch symbolische Ausführung der Maschinenbefehle durch Mikrobefehlssequenzen. Der Verifikation von Gordon in LCF_LSM folgten Verifikationen in VERIFY [Barr84], in HOL [JoBG86], in OBJ3 [Stav94] und in einem LISP-basierten System, genannt Silica Pithecus [Weis86]. Bei dieser letzten Arbeit wurden außerdem Aspekte des Prozessors auf niedrigen Ebenen (Gatter- und Transistor-Ebenen) untersucht. Später wurde eine in VLSI mit 5000 CMOS Transistoren implementierte Version des ursprünglichen Entwurfs (genannt Tamarack-1) von Gordon in HOL spezifiziert und verifiziert [Joyc88a]. Dieses von Gordon vorgeschlagene Verifikationsmodell wird außerdem als Beispiel in verschiedenen Simulations- und Verifikationsumgebungen benutzt und gilt damit als das am gründlichsten untersuchte [Cami88, Curz90, JLRS91, Herb92].

2.3.2 FM8501

Ein früheres und wesentliches Hardware-Verifikationsexperiment in den USA ist die Arbeit von Hunt [Hunt85] zur Verifikation eines 16-Bit Prozessors mit ähnlicher Komplexität wie die PDP-11 [Digi83], genannt FM8501. Hunt verwendete pures LISP zur formalen Spezifikation und den Boyer-Moore Theorembeweiser [BoMo88] zur Verifikation. Durch die Verifikation des FM8501 demonstrierte Hunt die Anwendung des Boyer-Moore Theorembeweisers zur Hardware-Verifikation. Der FM8501 ist mikroprogrammiert, hat 26 Befehle und 4 Adressierungsmodi, enthält einen Registersatz von 8 Registern und kann 64 KB adressieren.

Zur formalen Verifikation werden zwei Abstraktionsebenen herangezogen: eine Spezifikation (*instruction-level*) und eine Implementierung (*design-level*). Die Spezifikation präsentiert das Verhalten des FM8501 auf der Architekturebene. Die Implementierung besteht aus einer Strukturbeschreibung der Hardware auf Gatterebene. Die Verifikation umfaßt den Beweis der Äquivalenz beider Beschreibungen unter Annahme von entsprechenden Anfangsbedingungen. Ein interessanter Gesichtspunkt dieser Arbeit ist die Betrachtung eines asynchronen, abstrakten Speichermodells als separaten Prozeß des FM8501 unter Verwendung von Handshake-Signalen. Wegen des Fehlens von Existenzquantoren in Boyer-Moore benutzte Hunt eine “*Oracle*-Liste” [Hunt85], um temporale Aspekte zu modellieren.

Die ursprüngliche Arbeit erweiterte Hunt auf den FM8502, einen 32-Bit Mikroprozessor mit einem größeren Befehlssatz [Hunt89]. Diese Arbeit ist ein Teil eines großen Projekts zur Systemverifikation [BHMY89], bestehend aus einem Codegenerator für eine einfache höhere Programmiersprache [Youn89], einem Assemblierer, einem lenkenden Lader [Moor89] und einem einfachen Betriebssystemkern [Bevi89]. Jede Komponente in diesem “*short stack*” [Bevi89] wurde als ein abstrakter endlicher Automat formal spezifiziert und durch den Beweis der Übereinstimmung mit der Maschine auf niedriger Ebene verifiziert.

Unabhängig von Hunts Verifikation mit Boyer-Moore wurde der FM8501-Prozessor auch im SDVS-System [Marc91] formal spezifiziert und verifiziert [CCLO88]. Die Spezifikation des FM8501 ist in der Sprache ISPS [Barb81] und die Verifikation ist unter Verwendung der State-Delta-Logik [Croc77] durchgeführt worden. Analog zu [Hunt85] werden zur Verifikation zwei Abstraktionsebenen, genannt "*macro-level*" und "*micro-level*", zugrundegelegt. Zur Beschreibung von Mikroprozessoreigenschaften, hat sich die Verwendung von ISPS als besser geeignet herausgestellt als pures LISP in Boyer-Moore. Außerdem erlaubt die Semantik von SDVS eine bessere Darstellung von temporalen Aspekten als Boyer-Moore (z.B. die Modellierung des Speichers als einen externen, separaten Prozeß).

2.3.3 VIPER

Von all den Prozessoren, die in den letzten Jahren formal verifiziert worden sind, ist der britische VIPER Prozessor [Cull88] die einzige reale Schaltung, die kommerziell verfügbar ist und bei der die formale Verifikation von vorneherein beabsichtigt war. Obwohl VIPER im Vergleich zu modernen Prozessoren ein einfaches Modell ist, gilt dessen Verifikation als "Markstein" heutiger Mikroprozessorverifikation. Die Verifikation des VIPER ist jedoch aus Komplexitätsgründen nicht abgeschlossen worden.

Der VIPER-Prozessor ist von einer Abteilung der UK "Royal Signal and Radar Establishment" (RSRE) entworfen worden und wird in zivilen und militärischen sicherheitskritischen Anwendungen eingesetzt (z.B. Schienenverkehrsignalisierung). Es handelt sich dabei um einen festverdrahteten 32-Bit Prozessor, der einen Befehlssatz von 128 Befehlen hat, 1 MB adressieren kann und über 4 Register verfügt. Der Prozessor kennt keine Unterbrechungsbehandlung; bei einem Ausnahmefall geht er in einen definierten Fehlerzustand über und signalisiert dies.

Der Prozessor ist hierarchisch in drei Ebenen, die mit "*top-level*", "*major-state-machine*" und "*Electronic Block Model*" (kurz *EBM*) bezeichnet werden, in LCF_LSM bzw. HOL spezifiziert. Unterhalb des *EBM* wird die Struktur in der Hardware-Beschreibungssprache ELLA [CGED89] durch "*intelligente extensive Simulation*" verifiziert [Pygo85]. Zur Verifikation wurde zuerst ein Korrektheitsbeweis von Hand zwischen den Ebenen *major-state-machine* und *top-level* durchgeführt [Cull86], der später in HOL wiederholt wurde [Cohn88]. Ein Beweis zwischen *EBM* und der *major-state-machine* scheiterte. Cohn [Cohn89a] versuchte daraufhin, direkt den Zusammenhang zwischen der untersten und der obersten Ebene zu zeigen. Aufgrund der riesigen Anzahl der Fälle und der

Komplexität der einzelnen Fälle konnte eine vollständige Verifikation jedoch nicht abgeschlossen werden.

Ein besonderer Aspekt, den das VIPER Projekt gezeigt hat, ist die Schwierigkeit der Kommunikation zwischen Entwerfer, Verifizierer und Hersteller in der realen Welt. Ein erstes Problem, das Cohn [Cohn89a] begegnete, ist die Ableitung der *EBM*-Beschreibung aus einer ungenauen, teils schematisch und teils in Text gegebenen Dokumentation. Diese von den VIPER-Entwerfern übergebene informale Spezifikation ist z.T. unvollständig und enthält außerdem implizite Informationen, die nicht genau festgelegt wurden [Cohn89b]. Es wurden Fehler auf der *top-level* und auf niedriger Ebene entdeckt, diese haben sich jedoch nicht in relevanter Weise auf die Chip-Herstellung ausgewirkt. Außerdem zeigte sich in dieser Forschungsarbeit, welche kritische Rolle die Einbeziehung von *Abstraktionshierarchien* in der Verifikation von komplexen Systemen wie Mikroprozessoren spielt. Die *major-state-machine* bezeichnet zwar eine Zwischenabstraktion, diese stellte sich jedoch als die falsche heraus [Cohn89b]. Die Verwendung von HOL bildete auch wegen der fehlenden Automatisierung von Routinearbeiten und dem Fehlen von verallgemeinerten Strategien und Prozeduren ein großes Hindernis. Hier haben sich die Grenzen der Verwendung formaler Methoden in der Verifikation von realen Mikroprozessorsystemen gezeigt [Cohn89b].

Eine mikroprogrammierte Variante des VIPER-Prozessors, die einen Befehlssatz von 20 Befehlen umfaßt, wurde unter Verwendung einer Methodik, die auf einem hierarchischen *Interpreter-Modell* zur Verifikation von mikroprogrammierten Prozessoren [Wind90a] basiert, in HOL formalisiert und verifiziert [Aror90, ALLS94]. Das Interpreter-Modell wird ausführlich in Kapitel 5 beschrieben. Diese Arbeiten zeigen deutlich, warum der ursprüngliche VIPER-Prozessor nicht erfolgreich verifiziert werden konnte, indem die Vorteile der Verwendung von Hierarchien in der Strukturierung der Verifikation solcher komplexer Schaltungen vorgeführt werden.

In [Wong93] wird außerdem die ALU des VIPER-Prozessors als Anwendungsbeispiel für eine neue HOL-Bibliothek für Bit-Vektoren [Wong94] in HOL verifiziert.

2.3.4 Tamarack-3

Der Tamarack-3 Mikroprozessor gilt als Nachfolger von Tamarack-1 [Joyc88a] und wurde ebenfalls für experimentelle Zwecke entworfen. Tamarack-3 ist von Joyce [Joyc89] unter Benutzung von HOL verifiziert worden. Verschiedene Varianten davon wurden außerdem in einer Fülle von Verifikationsumgebungen verwendet [HaWi93, McAl93, ZhJS94].

Es handelt sich bei Tamarack-3 um eine 16-Bit Version von Gordon's Computer; ein sehr kleines mikroprogrammiertes Modell eines Prozessors mit zwei Registern und einem 8 Befehle umfassenden Befehlssatz. Der Prozessor besitzt eine 16-Bit Architektur, kann jedoch nur 8KB adressieren und erlaubt, im Gegensatz zu Tamarak-1, asynchrone Unterbrechungen.

In Anlehnung an die natürlichen Abstraktionsebenen eines Mikroprozessorentwurfs, wurde Tamarack-3 durch drei hierarchische Abstraktionsebenen, sogenannte *Interpreter* [Ance86], spezifiziert, die "*program-level*", "*microprogram-level*" und "*phase-level*" genannt werden und zur Modellierung der Architektur-, Mikrobefehls- bzw. Register-Transfer (RT)-Ebene dienen. Die Verifikation ist hierarchisch gegliedert, wobei jeder Interpreter den nächsthöheren Interpreter implementiert und gleichzeitig die Abstraktion des nächstniedrigeren Interpreters darstellt. Ein weiterer interessanter Aspekt dieser Arbeit von Joyce [Joyc89] ist die Behandlung eines asynchronen Speichermodells durch Verwendung einer Erweiterung von HOL mit Operatoren der temporalen Logik.

2.3.5 C/30

In früheren, aber erst später publizierten Arbeiten wurde mit dem SDVS-Verifikationssystem [Marc91] ein 16-Bit mikrokodierter Schaltprozessor des US-Verteidigungsdatennetzwerkes, genannt C/30, formal spezifiziert und zum Teil verifiziert [Cook90, LFMM92]. Der C/30 ist ein real eingesetzter Prozessor, der mit 128 Architekturbefehlen und 4193 Mikrobefehlen wesentlich komplexer als andere Verifikationsmodelle ist. Das Verhalten des C/30 ist in ISPS [Barb81] spezifiziert. Nach einer Übersetzung dieser ISPS-Beschreibung in eine Spezifikation in State-Delta-Logik ist der Korrektheitsbeweis in SDVS durchgeführt worden, jedoch wurde die Verifikation nicht vollständig ausgeführt.

2.3.6 SECD

Der SECD ist ein spezieller Koprozessor zur Reduktion von Lambda-Ausdrücken in LISP. Basierend auf der SECD Maschine von Landin [Land64] sind in [GrBi90] der Entwurf, die Formalisierung und die Verifikation des SECD-Chips in HOL beschrieben. Der SECD hat eine 32-Bit Architektur mit 21 Befehlen, ist mikroprogrammiert, hat 4 Register und ist in CMOS mit 20.000 Transistoren implementiert worden. Der SECD-Prozessor wird in drei Abstraktionsebenen beschrieben: "*top-level*", "*RT-level*" und "*lowest-level*". Die erste Ebene beschreibt das abstrakte Verhalten als endlicher Automat, die zweite besteht aus einer Beschreibung der Block-Komponenten und die dritte stellt eine Beschreibung auf der Taktphasenebene dar. Diese Arbeit ist ein Teil eines größeren Projektes zur Systemverifikation, das einen Lispkit-Compiler und das SECD-Chip einschließt.

2.3.7 AVM-1

AVM-1 ist ein Prozessormodell, das in [Wind90a] als Anwendungsbeispiel einer generischen Interpretertheorie präsentiert wurde. Dieser Prozessor ist mikroprogrammiert, hat einen Befehlssatz mit 30 Anweisungen, verfügt über einen Registersatz mit 32 Arbeitsregistern, kann 4 GB adressieren, erlaubt Unterbrechungen und kennt einen priviligierten Arbeitsmodus.

Im Unterschied zu Cohn [Cohn89a] versucht Windley den Beweis nicht direkt, sondern führt eine *Hierarchie* von *Interpretern* ein. Ein Interpreter besteht aus einer Menge von Zustandskomponenten und Zustandsänderungsfunktionen. Die Zustände werden durch Tupel repräsentiert, welche die auf der entsprechenden Ebene sichtbaren Variablen enthalten. Es wurden vier Abstraktionsebenen herangezogen: die "*macro-level*"-Ebene, welche eine Verhaltensbeschreibung aus der Sicht eines Assemblerprogrammierers widerspiegelt, die "*micro-level*"-Ebene, welche zu einem Mikroprogramm korrespondiert, die "*phase-level*"-Ebene, welche Zustandsänderungen mit den Phasen des Taktes entspricht, und das "*Electronic Block Model*" (*EBM*), das eine Strukturbeschreibung der Hardware darstellt. Im allgemeinen ergibt sich die gesamte Komplexität der Verifikation hauptsächlich aus der Anzahl der Maschinenbefehle in der *macro-level*-Ebene und aus der Größe der *EBM*-Beschreibung. Durch die Einführung von Teilbeweisen zwischen benachbarten Ebenen reduziert sich die gesamte Komplexität beträchtlich. Dies erlaubt eine stärkere Automatisierung sowie die mehrfache Nutzung von parametrisierten Theoremen. Durch die Vorstellung einer strukturierten Methode zur formalen Verifikation von Mikroprozessoren in HOL stellt Windley [Wind90a] zum ersten Mal eine allgemeine Methodik zur Automatisierung der Verifikation mit Theorembeweisern vor, die für Hardware-Entwerfer brauchbar ist.

2.3.8 MTI

MTI [PuRT87] ist ein mikroprogrammierter 16-Bit Prozessor, der einen 22 Befehle umfassenden Befehlssatz hat, aus 30.000 Transistoren besteht und in CMOS implementiert wurde. Ursprünglich zur Validierung von Testverfahren entwickelt, diente dieser Prozessor als Beispiel für eine Reihe von Arbeiten zur Mikroprozessorverifikation mit Hilfe von funktionalen Ansätzen. In [BCPP88] wird der MTI, unter Verwendung der funktionalen Sprache FP [Shee84], auf der

Maschinenbefehls-, Mikroprogramm- und Mikrobefehlsebene spezifiziert. Die Verifikation geschieht teils manuell, teils rechnergestützt durch einen Tautologieprüfer und einen Theorembeweiser (OBJ2 [FGJM85]).

In [Pail89] wird anhand des Lambda-Kalküls [Chur41] die funktionale Semantik des MTI Prozessors auf der Maschinenbefehlsebene beschrieben.

In [Alle93] ist die Verifikation des MTI unter Benutzung eines komplementären Hybridansatzes mit drei verschiedenen Systemen beschrieben. Sukzessiv werden für Termäquivalenzbeweise ein BDD-basierter Tautologieprüfer (TACHE [BaPa90]), für syntaktische Äquivalenzbeweise über Boolesche und arithmetische Formeln der Beweiser Boyer-Moore [BoMo88] und schließlich ein auf Termersetzungsregeln basierter Theorembeweiser namens OBJ3 [GoWi88] eingesetzt. Dieser Hybridansatz wird außerdem in ein Framework zur inkrementellen Spezifikation und Verifikation von Mikroprozessoren, genannt SPEED [BoCF91] integriert. Bisher haben sich diese Arbeiten entweder auf reine Spezifikationen oder auf eine Verifikationen zwischen den Mikrobefehls- und Architekturebenen beschränkt.

2.3.9 KUE-CHIP2

In Kapitel 1.2 wurden zur Hardware-Verifikation automatische und interaktive Ansätze beschrieben. Mikroprozessoren werden fast ausschließlich mit Theorembeweisern verifiziert. Dies ist eine direkte Folge der Komplexität von Mikroprozessoren und der Größe ihrer Daten- und Kontrollpfade. Neue Fortschritte im Bereich Modellüberprüfung (model checking) unter Verwendung von BDD’s haben jedoch zu einem Versuch zur Verifikation von Mikroprozessoren mit Hilfe dieser Verfahren geführt: KUE-CHIP2 [HaHY92]. Es handelt sich dabei um einen kleinen 8-Bit Prozessor mit einer Akkumulatorarchitektur, die einen 19 Befehle umfassenden Befehlssatz hat und deren Implementierung eine Komplexität von 16.000 Gattern aufweist. Das Verhalten des KUE-CHIP2 ist mit Hilfe verzweigender temporaler Logik (s. Kapitel 1.2.2.1) spezifiziert. Die Verifikation besteht aus der Überprüfung des Verhaltens *jedes* einzelnen Bits des Prozessorzustandes in *jeder* Phase der sequentiellen Befehlsbearbeitung. Die Grenzen der Verfahren der Modellüberprüfung sind hier ersichtlich: ein für diesen kleinen Prozessor überdimensionaler Spezifikationsumfang (über 2 MB groß) ist unrealistisch und eine bitweise Verifikation, die anschließend auf die restlichen 7 Bit übertragen wird, ist nicht überzeugend. In [HaHY92] wird dann auch zugegeben, daß die Verifikation einer Erweiterung des Prozessors auf 16 oder 32-Bit mit diesen Verfahren praktisch nicht möglich ist.

2.3.10 Mini Cayuga

Mini Cayuga ist ein Modell eines kleinen RISC-Prozessors, das als Beispiel zur formalen Spezifikation und Verifikation von Prozessoren in CLIO [BiMS89] entworfen wurde [SrBi90]. In dieser Arbeit wurde erstmals eine formale Verifikation einer RISC-Architektur durchgeführt. Bemerkenswert ist insbesondere, daß dieses RISC-Modell eine Pipeline besitzt. Das Mini Cayuga RISC-Modell hat eine 32-Bit Architektur mit 8 Befehlen, besitzt 32 Arbeitsregister in einem Registersatz, ist festverdrahtet und weist eine 3-stufige Pipeline auf. Anweisungen werden in der "*fetch*"-Stufe geladen und in der "*compute*"-Stufe interpretiert. Der Befehl, der sich in der "*write-back*"-Stufe befindet, wird als aktueller Befehl, d.h. als Befehl, der zu sichtbaren Zustandsänderungen führt, betrachtet.

Spezifikation und Beweis beschränken sich auf zwei Abstraktionsebenen: "*abstract-level*" und "*realization-level*". Die erste ist eine Verhaltensbeschreibung auf der Architekturebene und die zweite eine Strukturbeschreibung auf der RT-Ebene. Es werden jedoch keine Daten- und Zeitabstraktionen zwischen diesen Ebenen berücksichtigt. Die Verifikation besteht aus der Ableitung von Beziehungen zwischen den Darstellungen eines Prozessorzustandes auf beiden Ebenen. Im Unterschied zu Mini Cayuga, bei dem in der *compute*-Phase der Befehl in einem atomaren Schritt ausgeführt wird, sind bei realen RISC-Prozessoren die Aktionen auf mehreren Stufen (*Dekodieren (ID), Ausführen* (*EX*), *Speicherzugriff* (*MEM*), usw.) aufgeteilt. Dies führt zu partiellen Zustandsänderungen, die schwer mit den Änderungen auf der abstrakten Ebene in Einklang zu bringen sind.

Das Herunterbrechen der Abstraktion und die Einbeziehung von nur zwei Spezifikationsebenen führt beim Beweis zu einer umfangreichen Fallunterscheidung und damit zu einer Fallexplosion bei der Verifikation. Schließlich soll angemerkt werden, daß diese Arbeit auch unter Schwächen in der Modellierung der Hardware leidet, da z.B. die Zeit implizit betrachtet, die Implementierung durch abstrakte (black-box) Modelle repräsentiert und der Speicher als Teil des Prozessors aufgefaßt wird.

2.3.11 SPARC-Modell

Buckow [Buck92] präsentierte 1992 eine Fallstudie zur formalen Spezifikation und Verifikation eines realistischen RISC-Prozessors — eines SPARC-kompatiblen Modells — unter Verwendung des LAMBDA-Systems [AbHL90]. Hauptanliegen dieser Arbeit war die Behandlung der 4-stufigen Pipeline. Es ist jedoch nicht gelungen, einen überzeugenden Lösungsansatz vorzuführen.

Außerdem konnte die Verifikation nicht vollendet werden und das, obwohl sich diese Arbeit auf die Integer-Einheit des Prozessors beschränkte.

Der behandelte RISC ist ein DACAPO-Simulationsmodell [Ramm89] einer 32-Bit SPARC Architektur [SunM89], die 80 Befehle enthält, eine 4-stufige Pipeline und einen Registersatz mit Mehrfachfenstern benutzt und Unterbrechungen und einen priviligierten Arbeitsmodus unterstützt. Zur Spezifikation und Verifikation wurden zwei Abstraktionsebenen, “*macro-level*” und “*micro-level*”, herangezogen. Die Ebene *macro-level* ist eine Verhaltensspezifikation, die als endlicher Automat auf Architekturebene gegeben ist. Die Pipeline ist hier verborgen und Befehle werden sequentiell verarbeitet. Die Ebene *micro-level* ist eine Strukturbeschreibung auf RT-Ebene. Die Pipeline ist hier sichtbar und es findet eine parallele Verarbeitung statt. Eine vollständige Verifikation scheitert an zwei grundsätzlichen Problemen: erstens führt die explizite Expansion von Zustandsänderungen auf höheren Ebene zu einer Explosion der Anzahl der Fälle auf niedrigen Ebenen. Zweitens konnte diese Arbeit kein vernünftiges Zeitabstraktionskonzept liefern, das partielle Zustandsänderungen auf der *micro-level*-Ebene mit sequentiellen Zustandsänderungen auf der *macro-level*-Ebene verbindet.

2.3.12 Weitere Arbeiten

Außer den oben erwähnten Arbeiten gibt es noch eine Fülle von einzelnen Verifikationen, die auf verschiedenen Ansätzen formaler Methoden basieren [SeSr89, LoDi90, Herb92, LoDi92, ChCo93, HaTu93].

Betrachtet man das Spektrum aller Mikroprozessorverifikationsarbeiten, so stellt man fest, daß sich die Behandlung von kommerziellen Prozessoren in Grenzen hält. Mit Ausnahme der VIPER- und C/30-Prozessoren werden meist primitive, unrealistische Modelle zugrunde gelegt. Dies liegt unter anderem daran, daß alle Verifikationsarbeiten in einem akademischen Umfeld stattfinden, für das industrielle, firmeneigene Entwürfe und Implementierungen nicht freigegeben werden. Es existieren jedoch einige Arbeiten, die zwar kommerzielle Prozessoren betrachten, die sich aber entweder auf die formale Spezifikation des Befehlssatzes, der aus einem Handbuch abgeleitet wird, beschränken oder die nur einen Bestandteil eines Mikroprozessorsystems (z.B. eine Gleitkommaeinheit) verifizieren. Im folgendem werden einige dieser Beispiele erläutert.

In [Bowe87] wird die Spezifikationssprache Z [Spiv89] verwendet, um den gesamten Befehlssatz des Motorola Prozessors MC6800 [Moto75], einschließlich Unterbrechungen und Speicherkonfiguration, zu spezifizieren. Ähnlich dazu wird in [Gese89], unter Verwendung einer eigenen algebraischen Spezifikationssprache, die formale Spezifikation des Befehlssatzes des Mikroprozessors Intel 8085 beschrieben [Inte80]. In [BoYu91] wird die Semantik der Maschinenbefehle des Motorola MC68020 [Moto85] — ein beträchtlich größerer kommerzieller Prozessor — auf algorithmischer Ebene in Boyer-Moore formalisiert. Obwohl diese letzte Arbeit sich hauptsächlich mit Programmverifikation beschäftigt, zeigt sie praktische Schwierigkeiten bei der Ableitung einer formalen Spezifikation aus der einzig verfügbaren Referenz eines kommerziellen Prozessors: dem Handbuch. So konnten z.B. in dieser Arbeit nur 80% des Befehlssatzes formal spezifizieren werden, weil im Handbuch der Effekt der restlichen Befehle auf einige Zustandskomponenten des Prozessors unspezifiziert ist [BoYu91].

In [MaSh87] verwendete man algebraische Regeln der Programmiersprache OCCAM [RoHo86], um eine Spezifikation auf hoher Ebene der Gleitkommaeinheit des Transputers INMOS T800 [HMSS87] in eine Beschreibung auf der Mikrokodeebene zu transformieren. In [Barr89] wird diese Gleitkommaeinheit des T800 in Z [Spiv89] spezifiziert und, unter Verwendung von Transformationsregeln eines auf OCCAM basierenden Silicon-Compilers [MayD90b], auf Mikroarchitekturebene verifiziert [MayD90a].

Basierend auf dem Motorola Gleitkommaprozessor MC68881 [Moto87b] wird ein zwecks Verifikation entworfener Gleitkommaprozessor in HOL formalisiert und verifiziert [PanJ92]. Die Spezifikation und Verifikation unterliegen einem hierarchischen Interpreter-Modell [Joyc89, Wind90a]. Zur Modellierung von asynchronen Interaktionen auf derselben Abstraktionsebene werden hier, zusätzlich zu den vertikalen Interpreter-Hierarchien, horizontale, parallel ausführende Interpreter-Ebenen betrachtet. Schubert [Schu91b] führt eine ähnliche Arbeit mit dem Gleitkommaprozessor Intel 8087 [PaMo84] unter Verwendung einer Einbettung von CCS [Miln80] in HOL aus. Beide Arbeiten sind Teil eines größeren Projektes zur Systemverifikation [Schu91a], bestehend aus Mikroprozessoren, Koprozessoren, MMUs, DMAs, usw.

2.3.13 Vergleich der bisher verifizierten Prozessoren

Bei den meisten Mikroprozessorverifikationen wurden eigens zum Zweck der Verifikation kreierte und nicht implementierte Modelle behandelt, welche über einige Eigenschaften verfügen, die bei vorangegangenen Arbeiten noch nicht betrachtet wurden (z.B. Unterbrechungen, asynchroner Speicher oder Pipelining). In Tabelle 2-2 werden einige der bisher erwähnten Prozessoren anhand ausgewählter Eigenschaften verglichen. Beachtet wurden dabei insbesondere die Komplexität des Befehlssatzes, die Implementierungsgröße, gegeben durch die Anzahl der Gatter bzw. Transistoren, und die Unterscheidung zwischen mikroprogrammierten und festverdrahteten Prozessoren. Der Mikroprozessor DLX ist in der Tabelle mit aufgeführt, um einen Komplexitätsvergleich mit obigen Prozessoren zu ermöglichen. Er wird in dieser Arbeit als Beispielprozessor verwendet und gilt zugleich als erster in Hardware implementierter RISC-Prozessor, der formal spezifiziert und verifiziert wurde. Ausschlaggebend für seine Auswahl ist hier im wesentlichen die 5-stufige Pipeline, welche die Parallelverarbeitung von Befehlen ermöglicht. Diese Pipeline-Verarbeitung von Befehlen führt zu partiellen Zustandsänderungen, die gesondert behandelt werden müssen.

Tabelle 2-2: Komplexitätsvergleich von verifizierten Mikroprozessoren

Prozessoren / Eigenschaften	FM8501	VIPER	Tamarack-3	Mini Cayuga	AVM-1	MTI	DLX
Wortlänge	16-Bit	32-Bit	16-Bit	32-Bit	32-Bit	16-Bit	32-Bit
Befehle	26	128	8	8	30	22	51
mikroprogrammiert	ja	nein	ja	nein	ja	ja	nein
Mikrobefehle	14	-	32	-	64	38	-
Pipeline	nein	nein	nein	3-stufig	nein	nein	5-stufig
Register	16	4	2	32	32	32	32
Unterbrechungen	nein	nein	ja	ja	ja	ja	ja
Speichermodell	async.	sync.	async.	sync.	sync.	sync.	sync.
Speichergröße	64 KB	1 MB	8 KB	1 GB	1 GB	8MB	4 GB
implementiert	nein	ja	nein	nein	nein	ja	ja
Größe (Gatter od. Tr.)	1.700 G.	5.000 G.	-	-	-	30.000 T.	150.000 T.
Abstraktionsebenen	2	3	3	2	4	3	5

2.3.14 Zusammenfassung

In vielen Arbeiten zur Verifikation von Mikroprozessoren ist die Verifikation entweder unbeendet geblieben oder es wurde ein drastisch vereinfachtes Modell aus dem ursprünglichen Entwurf extrahiert und statt dessen verifiziert. Dies ist nicht nur auf die Größe und Komplexität des Problems zurückzuführen, sondern liegt auch an den verwendeten Methoden und Verfahren. So wird bei manchen Arbeiten eine zu abstrakte Betrachtung der Modelle auf niedriger Ebene zugrundegelegt, eine Betrachtung des Hauptspeichers als Block-Einheit der Hardware-

Struktur, also als Teil des Prozessors, oder als abstrakter Prozeß, der durch virtuelle Funktionen modelliert wird. Die Einführung von wenigen Abstraktionsebenen (s. Tabelle 2-2) führt zu einer großen Anzahl von Fallunterscheidungen bei der Ausführung von mehreren langen Pfaden von Sequenzen (insbesondere auf niedrigen Hardware-Strukturebenen) und damit zu einer exponentiell großen Anzahl von Fällen. Wenn es bei der Verifikation zu einem unvorhergesehen hohen Aufwand kommt und die Anzahl der Lemmata, die zu erzeugen und zu beweisen sind, sehr groß ist, muß dies nicht zwangsläufig an den schlechten Methoden liegen — es kann auch eine Folge des verwendeten Beweissystems sein. Bei der Verwendung von allgemeinen Theorembeweisern wie z.B. HOL oder Boyer-Moore müssen oft viele mühsame elementare Beweise (z.B. über natürliche Zahlen oder Boolesche Algebra) durchgeführt werden. Außerdem bieten allgemeine Beweissysteme kaum Unterstützung zur Modellierung von gebräuchlichen Hardware-Beschreibungsmitteln, wie Bit-Vektoren oder temporalen Operatoren, und zur automatischen Verifikation durch mächtige Beweistaktiken, was eine mühsame menschliche Interaktion notwendig macht. Die Verwendung von funktionalen Sprachen wie LISP oder ISPS leidet entweder unter der Mächtigkeit der Sprache oder dem Fehlen eines Formalismus.

Außerdem werden mit Ausnahme der Arbeiten von Joyce [Joyc89] und Windley [Wind90a] meist nur spezielle Fallbeispiele betrachtet und es wird kein allgemeines Konzept zur Verifikation von mikroprogrammierten Mikroprozessoren vorgestellt.

2.4 CISC, RISC und formale Verifikation

Der heutige Mikroprozessorenmarkt wird von zwei Prozessorklassen, den sogenannten CISC- und RISC-Prozessoren, geprägt [CiVa87, Goor89, Milu89, HaVZ90, Tane90]. CISC ist eine Abkürzung für Computer mit komplexem Befehlssatz (engl. “Complex Instruction Set Computer”) [Patt82]. RISC ist eine Abkürzung für Computer mit reduziertem Befehlssatz (engl. “Reduced Instruction Set Computer”) [Patt82]. Überwiegend werden CISC-Prozessoren entworfen und produziert, aber in den letzten Jahren gewinnen RISC-Prozessoren immer mehr Marktanteile [Lang93]. Dies liegt hauptsächlich daran, daß die letzteren durch ihre relativ einfachere Architektur die Verwendung von neuen Techniken, wie Pipelining oder superskalare Verarbeitung ermöglichen und somit bei geringeren Kosten eine größere Rechenleistung erbringen können.

2.4.1 CISC versus RISC

CISC ist die Architektur traditioneller Prozessoren. Prozessoren dieser Art haben üblicherweise einen umfangreichen Befehlssatz mit variablem Format und mehreren Adressierungsmodi und sind meist mikroprogrammiert. Der Begriff CISC wurde damals geprägt, um die traditionellen Rechnerarchitekturen gegenüber der RISC-Architektur abzugrenzen [Patt82].

Ein CISC-Prozessor ist durch einen großen und reichhaltigen Befehlssatz gekennzeichnet, dessen Befehle komplexe Operationen direkt in Hardware implementieren können. CISC-Befehle sind nicht regulär; verschiedene Befehle haben unterschiedliche Befehlslängen und Formate und benötigen eine unterschiedlich große Anzahl von Ausführungszyklen und Speicherzugriffen. Abhängig von den verschiedenen Adressierungs- und Arbeitsmodi ergeben sich unzählige Ausführungsvarianten eines Befehls, die entworfen, implementiert und anschließend simuliert oder verifiziert werden müssen.

Eine Untersuchung realer CISC-Prozessoren hat gezeigt, daß diese einem sehr komplexen und unstrukturierten Entwurf (sog. “*dirty design*”) unterliegen [Inte84, Moto85, Digi86, Inte87, Moto87a, Inte89a, Moto89, Inte93]. Dies liegt meist daran, daß zweierlei gefordert wird [FuSG89, HEKR90]. Zum einen sollen neue Leistungen und Fähigkeiten wie komplexere Befehle, Pipelining und Multitasking erbracht werden, wobei gleichzeitig immer mehr Funktionseinheiten wie Gleitkommaeinheiten, Caches und Speicherverwaltungseinheiten (engl. “Memory Management Unit”, kurz MMU) auf dem Chip integriert werden sollen. Zum anderen soll die Kompatibilität zu früheren Prozessoren der gleichen Familie erhalten bleiben. Diese letzte Forderung führt zu einem zusätzlichen Aufwand für die Dekodierung und Steuerung und insgesamt zu einer drastischen Erhöhung der Komplexität des Entwurfs. Das Ergebnis ist also eine sehr komplexe Implementierung mit einer Vielzahl von Funktionseinheiten auf dem Chip, wobei diese eine Fülle von außerordentlich großen verwickelten Steuerwerkeinheiten benötigen [Tred88], die bis zu 70% der Chip-Fläche verbrauchen können [Ance86]. Zum Beispiel besteht die Steuerung des Motorola Prozessors MC68040 [Moto89] aus mehr als neun PLAs, zwei ROM-Speichern (der Mikroprogrammspeicher allein ist 1028x130-Bit groß) und zusätzlicher festverdrahteter Logik [HEKR90]. Dank steigender Integrationsdichte stellt nicht der Flächenbedarf der integrierten Einheiten das Problem dar, sondern die Komplexität des Steuerwerks. Das Ziel, daß durch CISC-Prozessoren die sog. *semantische Lücke* [Benn87, Bode90, Tane90] (s. Bild 2-1) geschlossen wird, indem leistungsfähige Befehle in Hardware implementiert werden, führt zu großen Mikroprogrammspeichern in den Steuereinheiten. Gemäß den Angaben großer CISC-Hersteller, befinden sich die meisten Entwurfsfehler im Mikrokode [Tred88]. Wegen dessen Komplexität ist das Steuerwerk die Hauptquelle von Entwurfsfehlern [Bode90]. Das ist auch der Grund dafür, daß konventionelle Validierungsverfahren, wie z.B. die Simulation, den Flaschenhals von CISC-Entwurfsprojekten darstellen [Tred88]. Der unstrukturierte Entwurf vorhandener CISC-Prozessoren erschwert daher die formale Spezifikation und Verifikation [Bode90] und bildet der Hauptgrund, dafür, daß man bis jetzt noch keinen realen, kommerziellen Mikroprozessor, wie etwa Produkte von Intel oder Motorola, formal verifiziert hat.

Die RISC-Philosophie besagt, daß die Hardware-Komplexität auf die Software verlagert wird (s. Bild 2-1) [GiMi87], um so einen *einfachen* Entwurf zu haben, mit dem man einen *hohen Durchsatz* erzielen kann [Henn82, Patt82]. RISC-Prozessoren haben einen regulären einfachen Befehlssatz. RISC-Befehle haben feste Längen und Ausführungszeiten und es stehen nur wenige Adressierungs- und Arbeitsmodi zur Verfügung. Damit wird das Holen, Dekodieren und Ausführen von Befehlen vereinfacht und durch Verwendung von Pipelining eine Eintaktausführung ermöglicht. Komplexe Befehle, wie z.B. Gleitkomma- oder Systemoperationen, werden durch getrennte Einheiten ausgeführt. Während die Komplexität von CISC-Prozessoren im Mikroprogramm liegt, ist sie bei RISC-Prozessoren auf Code optimierende Compiler verschoben [GiMi87]. Daher verfügen RISC-Prozessoren, im Gegensatz zu CISCs, über einfachere, festverdrahtete Steuerungen [Kate84], was zu einer zusätzlichen Verringerung der Fehlermöglichkeiten führt [Bode90].

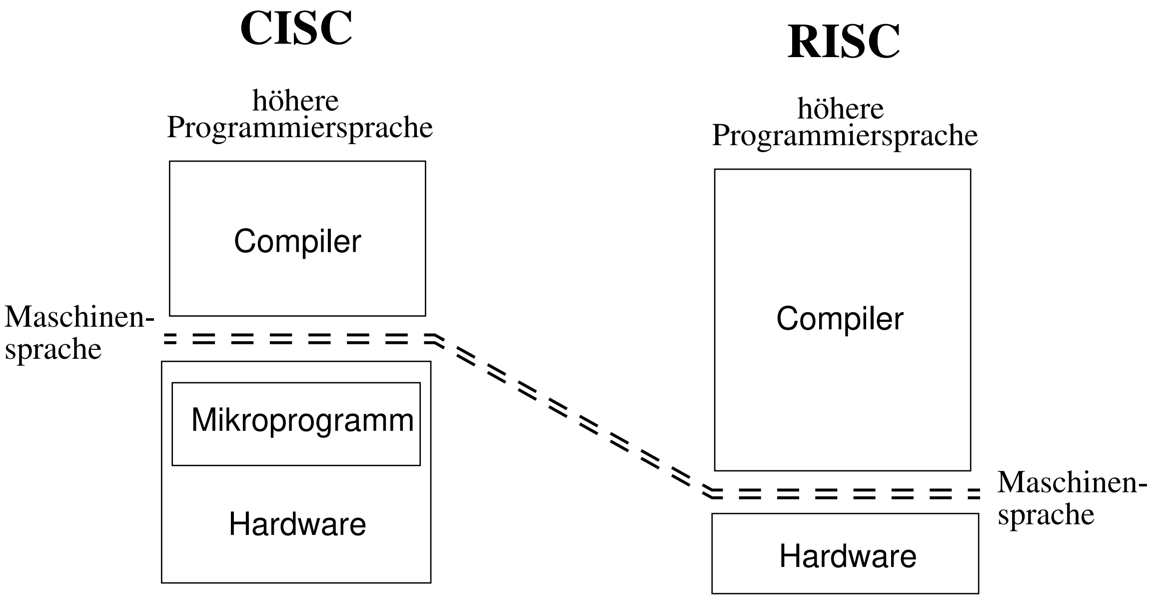


Bild 2-1: CISC- und RISC-Konzepte

Tabelle 2-3 faßt noch einmal die wichtigsten Merkmale, die CISCs von RISC-Prozessoren unterscheiden, zusammen. An Hand dieser Merkmale lassen sich nicht alle Prozessoren eindeutig zu CISC oder RISC zuordnen. Zum Beispiel existieren einige CISC-Prozessoren, die über einen großen Registersatz oder über eine festverdrahtete Steuerung verfügen [GiMi87, Craw90]. Umgekehrt gibt es ein paar kommerzielle RISCs, die für komplexe Befehle, wie etwa Gleitkommaoperationen, eine mikroprogrammierte Steuerung verwenden [GiMi87]. Die Größe des Befehlssatzes allein kann nicht als Definition von RISC und CISC verwendet werden, wichtiger ist die Charakteristik der funktionalen Komplexität der Befehle [Milu89].

Tabelle 2-3: CISC versus RISC

CISC	RISC
• großer, komplexer Befehlssatz	• kleiner, einfacher Befehlssatz (<128)
• variable Befehlslänge	• feste Befehlslänge (4 Byte)
• große Anzahl von Befehlsformaten	• wenige Befehlsformate (≤ 4)
• viele Adressierungsmodi	• wenige Adressierungsmodi (≤ 4)
• kleiner Registersatz	• großer Registersatz (32)
• register- und speicherorientiert	• Load-Store- (Register)-Architektur
• große Unterschiede der Ausführungszeit	• Ein-Zyklus-Ausführung als Zielvorstellung
• weniger Fließbandverarbeitung	• mehr Fließbandverarbeitung
• komplexe Steuerung (mikroprogrammiert)	• einfache Steuerung (festverdrahtet)
• Komplexität im Mikroprogramm	• Komplexität im Compiler
• unstrukturierte VLSI-Implementierung	• einfache modulare VLSI-Implementierung

Entsprechend der Anwendung unterschiedlicher Architekturtechniken ist es angebracht, drei Generationen in der RISC-Entwicklung zu unterscheiden [Bode90]:

- RISC-Architekturen der *ersten Generation* sind die ersten Entwürfe der Universitäten Berkeley (RISC I [PaSe82] und RISC II [Kate84]) und Stanford (MIPS [Henn82]) sowie von IBM (IBM 801 [Radi83]). Die Komplexität der Prozessorchips (20.000 - 100.000 Transistoren) war gering und die Befehlssätze waren sehr elementar (30-50 Befehle).
- RISC-Architekturen der *zweiten Generation* haben bereits deutlich leistungsfähigere Befehlssätze mit ca. 100 Befehlen, die meist Gleitkommaoperationen beinhalten. Darüberhinaus werden zusätzliche Funktionseinheiten wie Speicherverwaltung (MMU) oder Cache auf dem Prozessorchip integriert. Typischer Vertreter dieser heute kommerziell erfolgreichen Klassen mit

300.000 - 1.000.000 Transistoren/Chip sind AMD Am29000 [Adva88], Sun SPARC [SunM89], MIPS R3000 [Kane88], Motorola 88100 [Moto88] und HP PA [Hewl86].

- RISC-Architekturen der *dritten Generation* sind vor allem durch einen höheren chipinternen Parallelismus charakterisiert. Derzeit werden vor allem die Varianten Superskalar und Superpipeline [JoWa89] mit einer Integrationsdichte von bis zu 3 Millionen Transistoren entworfen. *Superskalare* Architekturen wie Intel i860 [Inte89b], Intel i960 CA [Inte89c], IBM RISC System/ 6000 [IBMc90], Motorola 88110 [DiAl92], Sun SuperSPARC [BlKr92] oder PowerPC 601 [Moor93] bieten auf einem Chip mehrere Rechenwerke, die es gestatten, nebenläufig mehrere Maschinenbefehle aus einem Befehlsstrom auszuführen. *Superpipeline* Architekturen, wie MIPS R4000 [MIPS91] oder DEC Alpha [Digi92] versuchen die Taktfrequenz dadurch deutlich zu erhöhen, indem der Maschinenbefehlszyklus in eine größere Anzahl von Pipelinestufen (7 bis 8 Stufen) mit kurzer Verweilzeit unterteilt wird.

Außerdem lassen sich RISC-Architekturen der zweiten und dritten Generation nach ihren Hauptanwendungen in *Allzweck*-Prozessoren (z.B. HP PA, MIPS R3000, Motorola 88100, Intel i860, Sun SPARC) und in *anwendungsspezifische* RISC-Architekturen (z.B. Intel i960 oder AMD Am29000) einteilen [Bode90]. Diese letzteren werden heutzutage insbesondere für systemintegrierte Steuerungen erfolgreich eingesetzt.

2.4.2 Vorteile der RISC-Prozessoren bei der formalen Verifikation

Ein offensichtlicher Unterschied zwischen CISC- und RISC-Prozessoren ist, daß RISC-Prozessoren im Gegensatz zu CISCs eine kohärente Entwurfsphilosophie besitzen [GiMi87, Milu89]. Die Regularität des Befehlssatzes, die reduzierte Anzahl von Befehlsformaten und Adressierungsmodi und die Verwendung einer Koprozessorarchitektur für komplexe Befehle führt zu einfachen und wenig aufwendigen Entwürfen bzw. Implementierungen. Wegen dieser verringerten Hardware-Komplexität stellen RISCs gegenüber CISCs eine vorteilhafte Ausgangsbasis zur Anwendung von Methoden der formalen Verifikation dar [Bode90]. Die Forderung eines hohen Durchsatzes, meist durch paralleles Pipelining, bringt hingegen andere Probleme mit sich, die in früheren CISC-Prozessoren nicht vorhanden waren und die einen Verifikationsprozeß erschweren. Obwohl RISC-Prozessoren eine umfangreiche Pipelineverarbeitung anwenden, sind sie in dieser Hinsicht auch wieder im Vorteil gegenüber modernen CISCs. Die letztgenannten verwenden zwar auch Pipelining, wegen der unterschiedlichen Befehlslängen, der ungleichmäßigen Ausführungszeiten und der komplexen Befehle sind CISC-Pipelines jedoch unbalanciert und auf eine sehr komplexe Weise implementiert [Craw90, HEKR90, CiGo92].

Bisherige Mikroprozessor-Verifikationen hatten fast ausschließlich mikroprogrammierte Prozessoren, also CISCs, zum Gegenstand (s. Kapitel 2.3). Mit Ausnahme der Arbeiten von [SrBi90] und [Buck92] (s. Kapitel 2.3.10 und 2.3.11), gibt es in der Literatur jedoch bisher keine Arbeit, die sich mit der formalen Verifikation von RISC-Prozessoren beschäftigt. Diese beiden Arbeiten weisen jedoch die in den Kapiteln 2.3.10 und 2.3.11 aufgeführten Schwächen und Einschränkungen auf.

2.5 Präzisierung der Zielsetzung und Abgrenzung der Arbeit

RISC-Architekturen der zweiten und dritten Generationen weisen durch Mehrfachrechenwerke und verstärkte Fließbandverarbeitung eine erhöhte Komplexität auf [Bode90]. Sie besitzen komplexe Befehle wie Gleitkomma- oder Systemoperationen und unterliegen einer aufwendigen Implementierung [Inte89b, Moto88, SunM89, Digi92]. RISC-Prozessoren bilden eine noch relativ neue Klasse von Schaltungen und sind dadurch dem Zwang der Familienkompatibilität nicht ausgesetzt [GiMi87]. Sie haben daher besser durchdachte und strukturierte Entwürfe, wobei neue VLSI-Entwurfsmethoden und Techniken großen Einsatz finden [Bode90]. Mit steigender Integrationsdichte wird die Integration von komplexen Funktionseinheiten durch eine starke Modularisierung gefördert [Mele88, Marg90, DiAl92]. Dadurch ist es möglich, eine wohl definierte, mehrschichtige Architektur (engl. “multiple layer architecture”) eines RISC-Prozessors festzulegen [Bake90]. Solche mehrschichtige Architekturen bestehen aus Kernarchitektur (engl. “core architecture”), numerischer Architektur (engl. “numeric architecture”) und geschützter Architektur (engl. “protected architecture”) (s. Bild 2-2). Der RISC-*Kern* führt den Grundbefehlssatz aus, er enthält die Basis-Pipeline des Prozessors und übernimmt sämtliche Steuerungen [PaDo89]. Die *numerische* Architektur implementiert komplexe arithmetische Operationen wie Gleitkomma- oder graphische Funktionen. Die *geschützte* Architektur liefert Systemoperationen, wie Speicherverwaltung, Multiprozessor- oder Multitasking-Funktionen. Betrachtet man das Spektrum realer RISC-Prozessoren, so stellt man fest, daß sich existierende RISCs umso mehr unterscheiden, je weiter man sich vom RISC-Kern entfernt (s. Bild 2-2). Außerdem zeichnen sich heutige anwendungsspezifische RISC-Prozessoren durch den zunehmenden Gebrauch einer Architektur mit einem RISC-Kern aus [LCLL92], in dem wenige primitive Operationen durch einen allgemeinen Prozessorkern mit schneller Ausführung und geringem Flächenbedarf realisiert werden, und indem anwendungsspezifische Operationen wie Signalverarbeitung oder Systemsteuerungen durch spezielle Funktionseinheiten in Form komplexer Befehle ausgeführt werden.

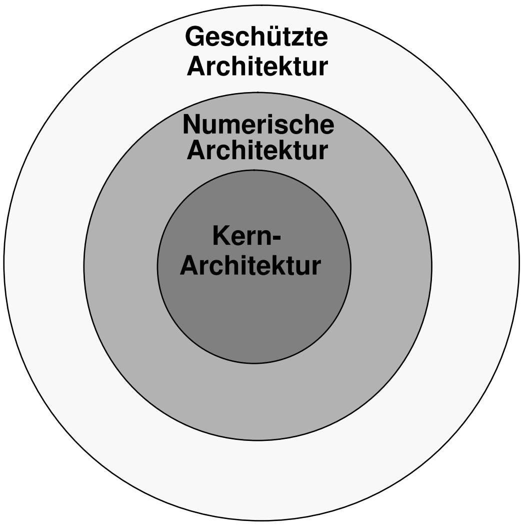


Bild 2-2: Mehrschichtige Architektur eines RISC-Prozessors

Mit dem Ziel der Entwicklung einer allgemeinen Methode zur formalen Verifikation einer großen Anzahl realer RISC-Prozessoren wird in dieser Arbeit zunächst nur die Kern-Architektur eines RISC-Prozessors betrachtet, welche die Basis eines RISC-Prozessors charakterisiert [Bake90].

Die formale Verifizierbarkeit von RISC-Prozessoren rückt durch die vereinfachte Hardware des Prozessorkernes[4] und den damit reduzierten Aufwand immer näher [Bode90]. Allerdings nutzen fast alle realen RISC-Architekturen zusätzliche Funktionseinheiten auf dem Chip, welche die Komplexität erhöhen. Vollständig formal verifizierte Bausteine existieren daher bis heute nicht. Als erster Schritt in diese Richtung werden in dieser Arbeit RISC-Kerne festgelegt und ausführlich behandelt. Die Behandlung der numerischen bzw. geschützten Architektur ist Thema von weiterführenden Arbeiten. Außerdem existieren in der Literatur schon einige Arbeiten, die sich speziell mit der Verifikation von Gleitkommaeinheiten (s. Kapitel 2.3.12), Cache-Modellen [LoDi90] oder Multiprozessor-Protokollen beschäftigen [LoDi92].

---

[4] Zum Beispiel benötigt der RISC-Kern des i860 nur 30% der 1,2 Millionen Transistoren, während der Rest von den Cache-Speichern, der Gleitkommaeinheit und anderen Funktionseinheiten verbraucht wird [PaDo89]. Ähnlich dazu benötigt der RISC-Kern des ARM3-Prozessors etwa 22% der gesamten Chipfläche [FTOH89].

Ziel der vorliegenden Arbeit ist *nicht* die Verifikation eines *speziellen* RISC-Prozessors, sondern die Entwicklung eines *allgemeinen* parametrisierbaren Konzepts zur formalen Verifikation von *realen RISC-Prozessorkernen.* Aus den Vorarbeiten anderer im Bereich der Mikroprozessorverifikation ist ersichtlich, daß die Verifikation eines komplexen Systems, wie es die Verifikation eines RISC-Prozessors darstellt, nur unter Verwendung einer hierarchischen Strukturierung zu handhaben ist. Für mikroprogrammierte Prozessoren konnten erfolgreiche Verifikationen durch Verwendung einer Hierarchie von sogenannten Interpretern [Ance86] strukturiert und verwirklicht werden [Joyc89, Wind90a]. Ähnlich dazu soll hier ein hierarchisches Modell entwickelt werden, welches das Verhalten und die Struktur eines RISC-Kernes widerspiegelt. Darüberhinaus liegt die Schwierigkeit der Verifikation von RISC-Prozessoren in der Behandlung der parallelen Pipeline-Verarbeitung (s. Kapitel 2.3.10 und 2.3.11). Das RISC-Modell soll also die Eigenschaften der Pipeline, durch entsprechende Abstraktionsformen (insbesondere die Zeitabstraktion), berücksichtigen. Basierend auf diesem Modell, werden Spezifikations- und Verifikationsmethoden angegeben, die es erlauben, den Verifikationsprozeß hierarchisch zu strukturieren und dadurch beherrschbar zu machen. Da die meisten Kosten einer Verifikation mit Hilfe von Theorembeweisern durch den Zeitaufwand bei der interaktiven Beweisdurchführung entstehen (s. Kapitel 1.2.2.2), ist man auf eine weitgehende Automatisierung angewiesen. Die Strukturierung des Beweises in eine Hierarchie von *einfachen* und *ähnlichen* Beweisen auf verschiedenen Abstraktionsebenen soll eine *automatische* Erzeugung von regulären Beweiszielen und die Verwendung von gemeinsamen, *parametrisierbaren* Beweistaktiken ermöglichen. Die Ableitung der formalen Beschreibungen, insbesondere auf der Ebene der Hardware-Implementierung, soll ebenfalls automatisiert werden. Darüberhinaus ist beabsichtigt, ein *konstruktives* Verifikationsverfahren zu implementieren, das bei einer Nichterfüllung der Korrektheitssätze die Fehlerursache entdeckt, lokalisiert und explizit anzeigt, um entsprechende Veränderungen am Entwurf vornehmen zu können. Das gesamte Verfahren wird als Basis für einen allgemeinen RISC-*Verifikationsrahmen* in HOL implementiert und in ein umfassendes Verifikationssystem, genannt MEPHISTO [KrKS93], das an eine kommerzielle Entwurfsumgebung angebunden ist, integriert.

Während der ganzen Arbeit wird der DLX-Prozessor [HePa90] als Beispiel verwendet. Es sei darauf hingewiesen, daß in dieser Arbeit anstelle des Begriffes “RISC-Kern” auch manchmal die Bezeichnung “RISC-Prozessor” synonym verwendet wird.

Kapitel 3: Grundlagen

In dieser Arbeit wird Prädikatenlogik höherer Ordnung als formale Grundlage der Spezifikation und Verifikation zugrundegelegt. Prädikatenlogik höherer Ordnung ist ausdrucksstark genug, um alle Aspekte des Hardware-Verhaltens und der Hardware-Struktur beschreiben zu können [Gord86, CaGM87]. Ihre Verwendung zur Hardware-Spezifikation und Verifikation wurde erstmals in [HaDa85] empfohlen. In diesem Kapitel werden zunächst einige Grundlagen der verwendeten Logik höherer Ordnung eingeführt. Anschließend wird auf die Beschreibung der Hardware mit Hilfe dieser Logik eingegangen.

3.1 Prädikatenlogik höherer Ordnung

Die Logik höherer Ordnung erweitert die Prädikatenlogik erster Ordnung um folgende Eigenschaften:

1. Funktionen und Prädikate können Variablen seien über welche qantifiziert werden kann.
2. Funktionen und Prädikate können als Argumente andere Funktionen oder Prädikate erhalten.

Die Syntax der Logik höherer Ordnung wird durch zwei Grundmengen bestimmt: eine Menge von elementaren Termen und eine Menge von Typen, die mit den Termen in der Logik assoziiert sind. Eine Signatur der Logik höher Ordnung wird im allgemeinen als Tripel bezeichnet $\Sigma = (V_\Sigma, C_\Sigma, \alpha_\Sigma)$, wobei V_Σ, C_Σ die Mengen der Variablen- bzw. Konstantensymbole sind und α_Σ eine Funktion $\alpha_\Sigma : V_\Sigma \cup C_\Sigma \mapsto T_\Sigma$, die jedem Variablen- bzw. Konstantensymbol einen *Typ* $\sigma \in T_\Sigma$ zuordnet.

3.1.1 Typen

Die hier verwendete Logik ist *typisiert*. Terme einer typisierten Logik müssen typkonsistent sein, d.h. sie werden so definiert, daß jeder Term einen Typ in einer konsistenten Weise zugeordnet bekommt. Es werden folgende Grundtypen eingeführt [GoMe93]:

- Typ-Variablen,
- Typ-Konstanten und
- Funktionstypen

Eine *Typvariable* steht für einen beliebigen Typ. Ein Typ, der Typ-Variablen enthält, wird auch als *polymorph* bezeichnet. Typ-Variablen und Polymorphismus spielen eine wichtige Rolle, da sie abstrakte Spezifikationen ohne irrelevante Details ermöglichen.

Eine *Typkonstante* bezeichnet eine endliche Menge von Werten, z.B. *bool*, *num* für Boolesche Werte bzw. natürliche Zahlen.

Funktionstypen haben die Form “$\sigma_1 \rightarrow \sigma_2$” und bezeichnen den Typ der totalen Funktionen mit Argumenten vom Typ σ_1 und Funktionswerten vom Typ σ_2. Ein Funktionstyp repräsentiert damit eine Menge von Funktionen, die durch die Mengen ihrer Definitions- und Wertebereiche vorgegeben ist.

Zur Kennzeichnung von Typen wird für eine Variable x vom Typ σ folgende Schreibweise vereinbart: “x:σ”. Um unnötige Schreibarbeit zu sparen, wird der Typ einer Variable bzw. eines Terms nur bei Bedarf explizit angegeben.

3.1.2 Terme

In der vorliegenden Arbeit werden folgende Arten von Termen der Logik höherer Ordnung zugrundegelegt [GoMe93]:

- Variablen,
- Konstanten,
- Funktionsanwendungen und
- λ-Terme

Variablen und Konstanten bezeichnen Sequenzen von Buchstaben oder Ziffern und haben entsprechende Namen und Typen. Der Unterschied zwischen Variablen und Konstanten ist der, daß Konstanten einen festen Wert bekommen. Beispiele von Konstanten sind die Boolesche Werte T und F vom Typ *bool*.

Funktionsanwendungen haben die Form “$f\ t$ ” und bezeichnen die Anwendung der Funktion f auf den Term t. Der Term “$f\ t$ ” (auch “$f(t)$ ” geschrieben) hat den Typ $\sigma' \rightarrow \sigma''$, wobei σ', der Typ des Terms t, dem Definitionsbereich entspricht und σ'' der Typ des Wertebereiches ist. Wenn f ein Prädikat darstellt, dann ist $f\ t$ eine logische Aussage, die den Typ *bool* hat und entweder *wahr* oder *falsch* ist.

Ein λ-Term hat die Form "$\lambda x.\ t$”, wobei x eine Variable ist und t einen Term bezeichnet. Dieser λ-Ausdruck charakterisiert die Funktion $a \mapsto t[a/x]$, wobei $t[a/x]$ das Resultat der Substitution von x durch a in t kennzeichnet. Beispiel eines λ-Ausdrucks sei die Funktion λx. x+3, welche die Zahl 3 zu ihrem Argument addiert.

Außer diesen Grundtermen ist es möglich, weitere spezielle Terme zu definieren, um Spezifikationen komfortabler zu machen. In dieser Arbeit werden, zusätzlich zu den beschriebenen Termen, zwei spezielle Arten von Termen, genannt *bedingte* und *Let*-Terme, definiert:

Bedingte Terme sind Ausdrücke der Form “$t \rightarrow t_1 \mid t_2$”, wobei t ein Boolescher Term ist und t_1 und t_2 zwei beliebige Terme vom gleichen Typ sind. Dieser bedingte Ausdruck bedeutet “*wenn t wahr ist, dann* t_1 *sonst* t_2”

Let-Terme sind lokale Deklarationen, sie haben die Form “$let\ x = t_1 in\ t_2$” und bezeichnen das Ergebnis der Ersetzung jedes freien Auftretens der Variablen x im Term t_2 durch den Term t_1. (Ein Let-Term ist äquivalent zu $(\lambda x\ .t_2)t_1$)

Komplexe Aussagen werden mit Junktoren und Quantoren aus elementaren Grundtermen zusammengesetzt. Für Junktoren und Quantoren werden die üblichen Symbole $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$ und $\exists$ für Negation, Konjunktion, Disjunktion, Implikation, Äquivalenz, Allquantor und Existenzquantor verwendet (s. Tabelle 3-1). Darüberhinaus vereinbart man folgende Bindungsprioritäten: $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

Quantifizierungen über mehrere Variablen können zusammengezogen werden, zum Beispiel sei $\mathcal{F}$ eine Formel und $x_1 \ldots x_n$ Variablen, so steht $\forall x_1 \ldots x_n\ \mathcal{F}$ für $\forall x_1 \ldots \forall x_n\ \mathcal{F}$.

In Tabelle 3-1 sind die wichtigsten Konstanten und Terme der Logik höherer Ordnung zusammengestellt, die hier verwendet werden.

Tabelle 3-1: Terme und Operatoren der Logik höherer Ordnung

Bezeichnung	Beispiel	Bedeutung
T	-	wahr (True)
F	-	falsch (False)
$\neg$	$\neg t$	NICHT t (Negation)
$\wedge$	$t_1 \wedge t_2$	t_1 UND t_2 (Konjunktion)
$\vee$	$t_1 \vee t_2$	t_1 ODER t_2 (Disjunktion)
$\Rightarrow$	$t_1 \Rightarrow t_2$	wenn t_1 dann t_2 (Implikation)
$\Leftrightarrow$	$t_1 \Leftrightarrow t_2$	t_1 genau dann wenn t_2 (Äquivalenz)
$\forall$	$\forall \mathrm{x}.\ t$	für alle x, t (Allquantor)
$\exists$	$\exists \mathrm{x}.\ t$	es gibt ein x, so daß t (Existenzquantor)
λ	$\lambda \mathrm{x}.\ t$	Abstraktionsfunktion $a \mapsto t[a/x|$
let	let $x = t_1$ in t_2	ersetze x in t_2 durch t_1
$\rightarrow$	$(t \rightarrow t_1 \mid t_2)$	wenn t wahr ist dann t_1 sonst t_2

3.1.3 Kalküle, Axiome, Theoreme und Definitionen

Ein *Kalkül* erweitert eine Logik um den syntaktischen Begriff des Ableitens, der den semantischen Folgerungsbegriff nachbilden soll. Ein Kalkül stellt eine Menge von logischen *Axiomen* und eine Menge von *Schlußregeln* (auch Inferenzregeln genannt) zur Verfügung, mit denen aus Formeln weitere Formeln abgeleitet werden können. Diese Regeln sind als rein syntaktische Symbolmanipulation definiert, die keine Kenntnis der Bedeutung der Formeln erfordern. Eine Schlußregel hat im allgemeinen die Form

$$\frac{\mathcal{F}_1 \ldots \mathcal{F}_n}{\mathcal{F}}$$

und erlaubt, aus den Formeln $\mathcal{F}_1, \ldots, \mathcal{F}_\mathrm{n}$ die neue Formel $\mathcal{F}$ abzuleiten. Ist $\mathcal{F}$ durch beliebig viele aufeinanderfolgende Anwendungen von Schlußregeln aus $\mathcal{F}_1, \ldots, \mathcal{F}_\mathrm{n}$ ableitbar, schreibt man

$$\mathcal{F}_1, \ldots, \mathcal{F}_\mathrm{n} \vdash \mathcal{F}$$

und nennt $\mathcal{F}_1, \ldots, \mathcal{F}_\mathrm{n} \vdash \mathcal{F}$ ein *Theorem.* Die Terme $\mathcal{F}_1, \ldots, \mathcal{F}_\mathrm{n}$ sind die *Annahmen* des Theorems, $\mathcal{F}$ ist dessen *Folgerung* (Conclusio). Theoreme, welche man ausschließlich aus den Grundaxiomen des Kalküls erhält werden durch folgendes Zeichen gekennzeichnet: $\vdash$.

Eine *Definition* ist eine spezielle Art von Axiomen, die eine gewisse Konsistenz gewährleisten und die Form “$f\ x_1 \ldots x_n = t$” haben, wobei f eine Konstante ist und alle freien Variablen des Terms t in $x_1 \ldots x_n$ enthalten sind. Definitionen sind konsistent, wenn sie keine neue Informationen hinzufügen, sondern nur Abkürzungen darstellen. Eingeführte Definitionen werden durch folgendes Zeichen gekennzeichnet: $\vdash_{def}$.

Zur Beurteilung eines Kalküls spielen die Begriffe Entscheidbarkeit und Vollständigkeit eine zentrale Rolle. Die Art der Logik legt fest, ob Kalküle mit bestimmten Eigenschaften möglich sind. Die Logik erster Ordnung ist nicht entscheidbar [Chur36, Fitt90] aber noch vollständig [Göde30, Fitt90]. Kalküle der Logik höherer Ordnung sind weder vollständig noch entscheidbar [Göde31, Andr86].

Wie im Laufe der vorliegenden Arbeit ersichtlich wird, wird aus der Prädikatenlogik höherer Ordnung die Möglichkeit der Quantifizierung über Funktionen ausgenutzt, welche abstraktere Spezifikationen als etwa in Prädikatenlogik erster Ordnung erlaubt. Jedoch da die hier verwendeten Funktionen zu einem begrenzten Raum von möglichen Instanzen gehören, wird die Verifikation tatsächlich auf die Prädikatenlogik erster Ordnung zurückgeführt.

3.2 Hardware-Beschreibung in der Logik höherer Ordnung

Ziel der Verifikation ist es, zu zeigen, daß die konkrete Implementierung einer Schaltung die abstrakte Spezifikation ihres Verhaltens erfüllt. Hierzu müssen formale Beschreibungen der Spezifikation und der Implementierung dem Theorembeweiser bereitgestellt werden, um den formalen Korrektheitsbeweis durchzuführen.

3.2.1 Hardware-Spezifikationsmittel

Üblicherweise werden verschiedene Spezifikationssprachen für verschiedene Abstraktionsebenen einer Hardware verwendet, um Simulation, Verifikation und Dokumentation zu unterstützen. Im Zusammenhang mit formaler Verifikation wurden entweder spezielle Hardware-Beschreibungssprachen wie z.B. ISPS [Barb81], LTS [BaFM85], ELLA [CGED89] und VHDL [IEEE93] oder allgemeine Programmiersprachen und Varianten davon wie z.B. LISP, Ada [USDD83] und Standard ML [Paul91] oder Beschreibungssprachen mit unterliegenden Formalismen, wie z.B. HOPE [BuMS80], PROLOG, FP [Shee84], νFP [PaSE85] und OBJ [GoWi88] oder direkt reine Logik, wie Aussagenlogik, temporale Logik verwendet.

Die Verwendung von Programmier- und Hardware-Beschreibungssprachen hat den Vorteil, daß Hardware-Entwerfer mit diesen imperativen Sprachen vertraut sind. Die meisten Hardware-Beschreibungssprachen (HDLs) sind jedoch zum Zweck der Hardware-Simulation und nicht für formale Verifikation entwickelt worden. HDLs können als formale Sprachen und nicht als formale Systeme angesehen werden. Daher kann man formal Beweise nicht direkt in HDLs führen [Evek87, CaPr88]. Benutzt man dennoch eine gebräuchliche Anwendungs- oder Hardware-Beschreibungssprache, so stellt sich die Notwendigkeit eines Übersetzers, der, durch Definition einer entsprechenden Semantik, die formulierte Beschreibung in die Ausdruckssyntax der Objektsprache des formalen Beweissystems (z.B. HOL, Boyer-Moore, SDVS, usw.) transformiert. Trotz der Fortschritte in diesem Bereich, die eine automatische Übersetzung ermöglichen [BGHT91, Fili91, BrHY92, BGGH92, Tass93], liefern solche Transformationen in der Praxis jedoch nur schwach strukturierte Beschreibungen, die für einen weiterführenden Beweis schlecht brauchbar sind [Levy91]. Es ist also aus Effizienzgründen besser, eine einheitliche formale Ausdruckssprache zu verwenden, die sowohl für die Spezifikation als auch für den formalen Beweis geeigneter ist.

In dieser Arbeit wird direkt in Logik höherer Ordnung spezifiziert. Wie in Kapitel 1.2.2.2 beschrieben, erlaubt diese Logik eine natürliche Beschreibung aller Aspekte der Hardware-Struktur und des Hardware-Verhaltens, die in weniger mächtigen Formalismen sehr umständlich sind. Die Idee der Verwendung der Logik höherer Ordnung zur Hardware-Beschreibung kam ursprünglich von Hanna und Daeche [HaDa85]. Ein Aufschwung dieser Logik in der Hardware-Verifikation kam jedoch erst mit den Arbeiten von Gordon zustande [Gord86]. Er hat gezeigt, wie diese Logik, die sowohl ein Hardware Beschreibungsmittel als auch ein deduktives System darstellt, vollkommen angemessen und effizient zur formalen Spezifikation und Verifikation von Hardware eingesetzt werden kann. Beschreibungen in Logik höherer Ordnung haben aber, im Gegensatz zu Hardware-Beschreibungssprachen, den Nachteil, daß sie nicht ausführbar sind. In [Cami88] ist jedoch ein Weg zur Ausführung wenigstens einer Teilmenge der Logik höherer Ordnung gezeigt worden.

3.2.2 Strukturbeschreibung in Logik höherer Ordnung

Die Struktur einer Hardware-Schaltung kann durch zwei grundsätzliche Formen beschrieben werden [Cloc87]:

- durch Funktionen, welche die Ausgangssignale anhand der Eingangssignale beschreiben oder
- durch Relationen (Prädikate), welche die Beziehungen zwischen den Schnittstellensignalen einer Komponente angeben. Relationen stellen spezielle Funktionen dar, welche den Booleschen Wert “wahr” oder “falsch” liefern.

3.2.2.1 Strukturbeschreibung durch Funktionen

Bei der Verwendung der funktionalen Methode wird die Verbindung zwischen den Komponenten durch die syntaktische Struktur der Funktionsanwendung gegeben. Um ein sequentielles Verhalten zu spezifizieren, wird Rekursion angewandt. Eine funktionale Beschreibung leidet jedoch unter folgenden Problemen [Cloc87]:

- Die Beschreibung von bidirektionalen Schnittstellen ist schwierig, da eine funktionale Beschreibung syntaktisch zwischen Ein- und Ausgangssignalen unterscheiden muß.
- Sequentielle Schaltungen können Rückführungen auf sich selbst enthalten. Die Verwendung von Rekursionen ist jedoch bei Mehrfachrückführungen nicht ausreichend.

Die Prädikatenmethode ist die am meisten verwendete Spezifikationstechnik der Logik höherer Ordnung [HaDa85, Gord86]. Sie stellt zwar eine den Hardware-Entwerfern ungewohnte Beschreibungsform dar, hat aber den Vorteil, daß sie die Spezifikation von einer Vielfalt von Schaltungsarten erlaubt, einschließlich solcher mit Mehrfachausgängen und bidirektionalen Schnittstellen.

In dieser Arbeit werden zur Spezifikation hauptsächlich relationale Beschreibungen verwendet, indem Strukturkomponenten durch Prädikate spezifiziert werden. Im nächsten Abschnitt wird die Strukturbeschreibung von Hardware in der Logik höherer Ordnung durch Prädikate näher beschrieben.

3.2.2.2 Strukturbeschreibung durch Prädikate

Eine Schaltung ist eine Sammlung von zusammengesetzten Komponenten, die wohldefinierte Schnittstellen mit Ein- und Ausgangsleitungen besitzen. Das Verhalten einer Komponente $\mathcal{K}$ mit den Eingangsleitungen e_1, …, e_n und den Ausgangsleitungen a_1, …, a_m (s. Bild 3-1) wird durch die Definition eines Prädikates K mit (n+m Argumenten) spezifiziert, so daß K (e_1, …, e_n, a_1, …, a_m) nur dann wahr ist, wenn die Variablen e_1, …, e_n, a_1, …, a_m erlaubte Signalwerte für die entsprechenden Leitungen der Komponente $\mathcal{K}$ sind. Das Prädikat K gibt das Verhalten der Komponente $\mathcal{K}$ durch eine Formel in Logik wieder.

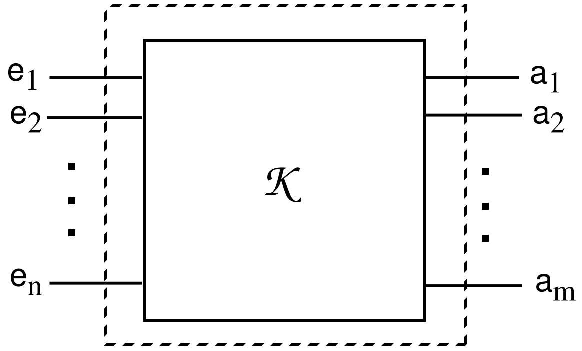


Bild 3-1: Hardware-Komponente

Das “black-box”-Verhalten einer Komponente wird anhand der Signale an ihren Schnittstellen ausgedrückt. In einer Schaltung kann jede Komponente wieder als hierarchische Zusammensetzung von weiteren Komponenten angesehen werden, usw. bis zu den primitivsten Grundelementen.

Gegeben sei die folgende Beispielstruktur einer Schaltung $\mathcal{BS}$, die aus drei Komponenten $\mathcal{K}_1$, $\mathcal{K}_2$ und $\mathcal{K}_3$ zusammengesetzt ist (s. Bild 3-2). Die externen Leitungen von $\mathcal{BS}$ sind a, b, c, d und e. Die internen Leitungen p und q sind nur innerhalb der Schaltung sichtbar.

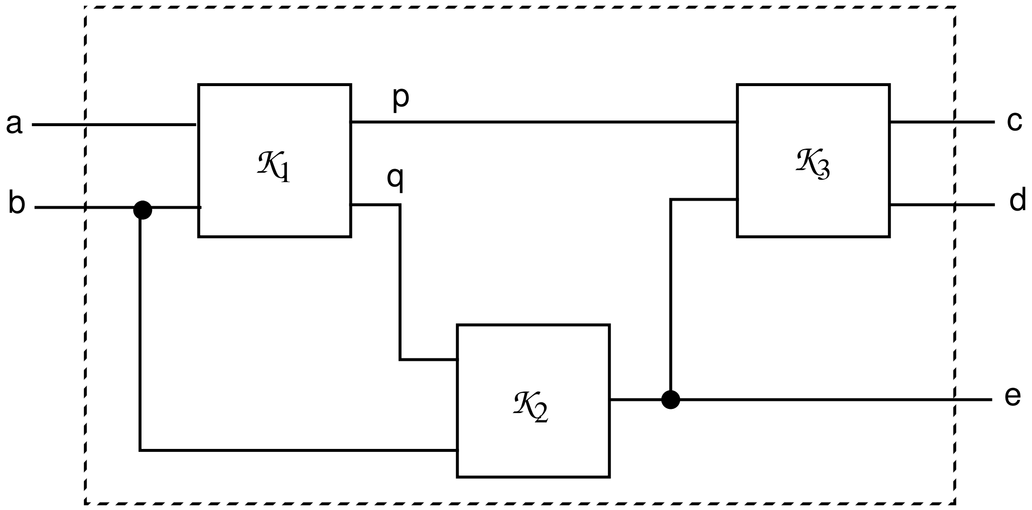


Bild 3-2: Schaltung einer Beispielsstruktur $\mathcal{BS}$

Angenommen das Verhalten von $\mathcal{K}_1$, $\mathcal{K}_2$ und $\mathcal{K}_3$ wird durch die Prädikate K_1, K_2 bzw. K_3 spezifiziert, dann wird das Verhalten der gesamten Schaltungsstruktur einfach durch die Konjunktion der Prädikate der zugehörigen Komponenten gebildet [HaDa85, Gord86]:

$$K_1\ (a,\ b,\ p,\ q) \wedge K_2\ (q,\ b,\ e) \wedge K_3\ (e,\ p,\ c,\ d)$$

Dieser Ausdruck bindet sowohl die Signalwerte der Ein- und Ausgangsleitungen a, b, c, d und e als auch die der internen Leitungen p und q. Für eine “black-box”-Verhaltensbeschreibung der Schaltung $\mathcal{BS}$ sind jedoch nur die Signalwerte der externen Leitungen interessant. Die internen Leitungen sollen unsichtbar bleiben. Um diesen Sachverhalt zu spezifizieren, benutzt man üblicherweise existenzquantifizierte Variablen, die interne Leitungen verstecken [Hoar81, Gord86]. Das Prädikat, das die gesamte Schaltung $\mathcal{BS}$ beschreibt, kann wie folgt angegeben werden:

$$\begin{aligned}&BS\ (a,\ b,\ c,\ d,\ e) :=\\ &\qquad \exists\ p\ q.\ K_1\ (a,\ b,\ p,\ q) \wedge K_2\ (q,\ b,\ e) \wedge K_3\ (e,\ p,\ c,\ d\end{aligned}$$

3.2.3 Verhaltensbeschreibung in Logik höherer Ordnung

Unterschiedliche Aspekte eines Hardware-Verhaltens werden durch eine Reihe von nichtformalen Darstellungsformen beschrieben (z.B. endliche Automaten, Pseudo-Code, Zeitdiagramme, usw.). Verschiedene Formen der nichtformalen Beschreibungen werden durch die geeignete Wahl von Prädikaten, Funktionen und abstrakten Typen in einheitliche formale Spezifikationen konvertiert, welche die nichtformale Beschreibung widerspiegeln sollen.

In den nächsten beiden Abschnitten wird anhand von einfachen Beispielen die Verhaltensbeschreibung von kombinatorischen und sequentiellen Schaltungen in der Logik höherer Ordnung beschrieben.

3.2.3.1 Kombinatorische Schaltungen

Als Beispiel einer kombinatorischen Schaltung betrachte man das Verhalten und die Struktur der folgenden Schaltung (Bild 3-3).

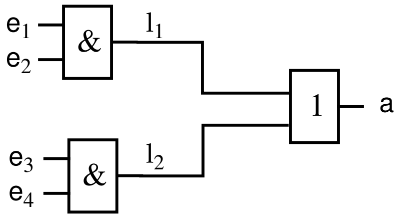


Bild 3-3: Implementierung der Schaltung S

Das Verhalten (Spezifikation) dieser Schaltung S kann durch die folgende Definition (Prädikat) spezifiziert werden:

$\vdash_{def}$ S_SPEC $(e_1, e_2, e_3, e_4, a) := (a = (e_1 \wedge e_2) \vee (e_3 \wedge e_4))$

Das Verhalten jeder Komponente dieser Schaltung wird wieder durch ein Prädikat spezifiziert, z.B. wird das obere UND-Gatter mit den Eingängen e1 und e2 und dem Ausgang l_1 durch folgendes Prädikat beschrieben:

$\vdash_{def}$ UND $(e_1, e_2, l_1) := (l_1 = e_1 \wedge e_2)$

Durch Konjunktionen der Prädikate der einzelnen Komponenten und Einführung von Existenzquantoren für die internen Leitungen wird die Struktur (Implementierung) der Schaltung S durch folgendes Prädikat repräsentiert:

$\vdash_{def}$ S_IMP $(e_1, e_2, e_3, e_4, a) :=$
$\exists\, l_1\, l_2.$ UND $(e_1, e_2, l_1) \wedge$ UND $(e_3, e_4, l_2) \wedge$ ODER (l_1, l_2, a)

Zum Vergleich: Eine Spezifikation der Schaltung S wird unter Verwendung von Funktionen wie folgt definiert:

$\vdash_{def}$ S_IMP $(e_1, e_2, e_3, e_4) :=$ ODER (UND (e_1, e_2), UND (e_3, e_4))

Im Unterschied zur Prädikatenform sind die Ausgänge hier nicht explizit angegeben und das Resultat der Funktion wird als Ausgang der Schaltung interpretiert.

Zur weiteren Veranschaulichung wird eine deklarative Spezifikation dieser Schaltung in einer imperativen Sprache (hier VHDL [IEEE93]) wie folgt angegeben:

```
Entity S_IMP is
  port(e1,e2,e3,e4:in Bit; a:out Bit)
end S_IMP
architecture Structure of D_IMP is
  component ANDGate port(e1,e2:in Bit; a:out Bit);
  component ORGate port(e1,e2:in Bit; a:out Bit);
  signal l1, l2:Bit
  G1: ANDGate port map (e1,e2,l1);
  G2: ANDGate port map (e3,e4,l2);
  G3: ORGate port map (l1,l2,a);
end Structure;
```

Auffallend ist an dieser imperativen Spezifikation die Häufigkeit der Schlüsselwörter im Vergleich zur Prädikatenform. Das größte Problem in der Verwendung von imperativen Sprachen wie VHDL ist jedoch, wie in Kapitel 3.2.1 schon angedeutet, das Fehlen einer eindeutigen formalen Semantik.

3.2.3.2 Sequentielle Schaltungen

Bei der Spezifikation der einfachen kombinatorischen Schaltung wurden im letzten Abschnitt Signale durch zeitunbehaftete Variablen beschrieben. Das Verhalten von sequentiellen Schaltungen in der Logik höherer Ordnung wird dagegen durch eine explizite Angabe der Zeit beschrieben. Zum Beispiel wird das Verhalten eines einfachen Latches wie folgt spezifiziert:

$$\vdash_{def} \text{LATCH}\ (e, a, s) := \forall\ t.\ a\ (t+1) = s\ t \rightarrow e\ t \mid a\ t$$

Diese Spezifikation besagt folgendes: für alle Zeitpunkte t, zu denen das Signal s den Wert wahr hat, nimmt der Ausgang a zum Zeitpunkt t+1 den Signalwert des Eingangs e zum Zeitpunkt t an, sonst bleibt a unverändert. In dieser Spezifikation sind e, a und s Funktionen über der Zeit, die den Signalwert der entsprechenden Leitung zum Zeitpunkt t bzw. $t+1$ angeben.

In Logik höherer Ordnung kann man auch Existenzquantoren verwenden, um temporale Operatoren zu beschreiben. Zum Beispiel wird ein EVENTUELL-

Operator, der besagt, daß ein Signal *sig* eventuell den Wert wahr annehmen wird, durch folgendes Prädikat definieren:

$$\vdash_{def} \mathsf{EVENTUELL}\ sig\ t_1 = \exists\ t_2.\ (t_2 > t_1) \wedge sig\ t_2$$

Angewendet auf das Signal *sig* und den aktuellen Zeitpunkt t_1 drückt das Prädikat die Existenz eines Zeitpunktes t_2 in der Zukunft aus, zu dem das Signal *sig* wahr wird. In der Logik höherer Ordnung wird die Existenzquantifizierung über die Zeit außerdem in der Spezifikation des Verhaltens von asynchronen Verbindungen zwischen Komponenten verwendet [Joyc89].

Die Erstellung der formalen Spezifikation ist eine Aufgabe, die nicht unterschätzt werden sollte. Für das Verhalten von Hardware wird die formale Beschreibung meist von Hand erstellt. Die Struktur hingegen kann automatisch generiert werden, da eine mechanische Übersetzung von maschinenlesbaren Hardware- oder Netzlisten-Beschreibungen wie z.B. EDIF [EDIF87] in eine Hierarchie von Prädikaten möglich ist [KrKS93].

Kapitel 4: RISC-Prozessoren und Pipelining

Die Verifikation komplexer Schaltungen, wie sie Mikroprozessoren darstellen, ist eine umfangreiche Aufgabe, die ein genaues Wissen über deren speziellen Hardware-Entwurf erfordert. Die formale Verifikation sollte daher mit den Entwurfsschritten gekoppelt sein, da spezialisierte Kenntnisse für die Auswahl einer geeigneten Verifikationsstrategie und für die Strukturierung der Korrektheitsbeweise von höchster praktischer Wichtigkeit sind [KaAL91]. In dieser Hinsicht werden in diesem Kapitel zunächst die wesentlichen Eigenschaften eines RISC-Prozessors und dann das Pipelineprinzip im allgemeinen und im Zusammenhang mit RISC-Prozessoren beschrieben.

4.1 Eigenschaften von RISC-Prozessoren

In Kapitel 2.4 wurden bereits einige charakteristische Merkmale von RISC-Prozessoren erwähnt. Diese werden hier näher erläutert, jedoch ohne daß tiefgreifende Details angegeben werden, da das den Rahmen dieser Arbeit sprengen würde und außerdem heute eine Reihe von Werken existiert, auf die man sich beziehen kann. In der Literatur findet man eine Fülle von charakteristischen Merkmalen, die einen RISC-Prozessor kennzeichnen [Henn84, Kate84, Patt85, CiVa87, GiMi87, Stal88, Goor89, Bode90, HePa90, Taba90, Tane90]. Es existieren jedoch keine einheitliche Kriterien zur Kennzeichnung einer RISC-Architektur. Daher gibt es zahlreiche kommerzielle Prozessoren, die als RISCs bezeichnet werden, obwohl sie abweichende Eigenschaften im Vergleich zu den ursprünglichen RISC-Entwürfen von Berkeley und Stanford aufweisen [GiMi87]. Außerdem gibt es Attribute, die eine RISC-Architektur kennzeichnen, die sich aber auch in CISC-Rechnern finden [GiMi87], wie z.B. großer Registersatz, Harvard-Architektur oder interne Koprozessorstruktur. Es lassen sich jedoch wesentliche Eigenschaften finden, die eine RISC-Architektur ausmachen und die heute als allgemein akzeptiert gelten, wenngleich sie nicht in jedem konkreten System eingehalten werden. Dazu gehören:

- *Reduzierter Befehlsatz*: Befehle werden bezüglich der statistischen Auftrittshäufigkeit und des Zeitverhaltens optimiert, so daß nur ein relativ kleiner Befehlssatz entsteht. Komplexe Befehle werden durch Folgen dieser elementaren Befehle realisiert.
- *Einzyklus-Maschinenbefehle*: Durch verschiedene Maßnahmen wird versucht, im eingeschwungenen Zustand in jedem Taktzyklus einen Befehl zu beenden.

Die Zielvorstellung der Ausführung in einem Zyklus wird jedoch im Mittel nur annähernd erreicht (s. Kapitel 4.2.2).

- *Load-Store-Architektur*: Alle Befehle, insbesondere die Rechenbefehle, arbeiten aus Zeitgründen grundsätzlich nur mit Registeroperanden. Auf den eigentlichen Datenspeicher (Arbeitsspeicher) greifen nur Lade- und Speicherbefehle zu, deren Häufigkeit man klein zu halten sucht.
- *Festverdrahtete Steuerlogik*: Aus Geschwindigkeitsgründen wird auf eine Mikroprogrammierung, die Maschinenbefehle interpretiert, verzichtet, da festverdrahtete Steuerwerke schneller sind.
- *Einfache Befehle und wenige Adressierungsarten*: Es werden prinzipiell nur elementare Befehle mit wenigen Adressierungsarten (meist zwei bis vier) implementiert. Damit werden die Steuerwerke klein und die Zeit für die Befehlsausführung wird kurz.
- *Feste Befehlslänge mit wenigen Formaten*: alle Befehle haben eine einheitliche Länge (meist ein Wort) und sind in wenigen Formaten (maximal vier) horizontal kodiert. Sie erlauben daher ein schnelles Holen aus dem Speicher und eine einstufige, gleichzeitige Dekodierung aller festen Felder des Formats.
- *Relativ großer Registersatz*: Die Einschränkung auf Register-Register-Befehle wird durch das Bereitstellen einer großen Anzahl von Allzweck-Registern ermöglicht, in denen auf alle relevanten Daten schnell zugegriffen werden kann.
- *Harvard-Architektur*: Im Unterschied zu einer klassischen von Neumann-Architektur werden hier Befehls- und Datenspeicher (Cache) physikalisch getrennt [HePa90, Goor89].
- *Koprozessorarchitektur*: Die Ausführungszeit spezifischer Anwendungen (wie Gleitkommarechnungen oder graphische Operationen) wird dadurch verkürzt, daß neben dem RISC-Kern entsprechende spezielle Koprozessoreinheiten auf dem Chip integriert werden, die über gemeinsame Ressourcen wie Registersatz, Cache oder MMU verfügen.

Zusammenfassend zeichnen sich RISC-Prozessoren vor allem durch zwei Hauptmerkmale aus: *Einfachheit* und extrem *hoher Durchsatz* [Bode90]:

- Einfachheit erreicht man durch kleine Befehlssätze, eine Koprozessorstruktur, eine Load-Store-Architektur und den Verzicht auf Mikroprogrammierung.
- Der extrem hohe Durchsatz ergibt sich aus den Einzyklusbefehlen, dem horizontalen Befehlsformat, dem großen Registersatz, der hierarchischen Speicherorganisation (vor allem durch die Harvard-Architektur) und der möglichst *nebenläufigen* Befehlsausführung.

Zur Steigerung des Durchsatzes spielt diese letztgenannte Eigenschaft der nebenläufigen Ausführung von Befehlen eine Schlüsselrolle, da die Hardware möglichst zu jedem beliebigen Zeitpunkt simultan mehrere Operationen ausführen soll [Kogg81]. Man unterscheidet in dieser Hinsicht zwei Prinzipien, nämlich *Parallelismus* und *Pipelining* [RaLi75, Kogg81, Goor89]. Unter Parallelismus versteht man die Ausnutzung von Nebenläufigkeit durch die Vervielfachung von (meist) identischen Hardware-Strukturen [Kogg81], wobei die Operationen der Maschine als Ganzes gleichzeitig ausgeführt werden können (s. Bild 4-1). Über die relativen Zeitverhältnisse des Beginns und des Endes der Ausführung sei dabei zunächst nichts ausgesagt. Pipelining liegt vor, wenn die Gesamtfunktion in eine sequentielle Folge von kleineren Einheiten (*Stufen*) aufgeteilt wird [Kogg81], die sich ihre Zwischenergebnisse jeweils weitergeben (s. Bild 4-1). Der Austausch der Zwischenergebnisse erfolgt dabei streng getaktet in Form einer Fließbandtechnik. Parallelismus und Pipelining können auch kombiniert auftreten [Kogg81, Gilo81]. Zum Beispiel werden bei superskalaren RISC-Architekturen beide Techniken zugleich angewandt, indem verschiedene Maschinenbefehle durch mehrere parallele Einheiten ausgeführt werden, die ihrerseits eine Ausführung in einer Pipeline aufweisen können [Bode90]. In dieser Arbeit wird jedoch in erster Linie das einfache Pipelineprinzip betrachtet, welches derzeit den Stand der Technik zur Erhöhung des Durchsatzes bei RISC-Kernen darstellt.

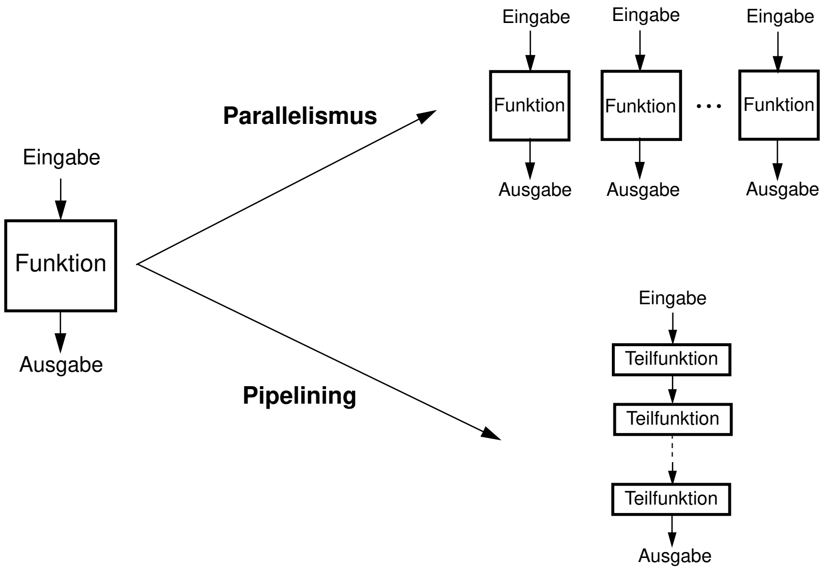


Bild 4-1: Parallelismus und Pipelining

RISC-Maschinen werden oft mit einer Pipeline-Implementierung assoziiert, da Pipelining eine naheliegende Technik ist, um das Ziel der Ausführung von Befehlen in einem einzigen Zyklus zu erreichen [Ston90]. Im folgenden Kapitel wird daher zunächst das Pipelineprinzip im allgemeinen genauer erläutert und dann wird speziell auf RISC-Pipelines eingegangen.

4.2 Pipelining

4.2.1 Definition und Klassifizierung

Im allgemeinen besteht das *Pipelining* (Fließbandverarbeitung) in der Zerlegung einer bestimmten Aufgabe in mehrere von einander unabhängige Teilaufgaben, welche dann taktsynchron in einer linearen Anordnung durch spezialisierte Einheiten hintereinander abgearbeitet werden [BuSo88]. Demnach bestimmt die Anzahl der Einheiten (*Stufen*) in dieser linearen Anordnung die *Pipelinetiefe*. Bei einer solchen Zerlegung einer Funktion in Teilaufgaben erhält man auf den ersten Blick keinen Geschwindigkeitsvorteil, weil man den Funktionsablauf lediglich in einzelne Teilaufgabe aufgeteilt hat. Da aber ein kontinuierlicher Strom von Eingabedaten berechnet werden muß, ergibt sich im Mittel dadurch ein Geschwindigkeitszuwachs, daß gleichzeitig alle Teilfunktionen ausgeführt werden, wenngleich auch auf verschiedenen, sequentiell ankommenden Eingabedaten. Voraussetzung dafür sind räumlich getrennte spezialisierte Verarbeitungseinheiten und eine Pufferung der Zwischenergebnisse. Demnach besteht jede Stufe einer Pipeline im allgemeinen aus zwei Elementen (s. Bild 4-2) [Kogg81, Haye88]:

- einem Verarbeitungsteil zur Ausführung der gewünschten Teilfunktion
- einem Puffer, der die Ausgabe des einen Verarbeitungsteils speichert und als Eingabe an den nächsten weiterreicht

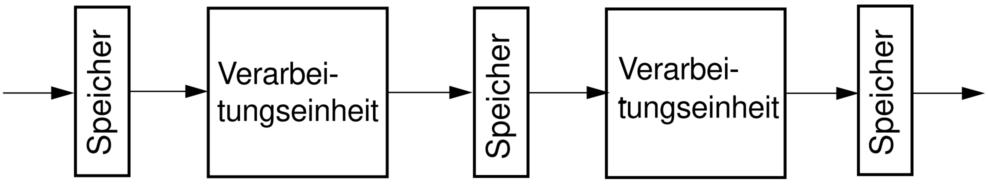
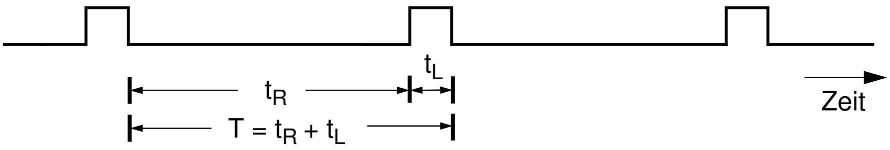


Bild 4-2: Grundaufbau einer Pipeline

Da der mittlere Durchsatz von der längsten Verweilzeit in einer Stufe abhängt, wird eine möglichst gleichlange Bearbeitungszeit in allen Stufen angestrebt. Betrachtet man einen eingeschwungenen Zustand einer Pipeline mit n Stufen, dann werden nach einer gewissen Auffüllzeit der Pipeline pro Zeiteinheit gleichzeitig n verschiedene Dateneingaben bearbeitet, allerdings in jeweils n verschiedenen Stufen. Dies bedeutet, daß eine Funktion, die bei sequentieller Bearbeitung c Takteinheiten benötigen würde, bei einer Pipeline mit n gleichlangen Stufen in c/n Takten abgearbeitet wird. Die Abarbeitung durch die n-stufige Pipeline ist also um den Faktor n schneller. Dieser Faktor n wird aber nicht immer erreicht, da der kontinuierliche Datenstrom durch dynamische Eigenschaften einer spezifischen Pipeline gestört wird. Nimmt man an, daß die Rechenzeit und die Speicherzeit (Pufferung) in jeder Pipelinestufe konstant t_R bzw. t_L sind (s. Bild 4-3), dann ist der mittlere Durchsatz umgekehrt proportional zur Verweilzeit T, welche die Taktperiode bestimmt (s. Bild 4-3) [Kogg81, Gilo81].



t_R = Rechenzeit einer Teilfunktion
t_L = Pufferzeit des Teilergebnisses
T = Taktperiode

Bild 4-3: Zeitlicher Ablauf einer Pipelineverarbeitung

Zur Beschreibung und Analyse der Aktivitäten innerhalb einer Pipeline benutzt man eine zweidimensionale Darstellung, das sog. *Raum-Zeit-Diagramm* [RaLi77, Haye88], welches die Verwendung der Pipelinestufen in Abhängigkeit von der Zeit angibt. Die vertikale Dimension stellt die verschiedenen Belegungen der Pipelinestufen durch die eingeschleusten Eingabedaten (E_1 bis E_n) dar, während die horizontale Achse die Zeit repräsentiert, deren Einheiten durch den Taktzyklus gegeben sind (s. Bild 4-4). Dem Raum-Zeit-Diagramm ist der aktuelle Zustand der Pipeline zu jedem Zeitpunkt eindeutig zu entnehmen, indem für jedes Eingabedatum angezeigt wird, in welcher Pipelinestufe es sich zu dem angegebenen Zeitpunkt (Takt) befindet [MüSc91]. Diese Darstellungsform wird im Laufe der vorliegenden Arbeit herangezogen, um das dynamische Verhalten einer Pipeline zu veranschaulichen.

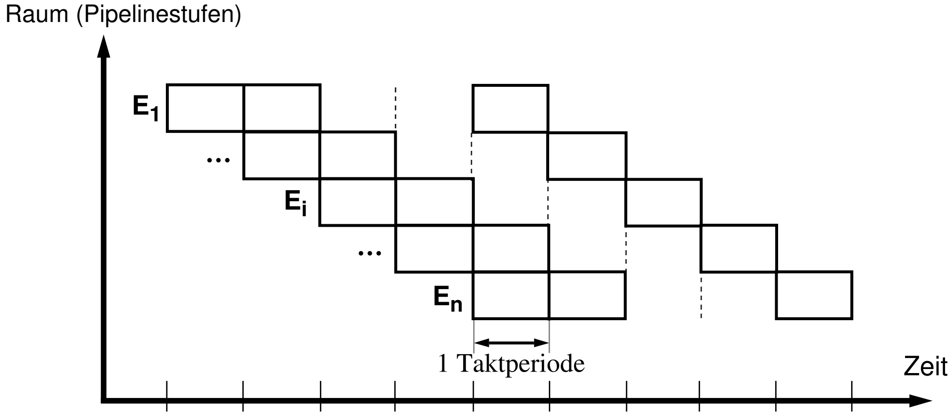


Bild 4-4: Raum-Zeit-Diagramm einer *n*-stufigen Pipeline

Verschiedene Arten von Pipelines werden allgemein nach unterschiedlichen Aspekten klassifiziert. In der Literatur findet man eine Reihe von Klassifizierungen, die jedoch nach verschiedenen Aspekten einer Pipeline gewichtet sind [Händ77, RaLi77, Kogg81, HwBr84]. Im folgenden wird dennoch versucht, eine etwas umfassendere Klassifizierung nach den wesentlichen Kriterien einer Pipeline anzugeben:

Nach der Form und den Berechnungsmöglichkeiten klassifiziert man Pipelines in *Einfunktions-* und *Mehrfunktionspipelines* [RaLi77, Kogg81, HwBr84]. Während eine Einfunktionspipeline immer dieselbe Folge von Operationen für alle möglichen Eingaben bearbeitet, werden bei Mehrfunktionspipelines mehrere unterschiedliche Funktionen in Abhängigkeit von einer gewissen Steuerung (z.B. Opcode eines Befehls) ausgeführt, wobei verschiedene Operationen verschiedene Hardware-Komponenten (Pipelinestufen) belegen.

Pipelines werden auch nach der Linearität klassifiziert [Kogg81, HwBr84, Bode90, MüSc91]. Demnach heißt eine Pipeline *linear*, wenn eine Verarbeitungseinheit nur in einer einzigen Pipelinestufe verwendet wird. Damit wird es möglich, die Pipeline zu jedem Zeittakt mit einer neuen Dateneingabe zu initiieren [Bode90]. Nichtlineare Pipelines sind gekennzeichnet durch Vorwärts- (“feedforward”) und Rückführungen (“feedback”) zwischen den Pipelinestufen [Kogg81, HwBr84, Haye88].

Nach dem aktuellen Gebrauch werden Pipelines als *statisch* oder *dynamisch* konfiguriert bezeichnet [RaLi77, Kogg81, HwBr84]. Bei den statischen Pipelines wird die auszuführende Funktion für längere Datenströme nicht geändert. Bei dynamischen Pipelines hingegen tritt eine Funktionsänderung häufig auf. Durch eine überlappte Ausführung von mehreren Funktionen können hier Funktionsänderungen im extremen Fall mit jeder neuen Dateneingabe geschehen. Nach dieser

Klassifizierung ist eine Einfunktionspipeline immer statisch [RaLi77]. Mehrfunktionspipelines hingegen können entweder statisch oder dynamisch sein [RaLi77].

Nach dem Zeitverhalten einer Pipeline unterscheidet man *synchrone* und *asynchrone* Pipelines [RaLi77, Gilo81, Kogg81]. Vorwiegend sind Pipelines synchron, wobei alle Elemente mit dem gleichen Takt angesteuert werden und jeder Puffer nur ein Datenwort enthält. Asynchrone Pipelines sind gekennzeichnet durch eine variable (jedoch im Mittel gleichlange) Bearbeitungszeit der Elemente und durch eine Pufferung mit einer FIFO-Struktur[5][Gilo81].

Das Pipelineprinzip kann sich auf verschiedenen Hierarchieebenen eines Systems vollziehen. Man spricht im allgemein auf der Ebene der einzelnen Operationen eines Rechenwerkes von *Datenpipelines* und auf der Ebene eines vollständigen Rechenwerkes von *Funktionspipelines* [BuSo88]. Betrachtet man ein Computersystem, so unterscheidet man in analoger Weise auf der Ebene der Funktionseinheiten eines Prozessors *arithmetische Pipelines* [Händ77, RaLi77, Kogg81, Haye88], auf der Ebene eines Befehlsstroms *Befehlspipelines* [Händ77, RaLi77, Kogg81, Haye88] und schließlich auf Systemebene *Prozessorpipelines*. [Händ77, Kogg81]. Während arithmetische Pipelines dem Benutzer völlig verborgen sind, könnten Befehlspipelines durch Sequenzen von Befehlen sichtbar gemacht werden [Kogg81], obwohl sie aus einer reinen Architektursicht dem Benutzer nicht sichtbar sein sollten [Kogg81, Haye88, HePa90]. Prozessorpipelines dagegen sind einem Benutzer immer sichtbar. Außerdem sind arithmetische Pipelines (z.B. in einer Gleitkommaeinheit) meist einfunktional, linear und statisch [Haye88]. Befehlspipelines andererseits sind meist mehrfunktional, nichtlinear und dynamisch konfiguriert [Kogg81].

4.2.2 Aufbau einer RISC-Pipeline

Um die Grundzykluszeit der RISC-Architekturen möglichst klein zu halten, wird der Maschinenbefehlszyklus in Stufen zerlegt, deren Ausführungszeit den Grundtakt bestimmt. Die Grundzykluszeit entspricht damit nicht der Ausführungszeit eines vollständigen Maschinenbefehls, sondern lediglich derjenigen einer Pipelinestufe. Die Stufen einer RISC-Pipeline entsprechen im allgemeinen der Partitionierung einer typischen Befehlsausführung (s. Bild 4-5) [CiVa87, Kane88, Bode90, HePa90]:

---

[5] Ein FIFO-Puffer enthält, im Gegensatz zu einem einfachen Puffer (latch), gleichzeitig mehrere Wörter, die in der Reihenfolge des Eintretens weitergereicht werden.

- *Befehl holen* (“instruction fetch”: *IF*): Zugriff auf den Speicher bzw. Cache, gesteuert durch den Befehlszähler, und Ablegen des Befehls im Befehlsregister.
- *Befehl dekodieren* (“instruction decode”: *ID*): Entnahme des Befehls aus dem Befehlsregister und Generierung der Steuersignale aus dem Operationscode, Weitergabe der Adressen aus dem Adreßteil an den Registersatz und Lesen der Operanden.
- *Befehl ausführen* (“instruction execute”: *EX*): Verknüpfung der Operanden im Rechenwerk bzw. Berechnung der Operandenadressen.
- *Speicherzugriff* (“memory access”: *MEM*): Daten aus dem Arbeitsspeicher laden bzw. in den Arbeitsspeicher speichern.
- *Abspeichern des Resultates* (“result write back”: *WB*): Zurückschreiben des Ergebnisses bzw. des geladenen Wertes in den Registersatz.

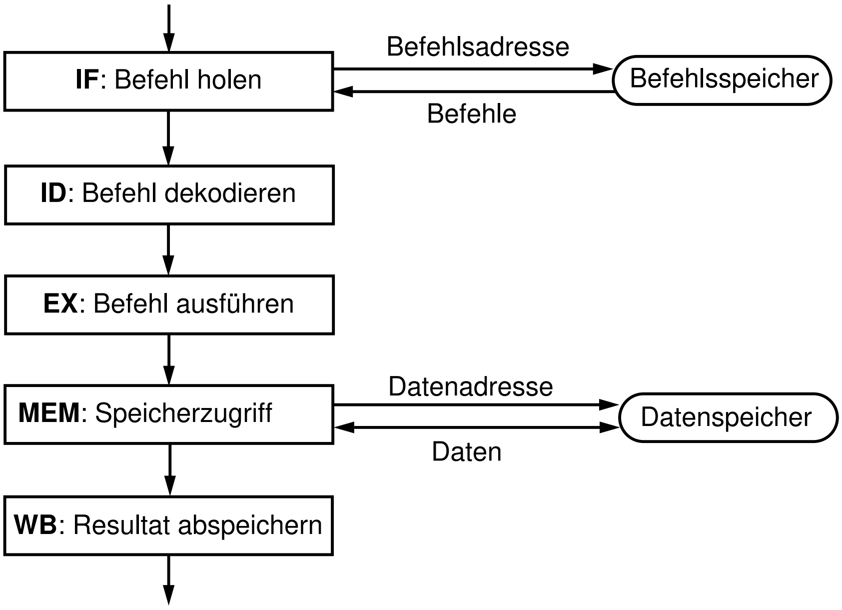


Bild 4-5: RISC-Befehlspipeline

Durch das Pipelining des Maschinenzyklus werden weniger Zyklen bzw. Takte pro Befehl benötigt (s. Bild 4-6). Anstatt, wie im sequentiellen Fall, den Maschinenbefehl *i+1* erst nach vollständiger Beendigung des Maschinenbefehls *i* zu bearbeiten, werden die Maschinenbefehle überlappend (“pipelined”) ausgeführt (s. Bild 4-6) [Stal88, Goor89, Bode90, HaVZ90, HePa90]. Auf diese Weise wird in der Regel (im eingeschwungenen Zustand) ein Durchsatz von einem Befehl pro Taktzyklus erreicht. Bei heutigen konventionellen RISC-Architekturen liegt die

Anzahl der Pipelinestufen in einer Größenordnung von 4 bis 5 [Kane88, Moto88, Inte89b, SunM89]. Superpipelined-Architekturen hingegen besitzen bereits Pipelinetiefen von über 7 [MIPS91, Digi92].

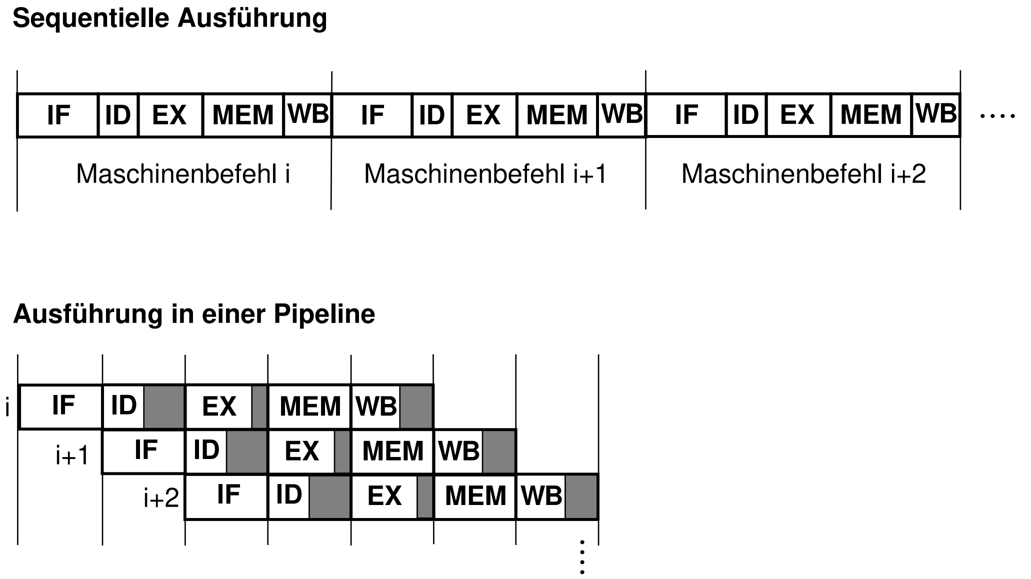


Bild 4-6: Sequentielle und Pipeline-Befehlsausführung

Nach der Klassifizierung in Kapitel 4.2.1 stellen konventionelle RISC-Pipelines *Befehlspipelines* dar. Demnach sind sie *mehrfunktional* und *dynamisch* konfiguriert [Kogg81]. Darüberhinaus sind RISC-Pipelines im allgemeinen *synchron* und *nichtlinear*, da aufgrund der Sonderbehandlung von Befehlen, deren Zeitverhalten nicht dem oben angegebenen Schema entspricht, sowie wegen *Pipelinekonflikte*n, die einen kontinuierlichen Befehlsstrom stören [Bode90, MüSc91]. Bei realen Programmen ist ein maximaler Durchsatz durch ein lineares Pipelineverhalten nicht erreichbar [Bode90, MüSc91].

Pipelinekonflikte entstehen, wenn es logische Abhängigkeiten der Pipelinestufen gibt, welche zu Doppelanforderungen der Verarbeitungseinheiten einer Pipelinestufe führen und eine Parallelisierung unmöglich machen [Bode90]. Für die Bearbeitung der Pipelinekonflikte gilt, daß die Pipeline angehalten werden muß bzw. durch Leerzyklen eine unerwünschte Ausführung verhindert werden muß, was die mittlere Anzahl der pro Maschinenbefehl benötigten Zyklen erhöht. Reale RISC-Architekturen sehen aber für jeden der Konflikte Lösungen in Hardware und/oder in Software (durch den Code-optimierenden Compiler) vor, welche die Anzahl der Leerzyklen verringern bzw. diese vollständig beseitigen. Die Erkennung und Beseitigung von Pipelinekonflikten bilden einen kritischen

Aspekt beim Entwurf und bei der Validierung eines RISC-Prozessors und sollen daher später bei der formalen Verifikation besonders behandelt werden. Eine ausführliche Beschreibung der Pipelinekonflikte und entsprechende Lösungstechniken werden auf das Kapitel 7.2 verschoben, in dem dann die entsprechende formale Spezifikation und Verifikation durchgeführt wird.

4.2.3 RISC-Pipelinestruktur

Die Architektur eines Prozessors soll hier vereinbarungsgemäß durch das Programmiermodell (Menge der sichtbaren Datenspeicherobjekte), den Befehlssatz (Menge der Befehle, die den Inhalt dieser Objekte verändern), sowie durch entsprechende Befehlsformate und Adressierungsarten beschrieben werden. Um die Hardware eines RISC-Prozessors, insbesondere die Pipeline, besser auszunutzen, wird diese Architekturschnittstelle nun in zwei Ebenen verfeinert [Milu89]: Die obere Ebene ist die *virtuelle* Architektur, die wie bisher nur das äußere Verhalten eines Maschinenbefehls angibt. Die untere Ebene ist die *reale* Architektur, in der im wesentlichen das Zeitverhalten des Befehls und eventuelle Pipelinekonflikte beschrieben werden. Die Maschinenbefehlsschnittstelle von RISC-Architekturen ist sehr maschinennahe und beinhaltet aufgrund der Pipelineverarbeitung komplexe Zeitbedingungen [Bode90]. Im Gegensatz zu CISC-Prozessoren ist die Pipelinearchitektur eines RISCs dem Programmierer bzw. Compiler nicht ganz verborgen, da manche Informationen wie Pipelinetiefe und parallele zustandsverändernde Pipelinestufenoperationen zu einer optimalen Codeerzeugung sowie zu der angesprochenen Pipelinekonflikterkennung und deren Beseitigung in der Software bereitgestellt werden müssen [Kogg81, Goor89, HePa90, MüSc91].

Die Architektur einer RISC-Pipeline wird meist durch eine *Pipelinestruktur* [HePa90] dargestellt. Eine Pipelinestruktur ist eine Tabelle, deren Spalten den verschiedenen Maschinenbefehlen entsprechen und deren Zeilen die Pipelinestufen repräsentieren. Anhand interner Ressourcen werden hierbei für jeden Befehl die zugehörigen Operationen jeder Pipelinestufe angegeben. Mit Hilfe der Pipelinestruktur ist es möglich, die zeitliche Aufteilung von verschiedenen Zustandsänderungen zu verfolgen und zugleich Abhängigkeiten von Stufenoperationen festzustellen.

Tabelle 4-1 zeigt den Grundaufbau der Pipelinestruktur der 5-stufigen (IF, ID, EX, MEM und WB) Pipeline des DLX-Prozessors für vier repräsentative Befehle: den Addier- (ADD), den Wortlade- (LW), den Wortspeicher- (SW) und den Sprung-Befehl (JMP).

Tabelle 4-1: Ausschnitt aus der Pipelinestruktur des DLX

	ADD	LW	SW	JMP
IF	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4
ID	A ← RF[rs1] B ← RF[rs2] IR1 ← IR	A ← RF[rs1] B ← RF[rs2] IR1 ← IR	A ← RF[rs1] B ← RF[rs2] IR1 ← IR	BTA ← PC+(IR) PC ← BTA
EX	ALUOUT ← A + B	DMAR ← A+(IR1)	DMAR← A+(IR1) SMDR ← B	
MEM	ALUOUT1 ←ALUOUT	LMDR ← D-MEM[DMAR]	D-MEM[DMAR]← SMDR	
WB	RF[rd] ← ALUOUT1	RF[rd] ← LMDR		

In Tabelle 4-1 stehen die Abkürzungen *I-MEM* und *D-MEM* für den Befehls- bzw. Datenspeicher. *PC* ist der Befehlszähler und *RF* bezeichnet den Registersatz. Die Abkürzungen *rs1*, *rs2* und *rd* sind Felder des Befehlsregisters *IR*; sie bezeichnen Quell- bzw. Zieladressen im Registersatz *RF*. *IR1* ist ein Puffer für das Befehlsregister *IR*. *DMAR* bezeichnet das Datenadressregister und *LMDR* bzw. *SMDR* sind Speicherregister für geladene bzw. zu speichernde Daten. Die Register *A* und *B* sind Eingangspuffer der ALU-Einheit und *ALUOUT* bzw. *ALUOUT1* sind Pufferregister für die ALU-Rechenergebnisse. Schließlich bezeichnet die Abkürzung *BTA* einen Zwischenpuffer für Sprungadressen.

Kapitel 5: Das hierarchische RISC-Modell

5.1 Einleitung

Ein wichtiger Aspekt bei der Verwendung der Logik höherer Ordnung ist die Möglichkeit der hierarchischen Verifikation bezüglich benachbarter Ebenen einer Hardware-Abstraktionshierarchie [Gord86, Joyc91]. Typischerweise wird der Entwurf von Systemen in verschiedenen Abstraktionsebenen organisiert. Die höchste Ebene stellt die abstrakteste Sicht des Systems dar, die unterste Ebene ist die konkrete Realisierung. Die Verifikationsaufgaben können dann auf natürliche Weise mit Hilfe derselben Abstraktionsebenen wie im Entwurf organisiert werden, wobei eine Implementierungsbeschreibung auf einer Ebene als Spezifikation für die nächstniedrigere Ebene dient (s. Bild 5-1). Auf diese Weise werden Spezifikationen schrittweise auf jeder Ebene implementiert und verifiziert, so daß man schließlich ein vollständig verifiziertes System erhält. Eine hierarchische Strukturierung erleichtert infolgedessen nicht nur den Verifikationsprozeß, sondern auch die Verifikationsaufgabe, da eine Reihe einzelner Beweisstufen von relativ geringer Komplexität behandelt werden können. Ein hierarchischer Ansatz erlaubt außerdem die Entdeckung von Fehlern in früheren Phasen des Verifikationsprozesses [NaSt88]. Sowohl die Organisation der Hierarchie als auch die Auswahl der Abstraktionen ist Aufgabe des Verifizierers, der die entsprechenden Abstraktionen und die dazwischenliegenden Beschreibungen ermitteln muß [KaAL91]. Diese Technik der hierarchischen Strukturierung hat es ermöglicht, einige Prozessoren mit beachtlicher Größe formal zu verifizieren [Hunt89, Joyc89, Wind90a, Alle93].

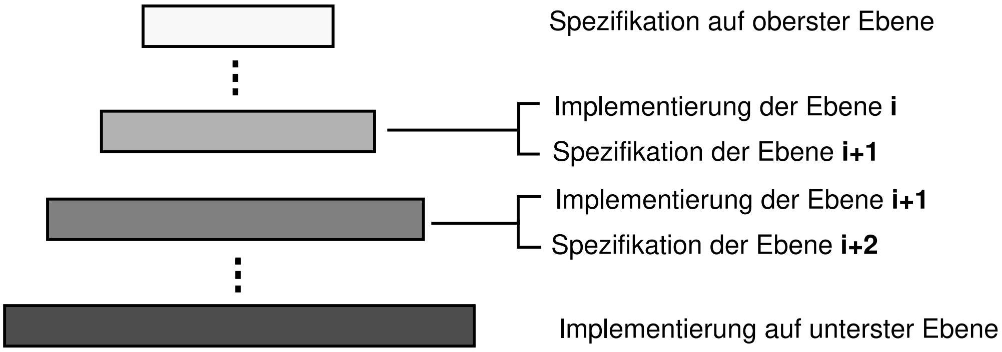


Bild 5-1: Hierarchische Verifikation

5.2 Der Prozeß der Mikroprozessorverifikation

Die Verifikation eines gegebenen Mikroprozessors besteht im allgemeinen aus dem Beweis dafür, daß die Hardware-Implementierung die Semantik jedes Befehls des Befehlsatzes erfüllt. In Anlehnung an die Definitionen in Kapitel 1.2.1 ist die *Spezifikation* damit die abstrakte Verhaltensbeschreibung der Architektur aus der Programmierersicht. Die *Implementierung* stellt dagegen die konkrete Strukturbeschreibung des Hardware-Entwurfs dar und wird entweder schematisch oder als Netzliste angegeben (s. Bild 5-2). Bei realen Mikroprozessoren, die einerseits einen vielfältigen und umfangreichen Befehlssatz, andererseits eine große, komplexe Hardware-Implementierung besitzen, führt eine direkte Verifikation der Implementierung bezüglich der Spezifikation zu hunderten von mehrseitigen Theoremen, die einzeln zu beweisen wären [Cohn89b]. Diese Schwierigkeit beruht auf der Existenz einer großen *semantischen Lücke* [Benn87] zwischen diesen beiden extremen Ebenen. Die Verifikation einer komplexen Schaltung soll also nicht nur zwischen einem Verhaltensmodell und einer Strukturbeschreibung, sondern vielmehr zwischen einer Vielzahl von passenden Abstraktionsebenen geschehen [Wind90b], die zwischen der Befehlssemantik und der Hardware-Realisierung eingefügt werden müssen. Jede Abstraktionsebene realisiert nur relativ wenige mögliche Fälle der Operations- ausführungen. Die Verifikation des Mikroprozessors vereinfacht sich dadurch, daß alle langen und komplexen Beweise durch eine Reihe von mehreren Routine-Beweisen ersetzt werden. Darüberhinaus stellt jede Abstraktionsebene eine in sich geschlossene und selbständige Abstraktion dar, die unabhängig von den übergeordneten Ebenen verifiziert werden kann.

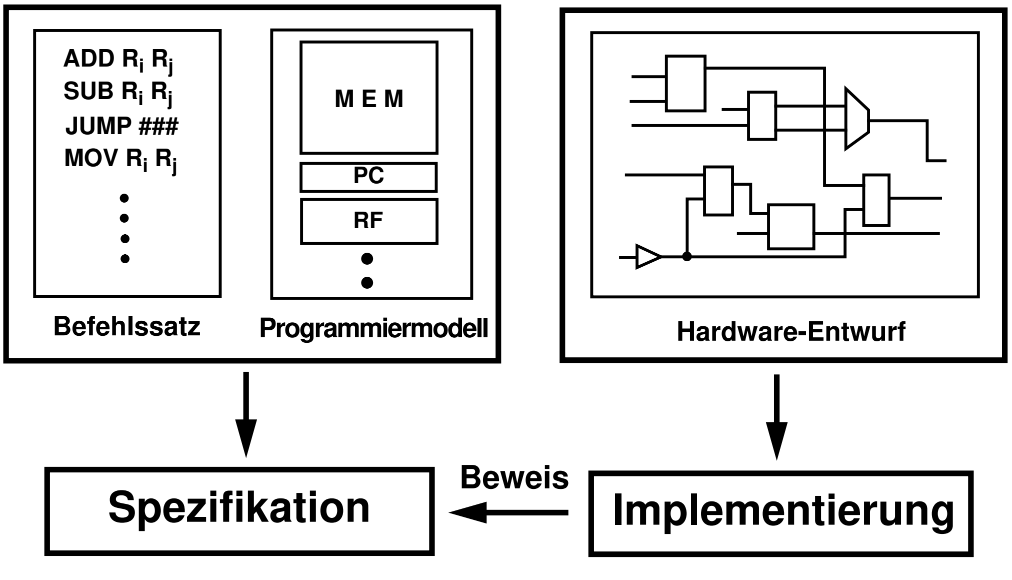


Bild 5-2: Ablauf der formalen Verifikation eines Mikroprozessors

Aus vorangegangenen Arbeiten (s. Kapitel 2.3) ist ersichtlich, daß ein direkter Korrektheitsbeweis zwischen einer Vielzahl von Befehlssemantiken und einer großen Hardware-Beschreibung zu einem exponentiellen Aufwand bezüglich der Anzahl der Befehle führt [Cohn89b, Wind90b]. Bei RISC-Prozessoren existiert wegen der Pipeline-Verarbeitung eine größere Zahl von Zuständen, in denen sich der Prozessor befinden kann. Der Versuch, alle möglichen Kombinationen von Befehlssequenzen zu verifizieren, führt zu einer größeren Anzahl der Fälle. Um den Korrektheitsbeweis von RISC-Prozessoren handhabbar zu machen, soll deshalb hier versucht werden, die Verifikation hierarchisch zu strukturieren, indem ein entsprechendes hierarchisches Modell für RISC-Prozessoren entwickelt wird.

In den nächsten Abschnitten wird ein hierarchisches Modell, genannt *Interpreter-Modell*, zunächst allgemein vorgestellt. Anschließend wird erläutert, wie dieses Modell bei der Verifikation von mikroprogrammierten Prozessoren eingesetzt wird. Danach wird, unter Verwendung der in Kapitel 4 gegebenen Erläuterung der wichtigsten Eigenschaften eines RISC-Prozessors, ein hierarchisches Modell zur Verifikation von RISC-Prozessoren vorgestellt.

5.3 Das Interpreter-Modell

Die Ausführung der Befehle eines Mikroprozessors ist durch eine bestimmte Sequenz von Operationen gegeben, die in einer Art hierarchischer Interpretation verschiedener Abstraktionsebenen dargestellt werden kann. Der Entwurf von Mikroprozessoren wird durch eine Hierarchie von sogenannten *Interpretern* aufgebaut, wobei jede Abstraktionsebene einem Interpreter entspricht [Ance86].

Allgemein ist ein Interpreter ein *Zustandsübergangssystem*, das hauptsächlich aus einer Menge von Zustandskomponenten, einer Umgebung, einer Menge von Zustandsübergangsfunktionen und einer Auswahlfunktion besteht [Wind90b]. Die *Zustandskomponenten* eines Interpreters entsprechen den sichtbaren Speicherelemente einer Abstraktionsebene (z.B. Akkumulator, Befehlszähler, usw.). Die *Umgebung* eines Interpreters besteht aus den sichtbaren Eingangssignalen, die aus der externen Umgebung zugeführt werden (z.B. Unterbrechungssignale). Die *Zustandsübergangsfunktionen* geben die Semantik der Befehle wieder, indem sie bezüglich des momentanen Zustands und einer Umgebung zu einem Zeitpunkt t den nächsten Zustand zum Zeitpunkt $t + 1$ in Abhängigkeit von der Semantik des aktuellen Befehls liefern. Ein Zustandswechsel geschieht zu jedem Zeittakt, welcher der abstrakten Zeiteinheit der betrachteten Ebene entspricht (z.B. ein Befehlszyklus auf der Architekturebene). Die *Auswahl-*

funktion selektiert nach einem bestimmten Kriterium (wie z.B. dem Opcode eines Maschinenbefehls) entsprechende Befehle. Unterschiedliche Ebenen der Interpreter-Hierarchie besitzen unterschiedliche Zeitskalen, deren Beziehungen durch Abbildungsfunktionen wiedergegeben werden.

Bild 5-3 stellt ein allgemeines Interpretermodell mit den Interpreterebenen $I_0 \ldots I_n$ dar, wobei der Pfeil zwischen zwei benachbarten Interpreterebenen angibt, daß die höhere Ebene eine Abstraktion der nächstniedrigeren ist und daß die niedrigere Ebene alle Befehle der höher liegenden Ebene implementiert. Hierarchisch gesehen schließen die Zustandskomponenten $\mathcal{Z}_{I_i}$ eines Interpreters I_i die Zustandskomponenten $\mathcal{Z}_{I_{i+1}}$ des übergeordneten Interpreters I_{i+1} ($\mathcal{Z}_{I_{i+1}}$ “$\subseteq$” $\mathcal{Z}_{I_i}$) ein. Gleiches gilt auch für die entsprechenden Umgebungkomponenten ($\mathcal{U}_{I_{i+1}}$ “$\subseteq$” $\mathcal{U}_{I_i}$).

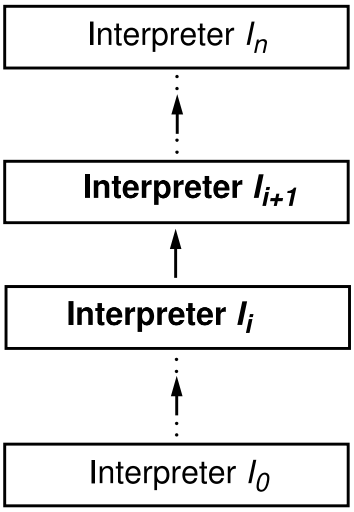


Bild 5-3: Interpretermodell

Seien $\mathcal{Z}_{I_i}$, $\mathcal{U}_{I_i}$ und $\mathcal{N}_{I_i}$ der Zustand, die Umgebung bzw. die Auswahlfunktion eines beliebigen Interpreters I_i, so kann dieser Interpreter formal in Logik höherer Ordnung durch ein Prädikat P_{I_i} spezifiziert werden, das für alle Zeiten den nächsten Zustand aus den aktuellen Zustands- und Umgebungsparametern bestimmt:

$$P_{I_i}(\mathcal{Z}_{I_i},\mathcal{U}_{I_i}) := \forall t.\ \mathcal{Z}_{I_i}(t+1) = \mathcal{N}_{I_i}(\mathcal{Z}_{I_i}(t),\ \mathcal{U}_{I_i}(t))$$

Die Korrektheit eines Interpreters I_{i+1} wird dadurch bewiesen, daß er von dem untergeordneten Interpreter I_i impliziert wird:

$$\forall \mathcal{Z}_{I_i}\mathcal{U}_{I_i}.\ P_{I_i}(\mathcal{Z}_{I_i}, \mathcal{U}_{I_i}) \Rightarrow P_{I_{i+1}}(\mathcal{Z}_{I_{i+1}}, \mathcal{U}_{I_{i+1}})$$

5.4 Das CISC-Interpretermodell

5.4.1 Einleitung

Bisherige Mikroprozessorverifikationen haben fast ausschließlich mikroprogrammierte Prozessoren, also CISCs, behandelt [Hunt85, Joyc89, Wind90a]. Der Entwurf dieser Art von Prozessoren wird durch eine Hierarchie von spezifischen Interpretern aufgebaut [Ance86]. Dieses in Bild 5-4 dargestellte Modell wurde erstmals von Joyce [Joyc89] und Windley [Wind90a] direkt ausgenutzt, um mikroprogrammierte Mikroprozessoren hierarchisch zu verifizieren. Dieses Modell besteht aus folgenden Hierarchieebenen:

- Die *Makroebene*, auch Instruktionsebene genannt, ist die höchste Interpreterebene der Hierarchie. Sie stellt die Architektur des Prozessors aus der Sicht eines Programmierers dar. Der Zustand der Makroebene wird durch das Programmiermodell gegeben, das die für den Programmierer direkt sichtbaren Komponenten der Architektur enthält, wie z.B. den Befehlszähler oder den Registersatz.

- Die *Mikroebene*, auch Mikroprogrammebene genannt, stellt die nächstniedrigere Interpreterebene dar. Sie implementiert die Maschinenbefehle durch entsprechende Sequenzen von Mikrobefehlen. Ein Zustandswechsel auf der Makroebene erzeugt in der Regel mehrere Zustandswechsel auf der Mikroebene, die durch ein entsprechendes Mikroprogramm hervorgerufen werden.

- Die *Phasenebene* zerlegt die Interpretation der einzelnen Mikrobefehle in elementare Operationen, die zum Teil parallel ausgeführt werden. Die Dauer eines Phasenbefehls entspricht der Phase eines nicht-überlappenden mehrphasigen Taktes (engl. “multiple phase non-overlapping clock”).

- Das *Elektronische Block-Modell* (kurz EBM) ist die niedrigste Abstraktionsebene des Interpretermodells. Es entspricht der Hardware-Realisierung des gesamten Systems. Das EBM selbst ist kein echter Interpreter, da es eher die Struktur des Prozessors als eine Verhaltensdarstellung darstellt. Das EBM implementiert das abstrakte Verhalten der Phasenbefehle, indem alle Phasenoperationen auf konkrete Hardware-Komponenten abgebildet werden.

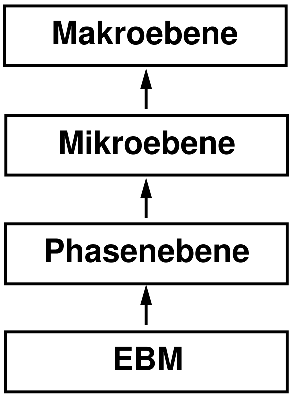


Bild 5-4: CISC-Interpretermodell

5.4.2 Abstraktionsformen

Man unterscheidet bei einem Interpretermodell im allgemeinen verschiedene Abstraktionsarten, die sich über alle Ebenen erstrecken [Melh88]:

- *Strukturabstraktion*: Die Struktur wird für jede Ebene durch die jeweiligen Zustandskomponenten der Interpreterebene dargestellt. Diese Abstraktion spiegelt die Sichtbarkeit der entsprechenden Zustandkomponenten auf den jeweiligen Ebenen wider.
- *Verhaltensabstraktion*: Das Verhalten einer Interpreterebene ist durch die Semantik ihrer Befehle gegeben. Die Befehle einer Interpreterebene abstrahieren das Verhalten der Schaltung auf der nächstniedrigen Ebene.
- *Zeitabstraktion*: Die Zeitgranularität jeder Interpreterebene wird auf der nächsttieferen Ebene verfeinert. Der Zeitgranularitäten der Makro-, Mikro- bzw. Phasenebene entsprechen Befehlszyklen, Taktzyklen (Mikrobefehlszyklen) bzw. einzelnen Taktphasen (s. Bild 5-5). Die Zeitgranularität des EBM ist die der Phasenebene, da einzelne Taktphasen die kleinsten synchronen Zeitintervalle des Systems darstellen.
- *Datenabstraktion*: Je nach Abstraktionsebene werden unterschiedliche Daten- und Operationsabstraktionen betrachtet. Auf der Makroebene werden arithmetische Operationen mit Hilfe von natürlichen Zahlen beschrieben. Auf der Mikro- und Phasenebene werden Operationen mit Bit-Vektoren gehandhabt. Auf der niedrigsten Ebene, hier das Elektronische Blockmodell EBM, werden einzelne Signale und logische Verknüpfungen implementiert.

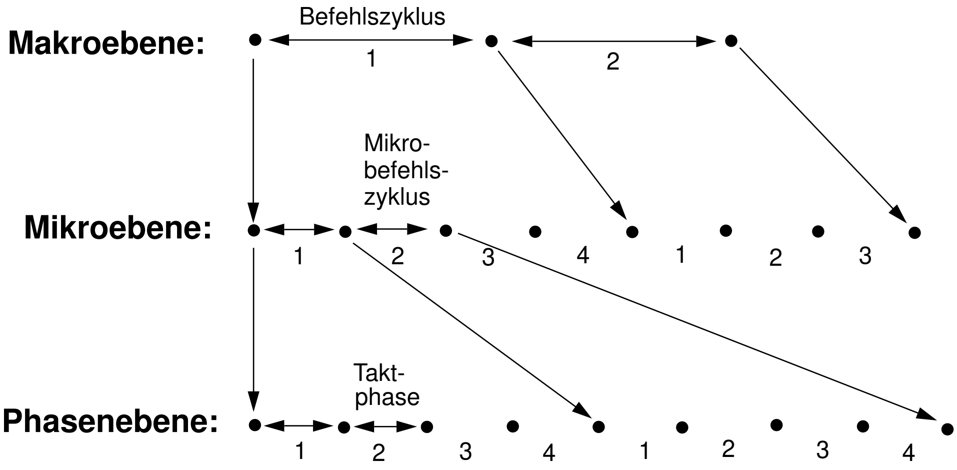


Bild 5-5: Zeitabstraktion bei CISC-Prozessoren

Den Abstraktionen dieses Modells entsprechen die sequentiellen Ausführungsphasen eines Befehls, wobei jeder Befehl in einer Sequenz von Mikrobefehlen implementiert wird, die wiederum durch Sequenzen von Phasenbefehlen ausgeführt werden. Dieses Abstraktionsmodell reflektiert außerdem die Art und Weise, wie ein Mikroprozessorentwurf praktisch realisiert wird: Der Entwerfer übersetzt die Funktionen sukzessive in mehreren Stufen [Ance86]. Aus früheren Verifikationsarbeiten ist ersichtlich, daß der größte Verifikationsaufwand beim Übergang vom EBM zur nächsthöheren Abstraktionsebene betrieben werden muß [Cohn89b, Wind90a]. Mit dem Interpretermodell ist der Abstraktionssprung zwischen dem EBM und der Phasenebene nicht mehr so groß. Außerdem braucht man nur noch eine geringe Anzahl von Phasenbefehlen (meist 4 bis 8) auf Korrektheit zu überprüfen. Dieses CISC-Interpretermodell wurde außerdem bei vielen anderen Arbeiten verwendet, um die formale Spezifikation, Simulation und Verifikation von Mikroprozessoren hierarchisch zu strukturieren [Joyc89, Wind90a, Aror90, Schu91b, HaWi93, PanJ92, ALLS94, FuWS94].

5.4.3 Verifikation mit dem Interpretermodell

Ziel der Verifikation ist es, zu zeigen, daß die Implementierung in Form des EBM die Semantik der Maschinenbefehle auf der Makroebene erfüllt. Frühere Mikroprozessor-Verifikationen wurden in einem Schritt gemacht, wobei das EBM direkt als die Implementierung der Makroebene betrachtet wurde ($P_{EBM} \Rightarrow P_{Makro}$) [Gord83b, JoBG86, Hunt85, Cohn89a]. Solche direkte Beweise sind schwer durchführbar, sogar manchmal unmöglich wegen der Fallexplosionen [Cohn89a]. Mit der hierarchischen Zerlegung wird der Beweis schrittweise gemacht $P_{EBM} \Rightarrow P_{Phase} \Rightarrow P_{Mikro} \Rightarrow P_{Makro}$. Dadurch wird auf einer Ebene nur eine reduzierte Expansion der formalen Beschreibungen nötig und es entsteht somit ein geringerer Beweisaufwand, z.B. müssen auf der Phasenebene nur 4 bis 8 Phasenbefehle vom EBM impliziert werden. Diese hierarchische Verifikation gliedert sich damit wie folgt in drei unabhängige Verifikationsschritte:

1. Verifikation der Makroebene $(P_{Mikro} \Rightarrow P_{Makro})$: Die Korrektheit aller Maschinenbefehle wird anhand der Mikroprogrammebene verifiziert. Die Implementierung (Mikrobefehle) ist nicht komplex und die semantische Lücke ist relativ klein.
2. Verifikation der Mikroebene $(P_{Phase} \Rightarrow P_{Mikro})$: Es wird überprüft ob die Mikrobefehle von den Phasenbefehlen korrekt implementiert werden. Wiederum ist die Implementierung (Phasenbefehle) einfach und die semantische Lücke nicht zu groß.
3. Verifikation der Phasenebene $(P_{EBM} \Rightarrow P_{Phase})$: Die Semantik der einzelnen Phasenbefehle wird von der Hardware-Implementierung impliziert. Die semantische Lücke ist hier groß und die Implementierung (EBM) ist im allgemeinen komplex, jedoch ist die Anzahl der Phasenbefehle relativ klein (meist 4 bis 8 Phasen), so daß der Verifikationsaufwand handhabbar bleibt.

Obwohl man hier mit drei Korrektheitsbeweisen zu tun hat, nämlich $(P_{EBM} \Rightarrow P_{Phase})$, $(P_{Phase} \Rightarrow P_{Mikro})$ und $(P_{Mikro} \Rightarrow P_{Makro})$, weisen diese Beweise eine *einheitliche* und *ähnliche* Struktur auf, so daß der gesamte Verifikationsaufwand annehmbar ist.

5.5 Aufbau des RISC-Modells

5.5.1 Einleitung

Aus den bisherigen Kapitel geht hervor, daß eine hierarchische Strukturierung der Verifikationsaufgabe durch Einführung von einer oder mehreren verschiedenen Verhaltensbeschreibungen zwischen der Hardware-Realisierung und der abstrakten Spezifikation eine geeignete Lösung zur Beherrschung des Komplexitätsproblems ist.

RISC-Prozessoren unterliegen einer anderen Entwurfsphilosophie als CISC-Prozessoren. Die Art und Weise wie RISC-Prozessoren strukturiert sind und wie sie sich verhalten, unterscheidet sich daher von den CISC-Prozessoren. Zum Beispiel existiert bei RISC-Prozessoren wegen der festverdrahteten Steuerung keine Mikroprogrammebene mehr. Eine hierarchische Strukturierung der Verifikation entsprechend dem CISC-Interpretermodell ist daher offensichtlich direkt nicht mehr übertragbar und man ist daher bestrebt, andere Modelle zu entwickeln.

In Anlehnung an die Entwurfsschritte eines Prozessors (s. Kapitel 2.1) wird ein RISC-Prozessor auf verschiedenen Ebenen spezifiziert und simuliert. In [CFHM93] werden zum Beispiel folgende Ebenen vorgeschlagen: *Architekturebene* (“instruction set architecture”) bestehend aus der Programmierersicht der Architektur, *Organisationsebene* (“organization level”) bestehend aus der Pipelinearchitektur und den Funktionseinheiten, *Datenpfadebene* (“datapath level”) bestehend aus elementaren Komponenten zur Implementierung der Struktur der Organisationsebene und *Logikebene* (“logic level”) bestehend aus logischen Gleichungen und endlichen Automaten. Solch einer hierarchischen Strukturierung entspricht aber eher die Struktur des RISC-Prozessors und nicht sein Verhalten. Betrachtet man im allgemeinen das Verhalten der Befehlsausführung von RISC-Prozessoren (s. Bild 5-6), so stellt man fest, daß außer der Architekturebene hier andere Abstraktionsebenen herangezogen werden müssen.

5.5.2 Abstraktionsebenen einer RISC-Befehlsausführung

Aus Kapitel 4.2.2 geht hervor, daß ein RISC-Prozessor jeden Befehl in einer Anzahl von Verarbeitungsschritten in den Pipelinestufen ausführt, deren Dauer jeweils einem Maschinentaktzyklus entspricht. Unter Verwendung eines mehrphasigen, nicht-überlappenden Taktes wird entsprechend den Phasen einer Taktperiode (z.B. ϕ_1, ϕ_2, ϕ_3 und ϕ_4 in Bild 5-6) jede Stufenoperation in eine Anzahl von weiteren Teiloperationen unterteilt [Kate84, Kane88, HePa90]. Wir definieren einen *Stufenbefehl* als die Menge von Operationen (Register-Transfers) eines spezifischen Maschinenbefehls, die in einer entsprechenden Pipelinestufe ausgeführt werden. Zum Beispiel bilden die beiden Stufenoperationen “$DMAR \leftarrow A + (IR1)$” und “$SMDR \leftarrow B$” in Tabelle 4-1 den EX-Stufenbefehl des Wortspeicherbefehls SW. Die *Stufenebene* wird demnach definiert als ein Interpreter, dessen Zustand durch alle sichtbare Speicherelemente der Pipelinearchitektur gegeben ist und dessen Zustandsübergangsfunktion der Semantik der Stufenbefehle entspricht. Analog dazu definieren wir einen *Phasenbefehl* eines spezifischen Stufenbefehls als die Menge der Teiloperationen, die während einer Taktphase abgearbeitet werden. Die *Phasenebene* wird auch als Interpreter definiert, dessen Zustand durch die sichtbaren Speicherelemente auf der Ebene der Taktphasen repräsentiert wird und dessen Zustandsübergangsfunktion den Phasenbefehlen entspricht. Der Zustand der Phasenebene erweitert den der Stufenebene durch zusätzliche Speicherelemente, die Zwischenergebnisse auf der

Phasenebene speichern (z.B. das *BTA*-Register, das in der ID-Stufe des JMP-Befehls verwendet wird, s. Tabelle 4-1).

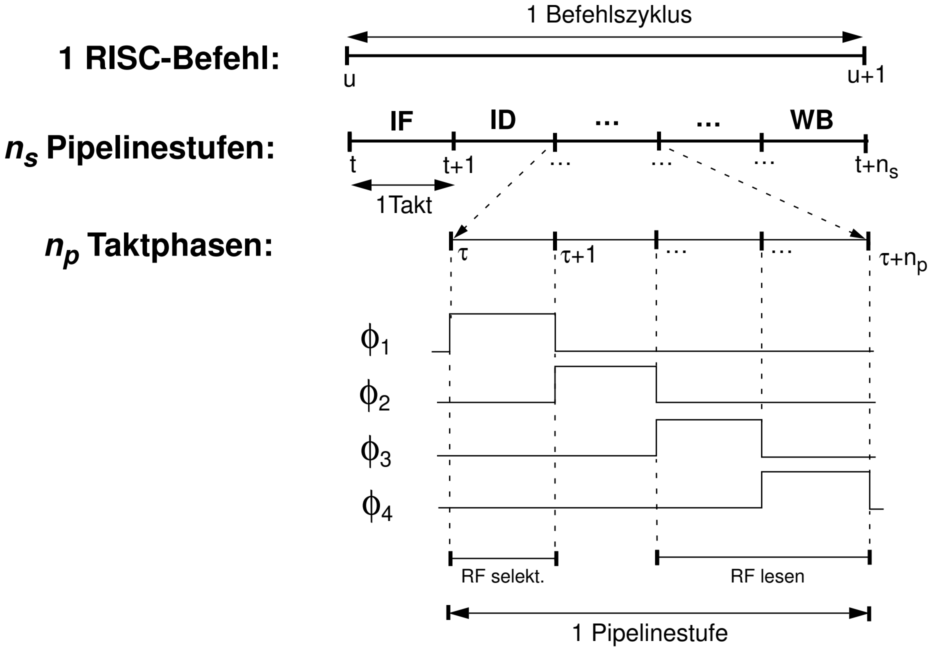


Bild 5-6: RISC-Befehlsausführung

Vergleicht man CISC- und RISC-Befehlsausführungen (s. Bild 5-5 und 5-6), so lassen sich einige Parallelen ziehen. Ein CISC-Maschinenbefehl wird durch eine Sequenz von Mikrobefehlen ausgeführt, die wiederum durch Sequenzen von Phasenbefehlen abgewickelt werden. Analog dazu wird ein Maschinenbefehl eines RISC-Prozessors durch eine Sequenz von Stufenbefehlen ausgeführt, welche wiederum durch eine Sequenz von Phasenbefehlen realisiert werden. Auf den ersten Blick sind also beide Hierarchien der Befehls-ausführungen identisch, jedoch täuscht dieser Eindruck aus folgenden Gründen:

1. Während die Anzahl der Phasenbefehle bei CISCs für alle Mikrobefehle konstant ist (z.B. Mikrobefehl holen, dekodieren, ausführen und Resultat abspeichern), ist sie bei RISC-Prozessoren von den Stufenbefehlen abhängig.
2. Während die Anzahl der Mikrobefehle von den einzelnen CISC-Maschinenbefehlen abhängt, ist die Anzahl der Stufenbefehle durch die gegebene Pipelinetiefe pro RISC-Maschinenbefehl annähernd konstant.

Diese unterschiedlichen Abhängigkeiten zwischen den Hierarchien von CISC- und RISC-Befehlausführungen sind in Bild 5-7 graphisch dargestellt.

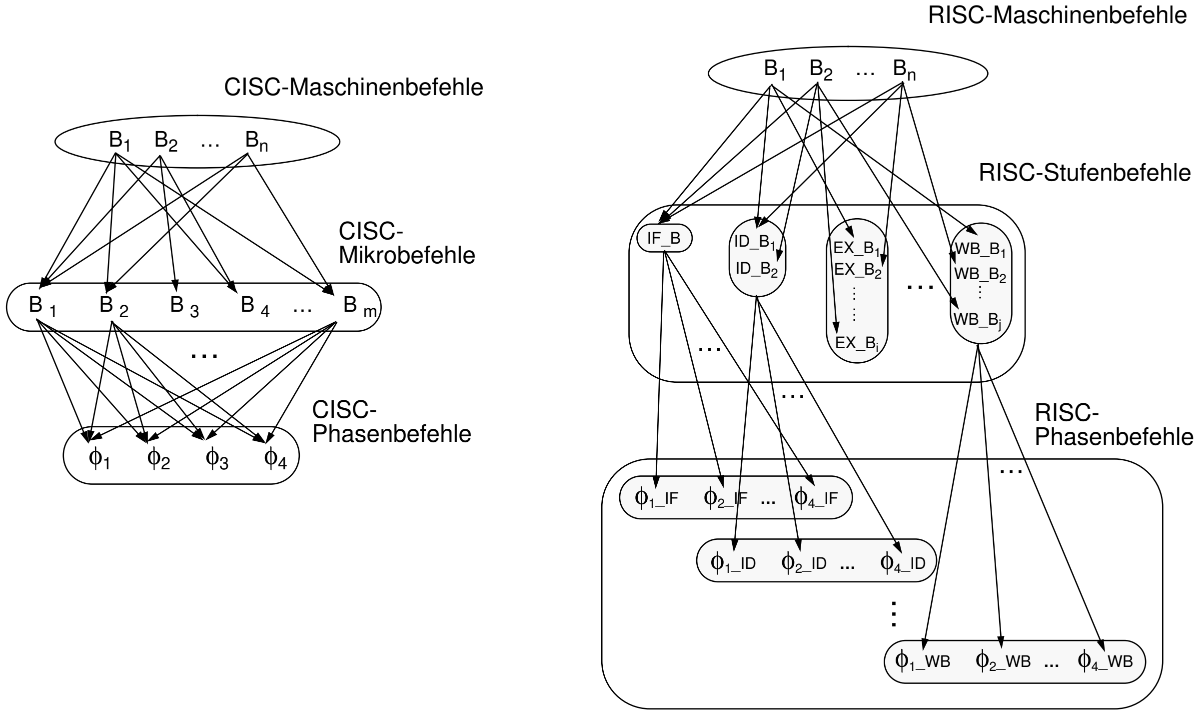


Bild 5-7: Hierarchieabhängigkeiten der Befehlsausführungen von CISC und RISC

Betrachtet man nun die syntaktischen und semantischen Eigenschaften der Befehle beider Prozessorklassen, so stellt man fest, daß die Maschinenbefehle eines RISC-Kerns wie CISC-Mikrobefehle elementar, einfach und horizontal kodiert sind, feste Formate haben und nach einer einheitlichen Schablone schnell dekodiert und ausgeführt werden [Kate84]. Ihre Komplexität und ihre logischen Eigenschaften können somit mit denen von CISC-Mikrobefehlen verglichen werden [Kate84, CiVa87, Goor89]. Ein offensichtlicher Unterschied bleibt jedoch die unterschiedliche Befehlslänge, da RISC-Befehle meist 32 Bit lang sind und Mikrobefehle über 128 Bit lang sein können. Diese Eigenschaft der Länge des Befehlworts wird aber auch heute bei neuartigen RISC-Architekturen, sog. VLIW [Ebci88] erfüllt [CGLT89, SoVa89]. Die Pipelinestufenoperationen ihrerseits sind mit Operationen der CISC-Phasen vergleichbar, da deren Anzahl durch die Pipelinetiefe, wie die Anzahl der CISC-Phasen, begrenzt ist und für alle Maschinenbefehle fast konstant ist. Darüberhinaus können RISC-Phasenoperationen mit CISC-Instanzen [Ance86] verglichen werden, da diese aufgrund der asynchronen Teiloperationen auf der physikalischen Ebene einer feineren Unterteilung der CISC-Phasen entsprechen. Im Gegensatz zu CISC-Prozessoren, wo Phasenbefehle wenigen Operationen (wie Mikrobefehl holen, dekodieren, ausführen, usw.) entsprechen, stellen Phasenbefehle von RISC-Prozessoren keine eindeutigen Operationen, sondern eine Aufteilung der

Stufenoperation in mehrere gleichlange physikalische Teilschritte dar [Bode90]. Die Phasenoperationen werden im allgemeinen nicht gleichmäßig benutzt und repräsentieren interne Synchronisationshilfsmittel, um eine überlappte Ausführung verschiedener Pipelinestufen durch eine Einschränkung einer Stufenoperation auf eine bestimmte Phase zu ermöglichen (siehe z.B. die Ausführungszeit der ID-Stufe in Bild 4-6) oder um Stufenoperationen zu beschleunigen, indem Teiloperationen in den ersten Phasen vorberechnet und zwischengespeichert werden [Kate84, Kane88, Chow89, Bode90, HePa90]. Man findet also hier auch wieder eine ähnliche hierarchische Strukturierung der Befehlsausführung wie bei den CISC-Prozessoren. Verwendet man diese Analogie, dann könnte man ein erstes Interpretermodell für RISC-Prozessoren angeben, das dem CISC-Interpretermodell ähnelt und das aus vier hierarchischen Ebenen für die Maschinen-, die Stufen- und die Phasenbefehle, sowie für die Implementierung EBM besteht (s. Bild 5-8).

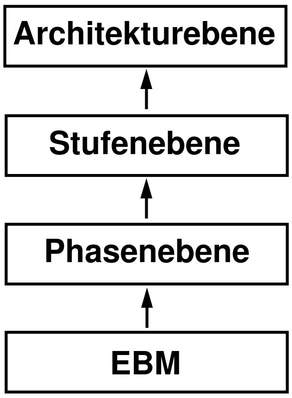


Bild 5-8: Einfaches RISC-Interpretermodell

Im Gegensatz zu den CISCs, wo die Anzahl der physikalischen Phasenoperationen für alle Mikrobefehle konstant ist, sind hier die Phasenoperationen von einer Stufenoperation zur anderen verschieden. Diese sind wiederum direkt von den Maschinenbefehlen (Maschinenbefehlen) abhängig (s. Bild 5-7). Seien N_a, N_s und N_p die Anzahlen der Maschinen-, Stufen- und Phasenbefehle eines gegebenen RISC-Prozessors, und seien n_s und n_p die Anzahlen der Pipelinestufen (Pipelinetiefe) bzw. der Phasen eines Taktes, dann würde die Anzahl der Phasenbefehle und folglich der Verifikationsschritte zwischen dem EBM und der Phasenebene theoretisch $(N_a \cdot n_s \cdot n_p)$ betragen. Da aber bekanntlich die Komplexität des Korrektheitsbeweises zwischen EBM und der nächst höheren Abstraktionsebene am größten ist [Cohn89a], führt die Verwendung dieses einfachen Modells zu keinem Vorteil. Zum Beispiel seien $N_a = 80$, $n_s = 5$ und $n_p = 4$. Demnach entspricht die Anzahl $(N_a \cdot n_s \cdot n_p)$ 1600 einzelnen Theoremen, die vom EBM impliziert werden müßten.

Als erste Lösung zur Reduzierung dieser Anzahl der Beweisschritte nutzt man die Idee der *Befehlsklassen*.

5.5.3 Klassenabstraktion

Eine Befehlsklasse entspricht der intuitiven Zuordnung von Befehlen mit ähnlicher Semantik zu einer Gruppe (z.B. arithmetisch-logische, Gleitkomma-, Lade-, und Kontrollbefehle, bezeichnet mit ALU, FLP, LOAD, CONTROL). Diese Klassifizierung findet sich auf natürliche Weise in dem Befehlssatz jedes herkömmlichen RISC-Prozessors [Kane88, Moto88, Inte89b, SunM89, Digi92, KaHe92] und bestimmt die Eigenschaft der Regularität der Befehle. Die Idee der Befehlsklassen wird in den ersten Entwurfsphasen eines gegebenen RISC-Prozessors befolgt, um einen strukturierten und modularen Entwurf zu erhalten und außerdem eine möglichst effizient balancierte Pipelineverarbeitung zu erzielen [Kane88].

Die Kriterien, wonach eine Befehlsklasse bei RISC-Prozessoren gebildet wird, können im allgemeinen wie folgt zusammengefaßt werden:

- Gleichartige Semantik der Befehle,
- gleiche Anzahl der benötigten Pipelinestufen,
- Verwendung von Stufenbefehlen mit ähnlicher Semantik,
- Befehlsausführung durch denselben Typ von Funktionseinheiten,
- Ähnlichkeit in der zeitlichen Ausführung im Datenpfad und
- Ähnlichkeit der Steuersequenz im Kontrollpfad.

Variabel hingegen können die Adressierungsmodi, die Operanden (Register), die Operatoren, das Befehlsformat (Opcode) und weitere Funktionalitäten, die keine strukturellen Veränderungen verursachen, sein.

Die Idee der Befehlsklassen wird nicht nur in diesem Zusammenhang, sondern auch in anderen verwandten Arbeiten über Mikroprozessoren verwendet, wie z.B. in der Simulation [VNMP89] und Synthese [ClTh93] von RISC-Prozessoren, in dem Entwurf von superskalaren Prozessoren [JoWa89] oder bei Mikroprozessortestverfahren [TaAS89]. Außerdem kann diese Abstraktion der Befehle auch auf das Interpretermodell von CISC-Prozessoren angewendet werden, jedoch nur in geringfügigem Maße, da CISC-Befehle meist nicht regulär sind (z.B. besteht der Befehlsatz des Intel-Prozessors i486 [Inte89a] aus ganzen 14 Befehlsklassen, die dort Befehlsgruppen genannt werden [Inte89a]).

5.5.4 Folgerungen der Befehlsabstraktion

Eine Befehlsklasse abstrahiert das Verhalten einer Gruppe von Maschinenbefehlen, die Struktur hingegen bleibt unverändert. Wir definieren einen *Klassenbefehl* als eine semantische Darstellung, welche die entsprechende Menge von Maschinenbefehlen abstrahiert, z.B. werden alle arithmetisch-logischen Befehle eines gegebenen Befehlssatzes durch folgenden ALU-Befehl

$$RF[rd] \leftarrow RF[rs1]\ op\ \ RF[rs2]$$

abstrahiert, wobei der Operator *op* für alle arithmetischen und logischen Operatoren (+, -, $\wedge$, $\vee$, usw.) steht. Ferner definieren wir die *Klassenebene* als einen Interpreter, dessen Zustand genau dem der Architekturebene — dem Programmiermodell — entspricht, und dessen Zustandsübergangsfunktionen durch die Semantiken der Klassenbefehle gegeben werden. Zwischen der Klassenebene und der Architekturebene existieren keine Struktur-, Zeit-, oder Datenabstraktionen. Sie werden daher beide auf der obersten Ebene eines Interpretermodells liegen.

Die Stufen- und Phasenbefehle können nun ihrerseits gemäß den Klassenabstraktionen parametrisiert werden, indem die gleiche Abstraktionsfunktion in der entsprechenden Stufen- bzw. Phasenoperation der zugehörigen Befehlsklasse verwendet wird. Beispielsweise kann die EX-Stufe der ALU-Klasse durch

$$ALUOUT \leftarrow A\ \ op\ \ B$$

repräsentiert werden. Die Stufen- und Phasenbefehle werden daher nicht mehr direkt von den Maschinenbefehlen abhängen, sondern nur noch von der Klassenebene.

Sei N_k die Anzahl der Klassenbefehle einer gegebenen Architektur, so reduziert sich die gesamte Anzahl der möglichen Befehle der Stufen- bzw. Phasenebene dann theoretisch auf $(N_k \cdot n_s)$ bzw. $(N_k \cdot n_s \cdot n_p)$.

Reale RISC-Prozessoren weisen zusätzliche Regelmäßigkeiten auf, die in das RISC-Interpretermodell aufgenommen werden können. Betrachtet man die Pipelinestruktur eines gegebenen RISC-Prozessors, so stellt man fest, daß manche Stufenbefehle gemeinsam von unterschiedlichen Klassenbefehlen verwendet werden. Beispielsweise wird bei allen Klassenbefehlen derselbe IF-Stufenbefehl benutzt (s. Tabelle 4-1). Außerdem wird von einigen Klassenbefehlen nicht die volle Pipelinetiefe benötigt (z.B. nur zwei Stufen für den CONTROL-Befehl), so daß insgesamt tatsächlich eine geringere Anzahl von möglichen Stufenbefehlen existiert als theoretisch angenommen wurde (d.h. $N_s < (N_k \cdot n_s)$). Die Anzahl der Phasenbefehle N_p, die in das Modell aufgenommen wird, beträgt somit $(N_s \cdot n_p)$.

Tabelle 5-1: Klassenabstrahierte Pipelinestruktur des DLX

	ALU	**LOAD**	**STORE**	**CONTROL**
IF	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4
ID	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$BTA \xleftarrow[\phi_1]{} \mathbf{f_C}(PC, IR, RF)$ $PC \xleftarrow[\phi_2]{} BTA$
EX	ALUOUT ← $\mathbf{f_A}$ (A, B)	DMAR ← A+(IR1)	DMAR← A+(IR1) SMDR ← B	
MEM	ALUOUT1 ← ALUOUT	LMDR ← $\mathbf{f_L}$ (D-MEM[DMAR])	D-MEM[DMAR]← $\mathbf{f_S}$ (SMDR)	
WB	$RF[rd] \xleftarrow[\phi_1]{} ALUOUT1$	$RF[rd] \xleftarrow[\phi_1]{} LMDR$		

Tabelle 5-1 zeigt die abstrahierte Pipelinestruktur des DLX-Prozessors mit den vier Befehlsklassen ALU, LOAD, STORE und CONTROL für arithmetisch-logische, Lade-, Speicher- und Kontrollbefehle, fünf Pipelinestufen IF, ID, EX, MEM, WB und zwei Taktphasen ϕ_1 und ϕ_2. Außerdem wird für jede Befehlsklasse die zugehörige Stufenoperation für jede Pipelinestufe angegeben. Darüberhinaus bezeichnet “←” in Tabelle 5-1 den Register-Transfer einer Stufenoperation, der nicht in Phasen aufgelöst wird. Solche Transfers werden durch eine zustandserhaltende Phasenoperation in der ersten Phase und eine Zustandsänderung in der zweiten Phase modelliert. “$\xleftarrow[\phi_1]{}$”, “$\xleftarrow[\phi_2]{}$” repräsentieren Transfers, die in der Phase 1 bzw. Phase 2 stattfinden. Phasenoperationen, die keine Zustandsveränderungen verursachen, werden nicht explizit angegeben.

Die Klassenabstraktion wird durch entsprechende Abstraktionsfunktionen f_A, f_L, f_S und f_C repräsentiert; f_A abstrahiert alle arithmetisch-logischen Operationen des Befehlssatzes, wobei f_A den oben angegebenen Operator *op* repräsentiert, indem statt der Infix- eine äquivalente Präfix Form verwendet wird ($f_A\ (A, B) \equiv A\ op\ B$); f_L und f_S abstrahieren den Speicherzugriffsmodus (Byte, Halfword oder Word), indem sie die entsprechenden Bits aus dem geladenen (*D-MEM [DMAR]*) bzw. zu speichernden (*SMDR*) Wort extrahieren, ausrichten und schließlich auf Wortlänge expandieren. Die Funktion f_C abstrahiert alle Kontrollbefehle wie direkte Sprünge (JMP), Register-Sprünge (JR) oder Verzweigungen (BRANCH), indem sie, in Abhängigkeit von der Art des Kontrollbefehls, die Zieladresse ausrechnet.

Zur Kennzeichnung der Stufen- und Phasenbefehle wird folgende Schreibweise vereinbart:

- Stufenbefehle: IF_A, ID_C, EX_S, MEM_L, usw.
- Phasenbefehle: $\phi_1 IF_A$, $\phi_2 ID_C$, $\phi_1 EX_S$, $\phi_2 MEM_L$, usw.

wobei die Indizes bei den Pipelinestufenbezeichnern jeweils den Anfangsbuchstaben der entsprechenden Befehlsklassen entsprechen. Da für alle Klassen ein gemeinsamer IF-Stufenbefehl existiert, wird stellvertretend gelegentlich die Bezeichnung IF_X verwendet, d.h. $IF_X = IF_A = IF_L = IF_S = IF_C$. Entsprechendes gilt auch für andere gemeinsame Befehle.

Phasenoperationen sind interne Synchronisationshilfsmittel, die (nach Bedarf) einer Unterteilung der Operationen der Pipelinestufen entsprechen und die technologisch bedingt sein können [Kate84]. Außerdem ist es bei vielen RISC-Prozessoren so, daß bei der Einführung eines Mehrphasentakts die Vermeidung eines Teils der Pipelinekonflikte bereits in der Entwurfsphase der RISC-Pipeline im Vordergrund steht. Durch die Aufteilung einer Stufenoperation in mehrere getrennte, zueinander komplementäre Phasenoperationen wird daher vermieden, daß Stufenoperationen um dieselben Ressourcen konkurrieren [Bode90, HePa90]. Zum Beispiel wird die Dekodierstufe ID in zwei Teile zerlegt (s. Bild 5-9), wobei in der ersten Phase des Maschinenzyklus der Befehl dekodiert wird und die Registeradressen berechnet werden, und in der zweiten Phase die Operanden aus dem Registersatz gelesen werden. Dadurch ist der Registersatz während der ersten Phase eines Zyklus unbenutzt. Deshalb kann dieser Teil eines Zyklus dazu verwendet werden, in der Schreibstufe (WB) das Resultat einer Berechnung in den Registersatz zu schreiben [Chow89, HePa90].

	erste Phase	**zweite Phase**
Dekodierstufe (ID)	Befehl dekodieren und Adressen berechnen	Operanden aus dem Registersatz lesen
Resultat-Schreibstufe (WB)	Resultat in den Registersatz schreiben	unbenutzt

Bild 5-9: Phasenaufteilung der Dekodier- (ID) und Schreibstufe (WB)

Eine nähere Betrachtung der Phasenebene eines RISC-Prozessors zeigt also, daß nicht alle Stufenoperationen in n_p Phasen aufgebrochen werden [Chow89, HePa90], wobei sich die entsprechenden Register-Transfers über den vollen

Taktzyklus erstrecken oder nur ein paar Phasen benötigen (s. Tabelle 5-1). Die verbleibenden (leeren) Phasenoperationen sind auf der Stufenebene nicht sichtbar. Diese Phasenoperationen werden dadurch modelliert, daß man den Zustand des Interpreters der Phasenebene unverändert läßt (z.B. $PC \xleftarrow[\phi_1]{} PC$). Die Anzahl von Phasenbefehlen, die semantische Zustandsänderungen hervorrufen, ist geringer als die gesamte Anzahl N_p, so daß der Aufwand der Spezifikation und Verifikation der Phasenebene insgesamt noch kleiner ausfällt.

Legt man die DLX-Architektur mit der entsprechenden Pipelinestruktur (s. Tabelle 5-1) als Beispiel zugrunde, so werden $N_a = 51$, $N_k = 4$, $N_s = 12$, $n_s = 5$ und $n_p = 2$. Damit würde eine einfache Berechnung $(51 \cdot 5 \cdot 2) = 510$ unterschiedliche Phasenbefehle liefern. Tatsächlich existieren nur $(N_s \cdot n_p) =$ $(12 \cdot 2) = 24$ Phasenbefehle und davon nur 4 mit semantischen Veränderungen.

5.5.5 Das hierarchische RISC-Modell

Das gesamte, bisher beschriebene hierarchische Modell für RISC-Prozessoren kann wie folgt dargestellt werden:

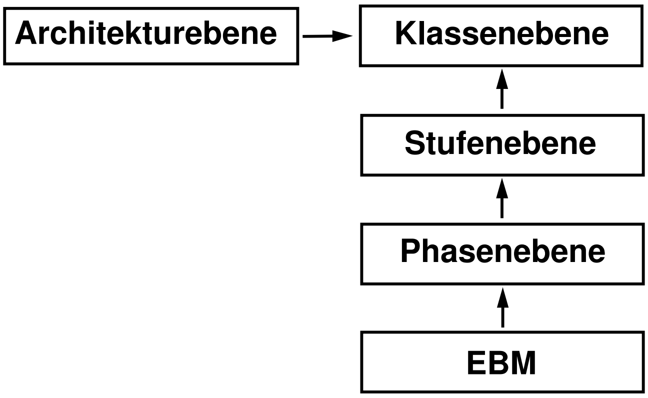


Bild 5-10: RISC-Interpretermodell

Hierbei stellt die Architekturebene die Menge der Maschinenbefehle des Befehlssatzes eines gegebenen RISC-Prozessors dar. Die Klassenebene ist die Menge der Klassenbefehle, die das Verhalten von Befehlsgruppen der gegebenen Architektur mit ähnlichen Semantiken abstrahieren. Die Stufenebene repräsentiert die Menge der Stufenbefehle, welche die einzelnen Klassenbefehle implementieren. Eine Sequenz von (parametrisierten) Stufenbefehlen bildet einen Maschinen- (Klassen-) Befehl. Ferner ist ein Stufenbefehl durch eine Sammlung von elementaren Operationen (Transfers) der entsprechenden Pipelinestufe auf Taktzyklusebene definiert. Die Phasenebene ist durch eine Menge von

Phasenbefehlen gegeben, welche die einzelnen Stufenbefehle realisieren. Eine Sequenz von (parametrisierten) Phasenbefehlen bildet einen (parametrisierten) Stufenbefehl. Schließlich besteht ein Phasenbefehl aus einer Sammlung von elementaren Operationen (Transfers) auf Taktphasenebene der entsprechenden Phase des Taktzyklus.

Jede Abstraktionsebene dieses Modells kann unabhängig von den anderen spezifiziert werden und soll damit einem Entwerfer dazu helfen, den Entwurf schrittweise zu verfeinern. Zusammenfassend besteht die gesamte Spezifikation der verschiedenen Abstraktionsebenen aus:

- einer Beschreibung der Architekturebene, die aus dem Befehlssatz des RISC-Prozessors abgeleitet wird,
- einer Beschreibung der Klassenebene, die durch eine Reihe von Abstraktionsfunktionen aus dem Befehlssatz abgeleitet wird,
- einer Beschreibung der Stufenebene, die aus der Pipelinestruktur abgeleitet wird,
- einer Beschreibung der Phasenebene, die durch Erweiterung der impliziten Phasenoperationen durch explizite Befehle aus der Pipelinestruktur abgeleitet wird und
- einer Beschreibung der Implementierung EBM, die aus einer Darstellung der Hardware-Realisierung auf der Logikebene abgeleitet wird.

Darüberhinaus wird es mit diesem RISC-Modell einem Verifizierer ermöglicht, den Verifikationsprozeß hierarchisch in sukzessiven Schritten zu bearbeiten. Dies geschieht wie folgt:

(1) $Stufenebene\ (N_s) \Rightarrow Klassenebene\ (N_k)$: Die Stufenbefehle implizieren alle N_k Klassenbefehle

(2) $Phasenebene\ (N_p) \Rightarrow Stufenebene\ (N_s)$: Die Phasenbefehle implizieren alle N_s Klassenbefehle

(3) $EBM \Rightarrow Phasenebene\ (N_p)$: Das EBM impliziert alle N_p Phasenbefehle

Unter Verwendung der Schritte (1), (2) und (3) ist man in der Lage, sukzessive die Korrektheit der Klassen-, Stufen-, und Phasenbefehle zu beweisen. Durch diese hierarchische Strukturierung und durch die Einführung der Klassenabstraktion wird nicht nur die semantische Lücke zwischen dem EBM und der Architekturebene geschlossen, sondern auch die Komplexität der Verifikationsaufgabe sowie deren Aufwandskosten reduziert.

Schritt (3) ist der schwierigste, da bei realen Prozessoren das EBM eine komplexe Strukturbeschreibung darstellt. Allerdings haben die Korrektheitsbeweise der Phasenebene eine ähnliche Natur, so daß sich hierfür eine gemeinsame Strategie entwickeln läßt (s. Kapitel 7.3.3). Die Schritte (1) und (2) bestehen aus dem Beweis einer Reihe von Implikationen zwischen Befehlen benachbarter Interpreter und sind damit relativ einfach zu handhaben (s. Kapitel 7.3.1 und 7.3.2).

Jeder dieser Verifikationsschritte kann unabhängig von den anderen durchgeführt werden. Somit ist es möglich, nicht nur eine "Bottom-Up" oder "Top-Down" Post-Validierung durch formale Verifikation, sondern auch einen "Top-Down" verifikationsbasierten Entwurf ("verification-driven design", oder auch "on-the-fly verification") durchzuführen, wobei ein gegebener Entwurf schrittweise gleichzeitig verfeinert und verifiziert werden kann (s. Kapitel 7.1.2).

Durch die Schritte (1), (2) und (3) haben wir bisher nur die Korrektheit der Klassenbefehle gezeigt. Das Endziel der Verifikation besteht jedoch im Beweis der Korrektheit der Maschinenbefehle. Dies wird dadurch erreicht, daß die Beweisschritte 1 bis 3 anschließend für die einzelnen Maschinenbefehle (siehe schattierte Pfeile 1' bis 3' in Bild 5-11) instanziiert werden. Diese Instanziierung geschieht durch die Ersetzung entsprechender Klassenabstraktionsfunktionen durch explizite Operatoren bzw. Funktionen (s. Kapitel 7.3.4). Zur Vollständigkeit muß noch die Aussage bewiesen werden, daß die Befehlsklassen alle Befehle des Befehlssatzes (Architekturebene) abstrahieren:

(4) $Architekturbene\,(N_a) \Rightarrow Klassenebene\,(N_k)$

Bild 5-11 repräsentiert schließlich das vollständige RISC-Modell, wobei die Hierarchieebenen *Stufenebene*' und *Phasenebene*' Instanziierungen der korrespondierenden Ebenen darstellen.

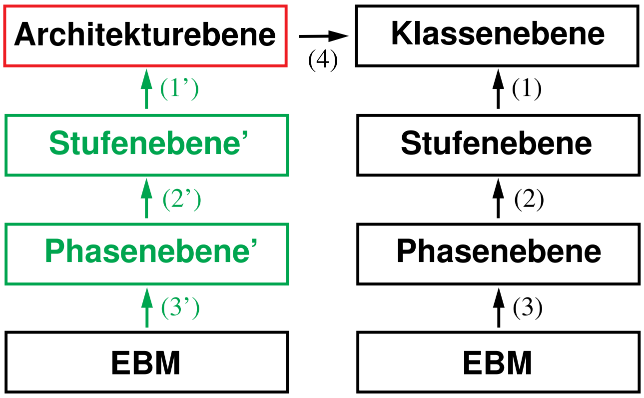


Bild 5-11: Vollständiges RISC-Interpretermodell

5.6 Vorteile des RISC-Modells

Die hierarchische Zerlegung der Spezifikations- und Verifikationsaufgabe durch eine Reihe von Interpretern bietet verschiedene Vorteile:

- Die hierarchische Zerlegung entspricht dem natürlichen Aufbau sowie dem natürlichen Pipelineverhalten eines RISC- Prozessors.
- Die Vielzahl der Abstraktionsebenen ermöglicht die Beherrschung der Verifikation durch Beseitigung der großen Anzahl der Fälle.
- Reduzierung der Komplexität der Befehlsinterpretation auf niedrigen Ebenen durch einfachere Befehle.
- Die Interpretationsebenen können als in sich geschlossene und selbständige Abstraktionen unabhängig voneinander spezifiziert und verifiziert werden.
- Korrektheitsbeweise zwischen verschiedenen Ebenen weisen einheitliche und ähnliche Strukturen auf und können somit durch verallgemeinerte parametrisierte Beweistaktiken bewiesen werden.
- Die hierarchische Strukturierung macht nicht nur die Verifikationsaufgabe handhabbar, sondern ermöglicht auch eine Automatisierung des Verifikationsprozesses.

5.7 Weiterer Einsatz des RISC-Interpretermodells

Das vorgestellte hierarchische Modell kann nicht nur zur Strukturierung des Verifikationsprozesses, sondern auch allgemein in einer VLSI-Entwurfsumgebung verwendet werden. Unabhängig von einer Formalisierung des Modells kann die hierarchische Strukturierung des RISC-Modells zum Beispiel direkt in der Simulation verwendet werden, wobei jede Spezifikationsebene eine virtuelle Maschine bildet, deren Verhalten vom Simulationsverfahren untersucht werden kann. Zum Beispiel wird in [Bere87] eine ähnliche Strukturierung der Simulation eines kommerziellen RISC Prozessors — des CRISP-Prozessors [BeDM87] — unternommen, wobei drei Hierarchieebenen (Architekturebene, Pipelinestufenebene und Hardware-Block-Implementierung) verwendet werden.

Das RISC-Modell kann außerdem in den Entwurfsprozeß einbezogen werden, indem die Einführung einer neuen Ebene durch eine wohldefinierte Abstraktion geleitet wird. In dieser Hinsicht kann also das Modell noch in der Synthese von RISC-Prozessoren angewandt werden.

Kapitel 6: Formalisierung des RISC-Modells

Nachdem im vorigen Kapitel ein hierarchisches Modell zur Spezifikation und Verifikation von RISC-Prozessoren vorgestellt wurde, wird in diesem Kapitel eine Formalisierung der verschiedenen Ebenen des RISC-Interpretermodells in Logik höherer Ordnung dargestellt, indem ein allgemeines Schema zur formalen Beschreibung der Befehle jeder Ebene des Modells angegeben wird. Dazu wird zunächst auf die verschiedenen Abstraktionsformen einer formalen Spezifikation kurz eingegangen. Dann wird ein Zeitabstraktionskonzept vorgestellt, das wegen des Pipelineverhaltens eines RISC-Prozessors eine große Rolle spielt. In den Abschnitten danach wird die formale Spezifikation der verschiedenen Abstraktionsebenen des Modells beschrieben.

6.1 Abstraktionsformen

Eine wesentliche Technik zur Beherrschung von umfangreichen Beweisen ist die Verwendung von Abstraktionsmechanismen. Durch sie sind formale Beweise zwischen genau definierten Abstraktionsebenen abgegrenzt und besser strukturiert und somit einfacher abzuleiten [KaAL91]. Die Betrachtung eines korrekten Abstraktionsmechanismus, der in diesem Kapitel näher beschrieben wird, spielt daher sowohl bei der formalen Spezifikation als auch bei der formalen Verifikation eine große Rolle. Innerhalb des im vorigen Kapitel dargestellten RISC-Verifikationsmodells werden unterschiedliche Abstraktionsebenen zugrundegelegt, die durch verschiedene Abstraktionsformen in Beziehung zueinander gebracht werden müssen. Melham [Melh88] präsentierte vier grundsätzliche Abstraktionsformen: *Strukturabstraktion*, *Verhaltensabstraktion*, *Datenabstraktion*, und *Zeitabstraktion*. Diese Abstraktionsformen wurden schon in Kapitel 5.4.2 für das CISC-Interpretermodell kurz erläutert und werden in den nächsten Abschnitten im Zusammenhang mit dem RISC-Modell einzeln beschrieben.

6.1.1 Strukturabstraktion

Eine Strukturabstraktion besagt, daß die *Spezifikation* einer Schaltung nur das extern sichtbare Verhalten reflektieren soll, während deren *Implementierung* explizite Informationen über die Struktur des internen Aufbaus beinhaltet [Melh88]. Der Mechanismus der Strukturabstraktion dient dazu, die interne

Struktur einer Hardware zu verbergen und im allgemeinen die Sichtbarkeit der Strukturkomponenten zu regeln.

Beim RISC-Interpretermodell spiegelt die Strukturabstraktion die Sichtbarkeit der Zustandskomponenten jeder Ebene wider. Beispielsweise sind auf der Architekturebene nur die Zustandskomponenten Befehlszähler *PC*, Registersatz *RF*, Befehlsspeicher *I-MEM* und Datenspeicher *D-MEM* sichtbar, die das Programmiermodell darstellen. Anhand dieser Struktur kann das Verhalten der Architekturebene durch entsprechende Zustandsübergangsfunktionen für jeden Maschinenbefehl beschrieben werden. Angesichts einer Verfeinerung der Implementierung werden im Modell hierarchisch von unten nach oben nur Teilmengen der Zustandskomponenten sichtbar gemacht (s. Bild 6-1 am Beispiel des DLX-Prozessors). Darüberhinaus existiert, wie bereits in den Kapiteln 5.4.2 und 5.5.4 erwähnt wurde, zwischen der Architektur- und der Klassenebene, sowie zwischen dem EBM und der Phasenebene keine Strukturabstraktion, da jeweils dieselben Zustandskomponenten sichtbar sind (s. Bild 6-1). Bei der formalen Spezifikation der Befehle jeder Abstraktionsebene werden nur die sichtbaren Strukturkomponenten in den entsprechenden Spezifikationsprädikaten als Parameter verwendet (s. Kapitel 6.3).

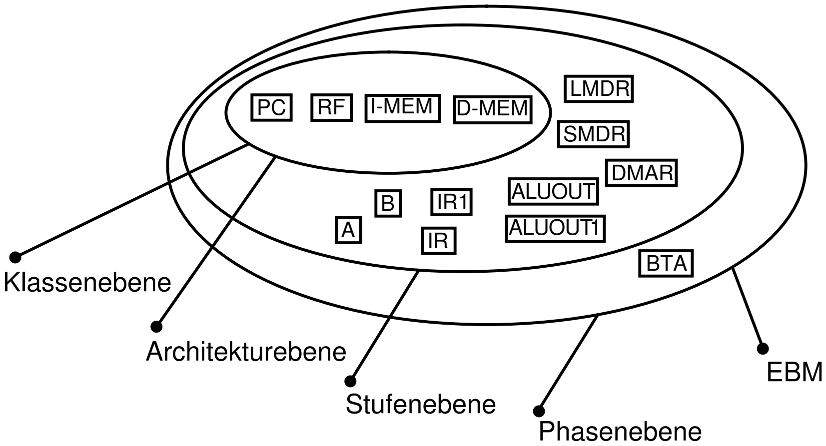


Bild 6-1: Strukturabstraktion des DLX-Prozessors

6.1.2 Verhaltensabstraktion

Die Verhaltensabstraktion ist eine natürliche Folge des hierarchischen Modells. Im allgemeinen ist eine Verhaltensabstraktion dadurch gegeben, daß eine Spezifikation nur das partielle Verhalten einer Komponente für die Zustände und die Umgebung, die von Interesse sind, beschreibt, wobei das Verhalten mancher Zustände unspezifiziert gelassen werden kann [Melh898]. Bei den formalen Korrektheitsbeweisen der verschiedenen Interpreterebenen wird üblicherweise eine Implikation von der konkreten zur abstrakten Ebene bewiesen (s. Kapitel 5.3).

Solche Korrektheitsbeweise entsprechen der Idee der Verhaltensabstraktion, wobei eine Spezifikation bezüglich einer Implementierung auf Korrektheit geprüft wird.

Die Architektur-, Klassen-, Stufen-, und Phasenebenen stellen Verhaltensbeschreibungen dar, während das EBM eine reine Struktur ist und daher keinen Interpreter darstellt (s. Kapitel 5.5.5). Jeder Interpreter abstrahiert das Verhalten der nächstniedrigeren Ebene durch die Semantik seiner Befehle. Beispielsweise wird unter Verwendung der Klassenabstraktionsfunktionen das Verhalten der Architekturebene durch das der Klassenebene abstrahiert, indem ganze Gruppen von Maschinenbefehlen durch wenige Klassenbefehle repräsentiert werden.

6.1.3 Datenabstraktion

Eine Datenabstraktion besteht in allgemeinen aus der Abbildung der Datentypen einer Implementierungsbeschreibung auf die abstrakteren Datentypen einer Spezifikation [Melh88]. Diese Abbildung wird dann dazu verwendet, zu zeigen, daß die benutzten Operationen auf dem Datentyp der niedrigeren Ebene die gewünschten Operationen auf dem Typ der höheren Ebene implementieren. Demnach beinhaltet eine formale Spezifikation die Definition der verschiedenen Datentypen und der entsprechenden Abbildungsfunktionen, die während des Korrektheitsbeweises die Datenabstraktion auflösen [Four90].

Üblicherweise werden in einem Computersystem verschiedene Datentypen für unterschiedliche Abstraktionsebenen verwendet (natürliche Zahlen, Bit-Vektoren, Bit-Signale, usw.). Da Bit-Vektoren und Bit-Vektoroperationen auf natürlicher Weise bei einem herkömmlichen Prozessor schon zur Beschreibung des Befehlssatzes im Benutzerhandbuch verwendet werden, wird in dieser Arbeit im Gegensatz zu den meisten Mikroprozessorverifikationsarbeiten (z.B. in [Gord83b, Hunt85, Cohn88, Joyce89a, Wind90a]) derselbe Datentyp — Bit-Vektoren — auf allen Ebenen zugrundegelegt und folglich keine Datenabstraktion vorgenommen. Demnach werden zur Spezifikation der Befehle aller Abstraktionsebenen generische Bit-Vektoren als einheitlicher Datentyp mit entsprechenden Bit-Vektoroperationen für alle Zustandskomponenten benutzt. Zusätzlich zum Vorteil der natürlichen Darstellungsform ergibt sich durch diese Entscheidung ein praktischer Aspekt, da formale Beweise durch die Einsparung von Abbildungsfunktionen zwischen abstrakten und konkreten Datentypen, die bei jedem Beweisschritt anzuwenden wären, erleichtert werden.

6.1.4 Zeitabstraktion

Formale Spezifikationen, die das Verhalten von zwei Interpretern mit verschiedenen Darstellungen diskreter Zeit beschreiben, werden durch eine Zeitabstraktion in Zusamenhang gebracht. Im allgemeinen wird eine Zeitabstraktion benötigt, wenn das sequentielle oder zeitabhängige Verhalten einer Komponente auf verschiedenen Granularitäten einer diskreten Zeit betrachtet wird, wobei die Implementierung mehr Details über das Zeitverhalten als ihre abstrakte Spezifikation aufweist [Melh88, Herb88]. Innerhalb des Mechanismus der Zeitabstraktion werden auf der Spezifikationsebene Informationen über das Verhalten der Implementierung zu bestimmten Zeitpunkten verborgen. Diese Eigenschaft wird durch eine Zeitabstraktionsfunktion f_z realisiert, welche die verschiedenen Zeitgranularitäten unterschiedlicher Abstraktionsebenen verbindet (s. Bild 6-2) [Melh88]. Durch diese Zeitabstraktionsfunktion werden Zeitpunkte oder Intervalle einer niedrigeren Ebene auf Zeitpunkte bzw. Intervalle einer höheren Ebene abgebildet. Demnach wird die Zusammenfassung verschiedener Abstraktionsebenen dadurch realisiert, daß gezeigt wird, daß eine Reihe von Rechenschritten auf niedriger Ebene einen einzigen Rechenschritt auf höherer Ebene implementiert [Melh88].

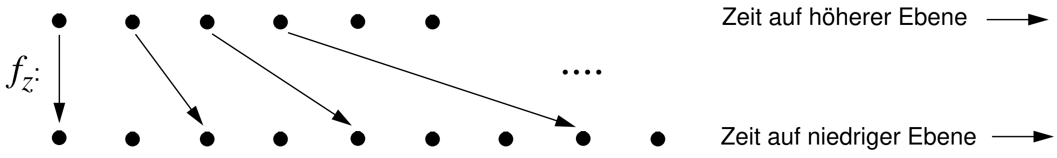


Bild 6-2: Zeitabstraktionsfunktion

Zur Modellierung des temporalen Verhaltens eines konventionellen Prozessors werden von der Ebene der Halbleiterkomponenten zu der Anwendungsebene hin viele Zeitgranularitäten identifiziert und herangezogen: Gatterlaufzeiten, Taktphasen, Taktzyklen, Mikrobefehlszyklen, Maschinenbefehlszyklen, usw. [Herb91, Joyc89, Wind90a]. Beim RISC-Modell werden entsprechende Zeitgranularitäten verwendet (s. Bild 6-3):

- Die Zeitgranularität der Architektur- und Klassenebenen entspricht dem Befehlszyklus der Ausführung einzelner RISC-Maschinenbefehle.
- Die Zeitgranularität der Stufenoperationen entspricht den Taktzyklen, welche die Ausführungsdauer einzelner Pipelinestufe darstellen.

- Die Zeitgranularität der Phasenebene ist durch die Dauer einzelner Phasen des Taktes gegeben.
- Die Zeitgranularität des EBM ist die der Phasenebene, da diese die kleinste synchrone Zeiteinheit des Systems darstellt, in der Register-Transfers des EBM stattfinden.

Folglich existieren zwischen der Klassen- und Architekturebene, sowie zwischen der Phasenebene und dem EBM keine Zeitabstraktionen.

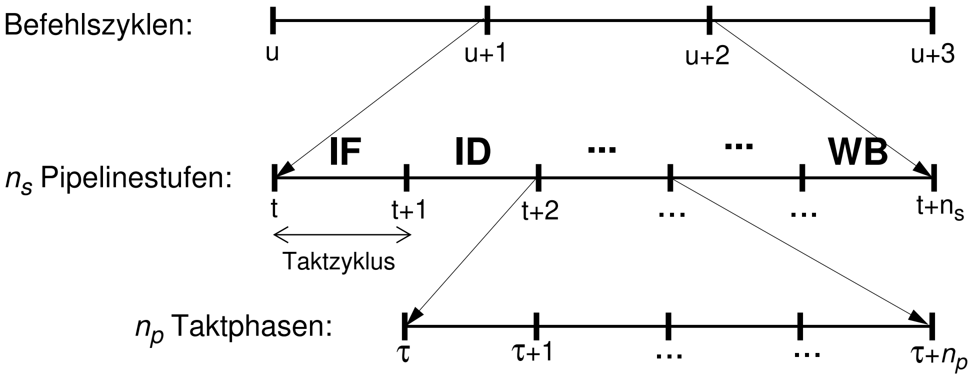


Bild 6-3: Zeitgranularitäten des RISC-Modells

Ein grundsätzlicher Schritt in der formalen Spezifikation des Interpretermodells ist die Bildung von mathematischen Beziehungen zwischen den abstrakten und den nächst konkreten Zeitskalen [Four90, Herb91]. Im einfachsten Fall einer Zeitabstraktion entspricht jede einzelne Einheit der diskreten Zeit auf der abstrakten Beschreibungsebene einer festen Anzahl von Zeiteinheiten auf der detaillierteren Beschreibungsebene [Four90]. In einer sequentiellen Ausführung von Befehlen begnügt man sich mit den Anfangs- und Endzeitpunkten des Ausführungszyklus (s. Bild 6-4), da Zustandsänderungen nur von einem einzigen Befehl hervorgerufen werden. Wegen überlappter Anfangs- und Endzeitpunkte können Zustandsänderungen auf der abstrakten Ebene zu irgend einem Zeitpunkt zwischen den diskreten Endpunkten des Zeitintervalls der Befehlsausführung stattfinden (s. Bild 6-4). Daher können bei einer Befehlsausführung in einer Pipeline Einheiten diskreter Zeit auf der abstrakten Ebene keine eindeutigen diskreten Zeitpunkte der konkreten Zeitskala zugewiesen bekommen.

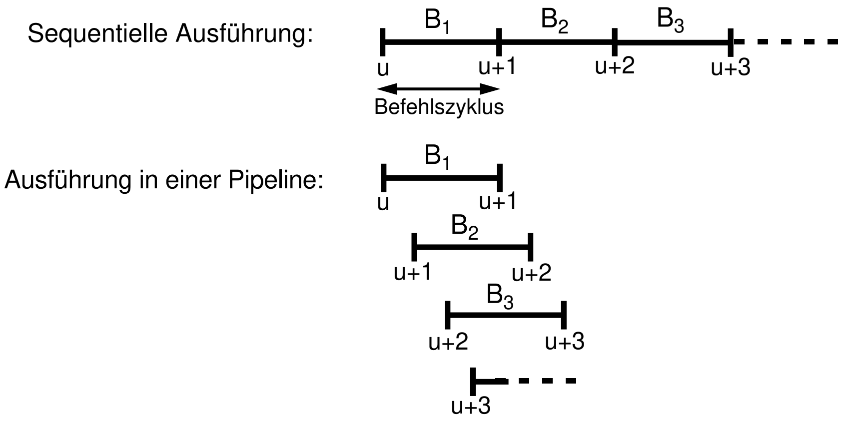


Bild 6-4: Temporale Abhängigkeiten von Befehlsausführungen

Eine Abbildungsfunktion, welche die Zeitabstraktion einer Befehlsausführung in einer Pipeline realisiert, ist nicht mehr linear. Es muß daher ein geeignetes Konzept zur Zeitabstraktion entwickelt werden, welches das temporale Verhalten einer Pipeline auf allen Ebenen des Modells berücksichtigt. Das nächste Kapitel widmet sich der Beschreibung eines solchen Konzepts.

6.2 Zeitabstraktionskonzept mit Pipelining

In einem herkömmlichen Prozessorhandbuch werden Maschinenbefehle durch Register-Transfers spezifiziert, wobei zeitliche Sachverhalte nicht explizit angegeben werden. Beispielsweise wird ein ADD-Befehl wie folgt definiert:

$$RF\,[rd] \leftarrow RF\,[rs1] + RF\,[rs2] \qquad \text{(Gl. 6-1)}$$

Unter Verwendung von Befehlszykluseinheiten “*u*”, kann eine zeitbehaftete Beschreibung dieses ADD-Befehls auf der Architekturebene wie folgt aussehen:

$$\forall\, u{:}\, Instr\_cycle.\ RF(u{+}1)\,[rd(u)] = RF(u)\,[rs1(u)] + RF(u)\,[rs2(u)] \qquad \text{(Gl. 6-2)}$$

Bezieht man sich auf die Pipelinestruktur in Tabelle 4-1, welche die genauen Lese- und Schreibzeiten enthält, dann kann eine genauere Beschreibung des ADD-Befehls unter Verwendung einer konkreteren Zeitgranularität in Form von Taktzyklen “*t*” wie folgt beschrieben werden:

$$\forall\, t{:}Clock\_cycle.\ RF(t{+}5)\,[rd(t)] = RF(t{+}1)\,[rs1(t)] + RF(t{+}1)\,[rs2(t)] \qquad \text{(Gl. 6-3)}$$

Eine Abbildungsfunktion, welche nun die abstrakte Zeitskala in Gleichung (6-2) auf die konkrete Skala in Gleichung (6-3) abbildet, ist offensichtlich nichtlinear, da eine Zeiteinheit auf der abstrakten Ebene nicht unbedingt einem diskreten Zeitpunkt auf der nächst konkreteren Ebene entspricht. Zum Beispiel soll derselbe

Zeitpunkt u, der in Gleichung (6-2) sowohl zur Berechnung der Registeradressen $rs1$, $rs2$ und rd als auch zum Lesen des Registersatzes RF verwendet wird, auf die verschiedenen Zeitpunkte t bzw. t+1 abgebildet werden.

Bei einer Ausführung in einer Pipeline finden Zustandsänderungen auf der abstrakten Ebene zu einem spezifischen Zeitpunkt zwischen den Endzeitpunkten des Zeitintervalls statt, d.h. in Abhängigkeit von der Implementierung zu einem Zeitpunkt zwischen u und u+1, der bei dieser abstrakten Zeitgranularität verborgen ist. Die zeitliche Information in der Spezifikation auf abstrakter Ebene ist unvollständig, und eine eventuelle Zeitabstraktionsfunktion muß daher zusätzliche Kontextinformationen aus der Implementierung heranziehen, um die abstrakte Zeit auf die konkrete abbilden zu können. Eine solche Kontextinformation muß die Anwendungsart (Lesen oder Schreiben) sowie die betreffende Pipelinestufe beinhalten und wird durch einen Kontextparameter dargestellt.

Ein *Kontextparameter* wird als Tupel repräsentiert, das aus einer Lese/Schreibe-Information und einem Pipelinestufenbezeichner besteht. Beispielsweise bezeichnen die Tupel [*read,ID*] und [*write,WB*] ein Lesen bzw. ein Schreiben während der ID- bzw. WB-Stufe.

6.2.1 Semantische und syntaktische Zeitdarstellung

Sei X eine beliebige Zustandskomponente, so können Zustandswerte in einer abstrakten Beschreibung der Befehle an Hand von Befehlszykluseinheiten "u" (s. Gleichung (6-2)) semantisch wie folgt interpretiert werden: $X(u)$ und $X(u+1)$ entsprechen die gelesenen bzw. geschriebenen Werte von X in dem halboffenen Intervall [u ; u+1 [bzw.] u ; u+1]. Syntaktisch müssen diese Werte in den entsprechenden Beweiszielen um eine Abstraktionsfunktion f_z und einen Kontextparameter C_X ergänzt werden, d.h. statt $X(u)$ würde man $(X \circ f_z)(u)$ einsetzen, wobei das Zeichen "$\circ$" die Funktionskomposition darstellt.

Die formale Spezifikation des ADD-Befehls auf der Architekturebene kann nun entweder die Form der Gleichung (6-2) annehmen [TaKu93b] oder, wie folgt, gleich die Zeitabstraktionsfunktion samt Kontextparameter einbeziehen [TaKu93a]:

$$\begin{aligned}
&\forall u\colon Instr\_cycle.\\
&\quad \exists\, C_1\, C_2\, C_3\colon Context. \qquad\qquad\qquad\qquad\qquad\qquad \text{(Gl. 6-4)}\\
&\quad (RF \circ f_z\, C_1)\,(u{+}1)\,[(rd \circ f_z\, C_2)\,(u)] = ((RF \circ f_z\, C_3)\,(u)\,[(rs1 \circ f_z\, C_2)\,(u)]) +\\
&\qquad\qquad\qquad\qquad\qquad\qquad ((RF \circ f_z\, C_3)\,(u)\,[(rs2 \circ f_z\, C_2)(u)])
\end{aligned}$$

Dabei ist f_z die Zeitabstraktionsfunktion und C_1 bis C_3 sind drei Kontextparameter, die wie oben angedeutet, auf die Verwendungsart (Lesen oder Schreiben) und auf die Verwendungsstelle bzw. die Pipelinestufe hindeuten sollen und die in Abhängigkeit von der implementierten Pipelinestruktur gesetzt werden. Zum Beispiel: wenn der Registersatz *RF* in der ID-Stufe gelesen wird (s. Tabelle 4-1), wird die Funktion f_z auf *RF* mit dem Kontextparameter $C_3 = [read, ID]$ angewandt. Ohne Kenntnisse von Implementierungsdetails deutet die Existenzquantifizierung der Kontextparameter nur auf die Existenz solcher Variablen hin, ohne daß dadurch ihr Wert angegeben wird. Unabhängig von der implementierten Pipelinearchitektur wird systematisch für jedes Auftreten einer Zustandskomponente (Register oder Speicher) ein Kontextparameter zugewiesen[6].

Die beiden alternativen Spezifikationen (s. Gleichungen (6-2) und (6-3)) haben den Vorteil, daß sie abstrakt und implementierungsunabhängig sind. Die erste Alternative hat zusätzlich den Vorteil, daß Spezifikationen übersichtlich und einfach abzuleiten sind. Allerdings müssen diese später während der Verifikation ergänzt werden [TaKu93b]. Im zweiten Fall sind Spezifikationen etwas komplizierter, aber vollständiger und ersparen eine nachträgliche Ergänzung der Beweisziele während der Verifikation. Die Einbeziehung der Abstraktionsfunktion in die abstrakte Spezifikation ist jedoch etwas ungewöhnlich [Melh88], da sie eine Abbildungsfunktion darstellt, die normalerweise erst beim formalen Beweis die abstrakten und konkreten Beschreibungen in Beziehung bringen soll. Aus diesem Grund und im Hinblick auf eine automatische Ableitung der formalen Spezifikation aus den informalen Beschreibungen wird in dieser Arbeit die erste Alternative weiterverfolgt.

Zwischen der Stufen- und Phasenebene existiert eine ähnliche temporale Beziehung, da Zustandsänderungen der Komponenten der Stufenebene an einigen spezifischen Phasen des Takts stattfinden können, die zu Zeitpunkten innerhalb des Taktzyklusintervalls gehören. In analoger Weise wird daher zwischen der Stufen- und Phasenebene eine ähnliche Zeitabstraktionsfunktion benötigt, die entsprechend der Phasenimplementierung der Stufenbefehle zusätzlich zu den Zeitvariablen Kontextparameter berücksichtigt. Ein Kontextparameter wird als Tupel definiert, das aus einer Lese/Schreibe-Information und einem Taktphasenbezeichner besteht. Beispielsweise bezeichnen die Tupel $[read, \phi_1]$ und $[write, \phi_2]$ das Lesen bzw. Schreiben während der Phase 1 bzw. 2. Zur Spezifikation der Stufenbefehle bieten sich hier auch die oben angegebenen Spezifikationsalternativen an: ohne [TaKu93b] oder unter Einbeziehung [TaKu93a] der Zeitabstraktionsfunktion in die Spezifikation. Aus den oben genannten Gründen wird in der vorliegenden Arbeit die erste Alternative bevorzugt.

---

[6] Da *rs1*, *rs2* und *rd* Felder eines einzigen Registers (Befehlsregister) darstellen, wurde in Gleichung (6-4) ein gemeinsamer Kontextparameter C_2 zugewiesen.

6.2.2 Temporale Abstraktionsfunktion

Wegen der Ähnlichkeit der Abstraktionsfunktion auf beiden Ebenen läßt sich eine allgemeine, parametrisierbare Zeitabstraktionsfunktion definieren. Eine solche Funktion soll, in Abhängigkeit von einem gegebenen Kontextparameter, für jede beliebige abstrakte Zeitvariable den konkreten Zeitpunkt berechnen. Beispielsweise ergibt eine Zeitabstraktionsfunktion f_z für ein Lesen bzw. ein Schreiben während der ID- bzw. WB-Stufe: $f_z([read, ID], u) = t+1$ bzw. $f_z([write, WB], u+1) = t+5$. Mit der Annahme, daß die Pipelinestufen- bzw. Taktphasenbezeichner in einer bestimmten Weise geordnet sind (z.B. $IF = 0$, $ID = 1$, usw.), wird eine Ordnungsfunktion (bezeichnet *ORD*) eingeführt, die jeder Pipelinestufe bzw. Taktphase einen eindeutigen Ordnungswert zuordnet (z.B. $ORD\ (ID) = 1$). Darüberhinaus soll eine Definition der Zeitabstraktionsfunktion voraussetzen, daß die Zeitvariablen ausgerichtet sind, d.h. gelesene und geschriebene Werte am Anfang bzw. am Ende eines diskreten Zeitintervalls betrachtet werden. Eine Zeitabstraktionsfunktion *Time_abs*, die sowohl abstrakte Zeitvariable der Architektur- als auch der Stufenbefehle auflöst, kann demzufolge wie folgt definiert werden, wobei die Parameter n, C und t der implementierten Pipelinetiefe bzw. der Anzahl der Taktphasen, dem Kontexttupel bzw. einer beliebigen Zeitvariablen entsprechen:

$$\begin{array}{ll} \vdash_{def}\ Time\_abs\ n\ \ C\ \ t := & \\ \quad \text{let}\ (rw = C[0] \wedge Id = C[1])\ \text{in} & \text{(Gl. 6-5)} \\ \quad\quad (n \bullet (t - rw) + ORD\ (Id) + rw) & \end{array}$$

Demnach wird bei gelesenen Werten ($rw = 0$) einfach die Zeitvariable mit der Pipelinetiefe bzw. mit der Anzahl der Phasen n multipliziert und mit dem entsprechenden Ordnungswert des Stufen- bzw. Phasenbezeichners Id addiert (z.B. mit $n_s = 5$ und $ORD\ (ID) = 1$ ist $Time\_abs\ (n_s, [read, ID], u) = 5 \bullet (u\text{-}0) + 1 + 0 = 5u + 1 = t+1$). Bei geschriebenen Werten ($rw = 1$) wird die Zeitvariable zusätzlich dekrementiert, um die alte Ausrichtung aufzuheben und anschließend eine 1 addiert, um den berechneten Wert neu auszurichten (z.B. mit $n_s = 5$ und $ORD\ (WB) = 4$ ist $Time\_abs\ (n_s, [write, WB], u+1) = 5 \bullet ((u+1) - 1) + 4 + 1 = 5u+5 = t+5$).

Durch die vorgenommene Parametrisierung hat diese Definition der Zeitabstraktionsfunktion den Vorteil, daß sie abstrakte und implementierungsunabhängige Spezifikationen erlaubt und außerdem sowohl für die Architektur- und Klassenebene als auch für die Stufenebene gilt. Der Parameter n wird für eine gegebene Ebene einmalig instanziiert. Der Kontextparameter C wird hingegen während der formalen Verifikation in den Beweiszielen für jede Zustandskomponente entsprechend gesetzt (s. Kapitel 7.3).

6.3 Spezifikation der Modellebenen

Die Architektur-, Klassen-, Stufen- und Phasenebenen des RISC-Modells stellen Interpreterebenen dar, deren Spezifikationen aus der formalen Beschreibung der Befehlssemantiken jeder Ebene bestehen. Die Semantik eines Befehls einer Interpreterebene wird durch eine entsprechende Zustandsübergangsfunktion wiedergegeben, die in Logik höherer Ordnung vereinbarungsgemäß durch ein Prädikat repräsentiert wird (s. Kapitel 5.3). Die Parameter der Spezifikationsprädikate entsprechen der sichtbaren Struktur der zugehörigen Ebene. Das EBM ist kein Interpreter, da es die Struktur der Hardware-Implementierung darstellt. Es wird entsprechend Kapitel 3.2.2 als Hierarchie von Prädikaten spezifiziert.

6.3.1 Architekturebene

Im Handbuch eines Prozessors wird die Semantik eines Maschinenbefehls in Form einer Zuweisung spezifiziert, wobei die zeitliche Information implizit durch den Befehlszyklus gegeben ist. Zum Beispiel werden für den DLX-Prozessor der ADD-Befehl (s. Gleichung (6-1)) und der BRANCH-Befehl, der eine bedingte Verzweigung beschreibt, wie folgt definiert:

$$ADD := RF\,[rd] \leftarrow RF\,[rs1] + RF\,[rs2] \qquad \text{(Gl. 6-6)}$$

$$\begin{aligned} BRANCH := \text{if } (RF\,[rs1] = 0) \ \ & \text{then } PC \leftarrow PC + offset16 \\ & \text{else } PC \leftarrow PC + 4 \qquad \text{(Gl. 6-7)} \end{aligned}$$

wobei *offset16* einem Feld des Befehlswortes entspricht. Gleichung (6-7) gibt an, daß dann, wenn die Registersatzstelle $RF\,[rs1]$ den Wert Null hat, dem Befehlszähler *PC* die Sprungadresse ($PC + offset16$) zugewiesen wird, ansonsten wird der *PC* einfach inkrementiert.

Gemäß der Diskussion in Kapitel 6.2 werden die ADD- und BRANCH-Befehle anhand von Befehlszykluseinheiten durch folgende Prädikate (entsprechend Gleichung (6-2)) formal spezifiziert:

$$\begin{aligned} &\vdash_{def} ADD\_SPEC\,(PC, RF, I\text{-}MEM, D\text{-}MEM) := \\ &\quad \forall\, u{:}\ Instr\_cycle. \\ &\qquad \text{let} \quad (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge \qquad \text{(Gl. 6-8)} \\ &\qquad\qquad rs2 = [I\text{-}MEM(PC)]_{20..16} \wedge \\ &\qquad\qquad rd = [I\text{-}MEM(PC)]_{15..11}) \text{ in} \\ &\qquad RF(u{+}1)\,[rd(u)] = RF(u)\,[rs1(u)] + RF(u)\,[rs2(u)] \end{aligned}$$

$$
\begin{array}{l}
\vdash_{def} \mathit{BRANCH\_SPEC}\ (\mathit{PC}, \mathit{RF}, \mathit{I\text{-}MEM}, \mathit{D\text{-}MEM}) := \\
\quad \forall\, u{:}\ \mathit{Instr\_cycle}. \\
\qquad \text{let}\ (\mathit{rs1} = [\mathit{I\text{-}MEM}(\mathit{PC})]_{25..21} \wedge \\
\qquad\quad \mathit{offset16} = [\mathit{I\text{-}MEM}(\mathit{PC})]_{15..0})\ \text{in} \\
\qquad\quad \mathit{PC}(u{+}1) = (\mathit{RF}(u)\ [\mathit{rs1}(u)] = 0) \quad \rightarrow \mathit{PC}(u) + \mathit{offset16}(u) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{PC}(u) + 4
\end{array}
\tag{Gl. 6-9}
$$

6.3.2 Klassenebene

Befehle der Klassenebene abstrahieren die Semantik der Maschinenbefehle durch entsprechende Klassenabstraktionsfunktionen, die spezifische Operatoren oder Verhaltensfunktionen repräsentieren. Zum Beispiel wird die ALU-Klasse, entsprechend einer Befehlsbeschreibung in einem Handbuch, durch Verwendung der Abstraktionsfunktion f_A, die für arithmetisch-logische Operationen gilt, wie folgt beschrieben:

$$\mathit{ALU} := \mathit{RF}\ [\mathit{rd}] \leftarrow f_A\ (\mathit{RF}\ [\mathit{rs1}],\ \mathit{RF}\ [\mathit{rs2}]) \tag{Gl. 6-10}$$

Ein Befehl der CONTROL-Klasse kann analog dazu durch die Abstraktionsfunktionen f_C, die alle Varianten der Kontrollbefehle (Sprünge, Verzweigungen, usw.) umfaßt, wie folgt angegeben werden:

$$\mathit{CONTROL} := \mathit{PC} \leftarrow f_C\ (\mathit{PC}, \mathit{offset16}, \mathit{offset26}, \mathit{RF}[\mathit{rs1}]) \tag{Gl. 6-11}$$

wobei *offset16* und *offset26* Felder des Befehlsregisters darstellen. Durch die Funktion f_C können alle auftretenden Sprungadressen anhand der Parameter *PC*, *offset16*, *offset26* und $\mathit{RF}[\mathit{rs1}]$ berechnet werden (z.B. liefert f_C für relative und Register-indirekte Sprünge $\mathit{PC}{+}\mathit{offset16}$ bzw. $\mathit{PC}{+}\mathit{RF}[\mathit{rs1}]$)

Wegen der Verwendung derselben Struktur und Zeitgranularität ähnelt eine formale Spezifikation der Klassenbefehle der der Maschinenbefehle. Jeder Klassenbefehl wird formal durch ein Prädikat spezifiziert, dessen Parameter denen der Architekturprädikate entsprechen. Zusätzlich dazu werden hier die Klassenabstraktionsfunktionen als Parameter der Spezifikationsprädikate eingeführt. Demnach werden die obigen ALU- und CONTROL-Befehle entsprechend Gleichungen (6-10) bzw. (6-11) wie folgt spezifiziert:

$$
\begin{array}{l}
\vdash_{def} \mathit{ALU\_SPEC}\ f_A\ (\mathit{PC}, \mathit{I\text{-}MEM}, \mathit{RF}, \mathit{D\text{-}MEM}) := \\
\quad \forall\, u{:}\ \mathit{Instr\_cycle}. \\
\qquad \text{let}\ (\mathit{rs1} = [\mathit{I\text{-}MEM}(\mathit{PC})]_{25..21} \wedge \\
\qquad\quad \mathit{rs2} = [\mathit{I\text{-}MEM}(\mathit{PC})]_{20..16} \wedge \\
\qquad\quad \mathit{rd} = [\mathit{I\text{-}MEM}(\mathit{PC})]_{15..11})\ \text{in} \\
\qquad \mathit{RF}(u{+}1)\ [\mathit{rd}(u)] = f_A\ (\mathit{RF}(u)\ [\mathit{rs1}(u)], \mathit{RF}(u)\ [\mathit{rs2}(u)])
\end{array}
\tag{Gl. 6-12}
$$

$$
\begin{array}{l}
\vdash_{def}\ CONTROL\_SPEC\ f_C\,(PC, I\text{-}MEM, RF, D\text{-}MEM) := \\
\quad \forall\, u\colon Instr\_cycle. \\
\qquad \text{let}\ (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge \\
\qquad\quad offset16 = [I\text{-}MEM(PC)]_{15..0} \wedge \\
\qquad\quad offset26 = [I\text{-}MEM(PC)]_{25..0})\ \text{in} \\
\qquad PC(u{+}1) = f_C\,(PC(u), offset16(u), offset26(u), RF(u)\,[rs1(u)])
\end{array}
\qquad \text{(Gl. 6-13)}
$$

6.3.3 Pipelinestufenebene

Einzelne Stufenbefehle werden mit der Zeitgranularität der Taktzyklen anhand der sichtbaren Zustandskomponenten der Stufenebene durch Prädikate spezifiziert. Die Spezifikationsprädikate bestehen aus Konjunktionen von Zustandsübergängen elementarer Stufenoperationen, welche die Semantik des entsprechenden Stufenbefehls implementieren (s. Pipelinestruktur in Tabelle 4-1 bzw. 5-1). Die einzelnen Zustandsübergänge können einfach aus der Pipelinestruktur abgelesen und formalisiert werden. Beispielsweise wird der ID-Stufenbefehl der ALU-Klasse ID_A aus der ID-Zeile und der Klassenspalte ALU direkt abgeleitet und wie folgt spezifiziert:

$$
\begin{array}{l}
\vdash_{def}\ ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) := \\
\quad \forall\, t\colon Clock\_cycle. \\
\qquad \text{let}\ (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{20..16})\ \text{in} \\
\qquad\quad A(t{+}1) = RF(t)\,[rs1(t)] \wedge \\
\qquad\quad B(t{+}1) = RF(t)\,[rs2(t)] \wedge \\
\qquad\quad IR1(t{+}1) = IR(t)
\end{array}
\qquad \text{(Gl. 6-14)}
$$

Für Stufenbefehle, deren Operationen mit einer Klassenabstraktionsfunktion behaftet sind, wird, analog zu den Klassenbefehlen, zusätzlich die Klassenfunktion als Parameter des Prädikates eingeführt. Beispielsweise wird der ID-Stufenbefehl der CONTROL-Klasse ID_C durch das Prädikat ID_C_SPEC wie folgt formal beschrieben:

$$
\begin{array}{l}
\vdash_{def}\ ID_C\_SPEC\ f_C\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) := \\
\quad \forall\, t\colon Clock\_cycle. \\
\qquad \text{let}\ (rs1 = [IR]_{25..21} \wedge \\
\qquad\quad offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0})\ \text{in} \\
\qquad PC(t{+}1) = f_C\,(PC(t), offset16(t), offset26(t), RF(t)\,[rs1(t)])
\end{array}
\qquad \text{(Gl. 6-15)}
$$

6.3.4 Taktphasenebene

Ähnlich wie in der Stufenebene werden Phasenbefehle formal durch Prädikate spezifiziert, deren Parameter den sichtbaren Strukturkomponenten der Phasenebene entsprechen. Diese Spezifikationsprädikate werden mit der Zeitgranularität der Taktphasen beschrieben und bestehen aus Konjunktionen von elementaren Zustandsübergängen der Phasenoperationen. Die Phasenoperationen können ebenfalls direkt aus der Pipelinestruktur abgeleitet und formalisiert werden. Für Phasenoperationen, die in der Pipelinestruktur nicht explizit angegeben sind, wie z.B. zwischen den IR- und IR1-Registern in der ID-Stufe (s. Tabelle 5-1), werden die Zustandswerte in den entsprechenden Phasen unverändert gehalten (s. das Prädikat $\phi_1 ID_A\_SPEC$ unten). Als Verfeinerung der oben beschriebenen Spezifikationen des ID_A-Stufenbefehls können die zwei korrespondierenden ID-Phasenbefehle aus der ID-Reihe und der ALU-Spalte der Pipelinestruktur des DLX-Prozessors in Tabelle 5-1 einfach durch folgende Prädikate spezifiziert werden:

$$
\begin{aligned}
&\vdash_{def}\ \phi_1 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=\\
&\quad \forall\,\tau{:}Clock\_phase.\\
&\quad\ IR(\tau{+}1) = IR(\tau)
\end{aligned}
\qquad \text{(Gl. 6-16)}
$$

$$
\begin{aligned}
&\vdash_{def}\ \phi_2 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=\\
&\quad \forall\,\tau{:}Clock\_phase.\\
&\quad\ \text{let}\ (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{20..16})\ \text{in}\\
&\qquad A(\tau{+}1) = RF(\tau)\,[rs1(\tau)] \wedge\\
&\qquad B(\tau{+}1) = RF(\tau)\,[rs2(\tau) \wedge\\
&\qquad IR1(\tau{+}1) = IR(\tau)
\end{aligned}
\qquad \text{(Gl. 6-17)}
$$

Analog dazu wird die Phasenimplementierung des oben angegebenen ID_C-Stufenbefehls wie folgt spezifiziert, wobei im Gegensatz zum vorigen Beispiel, die Stufenoperation (gekennzeichnet durch die Anwendung der Abstraktionsfunktion f_C) in der ersten Phase geschieht und das Resultat im Puffer BTA zwischengespeichert wird:

$$\begin{aligned}
&\vdash_{def}\ \phi_1 ID_C\_SPEC\ f_C\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=\\
&\quad \forall\, \tau{:}Clock\_phase.\\
&\qquad \text{let}\ (rs1 = [IR]_{25..21} \wedge\\
&\qquad\quad offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}\,)\ \text{in}\\
&\qquad\quad BTA(\tau+1) = f_C\,(PC(\tau), offset16(\tau), offset26(\tau), RF(\tau)\ [rs1(\tau)])
\end{aligned} \tag{Gl. 6-18}$$

$$\begin{aligned}
&\vdash_{def}\ \phi_2 ID_C\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=\\
&\quad \forall\, \tau{:}Clock\_phase.\\
&\qquad\quad PC(\tau+1) = BTA(\tau)
\end{aligned} \tag{Gl. 6-19}$$

6.3.5 Elektronisches Blockmodell (EBM)

Diese Spezifikationsebene unterscheidet sich von den vorherigen abstrakten Ebenen des Interpretermodells, da das EBM keinen Interpreter, sondern nur die Struktur der Hardware-Implementierung auf der RT-Ebene darstellt, wobei die Zustände implizit durch die Zustände der verschiedenen Speicherelemente charakterisiert werden. Die sichtbaren Zustandskomponenten sowie die verwendete Zeitgranularität des EBM entsprechen denen der Phasenebene. Zusätzlich zu den Zustandskomponenten werden hier Umgebungssignale des Prozessors in die Spezifikationsparameter einbezogen (wie z.B. Takt-, Unterbrechungs- oder Bussteuersignale (s. unten)).

Das EBM setzt sich im allgemeinen aus einer Hierarchie von Hardware-Einheiten zusammen, die wiederum aus weiteren Hierarchien von Hardware-Komponenten bestehen. Auf der höchsten Hierarchieebene besteht das EBM aus dem RISC-Prozessorkern und den angeschlossenen Befehls- und Datenspeichern bzw. Caches. Der Prozessor selbst wird konventionellerweise in Daten- und Kontrollpfad unterteilt, die ihrerseits aus Hierarchien von einfacheren Verarbeitungs- und Speichereinheiten wie z.B. Registersatz, ALU-Einheit, Dekodierer, usw. bestehen. Diese können auf niedrigerer Ebene aus weiteren primitiven Blöcken wie z.B. Registern, Multiplexern oder PLAs zusammengesetzt sein.

Bild 6-5 zeigt die vereinfachte Form des EBM für den DLX-Prozessor [DeTa94]. Das EBM besteht aus dem Prozessor selbst und den angeschlossenen Befehls- und Datenspeichern. Die Daten- und Kontrollpfade des Prozessors implementieren eine Pipelineverarbeitung und setzen sich demnach physikalisch entsprechend der Pipelinestufen aus einer Reihe von Verarbeitungseinheiten und Puffern zusammen (zur besseren Veranschaulichung sind die verschiedenen Ver-

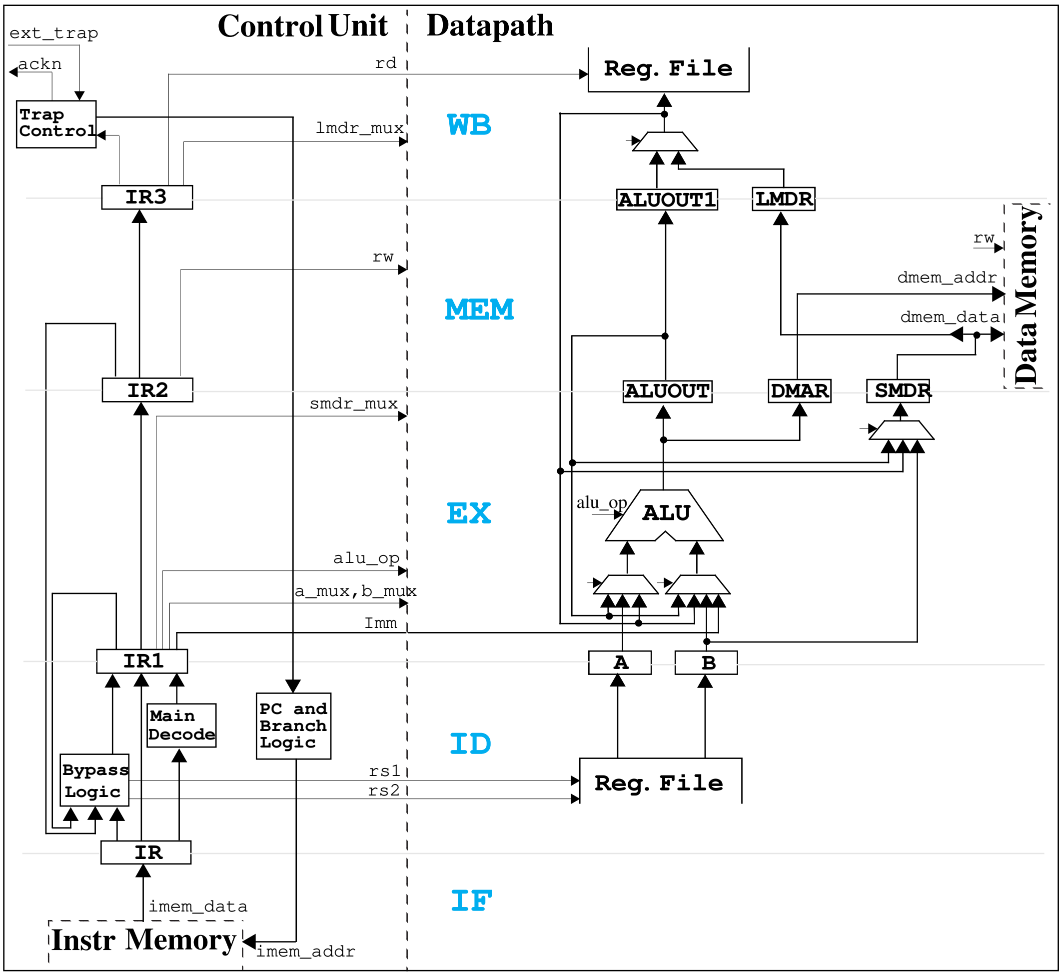


Bild 6-5: Elektronisches Blockmodell des DLX (vereinfacht)

arbeitungsteile in Bild 6-5 nach der Anordnung der zugehörigen Pipelinestufen gezeichnet). In der IF-Stufe wird auf den Befehlsspeicher zugegriffen, in der ID-Stufe werden Befehle dekodiert, Sprungadressen berechnet und sämtliche Steuerungen für die Bypasspfade erzeugt, in der EX-Stufe wird die ALU-Einheit aktiviert, in der MEM-Stufe wird eventuell auf den Datenspeicher zugegriffen und in der WB-Stufe werden Rechenergebnisse im Registersatz abgelegt und sämtliche Unterbrechungen bearbeitet.

Formal wird das EBM hierarchisch als Konjunktion von Prädikaten spezifiziert, die sich aus Konjunktionen von weiteren Prädikaten zusammensetzen. Eine detaillierte Beschreibung des EBM bis auf Gatterebene ist [Deho94] zu entnehmen. Die formale Implementierung des EBM auf der höchsten Hierarchieebene sieht jedoch wie folgt aus, wobei zur Veranschaulichung der Prozessorkern zusätzlich in Daten- und Kontrollpfad aufgeteilt ist:

$\vdash_{def}$ ***EBM*** (*PC, I-MEM, RF, D-MEM, A, B, ALUOUT, ALUOUT1, DMAR, SMDR,*
LMDR, IR, IR1, IR2, IR3, BTA, IAR, ext_trap, ackn, clk1, clk2) :=

$\exists$ *rs1, rs2, rd, Imm, a_mux, b_mux, alu_op, smdr_mux,*
lmdr_mux, imem_addr, imem_data, dmem_addr, dmem_data, rw.

DataPath (*RF, A, B, ALUOUT, ALUOUT1, DMAR, SMDR, LMDR,*
dmem_addr, dmem_data, rs1, rs2, rd, Imm, a_mux,
b_mux, alu_op, smdr_mux, lmdr_mux, clk1, clk2) $\wedge$

Control_Unit (*PC, IR, IR1, IR2, IR3, BTA, IAR, ext_trap, ackn, rw,*
imem_addr, imem_data, rs1, rs2, rd, Imm, a_mux,
b_mux, alu_op, smdr_mux, lmdr_mux, clk1, clk2) $\wedge$

Instr_Memory (*I-MEM, imem_addr, imem_data, clk2*) $\wedge$

Data_Memory (*D-MEM, dmem_addr, dmem_data, rw, clk2*)

In anderen Arbeiten, in denen ein Interpretermodell zur Mikroprozessorverifikation verwendet wurde [Joyc89, Wind90a, PanJ92], wird das EBM bezüglich abstrakter Unterblöcke beschrieben, deren korrekte Implementierung auf niedrigerer Ebene angenommen wird. Der in dieser Arbeit verfolgte Ansatz basiert auf Beschreibungen der Hardware auf Gatterebene, die aus Netzlisten eines CAD-Werkzeugs automatisch abgeleitet werden können. Die Unterblöcke des hierarchischen Entwurfs werden bis auf die elementaren Bibliothekszellen aufgelöst und formalisiert.

6.3.6 Ableitung der formalen Spezifikationen

Die Spezifikationsprädikate aller Ebenen haben eine einheitliche Form, die durch die Verwendung der Strukturkomponenten als Parameter und eine Quantifizierung der Zeitvariablen gekennzeichnet ist. Die Spezifikationsprädikate der Maschinenbefehle haben den gleichen Aufbau und können leicht aus einer Handbuchbeschreibung abgeleitet werden, wobei Befehlszykluseinheiten zugrundegelegt werden. Nach der Definition der verschiedenen Klassenabstraktionsfunktionen können die Prädikate der Klassenbefehle aus der Spezifikation der Maschinenbefehle abgeleitet werden, da diese dieselbe Struktur und Zeitgranularität verwenden. Die Spezifikationsprädikate der Stufen- bzw. Phasenbefehle werden direkt aus der Pipelinestruktur abgelesen und durch Konjunktion der einzelnen Stufen- bzw. Phasenoperationen formalisiert. Die zeitliche Basiseinheit der elementaren Operationen ist einheitlich der Taktzyklus bzw. die Taktphase.

Unter Verwendung eines Übersetzers können die Spezifikationsprädikate der Architektur, Stufen- und Phasenbefehle automatisch aus deren Beschreibungen in Form von Registerzuweisungen in dem gegebenen Befehlsatz bzw. der gegebenen Pipelinestruktur abgeleitet werden.

Im Gegensatz zu der Ableitung der Spezifikationsprädikate der abstrakten Ebenen aus informalen Beschreibungen ist die Implementierung EBM einfacher zu handhaben. Sie kann automatisch aus maschinenlesbaren Darstellungen in Form von Netzlisten (z.B. in EDIF oder HDL-Beschreibungen, wie z.B. VHDL) als Konjunktion von Prädikaten mit freien Variablen für Ein- und Ausgabesignale und existenzquantifizierten Variablen für interne Leitungen (s. Kapitel 3.2.2) generiert werden. Die vorhandenen Hierarchien des Schaltungsentwurfs werden dabei durch korrespondierende Hierarchien von Prädikaten realisiert [KrKS93].

6.3.7 Spezifikation der Hardware-Ereignisse

Zu einem RISC-Prozessor gehört eine Reihe von Hardware-Ereignissen, wie Hardware-Unterbrechungen oder “Pipeline-Stalls”, die formal spezifiziert und verifiziert werden müssen. Diese lassen sich jedoch nicht allgemein behandeln, da sowohl das Verhalten als auch die Implementierung stark prozessorabhängig sind. Zum Beispiel werden Unterbrechungen von verschiedenen Prozessoren auf unterschiedliche Art und Weise implementiert. Sie variieren darin, wo der bedingte Sprung zur Unterbrechungsbehandlungsroutine eingeführt wird, wo und wie die Pipeline gestoppt und wieder gestartet wird, usw. Ein anderes Beispiel sind die Pipeline-Stalls, welche dazu verwendet werden, um die Pipeline unter gewissen Umständen für eine bestimmte Anzahl von Taktzyklen anzuhalten. Auch sie werden ebenfalls unterschiedlich implementiert [HePa90].

Im allgemeinen wird das Verhalten eines spezifischen Hardware-Ereignisses durch ein Prädikat spezifiziert, dessen Korrektheit aus der Implementierung EBM impliziert werden kann. Sei *INTR_SPEC* ein Spezifikationsprädikat, welches das abstrakte Verhalten von Hardware-Unterbrechungen aus externer Sicht beschreibt [Deho94]. Dieses Prädikat beinhaltet bestimmte Eigenschaften wie z.B. die Art und Weise, wie der Prozessorzustand vor bzw. nach der Unterbrechungsbehandlung gerettet bzw. wiederhergestellt wird. Die Korrektheit des Unterbrechungsverhaltens wird durch den Beweis der Implikation des Spezifikationsprädikates aus der Implementierung EBM bewiesen:

$$\vdash\ EBM \Rightarrow\ INTR\_SPEC$$

6.4 Verwendung der Formalisierung des RISC-Modells

Im allgemeinen können formale Beschreibungen in Hardware-Systemen zur Spezifikation, Verifikation, Simulation oder auch in einer automatischen Synthese verwendet werden [GFAM89]. Demnach kann die Formalisierung des hierarchischen RISC-Modells in Logik höherer Ordnung außer für Verifikationszwecke auch direkt in der Simulation verwendet werden, jedoch sind Spezifikationen in Logik höherer Ordnung bekanntlich ohne weiteres nicht ausführbar [Joyc88b]. Camilleri [Cami88] beschreibt dennoch wie aus einer Hardware-Spezifikation in Logik höherer Ordnung eine Simulation generiert werden kann. Dadurch wird es ermöglicht, daß Spezifikationen in dieser Logik ausgeführt werden können, bevor man sie gegenüber einer Implementierung formal verifiziert. Die formalen Spezifikationen können auch als Eingabe für andere spezielle Werkzeuge, wie z.B. für die formale Synthese [FoMa89, HaLD89], verwendet werden. Dabei wird die Logik höherer Ordnung als universelle Hardware-Beschreibungssprache angesehen. Dieser Ansatz der formalen Synthese oder des sogenannten “verifikationsbasierten Entwurfs” (“verification-driven design”) [BJLM86, Miln90] sieht vor, daß die formale Verifikation mit dem Entwurf gekoppelt wird. In dieser Hinsicht kann das formalisierte Modell also auch im Entwurfsprozeß eines RISC-Prozessors verwendet werden.

Kapitel 7: RISC-Verifikationsverfahren

Im letzten Kapitel wurde die Formalisierung des hierarchischen RISC-Modells beschrieben, welche für verschiedene Zwecke verwendet werden kann, z.B. für die Simulation, Verifikation oder Synthese. Speziell für die formale Verifikation von RISC-Prozessoren wirkt sich günstig aus, daß ein allgemeines Schema angegeben werden kann, mit dem die Durchführung der Korrektheitsbeweise hierarchisch in einfachen und einheitlichen Schritten gegliedert wird. In diesem Kapitel werden zunächst die Verifikationsaufgaben abgegrenzt und dann die einzelnen Verifikationsschritte in entsprechenden Abschnitten genauer dargestellt.

7.1 Die Verifikationsaufgaben

Das Ziel der formalen Verifikation eines Mikroprozessors besteht im allgemeinen darin, zu zeigen, daß die formale Beschreibung der Hardware-Implementierung die formale Spezifikation des Verhaltens, gegeben durch die Semantik der verschiedenen Maschinenbefehle, erfüllt. Ein RISC-Prozessor zeichnet sich durch die Pipelineverarbeitung der Befehle aus, so daß dessen Verhalten zusätzlich zu der Semantik des Befehlssatzes durch die Pipelinearchitektur bestimmt ist.

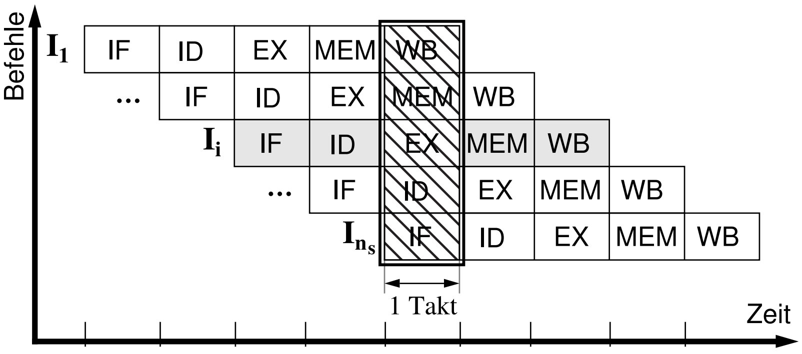


Bild 7-1: Befehlsausführung in einer fünfstufigen Pipeline

Sei n_s die Pipelinetiefe eines gegebenen RISC-Prozessors, so kann der Prozessor potentiell n_s Befehle in n_s unterschiedlichen Pipelinestufen parallel ausführen (s. Bild 7-1). Wegen der Verwendung von gemeinsamen Ressourcen und der Existenz von Daten- und Steuerabhängigkeiten zwischen den Befehlen in einer Pipeline können sich diese n_s Stufen (siehe schraffiertes vertikales Kästchen in Bild 7-1) gegenseitig stören, so daß die Ausführung eines einzelnen Befehls (siehe schattiertes horizontales Kästchen für Befehl I_i in Bild 7-1) gestört werden kann.

7.1.1 Fundamentale RISC-Korrektheitssätze

Die formale Verifikation eines RISC-Prozessors besteht deshalb aus dem Beweis, daß einerseits die Semantik des Befehlssatzes, andererseits der Arbeitsablauf in der Pipeline (“pipelined sequencing”) der Maschinenbefehle, korrekt implementiert sind. Demnach wird der Korrektheitsbeweis in zwei unabhängige Schritte wie folgt aufgeteilt:

1. *Korrektheit der Pipeline*: Gegeben seien einige Software-Randbedingungen (Software-Constraints) über die aktuelle Architektur und die Hardware-Implementierung EBM, aus denen abgeleitet wird, daß jede beliebige Sequenz von Maschinenbefehlen in der Pipeline korrekt abgearbeitet wird:

$$\boxed{\mathit{SW\_Constraints, EBM} \ \vdash \mathit{Correct\_Instr\_Pipelining} \qquad \text{(Gl. 7-1)}}$$

2. *Semantische Korrektheit*: Die Semantik jedes einzelnen Maschinenbefehls wird von der Hardware-Realisierung EBM korrekt implementiert:

$$\boxed{\vdash \mathit{EBM} \Rightarrow \mathit{Architekturebene} \qquad \text{(Gl. 7-2)}}$$

Für die Korrektheit des RISC-Prozessors ist jeder dieser Korrektheitssätze notwendig, aber nicht ausreichend. Sie können jedoch unabhängig voneinander formal behandelt werden.

Der erste Schritt besteht hauptsächlich in der formalen Verifikation der Nichterfüllung der verschiedenen Pipelinekonflikte, die zu einer Störung des Arbeitsablaufs in der Pipeline führen können. Spezifikation und Verifikation der Pipelinekonflikte werden an das hierarchische Modell angelehnt, indem sie auf verschiedenen Hierarchieebenen betrachtet werden. Die Software-Constraints — *SW_Constraints* — in Gleichung (7-1) entsprechen denjenigen Bedingungen, die bei einem Software-Entwurf zu erfüllen sind, um Konflikte zu vermeiden. Sie betreffen beispielsweise die Anzahl der Verzögerungstakte (“delay slots”), die bei der Verwendung der Technik des Software-Scheduling zwischen den Befehlen eingeführt werden müssen, um sie konfliktfrei zu machen. Darüberhinaus wird auch angenommen, daß das EBM einige Mechanismen zur Konfliktbeseitigung in Hardware vorsieht wie z.B. das Register-Bypassing, welches bewirkt, daß berechnete Daten für nachfolgende Befehle frühzeitig bereitgestellt werden.

Beim zweiten Verifikationsschritt wird, basierend auf dem hierarchischen RISC-Modell, die Verifikation der Architekturebene schrittweise hierarchisch durchgeführt. Hierbei wird zunächst die korrekte Implementierung der Semantik einzelner Maschinenbefehle durch die sequentielle Ausführung entsprechender

Stufenbefehle gezeigt. Dann wird jede dieser Teilausführungen auf eine korrekte Implementierung durch die Phasenebene geprüft. Anschließend wird gezeigt, daß die Hardware-Implementierung die Semantik einzelner Phasenbefehle impliziert.

7.1.2 Verschiedene Verifikationsansätze

Das hierarchische RISC-Modell wird dazu verwendet, um sowohl die semantische als auch die Pipeline-Korrektheit zu verifizieren. Unter Verwendung dieses Modells können verschiedene Verifikationsansätze eingesetzt werden, indem der Verifikationsprozeß hierarchisch Top-Down oder Bottom-Up derart verwaltet wird, daß ein *verifikationsbasierter Entwurf* oder eine *nachträgliche Verifikation* ("post-design verification") durchgeführt werden kann. Bei einem Top-Down verifikationsbasierten Entwurf werden die Verifikations- und die Entwurfsprozesse gekoppelt in dem Sinne, daß die Verifikation eines aktuellen Entwurfsschritts die notwendigen Randbedingungen (Constraints) für den nächsten Entwurfsschritt liefert. Eine nachträgliche Verifikation entspricht dagegen der Verifikation im konventionellen Sinne, indem Top-Down oder Bottom-Up der komplette fertige Entwurf auf allen Ebenen auf Korrektheit geprüft wird. In dieser Arbeit werden Korrektheitsbeweise hauptsächlich durch eine Top-Down verifikationsbasierte Entwurfsmethode beschrieben, jedoch lassen sich die entwickelten Verifikationsverfahren auch für den anderen Verifikationsansatz anwenden.

7.2 Korrektheit der Pipelineverarbeitung

7.2.1 Einleitung

Der Nachweis der Korrektheit der Pipelineverarbeitung besteht aus dem Beweis, daß alle möglichen Kombinationen von n_s Maschinenbefehlen in der Pipeline korrekt ausgeführt werden. Eine korrekte Pipelineausführung heißt, daß die betreffenden Befehle sich gegenseitig nicht stören, d. h., daß keine Pipelinekonflikte zwischen ihnen entstehen. Die Verifikation des Korrektheitssatzes (1) ist demnach äquivalent zum Beweis, daß die parallele Ausführung der Befehle zu keinem Zeitpunkt zu Konflikten führt. Man unterscheidet drei Klassen von *Konflikten* (auch *Hazards* [HePa90] oder *Hemmnisse* [Bode90] genannt), die während der Pipelineausführung einer RISC-Maschine vorkommen können, nämlich *Ressourcenkonflikte*, *Datenkonflikte* und *Kontrollkonflikte*[7]. Da die Korrektheit der Pipeline eine direkte Folgerung der Abwesenheit dieser Konflikte

---

[7] Eine Definition der verschiedenen Konflikte folgt später in entsprechenden Abschnitten.

ist, definiert der Korrektheitssatz (1) die Nicht-Existenz all dieser drei Arten von Konflikten. Seien *Res_Conflict*, *Data_Conflict* und *Control_Conflict* Spezifikationsprädikate für jeweils Ressourcen-, Daten- und Kontrollkonflikte. Das Prädikat *Correct_Instr_Pipelining* in Gleichung (7-1), welches der Nicht-Erfüllung all dieser Prädikate entspricht, wird durch folgende Konjunktion definiert:

$$\vdash_{def} \mathit{Correct\_Instr\_Pipelining} := \begin{aligned}(\neg\, &\mathit{Res\_Conflict} \wedge \\ \neg\, &\mathit{Data\_Conflict} \wedge \\ \neg\, &\mathit{Control\_Conflict})\end{aligned} \qquad \text{(Gl. 7-3)}$$

und der Korrektheitssatz (7-1) wird durch Termersetzung wie folgt geschrieben:

$$\mathit{SW\_Constraints, EBM} \quad \vdash \quad \begin{aligned}(\neg\, &\mathit{Res\_Conflict} \wedge \\ \neg\, &\mathit{Data\_Conflict} \wedge \\ \neg\, &\mathit{Control\_Conflict})\end{aligned} \qquad \text{(Gl. 7-4)}$$

Der gesamte Korrektheitsbeweis der Pipelineverarbeitung kann nun entsprechend jeder Konfliktart in drei unabhängige Teile aufgespalten werden. Diese Verifikationsteile werden in den nächsten Abschnitten einzeln ausführlich behandelt.

Der Beweis des Prädikates *Correct_Instr_Pipelining* gewährleistet, daß alle möglichen Kombinationen von Maschinenbefehlen, die sich in den n_s Pipelinestufen befinden können, korrekt ausgeführt werden. Nutzt man die Idee der Klassenbefehle wie in Kapitel 5.5.3 beschrieben, wird eine Explosion der Fallunterscheidungen vermieden, indem statt der Kombinationen aller möglichen Maschinenbefehle nur die Kombinationen einer kleinen Anzahl von Klassen betrachtet wird.

Alle Konfliktprädikate sind an das hierarchische Interpreter-Modell angelehnt, indem sie in bezug auf bestimmte sichtbare Zustandskomponenten oder Zeitgranularität einer spezifischen Abstraktionsebene definiert werden. Die obigen Konfliktprädikate *Res_Conflict*, *Data_Conflict* und *Control_Conflict* werden jedoch zunächst auf höherer Ebene bezüglich der Klassenbefehle spezifiziert. Bei der Formalisierung der Konflikte auf der Klassen-, Stufen- und Phasenebene wird die Existenz von Konflikten zwischen mehreren Befehlen durch Prädikate spezifiziert, die *Mehrfachkonfliktprädikate* (“multiple conflict predicates”) genannt werden. Der Beweis solcher Mehrfachkonflikte wird ferner durch die Bildung von Konfliktprädikaten zwischen Paaren von Befehlen vereinfacht, die als *Doppelkonfliktprädikate* (“dual conflict predicates”) bezeichnet werden.

Wie in Kapitel 7.1.2 erwähnt wurde, kann die formale Spezifikationen der gegebenen Konflikte in Abhängigkeit von der vorgenommenen Verifikationsart für unterschiedliche Zwecke verwendet werden:

- *Verifikationsbasierter Entwurf*: In einem bestimmten Stadium des Entwurfsprozesses helfen die Konfliktprädikate einem Entwerfer bei der Bestimmung der Hardware- und Software-Randbedingungen, die zur Konfliktbeseitigung benötigt werden. Die hier entwickelte Methode zur Verifikation der Pipelinekonflikte ist konstruktiv, d.h. wenn ein Konfliktfall entdeckt wird, dann wird dieser extrahiert und explizit ausgegeben. Wenn zum Beispiel ein Entwerfer feststellt, daß das Datenkonfliktprädikat für eine gegebene Ressource erfüllt ist, dann wird automatisch entweder eine entsprechende Software- oder Hardware-Bedingung generiert (s. Kapitel 7.2.4.2).
- *Nachträgliche Verifikation*: Die Konfliktprädikate helfen dem Entwerfer bei der Validierung von existierenden Software- oder Hardware-Randbedingungen, die zur Konfliktbeseitigung vorgegeben sind. Diese Randbedingungen werden auf deren Eignung hin überprüft, indem sie im Annahmeteil des Korrektheitsbeweises entweder als Software-Randbedingungen oder als Teil der Implementierung EBM aufgeführt werden.

7.2.2 Formale Definitionen

In Kapitel 6 wurde eine Formalisierung der Befehle aller Abstraktionsebenen vorgestellt. Zusätzlich zu diesen formalen Definitionen werden in diesem Abschnitt einige neue Typen und Funktionen eingeführt, die zur Behandlung der Konfliktprädikate benötigt werden. Gemäß dem hierarchischen RISC-Modell wird für jede Abstraktionsebene eine Menge von Aufzählungstypen für entsprechende Befehle, Strukturkomponenten (Ressourcen) und Pipelinecharakteristika wie z.B. die Pipelinestufen oder die Taktphasen definiert.

Anhand der Pipelinestruktur in Tabelle 5-1, welche die vier Befehlsklassen ALU, LOAD, STORE und CONTROL, die fünf Pipelinestufen IF, ID, EX, MEM und WB sowie die zwei Taktphasen ϕ_1 und ϕ_2 umfaßt, werden folgende Aufzählungstypen für das Beispiel des DLX-Prozessors definiert:

- Typen für die Pipelinestufen und Phasentakte:

$$pipeline\_stage = IF \mid ID \mid EX \mid MEM \mid WB$$
$$clock\_phase = \phi_1 \mid \phi_2$$

- Typen für die Mengen der Befehle auf den Abstraktionsebenen:

  $class\_instruction = ALU \mid LOAD \mid STORE \mid CONTROL$
  $stage\_instruction = IF_X \mid ID_X \mid ID_C \mid EX_A \mid \ldots \mid MEM_S \mid \ldots \mid WB_L$
  $phase\_instruction = \phi_1 IF_X \mid \ldots \mid \phi_1 EX_A \mid \ldots \mid \phi_1 WB_L \mid \phi_2 IF_X \mid \ldots \mid \phi_2 WB_L$

- Typen für die Strukturkomponenten (Ressourcen) gemäß der Strukturabstraktionen:

  $CL\_resource = PC \mid RF \mid I\text{-}MEM \mid D\text{-}MEM$
  $SL\_resource = PC \mid RF \mid I\text{-}MEM \mid \ldots \mid IR \mid A \mid B \mid ALUOUT \mid DMAR \mid \ldots$
  $PL\_resource = PC \mid RF \mid I\text{-}MEM \mid \ldots \mid IR \mid A \mid B \mid ALUOUT \mid \ldots \mid BTA$

Zur formalen Spezifikation der Konfliktprädikate wird eine Reihe von Funktionen und Prädikaten wie folgt definiert:

- Abstraktionsfunktionen, die entweder Befehle auf höherer Ebene aus Befehlen niedriger Ebene ausrechnen oder Befehle auf niedrigerer Ebene aus Befehlen höherer Ebene extrahieren:

  $ClassToStage:((pipeline\_stage, class\_instruction) \rightarrow stage\_instruction)$
  $StageToClass:(stage\_instruction \rightarrow class\_instruction)$
  $StageToPhase:((clock\_phase, stage\_instruction) \rightarrow phase\_instruction)$
  $PhaseToStage:(phase\_instruction \rightarrow stage\_instruction)$

  Beispiele: $ClassToStage\ (ID, CONTROL) = ID_C$, d.h. aus der Pipelinestufe ID und dem Klassenbefehl $CONTROL$ wird der Stufenbefehl ID_C bestimmt. $PhaseToStage\ (\phi_2 EX_A) = EX_A$, d.h. der Stufenbefehl EX_A wird aus dem Phasenbefehl $\phi_2 EX_A$ abgeleitet

- Funktionen, welche die logischen Pipelinestufen bzw. Taktphasen aus einem Stufen- bzw. Phasenbefehl bestimmen:

  $Stage\_Type:(stage\_instruction \rightarrow pipeline\_stage)$
  $Phase\_Type:(phase\_instruction \rightarrow clock\_phase)$

- Funktionen, welche die Ordnungsnummer einer gegebenen Pipelinestufe bzw. Taktphase berechnen[8]:

  $Stage\_Rank:(pipeline\_stage \rightarrow num)$
  $Phase\_Rank:(clock\_phase \rightarrow num)$

---

[8] Diese Funktionen werden benötigt, um die sequentielle Ordnung der Ausführung von Stufen- bzw. Phasenbefehlen auszudrücken (s. Kapitel 6.2.2).

Beispiele: *Stage_Rank* (*ID*) = 1, d.h. die Pipelinestufe *ID* hat die Ordnungsnummer 1. *Phase_Rank* (ϕ_1) = 0, d.h. die Taktphase ϕ_1 hat die Ordnungsnummer 0.

- Prädikate, die den Wert wahr annehmen, wenn eine gegebene Ressource bei einem Stufen- bzw. Phasenbefehl verwendet wird:

  Stage_Used:((*stage_instruction*, *SL_resource*) → *bool*)
  Phase_Used:((*phase_instruction*, *CL_resource*) → *bool*)

  Beispiele: *Stage_Used* (ID_C, *PC*) = *True*, d.h. die Ressource *PC* wird vom Stufenbefehl ID_C verwendet. Diese Prädikate werden aus den formalen Spezifikationsprädikaten der Stufen- bzw. Phasenbefehle extrahiert (s. Kapitel 6.3.3 bzw. 6.3.4).

- Prädikate, die implizieren, ob eine gegebene Strukturkomponente von einem gegebenen Klassen- bzw. Stufenbefehl in einer gegebenen Pipelinestufe bzw. Taktphase gelesen (“domain”) bzw. geschrieben (“range”) wird [Kogg81]:

  Stage_Domain:((*class_instruction*, *pipeline_stage*, *CL_resource*) → *bool*)
  Stage_Range:((*class_instruction*, *pipeline_stage*, *CL_resource*) → *bool*)
  Phase_Domain:((*stage_instruction*, *clock_phase*, *CL_resource*) → *bool*)
  Phase_Range:((*stage_instruction*, *clock_phase*, *CL_resource*) → *bool*)

  Beispiele: *Stage_Domain* (*ALU*, *ID*, *RF*) = *True*, d.h. die Ressource *RF* wird vom Klassenbefehl *ALU* in der Pipelinestufe *ID* gelesen (s. Tabelle 5-1). *Phase_Range* (ID_C, ϕ_2, *D-MEM*) = *False*, d.h. die Ressource *D-MEM* wird nicht vom Stufenbefehl ID_C in der Taktphase ϕ_2 geschrieben (s. Tabelle 5-1). Diese Prädikate werden aus den formalen Spezifikationsprädikaten der Stufen- bzw. Phasenbefehle extrahiert (s. Kapitel 6.3.2 bzw. 6.3.3).

7.2.3 Ressourcenkonflikte

Ressourcenkonflikte [Ston90, MüSc91] (auch *Struktur-Hazards* [Kogg81, Milu89, HePa90, Ston90] oder *Kollisionen* [DTSP75, Kogg81, HwBr84, Ston90] genannt) entstehen, wenn die Hardware nicht alle möglichen Kombinationen von Befehlen während der Pipelineausführung unterstützen kann. Dies geschieht, wenn einige Ressourcen oder Funktionseinheiten nicht in ausreichendem Maße vorhanden sind und zwei oder mehr Befehle versuchen, diese simultan zu verwenden. Eine Ressource kann ein Register, eine Speichereinheit, eine Funktionseinheit, ein Bus, usw. sein. Die Verwendung einer Ressource entspricht bei Speicherelementen einer Schreiboperation und bei Funktionseinheiten einer Zuteilung der entsprechenden Einheit. In den folgenden Abschnitten werden zunächst die Ressourcenkonflikte formal spezifiziert, dann wird deren formale Verifikation diskutiert.

7.2.3.1 Spezifikation der Ressourcenkonflikte

Ressourcenkonflikte werden formal durch Ressourcenkonfliktprädikate spezifiziert, die an das hierarchische RISC-Modell angelehnt werden. Demnach werden diese Prädikate bezüglich verschiedener Abstraktionsebenen definiert, indem nur über die sichtbaren Ressourcen einer bestimmten Ebene argumentiert wird. In den nächsten Abschnitten wird die Spezifikation der Ressourcenkonflikte hierarchisch auf der Klassen-, Stufen und Phasenebene dargestellt. Andere Spezifikationsformen, die an den nachträglichen Verifikationsansatz angelehnt sind, sind in [TaKu94b] zu finden.

Konflikte auf der Klassenebene

Das Ressourcenkonfliktprädikat *Res_Conflict* (s. Gleichung (7-3)) ist äquivalent zu einem Mehrfachkonflikt zwischen der maximalen Anzahl der Klassenbefehle, die in der Pipeline vorhanden sein können. Dieser Mehrfachkonflikt wird formal mit Hilfe eines Prädikats namens *Multiple_Res_Conflict* spezifiziert:

$$\vdash_{def} Res\_Conflict := Multiple\_Res\_Conflict\,(I_1, \ldots, I_{n_s}) \qquad \text{(Gl. 7-5)}$$

Das Prädikat *Multiple_Res_Conflict* ist genau dann wahr, wenn ein beliebiges Paar (I_i, I_j) von Klassenbefehlen in der Pipeline auf eine Ressource in den entsprechenden Pipelinestufen S_i bzw. S_j gemeinsam zugreifen will (s. Bild 7-2).

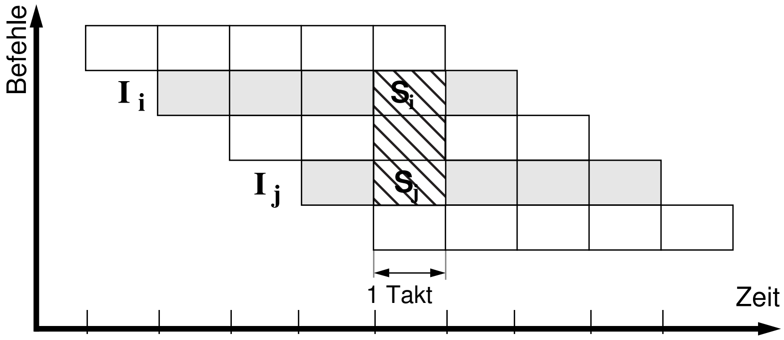


Bild 7-2: Konflikte zwischen zwei Stufenbefehlen

Demnach kann das Prädikat *Multiple_Res_Conflict* durch eine mehrfache Disjunktion über alle möglichen Konflikte von Stufenbefehlspaaren, die zu den Klassenbefehlen $I_1 \ldots I_{n_s}$ gehören, beschrieben werden. Sei *Dual_Stage_Conflict* ein Prädikat, welches Doppelkonflikte der Stufenebene beschreibt. Unter Verwendung der Funktion *ClassToStage* wird *Multiple_Res_Conflict* wie folgt spezifiziert, wobei ψ_i in Abhängigkeit vom Index i eine spezifische Pipelinestufe repräsentiert, d.h. $\psi_1 = IF$, $\psi_2 = ID$, $\psi_3 = EX$, usw.:

$$\begin{aligned}&\vdash_{def} Multiple\_Res\_Conflict\,(I_1, \ldots, I_{n_s}) := \\ &\bigvee_{\substack{i,\,j \\ (i,\,j\,=\,1 \ldots n_s) \\ (i\,<\,j)}} Dual\_Stage\_Conflict\,(ClassToStage\,(\psi_i, I_{n_s\text{-}i+1}), ClassToStage\,(\psi_j, I_{n_s\text{-}j+1}))\end{aligned} \tag{Gl. 7-6}$$

Konflikte auf der Stufenebene

Ein Doppelkonflikt der Stufenebene findet statt, wenn zwei beliebige Stufenbefehle versuchen, dieselbe Ressource zu verwenden. Da aber nur Stufenbefehle, die zu unterschiedlichen Pipelinestufen gehören, simultan ausgeführt werden können (s. schraffierter Kasten in Bild 7-1), muß bei der Bildung der Doppelkonflikte auf der Stufenebene sichergestellt werden, daß die betreffenden Stufenbefehle unterschiedliche logische Typen von Pipelinestufen aufweisen. Unter Verwendung der Funktion *Stage_Type* und des Prädikates *Stage_Used* wird das Prädikat *Dual_Stage_Conflict* formal wie folgt spezifiziert:

$$\begin{aligned}&\vdash_{def} Dual\_Stage\_Conflict\,(S_i, S_j) := \\ &\quad \exists\, r\!: SL\_resource. \\ &\qquad Stage\_Type\,(S_i)\ \ Stage\_Type\,(S_j) \wedge \\ &\qquad Stage\_Used\,(S_i, r) \wedge Stage\_Used\,(S_j, r)\end{aligned} \tag{Gl. 7-7}$$

Eine nähere Betrachtung eines Konfliktfalles zwischen zwei gegebenen Stufenbefehlen S_i und S_j auf der Phasenebene zeigt, daß ein Doppelkonflikt eigentlich nur dann gegeben ist, wenn die zwei Stufenbefehle S_i und S_j die Ressource r in derselben Phase des Taktes verwenden (s. Bild 7-3).

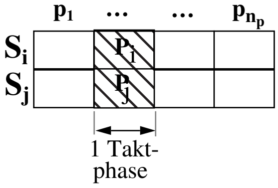


Bild 7-3: Parallele Ausführung der Taktphasen

Legt man eine Implementierung der Stufenbefehle auf der Phasenebene zugrunde, und betrachtet man alle Kombinationen von Phasenbefehlen für zwei beliebige Stufenbefehle, dann kann der Doppelkonflikt auf der Stufenebene durch ein Mehrfachkonfliktprädikat auf der Phasenebene (bezeichnet durch *Multiple_Phase_Conflict*) definiert werden:

$$\vdash_{def} Dual\_Stage\_Conflict := Multiple\_Phase\_Conflict\ (S_i, S_j) \qquad \text{(Gl. 7-8)}$$

Analog zur Klassenebene wird das Prädikat *Multiple_Phase_Conflict* formal durch Disjunktionen über alle mögliche Konflikte von Phasenbefehlspaaren (bezeichnet durch *Dual_Phase_Conflict*) definiert, wobei φ_k in Abhängigkeit vom Index *k* eine spezifische Taktphase repräsentiert, d.h. $\varphi_1 = \phi_1$, $\varphi_2 = \phi_2$, usw.:

$$\begin{aligned} &\vdash_{def} Multiple\_Phase\_Conflict\ (S_i, S_j) := && \text{(Gl. 7-9)} \\ &\quad \bigvee_{\substack{k \\ (k = 1 \ldots n_p)}} Dual\_Phase\_Conflict\ (StageToPhase\ (\varphi_k, S_i), StageToPhase\ (\varphi_k, S_j)) \end{aligned}$$

Konflikte auf der Phasenebene

Ein Doppelressourcenkonflikt auf der Phasenebene kommt zustande, wenn zwei gegebene Phasenbefehle, die denselben logischen Phasentyp aufweisen, um dieselbe Ressource konkurrieren (s. Bild 7-3). Unter Verwendung der Funktionen *Phase_Type*, *Stage_Type* und *StageToPhase* und des Prädikates *Phase_Used* wird ein Doppelkonflikt auf der Phasenebene durch das Prädikat *Dual_Phase_Conflict* wie folgt definiert:

$$\begin{aligned} &\vdash_{def} Dual\_Phase\_Conflict\ (P_i, P_j) := \\ &\quad \exists\ r{:}\ PL\_resource. \\ &\qquad Phase\_Type\ (P_i) = Phase\_Type\ (P_j) \wedge && \text{(Gl. 7-10)} \\ &\qquad Stage\_Type\ (PhaseToStage\ (P_i))\ \ Stage\_Type\ (PhaseToStage(P_j)) \wedge \\ &\qquad Phase\_Used\ (P_i, r) \wedge Phase\_Used\ (P_j, r) \end{aligned}$$

7.2.3.2 Verifikation der Ressourcenkonflikte

Das Endziel der Verifikation der Ressourcenkonflikte besteht darin, zu zeigen, daß für alle Kombinationen von Klassenbefehlen kein Ressourcenkonflikt entsteht. Das heißt, es ist formal zu beweisen, daß das Prädikat *Multiple_Res_Conflict* nie wahr wird:

$$\vdash \quad \forall\, I_1 \ldots I_{n_s} : \mathit{class\_instruction}\,. \quad \neg\ \mathit{Multiple\_Res\_Conflict}\,(I_1, \ldots, I_{n_s}) \qquad \text{(Gl. 7-11)}$$

Unter Verwendung der Definition von *Multiple_Res_Conflict* (s. Gleichung (7-6)) würde eine Expansion dieses Prädikates auf den Stufen- und Phasenebenen zu einer exponentiellen Anzahl der Fälle führen (s. Anhang B.1.1). Um diesen Aufwand zu umgehen, wird der Beweis des obigen Verifikationsziels in zwei Schritten durchgeführt:

1. Es wird gezeigt, daß zwischen allen möglichen Stufenbefehlspaaren keine Doppelkonflikte stattfinden:

$$\vdash \quad \forall\, S_i\, S_j : \mathit{stage\_instruction}\,. \quad \neg\ \mathit{Dual\_Stage\_Conflict}\,(S_i, S_j) \qquad \text{(Gl. 7-12)}$$

2. Aus dem Ergebnis des ersten Schrittes wird die Negation des Prädikates *Multiple_Res_Conflict* abgeleitet:

$$\vdash \quad (\forall\, S_i\, S_j : \mathit{stage\_instruction}\,.\ \neg\,\mathit{Dual\_Stage\_Conflict}\,(S_i, S_j)) \Rightarrow (\forall I_1 \ldots I_{n_s} : \mathit{class\_instruction}.\ \neg\ \mathit{Multiple\_Res\_Conflict}\,(I_1, \ldots, I_{n_s})) \qquad \text{(Gl. 7-13)}$$

Da in Gleichung (7-13) die Allquantifizierung über alle Stufenbefehlspaare beim Doppelkonfliktprädikat allgemeiner als die Disjunktion dieser Prädikate beim Mehrfachkonfliktprädikat ist, ist der Beweis des zweiten Schrittes einfach durchzuführen, denn es muß nicht einmal das Prädikat *Dual_Stage_Conflict* expandiert werden. Der Beweis des ersten Schrittes hingegen führt ohne Annahmen entweder zu dem Wert *True* oder zu einer Anzahl von Unterzielen, welche die spezifischen Stufenbefehle und welche die spezifische Ressource, um welche die Stufenbefehle in Konflikt stehen, enthält. Zum Beispiel wird ein Konflikt um die Ressource *PC* zwischen dem gemeinsamen IF-Stufenbefehl IF_X und dem ID-Stufenbefehl ID_C wie folgt ausgegeben:

$$(S_i = IF_X), (S_j = ID_C), (r = PC) \vdash \mathsf{F} \qquad \text{(Gl. 7-14)}$$

Um sicherzustellen, daß dieser Ressourcenkonflikt auf der Stufenebene tatsächlich existiert, muß die gegebene Implementierung der Stufenbefehle auf der Phasenebene diesen Konflikt bestätigen. Für das obige Beispiel wird die simultane Verwendung der Ressource *PC* auf der Phasenebene explizit durch das Setzen des folgenden Verifikationsziels überprüft:

$$\vdash \quad \neg\ Multiple\_Phase\_Conflict\,(IF_X, ID_C, PC) \qquad \text{(Gl. 7-15)}$$

Wird dieses Ziel korrekt bewiesen, dann ist die Konfliktfreiheit für diesen Fall vollständig gezeigt, anderenfalls bleibt der Ressourcenkonflikt bestehen und die Implementierung EBM muß entsprechend geändert werden, z.B. durch Duplizierung der Ressource *PC* oder durch das Einfügen einer zusätzlichen Pufferung.

Darüberhinaus ist es möglich, auf der Phasenebene alle Konflikte zu durchsuchen, indem man folgendes Verifikationsziel setzt:

$$\vdash \quad \forall\, P_i\, P_j : Phase\_instruction\,.\ \neg\ Dual\_Phase\_Conflict\,(P_i, P_j) \qquad \text{(Gl. 7-16)}$$

Wegen der großen Anzahl von Phasenbefehlen und Ressourcen, werden jedoch hierbei große Rechenzeiten benötigt, um alle möglichen Kombinationen aufzulösen (s. Anhang B).

7.2.4 Datenkonflikte

Datenkonflikte [MüSc91] (auch *Daten-Hazards* (“*data hazards*”) [Kogg81, HePa90, Ston90], *Zeit-Hazards* (“*timing hazards*”) [Milu89], *Datenabhängigkeiten* (“*data dependencies*”) [Goor89] oder *Datenläufe* (“*data races*”) [Furb89] genannt) entstehen, wenn ein Befehl vom Abarbeitungsresultat eines vorangehenden Befehls abhängt und die Reihenfolge, in der auf die Daten zugegriffen wird, in der Pipeline geändert wird, was zu fehlerhaftem Verhalten führen kann. Der Begriff “Daten” bezieht sich entweder auf den Inhalt eines Registers innerhalb des Prozessorkernes oder auf den Inhalt des Datenspeichers bzw. Caches. Im ersten Fall wird der Konflikt auch *Register-Hazard*, im zweiten Fall *Operanden-Hazard* genannt [RaLi77]. In dieser Arbeit werden sowohl die Inhalte der Register des Prozessors als auch die des Datenspeichers als Daten erfaßt. Da die Datenabhängigkeiten auf Maschinenbefehlsebene betrachtet werden, entsprechen den Daten nur Inhalte von Ressourcen der Architektur- (Klassen-) Ebene, d.h. dem Programmiermodell.

Man unterscheidet drei Arten von Datenkonflikten nämlich “*read after write*” (*RAW*), “*write after read*” (*WAR*) und “*write after write*” (*WAW*) [Kogg81, Goor89, HePa90, Ston90] (auch “*destination source*” (*DS*), “*source destination*” (*SD*) oder “*destination destination*” (*DD*) [Milu89] genannt). Gegeben seien zwei

Klassenbefehlen I_i und I_j, wobei I_j nach I_i in die Pipeline eingeschleust wird. Die drei Datenkonfliktarten RAW, WAR und WAW können wie folgt kurz beschrieben werden:

- RAW-Konflikt: I_j *liest* einen Quellenoperanden, *bevor* I_i diesen *geschrieben* hat.
- WAR-Konflikt: I_j *schreibt* einen Zieloperanden, *bevor* I_i diesen *gelesen* hat.
- WAW-Konflikt: I_j *schreibt* einen Zieloperanden, *bevor* I_i diesen *geschrieben* hat.

Bei einem herkömmlichen RISC-Prozessor stellen RAW-Konflikte die häufigste Art von Datenkonflikten dar, während WAR- und WAW-Konflikte selten und nur in einigen speziellen Fällen auftreten. Da die Semantik dieser Datenkonflikte eine ähnliche Form hat, ist zu erwarten, daß die entsprechenden formalen Spezifikationen und Verifikationen in einer ähnlichen Art und Weise behandelt werden können. Es läßt sich daher eine allgemeine Formalisierungs- und Verifikationsmethode hierfür angeben. Zur Veranschaulichung wird im folgenden hauptsächlich der RAW-Datenkonflikt als konkretes Beispiel verwendet und die resultierenden Ergebnisse werden dann auf die WAR- und WAW-Konflikte übertragen.

7.2.4.1 Spezifikation der Datenkonflikte

Wie bereits aus einigen abweichenden Bezeichnungen (s. oben) hervorgeht, schließen Datenkonflikte, im Gegensatz zu Ressourcenkonflikten, temporale Aspekte ein, über die, zusätzlich zu den Ressourcen, argumentiert werden muß. Diese stehen im direkten Zusammenhang mit den Zeitabstraktionen des hierarchischen RISC-Modells, so daß die formale Spezifikation der Datenkonflikte, analog zu den Ressourcenkonflikten, hierarchisch auf der Klassen-, Stufen und Phasenebene betrachtet werden muß. Andere Spezifikationsformen, die an den nachträglichen Verifikationsansatz angelehnt sind, sind in [TaKu94b] zu finden.

Konflikte auf der Klassenebene

Betrachtet man eine aufgefüllte Pipeline, wie beispielsweise in Bild 7-1, dann soll eine Spezifikation des Datenkonflikts im allgemeinen die maximale Anzahl n_s von Befehlen einschließen, die zu Datenkonflikten führen können. Das Datenkonfliktprädikat *Data_Conflict* (s. Gleichung (7-3)) wird daher durch ein Mehrfachprädikat (bezeichnet als *Multiple_Data_Conflict*) definiert, das die n_s Klassenbefehle I_1 … I_{n_s} mit entsprechenden sequentiellen Einschleuszeiten ("issue times") $t_1^0 \ldots t_{n_s}^0$ beinhaltet:

$$\vdash_{def} Data\_Conflict := Multiple\_Data\_Conflict\,(I_1, \ldots, I_{n_s}) \qquad \text{(Gl. 7-17)}$$

Analog zu den Ressourcenkonflikten wird dieses Mehrfachprädikat als Disjunktion aller möglichen Doppeldatenkonflikte spezifiziert. Sei *Dual_Data_Conflict* ein Prädikat, welches Doppelkonflikte beschreibt, dann wird das Datenkonfliktprädikat *Data_Conflict* wie folgt definiert:

$$\begin{aligned}&\vdash_{def} \mathit{Multiple\_Data\_Conflict}\,(I_1, \ldots, I_{n_s}) := \\ &\quad \exists\, t_1^0 \ldots t_{n_s}^0 : \mathit{Clock\_cycle}\,. \\ &\qquad \bigvee_{\substack{i,\,j \\ (i,\,j\,=\,1 \ldots n_s) \\ (i\,<\,j)}} \mathit{Dual\_Data\_Conflict}\,((I_i,), (I_j, t_i^0 + j - 1))\end{aligned} \qquad \text{(Gl. 7-18)}$$

Das Prädikat *Dual_Data_Conflict* ist erfüllt, wenn eine Ressource als Strukturkomponente der Klassenebene existiert, für die zwei beliebige Klassenbefehle I_i und I_j, eingeschleust in die Pipeline zu den Zeitpunkten t_i^0 und t_j^0, zu einem Konflikt führen. Bezüglich des RISC-Modells wird *Dual_Data_Conflict* auf der Stufenebene durch ein entsprechendes Datenkonfliktprädikat (bezeichnet *Stage_Data_Conflict*) formal wie folgt definiert:

$$\begin{aligned}&\vdash_{def} \mathit{Dual\_Data\_Conflict}\,((I_i, t_i^0), (I_j, t_j^0)) := \\ &\quad \exists\, r : \mathit{CL\_resource}\,. \\ &\qquad \mathit{Stage\_Data\_Conflict}\,((I_i, t_i^0), (I_j, t_j^0), r)\end{aligned} \qquad \text{(Gl. 7-19)}$$

Konflikte auf der Stufenebene

Sei I_i ein beliebiger Befehl, der zum Zeitpunkt t_i^0 in die Pipeline eingeschleust wird und auf eine gegebene Ressource r zum Zeitpunkt t_i^u ($t_i^0 \leq t_i^u$) *schreibt*, und sei I_j ein weiterer Befehl mit einer späteren Einschleuszeit t_j^0 ($t_i^0 < t_j^0$), der dieselbe Ressource r zum Zeitpunkt t_j^u ($t_j^0 \leq t_j^u$) *liest*. Ein RAW-Datenkonflikt entsteht, wenn der Befehl I_j die Ressource r liest *bevor* (und nicht *nachdem*) diese vom sequentiell vorangehenden Befehl I_i geschrieben wird (s. Bild 7-4). Seien s_i und s_j die entsprechenden Pipelinestufen, in denen die Ressource r geschrieben bzw. gelesen wird. Unter Annahme einer linearen Pipelineausführung der Befehle, d.h. daß die Pipeline zwischendurch nicht eingefroren ("frozen") oder angehalten ("stalled") wird, sind die Zeitpunkte t_i^u und t_j^u gleich ($t_i^0 + \theta(s_i)$) bzw. ($t_j^0 + \theta(s_j)$), wobei das Symbol θ die Funktion *Stage_Rank* repräsentiert, welche die Ordnungsnummer einer gegebenen Pipelinestufe berechnet (s. Kapitel 7.2.2). Die Zeitbedingung für den RAW-Konflikt ($t_j^u \leq t_i^u$) ist somit äquivalent zu ($t_j^0 - t_i^0$) $\leq$ ($\theta\,(s_i) - \theta\,(s_j)$).

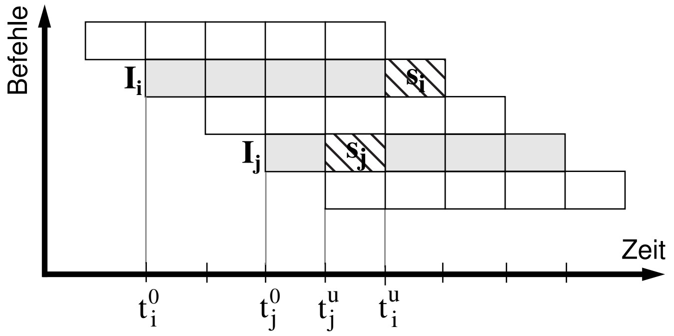


Bild 7-4: RAW-Datenkonflikt auf Taktzyklusebene

Unter Verwendung der Funktion *Stage_Rank* (repräsentiert durch das Symbol θ) und der Prädikate *Stage_Range* und *Stage_Domain*, kann der RAW-Datenkonflikt der Stufenebene wie folgt spezifiziert werden:

$$\begin{array}{l} \vdash_{def} Stage\_RAW\_Conflict\ ((I_i,\ t_i^0), (I_j,\ t_j^0), r) := \\ \qquad \exists\, s_i\, s_j : pipeline\_stage\ . \\ \qquad\qquad ((0 < (t_j^0 - t_i^0)) \wedge \\ \qquad\qquad ((t_j^0 - t_i^0) \le (\theta\,(s_i) - \theta\,(s_j))) \wedge \qquad\qquad \text{(Gl. 7-20)} \\ \qquad\qquad Stage\_Range\ (I_i,\ s_i,\ r) \wedge \\ \qquad\qquad Stage\_Domain\ (I_j,\ s_j,\ r) \end{array}$$

In analoger Weise werden die WAR- und WAW-Prädikate wie folgt definiert, wobei die Semantik des Datenkonflikts durch die Reihenfolge der Prädikate *Stage_Range* und *Stage_Domain* festgelegt wird:

$$\begin{array}{l} \vdash_{def} Stage\_WAR\_Conflict \\ \qquad\qquad ((I_i,\ t_i^0), (I_j,\ t_j^0), r) := \\ \quad \exists\, s_i\, s_j : pipeline\_stage\ . \\ \qquad ((0 < (t_j^0 - t_i^0)) \wedge \\ \qquad ((t_j^0 - t_i^0) \le (\theta\,(s_i) - \theta\,(s_j))) \wedge \\ \qquad Stage\_Domain\ (I_i,\ s_i,\ r) \wedge \\ \qquad Stage\_Range\ (I_j,\ s_j,\ r) \end{array}$$

$$\begin{array}{l} \vdash_{def} Stage\_WAW\_Conflict \\ \qquad\qquad ((I_i,\ t_i^0), (I_j,\ t_j^0), r) := \\ \quad \exists\, s_i\, s_j : pipeline\_stage\ . \\ \qquad ((0 < (t_j^0 - t_i^0)) \wedge \\ \qquad ((t_j^0 - t_i^0) \le (\theta\,(s_i) - \theta\,(s_j))) \wedge \\ \qquad Stage\_Range\ (I_i,\ s_i,\ r) \wedge \\ \qquad Stage\_Range\ (I_j,\ s_j,\ r) \end{array}$$

Ein spezieller Fall der Zeitbedingung des Datenkonflikts entsteht, wenn eine Ressource gleichzeitig von den Befehlen I_i und I_j verwendet wird, d.h. $t_i^u = t_j^u$. In dieser Situation muß der Datenkonflikt auf der Phasenebene untersucht werden.

Konflikt auf der Phasenebene

Seien S_i und S_j zwei beliebige Stufenbefehle von unterschiedlichem logischen Typ, bei denen der Rang von S_i größer als der von S_j ist, z.B. $S_i = WB_L$ und $S_j = ID_C$.

Nach Bild 7-5 entsteht ein RAW-Datenkonflikt auf der Phasenebene, wenn eine beliebige Ressource r vom Stufenbefehl S_i in einer Taktphase p_i beschrieben wird, die sequentiell nach der Taktphase p_j liegt, wo S_j dieselbe Ressource r liest, d.h. ($\tau_j^u \leq \tau_i^u$). Da Befehle auf der Phasenebene streng parallel ausgeführt werden, haben sie alle dieselbe Einschleuszeit in die Pipelinestufe τ_0 (s. Bild 7-5). Die Zeitbedingung ($\tau_j^u \leq \tau_i^u$) ist damit äquivalent zu $((\tau^0 + \xi(p_j)) \leq (\tau^0 + \xi(p_i))) = (\xi(p_j) \leq \xi(p_i))$, wobei das Symbol ξ die Funktion *Phase_Rank* repräsentiert, welche die Ordnungsnummer einer gegebenen Taktphase berechnet (s. Kapitel 7.2.2).

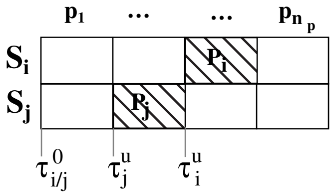


Bild 7-5: RAW-Datenkonflikt auf der Taktphasenebene

Unter Verwendung der Funktionen *Stage_Rank*, *Phase_Rank* und *Stage_Type* sowie der Prädikate *Phase_Range* und *Phase_Domain* kann ein Datenkonflikt auf der Phasenebene durch folgendes Prädikat formal spezifiziert werden, wobei das Symbol ϑ die Funktion *Stage_Type* repräsentiert (s. Kapitel 7.2.2):

$$\begin{aligned}
&\vdash_{def} \mathit{Phase\_RAW\_Conflict}\,(S_i, S_j, r) := \\
&\quad \exists\, p_i\, p_j : \mathit{clock\_phase}\,. \\
&\qquad (\xi(p_j) < \xi(p_i)) \wedge \\
&\qquad (\theta\,(\vartheta(S_j)) < (\theta\,(\vartheta(\,S_i)) \wedge \\
&\qquad \mathit{Phase\_Range}\,(S_i, p_i, r) \wedge \\
&\qquad \mathit{Phase\_Domain}\,(S_j, p_j, r)
\end{aligned} \tag{Gl. 7-21}$$

Analog dazu werden die WAR- und WAW-Konflikte auf der Phasenebene gemäß der entsprechenden Definitionen auf der Stufenebene wie folgt definiert:

$$\begin{aligned}
&\vdash_{def} \mathit{Phase\_WAR\_Conflict}\,(S_i, S_j, r) := \\
&\quad \exists\, p_i\, p_j : \mathit{clock\_phase}\,. \\
&\qquad (\xi(p_j) < \xi(p_i)) \wedge \\
&\qquad (\theta\,(\vartheta(S_j)) < (\theta\,(\vartheta(\,S_i)) \wedge \\
&\qquad \mathit{Phase\_Range}\,(S_i, p_i, r) \wedge \\
&\qquad \mathit{Phase\_Range}\,(S_j, p_j, r)
\end{aligned}$$

$$\begin{aligned}
&\vdash_{def} \mathit{Phase\_WAW\_Conflict}\,(S_i, S_j, r) := \\
&\quad \exists\, p_i\, p_j : \mathit{clock\_phase}\,. \\
&\qquad (\xi(p_j) < \xi(p_i)) \wedge \\
&\qquad (\theta\,(\vartheta(S_j)) < (\theta\,(\vartheta(\,S_i)) \wedge \\
&\qquad \mathit{Phase\_Range}\,(S_i, p_i, r) \wedge \\
&\qquad \mathit{Phase\_Range}\,(S_j, p_j, r)
\end{aligned}$$

7.2.4.2 Verifikation der Datenkonflikte

Das Endziel der Verifikation der Datenkonflikte besteht in dem Beweis, daß keiner der Datenkonflikte (RAW, WAR und WAW) existiert:

$$\vdash \neg \mathit{Data\_Conflict} \Leftrightarrow \begin{array}{l}(\neg \mathit{RAW\_Conflict} \wedge \\ \neg \mathit{WAR\_Conflict} \wedge \\ \neg \mathit{WAW\_Conflict})\end{array} \qquad \text{(Gl. 7-22)}$$

Der Verifikationsbeweis wird daher für jeden Typ von Datenkonflikten in drei unabhängigen Teilen durchgeführt. Wegen der Ähnlichkeit der Spezifikationsprädikate aller drei Konfliktarten weisen diese Beweise eine ähnliche Form auf. Daher wird zur Veranschaulichung im folgenden lediglich die Verifikation des RAW-Konflikts beschrieben.

Auf der höchsten Hierarchieebene ist das Beweisziel für den RAW-Konflikt anhand des Mehrfachprädikates *Multiple_RAW_Conflict* wie folgt gegeben:

$$\begin{array}{l}\vdash \forall I_1 \ldots I_{n_s} : \mathit{class\_instruction}. \\ \qquad \neg \mathit{Multiple\_RAW\_Conflict}\,(I_1, \ldots, I_{n_s})\end{array}$$

Dieses Beweisziel enthält eine Allquantifizierung über alle möglichen Konfliktkombinationen von n_s Klassenbefehlen, die in der Pipeline entstehen können. Ein direkter Beweis dieses Ziels führt jedoch zu einer großen Anzahl von Fallunterscheidungen (s. Anhang B). Daher wird der Beweis analog zu den Ressourcenkonflikten (s. Kapitel 7.2.3.2) in zwei Schritten durchgeführt. Dies geschieht wie folgt:

1. Es wird gezeigt, daß keine Doppelkonflikte stattfinden:

$$\begin{array}{l}\vdash \forall I_i\, I_j : \mathit{class\_instruction}. \\ \quad \forall t_i^0\, t_j^0 : \mathit{Clock\_cycle}. \\ \qquad \neg \mathit{Dual\_RAW\_Conflict}\,((I_i, t_i^0), (I_j, t_j^0))\end{array} \qquad \text{(Gl. 7-23)}$$

2. Die Negation des Prädikates *Multiple_RAW_Conflict* wird aus dem ersten Schritt abgeleitet:

$$\begin{array}{l}\vdash (\forall I_i\, I_j : \mathit{class\_instruction}. \\ \qquad \forall t_i^0\, t_j^0 : \mathit{Clock\_cycle}. \\ \qquad\quad \neg \mathit{Dual\_RAW\_Conflict}\,((I_i, t_i^0), (I_j, t_j^0))) \\ \quad \Rightarrow (\forall I_1 \ldots I_{n_s} : \mathit{class\_instruction}. \\ \qquad\qquad \neg \mathit{Multiple\_RAW\_Conflict}\,(I_1, \ldots, I_{n_s}))\end{array} \qquad \text{(Gl. 7-24)}$$

Der Beweis des zweiten Schritts ist einfach zu realisieren, da eine Allquantifizierung über alle möglichen Paare (I_i,I_j) allgemeiner als die Disjunktion einer festen Anzahl von Paaren in Abhängigkeit von n_s ist. Unter Verwendung der Definitionen von *Dual_Data_Conflict* (s. Gleichung (7-19)) wird der erste Verifikationsschritt für den RAW-Konflikt wie folgt formuliert:

$$\begin{aligned} \vdash \quad & \forall\, I_i\, I_j : class\_instruction\,. \\ & \quad \forall\, t_i^0\, t_j^0 : Clock\_cycle\,. \\ & \quad\quad \forall\, r : CL\_resource\,. \\ & \quad\quad\quad \neg\ Stage\_RAW\_Conflict\,((I_i,\, t_i^0),\, (I_j,\, t_j^0),\, r) \end{aligned} \qquad \text{(Gl. 7-25)}$$

Eine Expansion dieses Beweisziels auf der Stufenebene durch Verwendung der Definition des Prädikates *Stage_RAW_Conflict* (s. Gleichung (7-20)) führt entweder zu *True* oder zu einer Anzahl von Unterzielen, welche die spezifische Ressource und die Klassenbefehle enthält, die um diese Ressource in Konflikt stehen. Zum Beispiel wird ein RAW-Datenkonflikt zwischen dem LOAD- und ALU-Befehl infolge eines Schreibens des Registersatzes *RF* in der WB-Stufe vom LOAD-Befehl und eines Lesens dieser Ressource *RF* in der ID-Stufe vom ALU-Befehl wie folgt entdeckt und ausgegeben. Dabei entspricht die Zahl “3” der Differenz $(\theta\,(s_i)$ - $\theta\,(s_j) = (\theta\,(WB)$ - $\theta\,(ID)$:

$$\begin{array}{l} (I_i = LOAD),\, (I_j = ALU),\ (0 < (t_j^0 - t_i^0)) \\ (s_i = WB),\, (s_j = ID),\, (r = RF) \end{array} \quad \vdash\ \neg\,((t_j^0 - t_i^0) \le 3) \qquad \text{(Gl. 7-26)}$$

Diese Ausgabe wird wie folgt interpretiert: solange die Einschleuszeiten der in Konflikt stehenden LOAD- und ALU-Befehle die Bedingung “$(t_j^0 - t_i^0) \le 3$” erfüllen, bleibt der RAW-Datenkonflikt um die Ressource *RF* bestehen. Um diesen Konflikt zu beseitigen, muß diese Zeitbedingung neutralisiert werden. Dies kann unter Betrachtung von folgenden zwei Fällen realisiert werden:

1. “$(t_j^0 - t_i^0) = 3$”: Mit $t_i^u = t_i^0 + \theta(WB)$ und $t_j^u = t_j^0 + \theta(ID)$ (s. Kapitel 7.2.4.1), ist diese Zeitbedingung äquivalent zu $(t_j^u - \theta(ID)) - (t_i^u - \theta(WB)) = 3$ d.h. $t_i^u = t_j^u$. Demnach muß der Datenkonflikt auf niedrigerer Zeitgranularität, nämlich der Phasenebene, untersucht werden. Dies wird durch das explizite Setzen des folgenden Zieles gegeben:

$$\begin{aligned} \vdash\ & \neg\, Phase\_RAW\_Conflict \\ & \quad (ClassToStage\ (WB,\ LOAD),\ ClassToStage\ (ID,\ ALU),\ RF) \end{aligned} \qquad \text{(Gl. 7-27)}$$

Wenn dieses Verifikationsziel korrekt bewiesen wird, dann ist die Datenkonfliktfreiheit für diesen Fall definitiv gezeigt. Andernfalls verbleibt der Konflikt zwischen der LOAD- und ALU-Klasse. Zur Konfliktbeseitigung bieten sich zwei Alternativen an: Entweder wird die Hardware-Implementierung geändert, z.B. durch Einfügung von mehr Taktphasen, oder man verwendet die Technik des Software-Scheduling (s. unten).

2. “$(t_j^0 - t_i^0) < 3$”: Ein Datenkonflikt zwischen der LOAD- und ALU-Klasse ist dann gegeben. Die ausgegebene Zeitbedingung liefert eine exakte Information über die maximale Anzahl der Verzögerungstakte (“delay slots”) oder Bypass-Pfade, die von einem Zeitplaner (“Software-Scheduler”) bzw. von der Implementierung EBM bereitgestellt werden müssen, nämlich “3-1 = **2**”.

Verwendet man die Technik des “*Software-Scheduling*” (auch “*instruction scheduling*” genannt [HePa90]), dann kann die Software-Randbedingung, welche zur Beseitigung des obigen Konfliktbeispiels benötigt wird, wie folgt definiert werden:

$$\vdash_{def}\ SW\_Constraint := \quad \begin{array}{l}((I_i = LOAD) \wedge (I_j = ALU) \wedge \\ (r = RF) \wedge (0 < (t_j^0 - t_i^0))\end{array} \Rightarrow ((t_j^0 - t_i^0) > 3) \qquad \text{(Gl. 7-28)}$$

Benutzt man die Technik des “*Bypassing*” (auch “*Forwarding*” genannt [HePa90]), dann muß sichergestellt werden, daß das benötigte Datum, sobald es berechnet ist (Ende der EX-Stufe), zum nächsten Befehl (Anfang der EX-Stufe) weitergeleitet wird. Dieses Verhalten wird durch die Hardware EBM realisiert, indem einige Register und entsprechende Rückführpfade zum Speichern bzw. Weiterleiten dieses Datums implementiert werden.

Die Existenz dieser Register und der Bypass-Pfade wird einfach wie folgt formalisiert:

$$\begin{array}{l}\vdash_{def}\ Bypassing := \\ \quad \exists\, rb\,.\ (rb = RF) \wedge Stage\_Range\ (I_i, EX, rb) \wedge \\ \qquad Stage\_Domain\ (I_j, EX, rb)\end{array} \qquad \text{(Gl. 7-29)}$$

Unter Annahme dieser Bypassing-Bedingung im Beweisziel des Datenkonflikts (s. Gleichung (7-25)) werden die existenzquantifizierten Variablen s_i und s_j der Pipelinestufen in der Definition von *Stage_RAW_Conflict* (s. Gleichung

(7-20)) durch den Wert der Pipelinestufe EX ersetzt und die Zeitbedingung reduziert sich somit zu der Gleichung:

$$\ldots, (0 < (t_j^0 - t_i^0)) \vdash \neg ((t_j^0 - t_i^0) \leq 0)$$

Diese ist immer erfüllt und demzufolge wird das ursprüngliche Verifikationsziel der Gleichung (7-25) korrekt bewiesen.

Zusammenfassung: Gegeben seien einige spezifische Software-Constraints in Form von Zeitbedingungen oder die Implementierung EBM, welche Bypass-Pfade mit entsprechenden Registern und verarbeitenden Schaltungen enthält, dann wird die Nicht-Erfüllung aller Datenkonflikte RAW, WAR und WAW für alle möglichen Befehlskombinationen, Einschleuszeiten und Ressourcen der Klassenebene verifiziert. Formal:

$$\boxed{SW\_Constraints, EBM \vdash \begin{array}{l} (\neg\, RAW\_Conflict \;\wedge \\ \neg\, WAR\_Conflict \;\wedge \\ \neg\, WAW\_Conflict\,) \end{array} \qquad \text{(Gl. 7-30)}}$$

7.2.5 Kontrollkonflikte

Kontrollkonflikte [MüSc91, Stone90] (auch *Kontroll-Hazards* [HePa90, Ston90], *Branch-Hazards* [HePa90], Abarbeitungshazards (“*sequencing hazards*”) [Milu89], *Branch-Abhängigkeiten* (“*branch dependencies*”) [Goor89] oder *Instruktionshazard* (“*instruction hazards*”) [RaLi77] genannt) entstehen aus dem Pipelining von Sprung- und Verzweigungsbefehlen sowie anderen Befehlen, die den Befehlszähler *PC* verändern und damit den linearen Befehlsfluß unterbrechen.

7.2.5.1 Spezifikation der Kontrollkonflikte

Gegeben sei ein beliebiger Befehl I_i, eingeschleust in die Pipeline zum Zeitpunkt t_i^0. Sei $A_f(I_i, t_i^0)$ die Holadresse (“fetch address”) von I_i, d.h. $A_f(I_i, t_i^0) = PC(t_i^0)$, und sei $A_n(I_i, t_i^0)$ die Adresse des sequentiell folgenden Befehls, d.h. $A_n(I_i, t_i^0) = A_f(I_{i+1}, t_{i+1}^0) = PC(t_{i+1}^0)$. Bei einer Befehlsausführung in einer Pipeline wird zu jedem Taktzyklus ein neuer Befehl initiiert (geholt), d.h. $PC(t_{i+1}^0) = PC(t_i^0+1)$. Wenn I_i ein Kontrollbefehl ist, dann ist die sequentiell folgende Adresse eine spezifische Zieladresse A_t, d.h. $A_n(I_i, t_i^0) = A_t(I_i, t_i^0)$. Wegen der sequentiellen Ausführung eines einzelnen Befehls durch eine Anzahl von Teilschritten kann die Zieladresse erst dann geholt werden, wenn der Befehl I_i bereits geholt und dekodiert wurde, und die Berechnung der Zieladresse abgeschlossen ist. Dies alles kann aber nicht in einem einzigen Schritt (Takt) geschehen und die Zieladresse A_t ist somit gleich $A_t\,(I_i, t_i^0) = PC\,(t_i^0{+}n)$, wobei $n > 1$ ist. Demzufolge entspricht die Nächstadresse nicht mehr der Zieladresse, d.h.:

$$A_n\,(I_i, t_i^0) = PC\,(t_{i+1}^0) = PC\,(t_i^0{+}1) \neq PC\,(t_i^0{+}n) = A_t\,(I_i, t_i^0)$$

und folglich wird ein falscher Befehl geholt.

Demnach entsteht ein Kontrollkonflikt im allgemeinen, wenn immer ein zu initiierender Befehl aus einer Adresse geholt wird, die von einem vorangehenden Befehl noch nicht zu Ende berechnet wurde [RaLi77]. Ein Kontrollkonflikt entspricht somit der Situation, bei der ein Befehl versucht die Ressource *PC* zu lesen, die von einem vorangehenden Befehl noch nicht geschrieben bzw. aktualisiert worden ist. Eine nähere Betrachtung dieser Sachlage zeigt, daß diese Eigenschaft eines Kontrollkonflikts mit der Definition eines RAW-Datenkonflikts bei der Ressource *PC* übereinstimmt [RaLi77, Kogg81]. Das Konfliktprädikat *Control_Conflict* (s. Gleichung (7-3)) kann daher formal wie folgt definiert werden:

$$\begin{aligned} &\vdash_{def} \mathit{Control\_Conflict} := \\ &\qquad \mathit{Stage\_RAW\_Conflict}\,((\mathit{CONTROL}, t_i^0), (I_j, t_j^0), \mathit{PC}) \end{aligned} \qquad \text{(Gl. 7-31)}$$

7.2.5.1 Verifikation der Kontrollkonflikte

Nach Gleichung (7-31) stellt die Verifikation der Kontrollkonflikte nur einen speziellen Fall der Datenkonfliktbeweise dar. Demnach wird das Verifikationsziel der Kontrollkonflikte einfach wie folgt gesetzt:

$$\begin{aligned} &\vdash \forall I_j : \mathit{class\_instruction}\,. \\ &\qquad \forall t_i^0\ t_j^0 : \mathit{Clock\_cycle}\,. \\ &\qquad\qquad \neg\, \mathit{Stage\_RAW\_Conflict}\,((\mathit{CONTROL}, t_i^0), (I_j, t_j^0), \mathit{PC}) \end{aligned} \qquad \text{(Gl. 7-32)}$$

und führt zum Beispiel beim DLX-Prozessor zu Unterzielen der folgenden Form:

$$(I_j = \mathit{ALU}), (s_i = \mathit{ID}), (s_j = \mathit{IF}), (0 < (t_j^0 - t_i^0)) \vdash \neg\, ((t_j^0 - t_i^0) \leq 1) \qquad \text{(Gl. 7-33)}$$

Zur Konfliktbeseitigung ist hier kein Bypassing möglich, da die Berechnung der Zieladresse, die weitergeleitet werden soll, zu einem früheren Zeitpunkt nicht stattfinden kann. Eine verbreitete Technik dafür ist Software-Scheduling (s. Kapitel 7.2.4.2), wobei der betreffende Kontrollbefehl verzögert ausgeführt wird. Daher auch der Name “*delayed Branch*” [HePa90]. Es existiert in der Literatur eine Reihe von anderen Methoden, die sich dazu anbieten, die Verzögerung der Ausführung gering zu halten [HePa90]. Bei der gegebenen Implementierung des DLX-Prozessors [DeTa94] wird nur ein einziger Verzögerungstakt ("delay slot") benötigt, um den Kontrollkonflikt zu beseitigen ("$(t_j^0 - t_i^0) \leq 1$" in Gleichung (7-33)). Die in diesem Fall benötigte Zeitbedingung (Software-Constraint) wird wie folgt definiert:

$$\vdash_{def} SW\_Constraint := (0 < (t_j^0 - t_i^0)) \Rightarrow ((t_j^0 - t_i^0) > 1) \qquad \text{(Gl. 7-34)}$$

7.2.6 Zusammenfassung

Das Ziel der Verifikation der Pipelineverarbeitung besteht in dem Beweis der Nicht-Existenz aller möglichen Pipelinekonflikte (Ressourcen-, Daten- und Kontrollkonflikte) für alle möglichen Kombinationen von Klassenbefehlen, die in der Pipeline parallel ausgeführt werden können. Gegeben seien die Implementierung EBM und einige Software-Constraints in Form von Zeitbedingungen. Die Korrektheit der Pipelineverarbeitung folgt dann aus den Theoremen in den Gleichungen (7-11), (7-30) und (7-32):

$$SW\_Constraints, EBM \quad \vdash \quad \begin{array}{l} (\neg Res\_Conflict \wedge \\ \neg Data\_Conflict \wedge \\ \neg Control\_Conflict) \end{array} \qquad \text{(Gl. 7-35)}$$

Die erzielten Beweise der Konfliktfreiheit sind, wie bereits oben erwähnt, an das RISC-Modell stark angelehnt, wobei auf höherer Hierarchieebene über Klassenbefehle argumentiert wird. Die resultierenden Theoreme lassen sich jedoch auf die Architekturebene bzw. deren Befehle direkt übertragen, da zwischen der Architektur- und Klassenebene keine Struktur- oder Zeitabstraktion existiert und sie daher dieselben Ressourcen besitzen und dasselbe Zeitverhalten aufweisen. Demnach ist die Korrektheit der Pipelineverarbeitung für alle mögliche Kombinationen von Maschinenbefehlen gegeben.

7.3 Semantische Korrektheit der Befehle

Unter Verwendung der Idee der Befehlsklassen und des hierarchischen Modells wird, wie in den Kapiteln 5.5.3 bzw. 5.5.5 beschrieben, die semantische Korrektheit in zwei Schritten bewiesen. Dabei wird zuerst die Korrektheit der Klassenebene sukzessiv gezeigt:

$$EBM \Rightarrow Phasenebene \Rightarrow Stufenebene \Rightarrow Klassenebene \qquad \text{(Gl. 7-36)}$$

Dann wird durch Instanziierung der erhaltenen Theoreme für jeden Befehl der Architektur die Korrektheit der Architekturebene sukzessiv bewiesen:

$$EBM \Rightarrow Phasenebene' \Rightarrow Stufenebene' \Rightarrow Architekturebene \qquad \text{(Gl. 7-37)}$$

Angesichts der abstrakten Ebenen des Modells wird jeder dieser Schritte in drei Teilschritte aufgeteilt (s. Kapitel 5.5.5). Beispielsweise folgt für Gleichung (7-36):

$$Stufenebene \Rightarrow Klassenebene \qquad \text{(Gl. 7-38)}$$

$$Phasenebene \Rightarrow Stufenebene \qquad \text{(Gl. 7-39)}$$

$$EBM \Rightarrow Phasenebene \qquad \text{(Gl. 7-40)}$$

Durch Strukturierung der Verifikationsaufgabe sind die Beweisziele auf verschiedenen Ebenen einfach und weisen viele Ähnlichkeiten auf. Die Beweise sind einfacher zu leisten und es lassen sich allgemeine Beweisstrategien entwickeln. Das Setzen und Beweisen der Verifikationsziele wird automatisch durchgeführt, indem parametrisierte Funktionen bzw. Taktiken verwendet werden. Darüberhinaus kann jede Abstraktionsebene (s. Gleichungen (7-38) bis (7-40)) unabhängig verifiziert werden, so daß es einem Entwerfer ermöglicht wird, den Entwurf schrittweise zu verfeinern und zu verifizieren. In den nächsten Abschnitten wird der prinzipielle Verifikationsprozeß auf jeder Abstraktionsebene anhand einiger repräsentativer Beispiele beschrieben. Dann wird dargestellt, wie die Instanziierungen behandelt werden.

7.3.1 Stufen- zu Klassenebene

Um die Korrektheit der Klassenebene formal zu zeigen, muß bewiesen werden, daß jeder einzelne Klassenbefehl durch die sequentielle Ausführung der korrespondierenden Stufenbefehle korrekt implementiert wird. Formal entspricht das dem Beweis der Implikation des Spezifikationsprädikates eines Klassenbefehls durch eine Konjunktion von Stufenprädikaten:

$$IF\_SPEC \wedge ID\_SPEC \wedge \ldots \wedge WB\_SPEC \Rightarrow CLASS\_SPEC \qquad \text{(Gl. 7-41)}$$

Wie in Kapitel 6.1 erläutert wurde, beinhalten die Stufen- und Klassenebenen verschiedene Zeitgranularitäten und Strukturen sowie verschiedenes Verhalten. Während die Struktur- und Verhaltensabstraktion durch die Natur des Implikationsbeweises gegeben sind, wird die Zeitabstraktion durch eine temporale Abstraktionsfunktion realisiert, die von Kontextparametern abhängt, welche aus Lese/Schreibe-Informationen und Pipelinestufenbezeichnern bestehen (s. Kapitel 6.2). Daher sollen die formalen Spezifikationen der Klassenbefehle, wie in Kapitel 6.2.1 bereits diskutiert wurde, im Beweisziel durch eine Zeitabstraktionsfunktion und durch entsprechende existenzquantifizierte Kontextparameter ergänzt werden. Die temporale Abstraktionsfunktion *Time_abs* (s. Gleichung (6-5) in Kapitel 6.2.2) wird durch die Pipelinetiefe n_s instanziiert und auf spezifische Kontextparameter angewendet. Diese implementierungsabhängigen Kontextparameter werden im Beweisziel in der Spezifikationsformel des Klassenbefehls für jede Zustandskomponente eingeführt und später beim Beweis explizit instanziiert.

Unter Verwendung der formalen Spezifikation der Klassen- bzw. Phasenebene (s. Kapitel 6.3.2 bzw. 6.3.3) sieht z.B. das Beweisziel der Verifikation des ALU-Klassenbefehls wie folgt aus: (zur besseren Veranschaulichung wurde der Let-Term für die Registersatzadressen beim ursprünglichen Spezifikationsprädikat des ALU-Klassenbefehls (s. Gleichung (6-12) in Kapitel 6.3.2) weggelassen)

$$\begin{aligned}
&\vdash \forall f_A.\\
&\qquad IF_{A}\_SPEC\,(A, B, \dots, PC, RF, \dots, IR, \dots) \wedge\\
&\qquad ID_{A}\_SPEC\,(A, B, \dots, PC, RF, \dots, IR, \dots) \wedge\\
&\qquad EX_{A}\_SPEC\ f_A\,(A, B, \dots, PC, RF, \dots, IR, \dots) \wedge\\
&\qquad MEM_{A}\_SPEC\,(A, B, \dots, PC, RF, \dots, IR, \dots) \wedge\\
&\qquad WB_{A}\_SPEC\,(A, B, \dots, PC, RF, \dots, IR, \dots)\\
&\qquad \Rightarrow \exists\, C_1\, C_2\, C_3\!: Context.\\
&\qquad\qquad \text{let } f_z = (Time\_abs\ n_s) \text{ in}\\
&\qquad\qquad \forall u\!: Instr\_cycle.\\
&\qquad\qquad\quad (RF \circ f_z\, C_1)\,(u{+}1)\,[(rd \circ f_z\, C_2)\,(u)] =\\
&\qquad\qquad\qquad f_A\,((RF \circ f_z\, C_3)\,(u)\,[(rs1 \circ f_z\, C_2)\,(u)], (RF \circ f_z\, C_3)\,(u)\,[(rs2 \circ f_z\, C_2)\,(u)])
\end{aligned} \tag{Gl. 7-42}$$

Analog dazu kann das Beweisziel der Verifikation des Klassenbefehls CONTROL, der aus der Konjunktion der IF- und ID-Stufenbefehle impliziert wird, wie folgt angegeben werden:

$$\begin{aligned}
&\vdash \forall f_C.\\
&\qquad IF_{C}\_SPEC\,(A, B, \dots, PC, RF, \dots, IR, \dots) \wedge\\
&\qquad ID_{C}\_SPEC\ f_C\,(A, B, \dots, PC, RF, \dots, IR, \dots)\\
&\quad \Rightarrow \exists\, C_1\, C_2\, C_3\, C_4\!: Context.\\
&\qquad\quad \text{let } f_z = (Time\_abs\ n_s) \text{ in}\\
&\qquad\qquad \forall u\!: Instr\_cycle.\\
&\qquad\qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}\,) \text{ in}\\
&\qquad\qquad\qquad (PC \circ f_z\, C_1)\,(u{+}1) =\\
&\qquad\qquad\qquad\qquad f_C\,((PC \circ f_z\, C_2)\,(u), (offset16 \circ f_z\, C_4)\,(u),\\
&\qquad\qquad\qquad\qquad\quad (offset26 \circ f_z\, C_4)\,(u), (RF \circ f_z\, C_3)\,(u)\,[(rs1 \circ f_z\, C_4)\,(u)])
\end{aligned} \tag{Gl. 7-43}$$

Die Allquantifizierung der Klassenabstraktionsfunktionen f_A bzw. f_C über das gesamte Beweisziel deutet auf die Allgemeinheit des zu bestimmenden Theorems, das durch Instanziierung der Klassenabstraktionfunktion für einen spezifischen Maschinenbefehl spezialisiert werden kann, hin. Da die Verifikationsziele dieser Ebene ein ähnliches Muster aufweisen, kann darüberhinaus das Setzen der Beweisziele automatisch geschehen, indem eine parametrisierte Funktion verwendet wird, deren Parameter durch die implementierte Pipelinetiefe n_s, durch den Prädikatenbezeichner des zu verifizierenden Klassenbefehls, durch die zugehörige Klassenabstraktionsfunktion und durch die Prädikatenbezeichner der entsprechenden Spezifikationen der Stufenbefehle gegeben sind. Sei $\mathcal{S}_k$ diese Funktion, so könnte das Beweisziel in Gleichung (7-43) durch folgende Parametrisierung von $\mathcal{S}_k$ automatisch generiert werden:

$$\mathcal{S}_k(n_s, f_C, CONTROL\_SPEC, IF_C\_SPEC, ID_C\_SPEC) \tag{Gl. 7-44}$$

Für den Korrektheitsbeweis aller Klassenbefehle wurde eine gemeinsame parametrisierte Taktik mit folgenden Parametern entwickelt: Pipelinetiefe n_s, der Prädikatenbezeichner des zu verifizierenden Klassenbefehls, eine Liste der Prädikatenbezeichner der implizierenden Stufenbefehle und eine explizite Liste der benötigten Kontextparameter. Die Taktik löst die Allquantifizierung und Let-Terme auf, ersetzt die existenzquantifizierten Variablen durch entsprechende explizite Werte, expandiert die Formeln der Stufenprädikate auf der Zeitgranularität der Taktzyklen und bildet die Zeitvariablen der Klassenebene in die der Stufenebene durch Anwendung der Zeitabstraktionsfunktion ab, löst die Strukturabstraktion durch Auflösung der Konjunktion der Stufenprädikate auf und führt schließlich eine Reihe von arithmetischen und logischen Vereinfachungen sowie Termersetzungsoperationen aus [TaKu94a]. Sei $\mathcal{T}$ diese allgemeine Taktik für den Korrektheitsbeweis der ALU-Klasse, dann wird $\mathcal{T}$ mit folgenden Parametern ausgeführt:

$$\begin{aligned} \mathcal{T}\ (&n_s, \\ &ALU\_SPEC, \\ &[IF_A\_SPEC, ID_A\_SPEC, EX_A\_SPEC, MEM_A\_SPEC, WB_A\_SPEC], \\ &[[write, WB], [read, IF], [read, ID]]\ ) \end{aligned} \tag{Gl. 7-45}$$

Die im Beweis benutzten Kontextparameter werden, wie bereits in Kapitel 6.2.1 angedeutet wurde, direkt aus der implementierten Pipelinestruktur abgeleitet. Zum Beispiel wird die Kontextvariable C_3 in Gleichung (7-42) durch den Wert “[*read*, *ID*]” ersetzt, der gemäß der ID-Reihe der ALU-Spalte einer Leseoperation während der ID-Stufe entspricht (s. Tabelle 5-1).

7.3.2 Phasen- zu Stufenebene

Da zwischen der Phasen- und Stufenebene eine ähnliche Struktur-, Verhaltens-, und Zeitabstraktion wie zwischen der Stufen- und Klassenebene existiert, wird der Korrektheitsbeweis der Stufenebene in ähnlicher Weise durchgeführt wie dies im vorigen Kapitel für die Klassenebene beschrieben wurde. Es wird dieselbe Zeitabstraktionsfunktion *Time_abs* und dieselbe Beweistaktik $\mathcal{T}$ (jedoch mit unterschiedlicher Parametrisierung) verwendet. Für die Festlegung der Verifikationsziele wird eine parametrisierte Funktion $\mathcal{S}_s$ benutzt, die ähnlich der Funktion $\mathcal{S}_k$ der Klassenebene realisiert wird.

Die Verifikation der Stufenebene besteht darin, zu zeigen, daß jeder Stufenbefehl durch eine Sequenz von zugehörigen Phasenbefehlen implementiert wird. Formal entspricht dies dem Beweis, daß das Spezifikationsprädikat eines gegebenen Stufenbefehls von einer Konjunktion von Phasenprädikaten impliziert wird. Beispielsweise ist für einen beliebigen ID-Stufenbefehl folgendes zu zeigen:

$$\phi_1 ID\_SPEC \wedge \ldots \wedge \phi_{n_p} ID\_SPEC \Rightarrow ID\_SPEC \qquad \text{(Gl. 7-46)}$$

Analog zur Verifikation der Klassenebene beinhaltet das Verifikationsziel der Stufenebene Struktur-, Verhaltens- und vor allem Zeitabstraktion. Die Zeitabstraktionsfunktion wird hier jedoch mit der implementierten Anzahl der Taktphasen n_p instanziiert und gemäß der Phasenebene auf entsprechende Kontextparameter angewandt. Die Kontextparameter werden durch existenzquantifizierte Variablen für jede Zustandskomponente eingeführt.

Unter Verwendung der formalen Spezifikationen der Stufen- und Phasenbefehle (s. Kapitel 6.3.3 bzw. 6.3.4) sieht das Verifikationsziel z.B. für die ID-Stufe des ALU-Befehls wie folgt aus:

$$\boxed{\begin{array}{l}
\vdash \phi_1 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, BTA) \wedge \\
\quad \phi_2 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, BTA) \\
\qquad \Rightarrow \exists\, C_1\, C_2\, C_3\, C_4\, C_5\!: Context. \\
\qquad\quad \text{let}\, f_z = (Time\_abs\; n_p)\ \text{in} \\
\qquad\qquad \forall t\!: Clock\_cycle. \hspace{8em} \text{(Gl. 7-47)} \\
\qquad\qquad\quad \text{let}\, (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{22..16})\ \text{in} \\
\qquad\qquad\qquad (A \circ f_z\, C_1)\,(t{+}1) = (RF \circ f_z\, C_3)\,(t)\,[(rs1 \circ f_z\, C_4)\,(t)] \wedge \\
\qquad\qquad\qquad (B \circ f_z\, C_2)\,(t{+}1) = (RF \circ f_z\, C_3)\,(t)\,[(rs2 \circ f_z\, C_4)\,(t)] \wedge \\
\qquad\qquad\qquad (IR1 \circ f_z\, C_5)\,(t{+}1) = (IR \circ f_z\, C_4)\,(t)
\end{array}}$$

Die Verifikationsziele der Phasenebene weisen ein ähnliches Muster auf und können durch Verwendung einer Funktion $\mathcal{S}_s$ automatisch festgelegt werden. Diese Funktion erhält folgende Parameter: die implementierte Anzahl der Taktphasen n_p, den Prädikatenbezeichner des zu verifizierenden Stufenbefehls, gegebenfalls die zugehörige Klassenabstraktionsfunktion und die Prädikatenbezeichner der entsprechenden Spezifikationen der Phasenbefehle. Zum Beispiel wird durch diese Parametrisierung von $\mathcal{S}_s$:

$$\mathcal{S}_s\,(n_p, f_C, ID_C\_SPEC, \phi_1 ID_C\_SPEC, \phi_2 ID_C\_SPEC) \qquad \text{(Gl. 7-48)}$$

folgendes Beweisziel generiert:

$$\boxed{\begin{array}{l}
\vdash \forall f_C. \\
\quad \phi_1 ID_C\_SPEC\ f_C\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \wedge \\
\quad \phi_2 ID_C\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \\
\quad \Rightarrow \exists\, C_1\, C_2\, C_3\, C_4\!: Context. \\
\qquad \text{let}\, f_z = (Time\_abs\ n_p)\ \text{in} \qquad\qquad\qquad\qquad \text{(Gl. 7-49)} \\
\qquad\quad \forall t\!: Clock\_cycle. \\
\qquad\qquad \text{let}\,(rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0})\ \text{in} \\
\qquad\qquad\quad (PC \circ f_z\, C_1)\,(t{+}1) = \\
\qquad\qquad\qquad\quad f_C\,((PC \circ f_z\, C_2)\,(t), (offset16 \circ f_z\, C_4)\,(t), \\
\qquad\qquad\qquad\qquad (offset26 \circ f_z\, C_4)\,(u), (RF \circ f_z\, C_3)\,(t)\,[(rs1 \circ f_z\, C_4)\,(t)])
\end{array}}$$

Zum Beweis wird dieselbe Taktik $\mathcal{T}$, die für die Korrektheitsbeweise der Klassenbefehle verwendet wurde, mit folgenden Parametern benutzt: Anzahl der Taktphasen n_p, der Prädikatenbezeichner des zu verifizierenden Stufenbefehls, eine Liste der Prädikatenbezeichner der implizierenden Phasenbefehle und eine Liste der nötigen Kontextparameter. Für das obige Beweisziel des ID-Stufenbefehls der ALU-Klasse (s. Gleichung (7-47)) wird die Taktik $\mathcal{T}$ wie folgt parametrisiert:

$$\begin{array}{l}
\mathcal{T}\,(\,n_p, \\
\quad ID_A\_SPEC, \qquad\qquad\qquad\qquad \text{(Gl. 7-50)} \\
\quad [\phi_1 ID_A\_SPEC, \phi_2 ID_A\_SPEC], \\
\quad [[write, \phi_2], [write, \phi_2], [read, \phi_2], [read, \phi_1], [write, \phi_2]])
\end{array}$$

Die Kontextparameter können ganz leicht direkt aus der Pipelinestruktur (s. Tabelle 5-1) abgelesen werden. Wenn zum Beispiel das B-Register in der Phase 2 geschrieben wird, wird die Zeitabstraktionsfunktion in Gleichung (7-46) mit dem Kontextparameter $C_2 = [write, \phi_2]$ angewandt.

7.3.3 EBM zu Phasenebene

Die Phasenebene stellt die niedrigste Abstraktionsebene des RISC-Modells dar und liegt direkt über der Implementierung EBM. Der Verifikationsschritt zwischen dem EBM und der Phasenebene unterscheidet sich von den vorigen Verifikationen, da einerseits das EBM eine Hardware-Struktur darstellt, während die Phasenebene eine Verhaltensbeschreibung ist und andererseits weder eine Struktur- noch eine Zeitabstraktion zwischen dem EBM und der Phasenebene existieren. Das EBM stellt in der Regel eine komplexe Hardware-Beschreibung dar, jedoch braucht man dank der Verwendung des hierarchischen Interpretermodells nur die Korrektheit einer kleinen Anzahl von Phasenbefehlen, bestehend aus einfachen Register-Transfers, auf Korrektheit zu prüfen (s. Kapitel 5.5.5).

Da das EBM der konkreten Realisierung entspricht, können dessen Komponenten nicht parametrisiert werden. Eine Allquantifizierung über Klassenabstraktionsfunktionen macht hier wenig Sinn, da nicht alle denkbaren Operationen einer Klassenabstraktion vom EBM implementiert werden. Beispielsweise soll die Klassenabstraktionsfunktion f_A keinen Gleitkommaoperationen entsprechen, da diese vom EBM des DLX nicht unterstützt werden. Man unterscheidet auf dieser Ebene daher zwischen klassenabstraktionsbehafteten und nicht klassenabstraktionsbehafteten Phasenbefehlen (s. Kapitel 6.3.4). Für die letztgenannten können die Verifikationsziele unabhängig von der Klassenabstraktion bewiesen werden. Bei den erstgenannten müssen jedoch in Abhängigkeit von der Implementierung in Form des EBM Einschränkungen hinsichtlich der unterstützten konkreten Operationen der entsprechenden Klasse vorgenommen werden. Für Phasenbefehle dieser Art bieten sich zur Verifikation zwei Möglichkeiten an: Entweder werden nur instanziierte Phasenbefehle auf Korrektheit überprüft oder man muß eine Einschränkung in Form von Annahmen $\mathcal{A}$ als Teil des Beweisziels einführen. Beispiel:

$$\mathcal{A} := (f_A = \mathsf{ADD} \vee \mathsf{SUB} \vee \mathsf{AND} \vee \mathsf{OR} \vee \mathsf{SHL} \vee \mathsf{SHR} \vee \ldots)$$

Zum Setzen der Verifikationsziele wird, analog zu den Klassen- und Phasenebenen, eine einfache parametrisierte Funktion verwendet, deren Parameter durch den betreffenden Prädikatenbezeichner des Phasenbefehls, den zugehörigen Phasenbezeichner und den Prädikatenbezeichner des implementierten EBM gegeben sind. Der Phasenbezeichner wird dazu benötigt, um die Taktphasen im Kopf des EBM-Prädikates entsprechend zu setzen. Beispiel: Phase 1 entspricht

"$clk1 = \mathsf{F} \wedge clk2 = \mathsf{T}$". Sei $\mathcal{S}_p$ diese Funktion, so wendet man $\mathcal{S}_p$ mit folgenden Parametern an:

$$\mathcal{S}_p\,(\phi_2 ID_A\_SPEC,\ \phi_2,\ EBM) \tag{Gl. 7-51}$$

dann erhält man folgendes Verifikationsziel für den Phasenbefehl $\phi_2 ID_A\_SPEC$ der ID-Stufe der ALU-Klasse:

$$\begin{aligned} &\vdash EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathsf{F}, \mathsf{T}) \\ &\qquad \Rightarrow \phi_2 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{aligned} \tag{Gl. 7-52}$$

Der Phasenbefehl $\phi_2 ID_A\_SPEC$ enthält keine Klassenabstraktionsfunktion und kann allgemein ohne Annahmen über das Beweisziel bewiesen werden. Ein anderes Beispiel ist die Phase 1 der ID-Stufe der CONTROL-Klasse $\phi_1 ID_C\_SPEC$ (s. Kapitel 6.3.4). Hierbei wird die Klassenabstraktionsfunktion für die unterstützten Kontrollbefehle JMP, JR, JL, JLR, BEQZ, BNEQ und TRAP (s. Anhang A) einschränkend als Annahme im Beweisziel eingeführt:

$$\begin{aligned} &(f_C = f_{jmp} \vee f_{jr} \vee f_{jl} \vee f_{jlr} \vee f_{beqz} \vee f_{bneq} \vee f_{trap}) \vdash \\ &\qquad EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathsf{F}, \mathsf{T}) \\ &\qquad \Rightarrow \phi_1 ID_C\_SPEC\ f_C\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{aligned} \tag{Gl. 7-53}$$

Für den Korrektheitsbeweis der Phasenbefehle wird eine nicht parametrisierte allgemeine Taktik verwendet [TaKu94a], die unabhängig vom Phasenbefehl und von der Implementierung EBM für beliebige Hardware-Formeln und Strukturbeschreibungen anwendbar ist.

In der letzten Zeit wurden große Fortschritte bei der Automatisierung der Verifikation auf der RT-Ebene realisiert (z.B. MEPHISTO [KuSK93]). Das Verhalten der Phasenbefehle wird dabei formal auf RT-Ebene beschrieben und durch Konjunktionen von einfachen Transfers dargestellt, die als Hardware-Formeln [KuSK93] aufgefaßt werden können, so daß, obwohl das EBM eine komplexe Struktur aufweist, die Verifikation der einzelnen Phasenbefehle automatisch geschehen kann. Das Verifikationsziel läßt sich demnach schrittweise automatisch beweisen, indem sukzessiv die Hierarchie des EBM aufgelöst, dessen Spezifikationsprädikate expandiert, die kombinatorischen Leitungen eliminiert und anschließend die Konjunktion der Phasenbefehle aufgelöst und Termersetzungsoperationen auf die Formeln einzelner Phasenoperationen angewendet werden.

7.3.4 Instanziierung

In diesem Teil der Verifikationsaufgabe wird der Korrektheitsbeweis der Maschinenbefehle behandelt. Dies ist einfach dadurch gegeben, daß die resultierenden Beweise (Theoreme) auf der Klassen-, Stufen- und Phasenebene für jeden Befehl der gegebenen Architektur instanziiert werden. Wie in Kapitel 7.1 schon erwähnt wurde, wird bei diesem Verifikationsschritt hierarchisch vorgegangen, indem die Instanziierung zuerst auf der Klassen-, dann auf der Stufen- und schließlich auf der Phasenebene realisiert wird.

Da die Klassenebene eine Abstraktion der Architekturebene ist, wird in einem ersten Schritt die Äquivalenz der formalen Spezifikationen spezifischer Maschinenbefehle mit den instanziierten Spezifikationen der entsprechenden Befehlsklassen gezeigt (s. Kapitel 5.5.5). Zum Beispiel erhält man für eine gegebene formale Spezifikation des ADD-Befehls (s. Kapitel 6.3.1) folgendes Theorem, wobei die Klassenabstraktionsfunktion durch den Additionsoperator *add* instanziiert wird:

$$\boxed{\begin{array}{l} \vdash ADD\_SPEC\,(PC, RF, I\text{-}MEM, D\text{-}MEM) = \\ \qquad ALU\_SPEC\;\; \mathbf{add}\,(PC, I\text{-}MEM, RF, D\text{-}MEM) \qquad\qquad \text{(Gl. 7-54)} \end{array}}$$

Da alle Korrektheitsbeweise auf der Klassenebene eine Allquantifizierung über die Klassenabstraktionsfunktionen beinhalten (s. Kapitel 7.3.1), ist es erlaubt, für jede Abstraktionsfunktion eine Operatorkonstante des gleichen Typs explizit einzusetzen, so daß diese Korrektheitsbeweise (Theoreme) für einen spezifischen Maschinenbefehl instanziiert werden können. Beispielsweise erhält man durch die Instanziierung des bewiesenen Korrektheitstheorems der ALU-Klasse (s. Gleichung (7-42)) mit der Operatorkonstante *add* folgendes Theorem für den ADD-Befehl:

$$\boxed{\begin{array}{ll} \vdash & IF_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ & ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ & EX_A\_SPEC\;\; add\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ & MEM_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \qquad\qquad \text{(Gl. 7-55)} \\ & WB_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\ & \Rightarrow ALU\_SPEC\;\; \mathbf{add}\,(PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array}}$$

Die Korrektheit des ADD-Befehls folgt demnach einfach aus den obigen beiden Theoremen (s. Gleichungen (7-54) und (7-55)) durch Termersetzung. Man erhält folgendes Korrektheitstheorem:

$$\begin{array}{l} \vdash \; IF_{A}\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ \quad ID_{A}\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ \quad EX_{A}\_SPEC\ \mathbf{add}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ \quad MEM_{A}\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\ \quad WB_{A}\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\ \qquad \Rightarrow ADD\_SPEC\ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array} \qquad \text{(Gl. 7-56)}$$

Auf der Stufenebene braucht man nur klassenabstraktionsbehaftete Korrektheitsbeweise zu instanziieren wie z.B. die Verifikation des Stufenbefehls EX_A der ALU-Klassen in dem obigen Theorem (s. Gleichung (7-56)). Die benötigten Theoreme erhält man in ähnlicher Weise wie dies oben für die Klassenebene beschrieben wurde durch einfache Ersetzung der entsprechenden Klassenabstraktionsfunktion. Beispielsweise wird das instanziierte Korrektheitstheorem für den EX_A Stufenbefehl wie folgt bestimmt:

$$\begin{array}{l} \vdash \; \phi_1 ID_{A}\_SPEC\ (A, B, \ldots, PC, RF, \ldots, BTA) \wedge \\ \quad \phi_2 ID_{A}\_SPEC\,\mathbf{add}\,(A, B, \ldots, PC, RF, \ldots, BTA) \\ \qquad \Rightarrow EX_{A}\_SPEC\ \mathbf{add}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots) \end{array} \qquad \text{(Gl. 7-57)}$$

Auf der Phasenebene sind ebenfalls nur die von den Klassenabstraktionsfunktionen abhängigen Phasenbefehle von Interesse. Wie aus der Diskussion in Kapitel 7.3.3 hervorgeht, kann man hierbei entweder die resultierenden Theoreme aus den eingeschränkten Beweisen direkt instanziieren oder es muß die Korrektheit einzelner Phasenbefehle aus dem EBM impliziert werden. Beispielweise erhält man durch Verwendung der in Kapitel 7.3.3 beschriebenen Taktik folgendes Theorem für den Phasenbefehl $\phi_2 ID_A$ der ID-Stufe der ALU-Klasse:

$$\begin{array}{l} \vdash \; EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathsf{T}, \mathsf{F}) \\ \qquad \Rightarrow \phi_2 ID_{A}\_SPEC\ \mathbf{add}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{array} \qquad \text{(Gl. 7-58)}$$

Aus den Beweisschritten auf der Phasen-, Stufen- und Klassenebene ergibt sich schließlich die Korrektheit der Maschinenbefehle aus dem EBM durch Transitivität der bewiesenen Implikationen. Beispielsweise erhält man aus den obigen Theoremen in Gleichungen (7-56), (7-57) und (7-58) folgendes Korrektheitstheorem für den ADD-Befehl:

$$\begin{array}{l} \vdash \; EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, clk1, clk2) \\ \qquad \Rightarrow ADD\_SPEC\ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array} \qquad \text{(Gl. 7-59)}$$

7.3.5 Zusammenfassung

Aus den Korrektheitsbeweisen der Klassen-, Stufen-, und Phasenebene und den anschließenden Instanziierungen der erhaltenen Theoreme für spezifische Befehle des Befehlssatzes folgt die Korrektheit der Architekturebene aus der Hardware-Implementierung:

$$\vdash \mathit{EBM} \quad \Rightarrow \mathit{Architekturebene}$$

Kapitel 8: Implementierung

In dieser Arbeit wird Logik höherer Ordnung zur Hardware-Spezifikation und Verifikation verwendet. Diese Logik wurde erstmals von Hanna und Daeche [HaDa85] und später von Gordon [Gord86, CaGM86] als geeigneter Formalismus zur Hardware-Spezifikation und Verifikation vorgeschlagen. Mittlerweile existiert eine Reihe von interaktiven Theorembeweisern, die auf Logik höherer Ordnung basieren und die in der Hardware-Verifikation Anwendung gefunden haben, wie z.B. die Systeme VERITAS [HaDa92], HOL [GoMe93], EHDM [HCLR88], NUPRL [Cons86], SDVS [Marc91], LAMBDA [AbHL90], PVS [OwSR93] und ISABELLE [Paul94]. Aus dieser Fülle von Beweissystemen weist heutzutage das HOL-System bei der Hardware-Verifikation die größte Verbreitung auf und wird deshalb hier zur Implementierung der in dieser Arbeit entwickelten Spezifikations- und Verifikationsmethoden eingesetzt. Zur Auswahl des HOL-Systems haben dessen vielfältige Eigenschaften wesentlich beigetragen. Diese Eigenschaften werden im ersten Abschnitt dieses Kapitels beschrieben. Es folgen dann Darstellungen der verschiedenen Spezifikations- und Verifikationsimplementierungen. In einem anschließenden Abschnitt wird die Einbettung der implementierten Methoden in eine Entwurfs- und Verifikationsumgebung kurz erläutert. Zur Validierung der entwickelten Techniken wird im letzten Abschnitt dieses Kapitels die Verifikation eines realen Prozessors mit der Bezeichnung DLX beschrieben.

8.1 Das HOL-Beweissystem

Das HOL-System [GoMe93] ist eine Erweiterung des “Edinburgh ML/LCF” [GoMN79] und stellt einen interaktiven Theorembeweiser-Assistent dar, der auf Logik höherer Ordnung basiert. Die Programmierumgebung des HOL-Systems ist gegeben durch eine typisierte funktionale Sprache ML (SML) [Paul91], welche die Konstruktion von formalen Beweisen in Logik höherer Ordnung ermöglicht. Terme, Typen, Theoreme und andere Eigenschaften der Logik höherer Ordnung stellen in ML spezielle Datenobjekte dar, die durch ML-Funktionen manipuliert werden.

Das HOL-System beruht auf natürlicher Deduktion und baut auf fünf Grundaxiomen und acht primitiven Inferenzregeln zur Ableitung von Theoremen auf. Alle Beweise in HOL werden auf die Verwendung dieser kleinen Menge von Axiomen und Inferenzregeln zurückgeführt, wodurch eine hohe Beweissicherheit garantiert ist, denn jedes Theorem, das von HOL generiert wird, hängt nur von dieser Axiomatisierung ab und stellt ein Theorem der Logik höherer Ordnung dar. Auf diese Weise wird die Korrektheit von nichttrivialen Beweisen, die eine große Anzahl von Inferenzregeln benötigen, wie z.B. die Verifikation von Mikroprozessoren durch Anwendung von Millionen von Regeln, garantiert. Diese Eigenschaft des HOL-Systems steht im Gegensatz zu Ansätzen vieler Verifikationssysteme, die als Sammlung von Entscheidungsprozeduren implementiert sind. Darüberhinaus verfügt das HOL-System über einen Typüberprüfungsmechanismus, der eine Konsistenzprüfung von Spezifikationen ermöglicht. Damit kann man eine erste Validierung von Spezifikationen durchführen bevor überhaupt mit dem formalen Beweis begonnen wird.

Theorien in HOL entsprechen Theorien der Mathematik. Eine HOL-Theorie stellt eine Menge von Konstanten, Typen, Axiomen und Theoremen dar. Das HOL-System unterstützt sowohl Vorwärts- als auch Rückwärtsbeweistechniken und kann durch die Metasprache ML programmiert und erweitert werden. Ferner ist der Theorembeweiser von HOL taktikbasiert. Eine Taktik reduziert ein Beweisziel auf eines oder mehrere Unterziele durch Verwendung von Inferenzregeln. Sie führen Aufgaben, wie Induktion, Termersetzung oder Fallunterscheidung durch. Unter Verwendung von benutzerdefinierten ML-Funktionen können Taktiken programmiert werden, indem Inferenzregeln und Theoreme derart kombiniert werden, daß Teilbeweise automatisiert werden können. Darüberhinaus unterstützt das HOL-System die Bildung von Bibliotheken. Eine Bibliothek besteht aus einer Menge von Theorien, Taktiken und nützlichen ML-Funktionen. Das HOL-System selbst stellt eine Reihe von Bibliotheken zur Verfügung, die sowohl anwendungsspezifische Werkzeuge als auch Sammlungen von mathematischen Theorien darstellen. In dieser Arbeit wird die aktuelle Version HOL90.6 verwendet, welche in der Programmiersprache SML [Paul91] implementiert ist.

Zusammenfassend können die wesentlichen Vorteile des HOL-Systems wie folgt angegeben werden:

- Beweissicherheit
- integrierte, nützliche Theorien und Bibliotheken
- Eine reiche Sammlung von Taktiken
- gutes Beweisverwaltungssystem
- Flexibilität durch ML-Programmierung

Ein beachtlicher Nachteil des HOL-Systems bleibt, wie bei jedem Theorembeweiser, die Notwendigkeit von umfangreichen Kenntnissen in der mathematischen Logik. Diese werden für die Ableitung von Beweisen und für die

Programmierung von Taktiken benötigt, da alle Beweise auf die Grundaxiome und Regeln zurückgeführt werden müssen [Loew91].

8.2 Implementierung der formalen Spezifikationen

Die formalen Spezifikationen bestehen aus der Beschreibung der Interpreterebenen, der Hardware-Implementierung und der verschiedenen Konfliktprädikate. Darüberhinaus müssen verschiedene Typen, Konstanten, Hilfsfunktionen und Hilfsprädikate eingeführt werden.

Die Befehle der verschiedenen Ebenen des RISC-Modells werden in HOL durch Definitionen spezifiziert. Beispielsweise wird die Spezifikation des ALU-Klassenbefehls des DLX-Prozessors, gemäß der Definition in Kapitel 6.3.2, wie folgt implementiert:

```
val ALU_SPEC =
      new_definition ("ALU_SPEC",
       --`ALU_SPEC op PC I_MEM RF D_MEM =
            let rs1 u = WSEG 5 21 (I_MEM u (PC u)) in
            let rs2 u = WSEG 5 16 (I_MEM u (PC u)) in
            let rd u = WSEG 5 11 (I_MEM u (PC u)) in
                !u. RF (u + 1) (rd u) = op (RF u (rs1 u))
                                           (RF u (rs2 u))`--);
```

Die Befehlsspezifikationen werden manuell erstellt, jedoch lassen sich diese aus nichtformalen Beschreibungen einfach ableiten, da sie ein einheitliches Muster aufweisen. Die formale Beschreibung der Hardware-Implementierung hingegen kann aus Netzlisten oder HDL-Beschreibungen automatisch generiert werden [KrSK93]. Die Datentypen der Spezifikationen sind, wie schon in Kapitel 6.1.3 angedeutet wurde, generische Bit-Vektoren. Diese werden anhand einer HOL-Bibliothek für Bit-Vektoren [Wong94] eingeführt, welche verschiedene nützliche Bit-Vektorfunktionen, wie z.B. die Segmentierfunktion WSEG im obigen Beispiel, enthält.

Die Spezifikation der Pipelinekonflikte wird in HOL ebenso durch entsprechende Definitionen für jede Konfliktart implementiert, z.B. wird das allgemeine Prädikat *Stage_RAW_Conflict* der RAW-Datenkonflikte (s. Kapitel 7.2.4.1) wie folgt in HOL implementiert:

```
val STAGE_RAW_CONFLICT =
      new_definition ("STAGE_RAW_CONFLICT",
         --`STAGE_RAW_CONFLICT (Ii,t0i) (Ij,t0j) r =
               ?si sj.
                   (0 < t0j - t0i) /\
                   (t0j - t0i <= STAGE_RANK si - STAGE_RANK sj) /\
                   (STAGE_RANGE Ii si r) /\
                   (STAGE_DOMAIN Ij sj r)`--);
```

8.3 Implementierung der formalen Verifikation

Die Verifikation von komplexen Schaltungen, wie etwa Mikroprozessoren, schließt den Beweis einer großen Anzahl von Theoremen ein. Die Automatisierung solcher aufwendiger Verifikationsaufgaben ist daher unabdingbar und wird in dieser Arbeit angestrebt. Der Verifikationsprozeß läßt sich jedoch nicht vollständig automatisieren. Eine Interaktion ist aber manchmal hilfreich, indem ein Entwurf schrittweise verfeinert und verifiziert wird und somit einem Entwerfer die genaue Betrachtung von Fehlerfällen ermöglicht. Im Rahmen dieser Arbeit wurde eine Vielzahl von allgemeinen parametrisierten Taktiken für die verschiedenen Verifikationsaufgaben in HOL implementiert. Diese Taktiken sind prozessorunabhängig anwendbar und automatisieren die wesentlichen Beweisschritte.

Die Pipeline-Korrektheit wird mit Hilfe von vier parametrisierten Taktiken (MULTIPLE_CONF_TAC, RES_CONF_TAC, DATA_CONF_TAC und CONTROL_CONF_TAC) realisiert [TaKu94b]. Beispielsweise wird die Taktik DATA_CONF_TAC durch folgendes (etwas vereinfachtes) SML-Programm implementiert:

```
fun DATA_CONF_TAC (conflict_pred) =
     REWRITE_TAC [conflict_pred,STAGE_RANK]
      THEN REPEAT STRIP_TAC
      THEN UNDISCH_ALL_TAC
      THEN REPEAT COND_CASES_TAC
      THEN CONV_TAC (REDEPTH_CONV num_CONV)
      THEN ASM_REWRITE_TAC [GREATER,LESS_REFL,LESS_REFL,LESS_0,
                                NOT_LESS_0,LESS_MONO_EQ,,CONJ_lemma]
      THEN REWRITE_TAC[SUB_0,SUB_EQUAL_0,LESS_EQ_0,SUB_MONO_EQ]
      THEN REPEAT STRIP_TAC
      THEN UNDISCH_TAC (--`0 < t0j - t0i`--)
      THEN ASM_REWRITE_TAC[NOT_LESS_0,NOT_LESS,LESS_EQ_0]
      THEN MAP_EVERY ASSUME_TAC
          [SPEC (--`r:resource`--) RESOURCE_CASES_COND_THM,
           SPEC (--`Ii:class`--) CLASS_CASES_COND_THM,
           SPEC (--`Ij:class`--) CLASS_CASES_COND_THM]
      THEN REPEAT UNDISCH_NO_TAC 0
      THEN REPEAT COND_CASES_TAC
      THEN ASM_REWRITE_TAC [STAGE_RANGE_thms STAGE_DOMAIN_thms];
```

Die Beweise der semantischen Korrektheit werden hauptsächlich mit Hilfe von zwei parametrisierten Taktiken (COMN_SPEC_TAC und PHASE_HW_TAC) realisiert [TaKu94a]. COMN_SPEC_TAC entspricht der in den Kapiteln 7.3.1 und 7.3.2 bereits beschriebenen Taktik $\mathcal{T}$ und wird durch folgendes (etwas vereinfachtes) SML-Programmstück implementiert:

```
fun COMN_SPEC_TAC (depth, spec, imp_list, context_list) =
    THEN MAP_EVERY EXISTS_TAC context_list
    THEN REPEAT (CONV_TAC let_CONV)
    THEN REWRITE_TAC [spec, imp_list]
    THEN REPEAT STRIP_TAC
    THEN MAP_EVERY ASM_REWRITE_TAC [[depth,R,W],Time_abs,
                                    [o_THM,EL,HD,TL,num_CONV(--`1`--)]]
    THEN BETA_TAC
    THEN CONV_TAC(DEPTH_CONV let_CONV)
    THEN ASM_REWRITE_TAC [ADD1,ADD_CLAUSES,ADD_SUB,SUB_0,
    THEN LEFT_ADD_DISTRIB,MULT_CLAUSES]
    THEN ASM_STRIP_LIST_TAC
    THEN ASM_SYM_LIST_TAC
    THEN ASM_REWRITE_TAC []
    THEN UNDISCH_ALL_TAC
    THEN CONV_TAC (REDEPTH_CONV num_CONV)
    THEN REWRITE_TAC [ADD_CLAUSES,LEFT_ADD_DISTRIB,ADD_SUB,
                     GSYM TIMES2,MULT_CLAUSES,ADD_ASSOC,ADD1]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC [];
```

Die Taktik PHASE_HW_TAC wird zur Korrektheit der Phasenebene verwendet und setzt sich aus existierenden Taktiken des MEPHISTO-Systems [KuSK93] wie folgt zusammen:

```
fun PHASES_HW_TAC (phase_spec, ebm_imp) =
     REWRITE_TAC[phase_spec,ebm_imp]
     THEN EXPAND_DEFS_TAC
     THEN FLAT_HW_TAC
     THEN COMB_ELIM_TAC
     THEN ASM_REWRITE_TAC[]
     THEN CONV_TAC (DEPTH_CONV let_CONV) ORELSE ALL_TAC
     THEN REPEAT STRIP_TAC
     THEN ASM_REWRITE_TAC[];
```

Die Verifikationsziele der Pipeline-Korrektheitsbeweise (insgesamt 9 Sätze) sind für beliebige RISC-Prozessoren allgemein gültig und werden einmalig erstellt. Für die semantische Korrektheit müssen die Verifikationsziele hingegen für jeden Befehl der verschiedenen Interpreterebenen gesetzt werden. Die Anzahl der zu verifizierenden Sätze ist zwar proportional zu der Summe der Befehle der Klassen-, Stufen und Phasenebenen, aber diese Beweisziele weisen ähnliche Muster auf, so daß sie durch parametrisierte SML-Funktionen automatisch gesetzt werden können. Hierfür wurden im Rahmen dieser Arbeit drei Funktionen (mk_class_goal, mk_stage_goal und mk_phase_goal) implementiert, welche alle Verifikationsziele der entsprechenden Ebene erzeugen. Beispielsweise sieht die Implementierung der Funktion mk_phase_goal der Phasenebene wie folgt aus:

```
fun mk_phase_goal  (abs_func imp spec_list) =
      let val t1 = list_mk_conj (map PRED spec_list)
          val term = mk_imp {ant = t1, conseq = (PRED imp)} in
          set_goal ([], list_mk_forall (abs_func, term))
      end;
```

8.4 Einbettung in das gesamte System

Der soweit implementierte Spezifikations- und Verifikationsrahmen wurde nicht nur als allein stehendes Werkzeug implementiert, sondern zusätzlich in ein umfassendes Verifikationssystem — MEPHISTO [KrKS93] — integriert. Das MEPHISTO-System stellt einen Rahmen zur Hardware-Verifikation dar und besteht aus einer Vielzahl von Modulen. Es basiert auf dem HOL-Theorembeweiser und ist an die kommerzielle Entwurfsumgebung CADENCE [Cade91] eingebunden[9]. Durch diese Anbindung ist beim MEPHISTO-System die Möglichkeit gegeben, formale Hardware-Beschreibungen aus schematischen Eingaben in CADENCE automatisch abzuleiten, indem ein Zwischenformat für Netzlistenbeschreibungen — EDIF [EDIF87] — verwendet wird. Darüberhinaus verfügt das MEPHISTO-System über eine Reihe von Taktiken, welche die Beweisautomatisierung von Hardware-Schaltungen auf der RT-Ebene unterstützen (s. die Taktik PHASE_HW_TAC oben).

8.5 DLX-Verifikation

Zur Validierung der in dieser Arbeit vorgestellten Verifikationsmethoden und Techniken wurde im Rahmen dieser Arbeit ein realer RISC-Prozessor — DLX [HePa90] — implementiert und verifiziert. Der DLX ist ein allgemeiner RISC-Prozessor, dessen Architektur von den RISC-Pionieren Hennessy und Patterson entworfen wurde [HePa90]. Folgende Eigenschaften haben unter anderem zur Auswahl dieses Prozessors als Verifikationsbeispiel beigetragen:

- vorhandene, fest definierte und gut dokumentierte DLX-Architektur [HePa90],
- der DLX enthält eine einfache, prototypische Basisarchitektur, die dem Rahmen einer akademischen Arbeit angemessen ist,
- der DLX umfaßt Eigenschaften von herkömmlichen RISC-Prozessoren, wie Intel i860 [Inte89], Sun SPARC [SunM87], Motorola M88100 [Moto88] und MIPS R3000 [Kane88] und ist also damit als ein sehr praxisnahes Beispiel zu betrachten,
- es existieren bereits Implementierungen von Varianten des DLX-Prozessors, welche unter Verwendung unterschiedlicher Werkzeuge, wie VHDL [IEEE93] oder GENESIL [Ment92] für verschiedene Zwecke wie Leistungs-

---

[9] Das MEPHISTO-System ähnelt weiteren Verifikationsrahmen, wie z.B. der SSVE-Verifikationsumgebung [SGSE92] oder dem SDVS-Verifikationssystem [LFMM92]. SSVE besteht aus der Spezifikationssprache ELLA [ELLA86] und den Beweisern FUNNEL, OBJ3 und 2OBJ [SGSE92]. Das SDVS stellt einen Rahmen für die Spezifikationssprachen Ada [USDD83], ISPS [Barb81] und VHDL [IEEE93] sowie den State-Delta-Beweiser [Marc91] dar.

analyse, Simulation oder Synthese [GuSt91, BlNS92, Wint92, Ashe93, CAOV93] realisiert wurden.

Der im Rahmen dieser Arbeit implementierte und verifizierte DLX RISC-Prozessor hat einen Befehlssatz von 51 Befehlen, besitzt eine Pipeline mit 5 Stufen und wird mit einem Zweiphasentakt betrieben. Die 51 Befehle werden durch drei Befehlsformate kodiert, umfassen elementare Befehle eines herkömmlichen RISC-Kernes und gliedern sich in 5 Klassen: arithmetisch-logische Registerbefehle, arithmetisch-logische unmittelbare Befehle, Ladebefehle, Speicherbefehle und Kontrollbefehle.

Die Hardware-Implementierung des DLX-Prozessors wurde mit Hilfe des Standardzellenentwurfssystems CADENCE [Cade91] in einer 1,0 µm CMOS-Technologie realisiert und läuft mit einer Taktfrequenz von 12,5 MHz. Die Hardware-Realisierung umfaßt 150.000 Funktionstransistoren, benötigt eine Chipfläche von ungefähr 62,5 mm^2 und hat 172 I/O Pads (s. Bild 8-1). Eine ausführliche Dokumentation der Implementierung des DLX in CADENCE ist in [DeTa94] beschrieben.

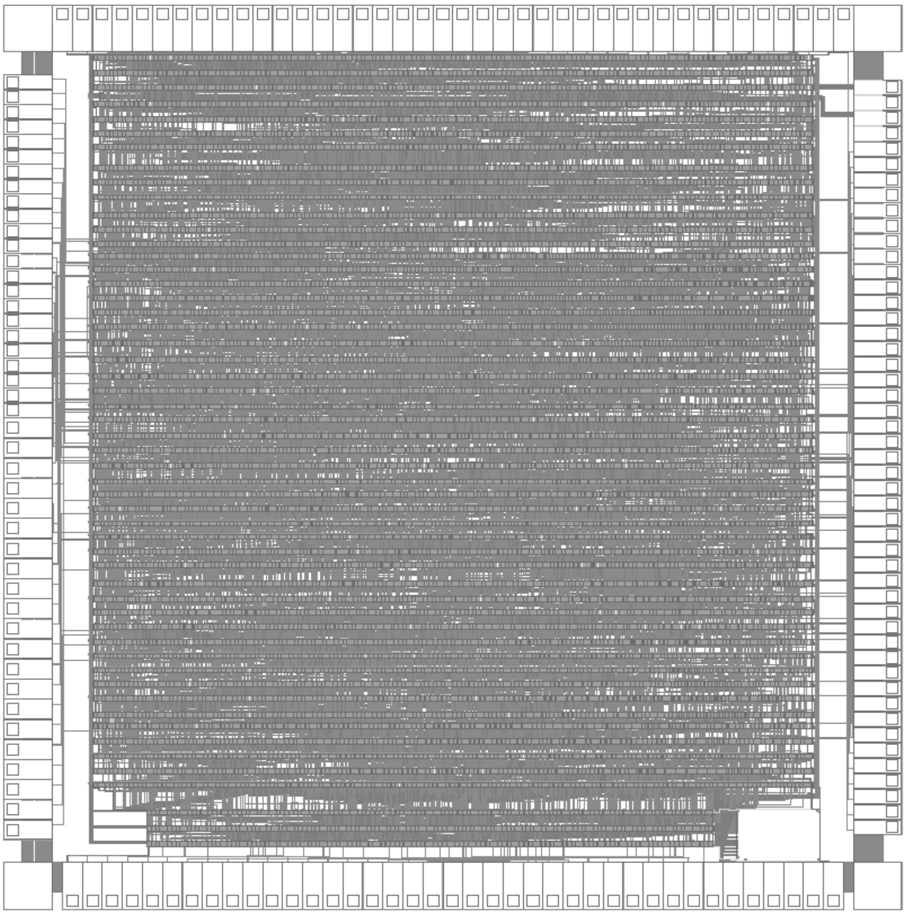


Bild 8-1: Layout des implementierten DLX-Prozessors

Der Befehlssatz, die Befehlsformate, die Pipelinestruktur, die Mikroarchitektur, die Chip-Pinbelegung und die Chip-Systemumgebung des implementierten DLX-RISC können dem Anhang A entnommen werden. Die Größen der verschiedenen Spezifikationsbeschreibungen werden anhand der benötigten Anzahl der Codezeilen und der entsprechenden Dateigrößen in Tabelle 8-1 veranschaulicht. Auffallend ist hierbei die Größe der Implementierungsbeschreibung EBM, welche ungefähr 70% der gesamten formalen Spezifikation des ganzen Prozessors ausmacht.

Tabelle 8-1: Spezifikationbeschreibungen des DLX in HOL

Spezifikation	# Zeilen	# Bytes	Bemerkungen
Architekturebene	495	18886	51 Befehle
Klassenebene	161	6289	5 Befehle
Stufenebene	218	8363	13 Befehle
Phasenebene.	213	8024	26 Befehle
EBM	2764	105578	-
Ressourcenkonflikt	481	15932	-
Datenkonflikte	172	6714	-
Kontrollkonflikte	47	1816	-
Hilfsfunktionen	1846	61096	-
Σ Spezifikation	4911	178776	-

Der gesamte Verifikationsaufwand für den DLX-Prozessor ist in Tabelle 8-2 durch die Laufzeiten der Beweise und die Anzahl der erzeugten Inferenzen auf einer Sun SPARC10 Maschine mit 128 MB Hauptspeicher angegeben. Hierbei ist es beachtenswert, daß der Verifikationsaufwand der Pipeline-Korrektheit das zwanzigfache des Aufwandes der semantischen Korrektheit beträgt. Die detaillierten Verifikationsdaten der verschiedenen Beweise der Pipeline- bzw. semantischen Korrektheit können dem Anhang B entnommen werden.

Tabelle 8-2: Gesamte Verifikationsergebnisse

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Pipeline-Korrektheit	8312, 13	20.283.706	9 Haupttheoreme
Semantische Korrektheit	457, 66	121.997	44 Haupttheoreme
Σ Verifikation	8769, 79	20.405.703	53 Haupttheoreme

Die formalen Spezifikationen der Architektur-, Klassen-, Stufen- und Phasenebenen dieses implementierten DLX-Prozessors sowie die formale Beschreibung von dessen Hardware EBM auf Gatterebene sind in [Deho94] beschrieben.

Kapitel 9: Zusammenfassung und Ausblick

9.1 Erzielte Ergebnisse

In der vorliegenden Arbeit wurde erstmals eine Methode entwickelt, welche basierend auf einem hierarchischen Modell, einen allgemeinen Rahmen zur formalen Spezifikation und Verifikation von herkömmlichen RISC-Prozessorkernen beschreibt. Ein wichtiger Aspekt dieser Arbeit ist der praktische Einsatz formaler Methoden in einer VLSI-Entwurfsmethodik. In dieser Arbeit wird die Logik höherer Ordnung wegen ihrer Ausdrucksmächtigkeit verwendet. Aufgrund der Nichtvollständigkeit und der Nichtentscheidbarkeit dieser Logik werden die Kosten der Verifikation nicht durch die Geschwindigkeit des Theorembeweisers, sondern durch die Zeit des Experten dominiert. Jedoch können fertige Beweise leicht modifiziert werden, um sie kleinen Hardware-Veränderungen anzupassen. Im Vergleich dazu benötigt eine Veränderung bei der Simulation einen kompletten Durchlauf der Testfolgen, was sehr zeitintensiv ist. Langfristig lohnen sich daher die Anfangskosten der formalen Methoden.

Darüberhinaus werden RISC-Prozessoren zunehmend bei sicherheitskritischen Anwendungen eingesetzt, die eine korrekte Implementierung erfordern. Bei vielen sicherheitskritischen Anwendungen spielen Zuverlässigkeitseigenschaften eine größere Rolle als die Funktionalität [TuBH87], und es werden deshalb möglichst einfache aber korrekte Prozessoren akzeptiert. Mit den Ergebnissen dieser Arbeit ist die Möglichkeit gegeben, RISC-Prozessorkerne, wie sie eventuell in eingebetteten Steuersystemen eingesetzt werden können, formal zu spezifizieren und zu verifizieren.

Das entwickelte hierarchische Interpreter-Modell entspricht der Natur eines RISC-Prozessorentwurfs und berücksichtigt vor allem die parallele Pipelineverarbeitung der RISC-Prozessoren. Zur Beherrschung der Verifikationskomplexität werden Befehle mit ähnlicher Semantik in Klassen zusammengefaßt, wonach die verschiedenen Abstraktionsebenen parametrisiert werden.

Die formalen Spezifikationen der verschiedenen Befehle der Interpreterebenen (Architektur-, Klassen-, Stufen- und Phasenebene) weisen eine einheitliche Form auf und können leicht aus nichtformalen Beschreibungen der Architektur bzw. der Pipelinestruktur abgeleitet werden. Zur Beschreibung der temporalen Gegebenheiten einer Pipelineverarbeitung wurde ein Zeitabstraktionsmechanismus entwickelt, welcher abstrakte und implementierungsunabhängige Spezifikationen auf verschiedenen Zeitgranularitäten gestattet. Dadurch gibt das Modell einem Prozessorentwerfer die Möglichkeit, einen Entwurf schrittweise zu verfeinern und zu spezifizieren. Das hierarchische Modell und die entsprechende Formalisierung sind allgemein und können für verschiedene Zwecke, wie Verifikation, Simulation oder formale Synthese verwendet werden.

Die Verifikation eines RISC-Prozessors besteht im allgemeinen einerseits aus dem Beweis der korrekten Implementierung der Maschinenbefehle durch die Hardware, andererseits aus der Korrektheitsprüfung der Pipelineabarbeitung der Befehle. Basierend auf diesem hierarchischen Modell wird der Korrektheitsbeweis eines RISC-Prozessors in zwei unabhängigen Schritten realisiert. Im ersten Schritt wird die Korrektheit der Pipelineverarbeitung bewiesen, indem unter Annahme von Software-Constraints einer gegebenen Architektur gezeigt wird, daß die Hardware keine Ressourcen-, Daten- oder Kontrollkonflikte zuläßt. Im zweiten Schritt wird die semantische Korrektheit der Befehle verifiziert, indem hierarchisch gezeigt wird, daß die Hardware-Implementierung die Semantik jedes Maschinenbefehls erfüllt.

Dank des hierarchischen RISC-Modells weisen einzelne Verifikationsaufgaben einheitliche Muster auf, wodurch eine Automatisierung der Beweisdurchführung ermöglicht wird. Hierfür wurde eine Reihe von parametrisierten Taktiken entwickelt, die unabhängig von der gegebenen Architektur allgemein anwendbar sind. Darüberhinaus ist das implementierte Verifikationsverfahren konstruktiv, d.h. bei einem Fehlschlag des Korrektheitsbeweises werden nützliche Hinweise auf das Fehlverhalten ausgegeben, um entsprechende Veränderungen am Entwurf vornehmen zu können. Die gesamte Verifikationsmethode kann sowohl zur nachträglichen Verifikation eines Entwurfs eingesetzt als auch in einem verifikationsbasierten Entwurf verwendet werden.

Die vorgestellten Techniken und Verfahren wurden im HOL-System als Rahmen zur Spezifikation und Verifikation von RISC-Prozessoren implementiert. Dieser Rahmen wurde zusätzlich in das Hardware-Verifikationssystem MEPHISTO integriert, welches in kommerzielle Entwurfsumgebungen eingebunden ist. Zur Validierung der entwickelten Spezifikations- und Verifikationsmethoden wurde im Rahmen dieser Arbeit schließlich ein RISC-Prozessor der Größenordnung herkömmlicher RISC-Kerne implementiert, formal spezifiziert und verifiziert.

9.2 Weiterführende Arbeiten

Vollständig verifizierte RISC-Chips gibt es bis heute noch nicht. Als wesentlicher Schritt in diese Richtung wurde in dieser Arbeit ein erster Baustein gelegt, in dem RISC-Prozessorkerne ausführlich behandelt werden. Die Behandlung von zusätzlichen Einheiten auf dem Chip, wie FPU, Cache oder MMU kann im Rahmen von weiterführenden Arbeiten geschehen. In der Literatur existieren schon einige Ansätze, die sich speziell mit der Verifikation von Gleitkommaeinheiten oder Cache-Modellen beschäftigen. Hierbei wird das Verhalten der Funktionseinheit formal spezifiziert und auf eine korrekte Implementierung durch die gegebene Hardware verifiziert. Unter Betrachtung des modularen Entwurfes und der modularen Realisierung solcher Funktionseinheiten werden diese als eigenständige Blockeinheiten mit einer festgelegten Schnittstelle betrachtet und verifiziert.

Heutige RISC-Prozessoren zeichnen sich zusätzlich durch eine zunehmende Anwendung neuer Techniken wie Superpipelining, Superskalar oder VLIW aus.

Superpipeline-Prozessoren steigern den Grad der Parallelität durch eine größere Pipelinetiefe als konventionelle RISC-Prozessoren, wobei die Dauer einer Stufenoperation im allgemeinen geringer als die längste Ausführungszeit der einzelnen Funktionseinheiten ist. Diese Art von Prozessoren wird durch die entwickelten Methoden dieser Arbeit abgedeckt, da die vorgestellten Verfahren und Techniken unabhängig von der Pipelinetiefe und für beliebige Stufenoperationen anwendbar sind.

Superskalare Architekturen zeichnen sich dadurch aus, daß mehrere Ausführungseinheiten parallel mehrere Befehle gleichzeitig verarbeiten können. Die Ausführungseinheiten haben meist eine unterschiedliche Funktionalität und können Befehle durch entsprechende Pipelines verarbeiten. Maschinenbefehle werden hierfür gemäß den Funktionseinheiten in Gruppen zusammengefaßt, die von einem Verwalter (“scheduler”) dekodiert und auf die Ausführungseinheiten verteilt werden. Die Idee der Klassenabstraktion von Maschinenbefehlen, die im RISC-Modell benutzt wird, kann durch Ausnutzung der gegebenen Befehlsgruppen hier auch verwendet werden. Ein Modell für superskalare Prozessoren muß jedoch, gemäß der nebenläufigen Funktionseinheiten, für jede Befehlsklasse eine entsprechende Stufen- bzw. Phasenebene vorsehen. Die Hardware-Implementierung muß somit die Korrektheit aller Stufen- und Phasenebenen erfüllen.

Viel schwieriger als die Modellierung und die Verifikation der semantischen Korrektheit von superskalaren Architekturen ist die Verifikation der korrekten Befehlsbearbeitung, denn hier muß zusätzlich zur Pipelineverarbeitung der einzelnen Ausführungseinheiten die Parallelität ganzer Funktionsblöcke betrachtet werden. Die Ressourcen-, Daten- und Kontrollkonflikte müssen hierbei einerseits auf der Ebene der Befehlspipelines der Funktionseinheiten, andererseits auf der höheren Abstraktionsebene der Befehlsgruppen spezifiziert und verifiziert werden.

VLIW-Prozessoren sind durch ein sehr langes Befehlwort gekennzeichnet. Hierbei wird eine vorgegebene Anzahl von Maschinenbefehlen als eine Befehlsgruppe in einem einzigen Format zusammengefaßt und kodiert. VLIW-Befehle werden einheitlich geholt und dekodiert, aber ähnlich wie bei superskalaren Prozessoren durch verschiedene parallele Funktionseinheiten ausgeführt. Wegen der gegebenen Befehlskodierung geht bei VLIW-Prozessoren zusätzlich zur Schwierigkeit der Behandlung der mehrfachen Parallelitäten die Unterteilung der Befehle in Befehlsklassen verloren.

Anhang A: DLX-Prozessor

A.1 Befehlssatz

Befehlsart	Bedeutung
Datentransfer	**Bewegen von Daten zwischen Speicher und Registersatz, oder dem special Register**
LB, LBU, SB	Lade Byte, Lade Byte vorzeichenlos, speichere Byte
LH, LHU, SH	Lade Halbwort, lade Halbwort vorzeichenlos, speichere Halbwort
LW, SW	Lade Wort, speichere Wort
MOVI2S, MOVS2I	Von special/integer-Register nach integer/special-Register
Arithmetisch, Logisch	**Operationen mit Integerzahlen oder Registerinhalten, vorzeichenbehaftete Operationen können eine Unterbrechung auslösen**
ADD, ADDI, ADDU, ADDUI	Addiere, addiere unmittelbar (je 16 Bit), ohne und mit Vorzeichen
SUB, SUBI, SUBU, SUBUI	Subtrahiere, subtrahiere unmittelbar, ohne und mit Vorzeichen
AND, ANDI	Logisches UND, unmittelbar
OR, ORI, XOR, XORI	Logisches ODER, bzw. XOR, unmittelbar
LHI	Lädt das obere Halbwort mit 16-Bit unmittelbaren Wert
SLL, SRL, SRA, SLLI, SRLI, SRAI	Schieben nach links, logisch, unmittelbar, nach rechts, logisch oder arithmetisch, unmittelbar
S__, S__I	Setze Bedingung, hier EQ, NE, LT, GT, LE, GE
Kontrollbefehle	**Bedingte und direkte Sprünge; relativ zum PC oder über Register**
BEQZ, BNEZ	Springe wenn Register (un)gleich Null, 16-Bit Wert relativ zum PC
J, JR	Sprung, 26-Bit Wert relativ zum PC (J) oder mit Zieladresse von einem Register (JR)
JAL, JALR	Springe und sichere Adresse nach Register R31, 26-Bit Wert relativ zum PC (JAL) oder mit Zieladresse von einem Register (JLAR)
TRAP	Unterprogrammaufruf über Vektoradresse
RFE	Rückkehr von einer Unterbrechung

A.2 Befehlsformate

I-type instruction

6	5	5	16
OpCode	rs1	rd	immediate value

R-type instruction

6	5	5	5	11
OpCode	rs1	rs2	rd	function field

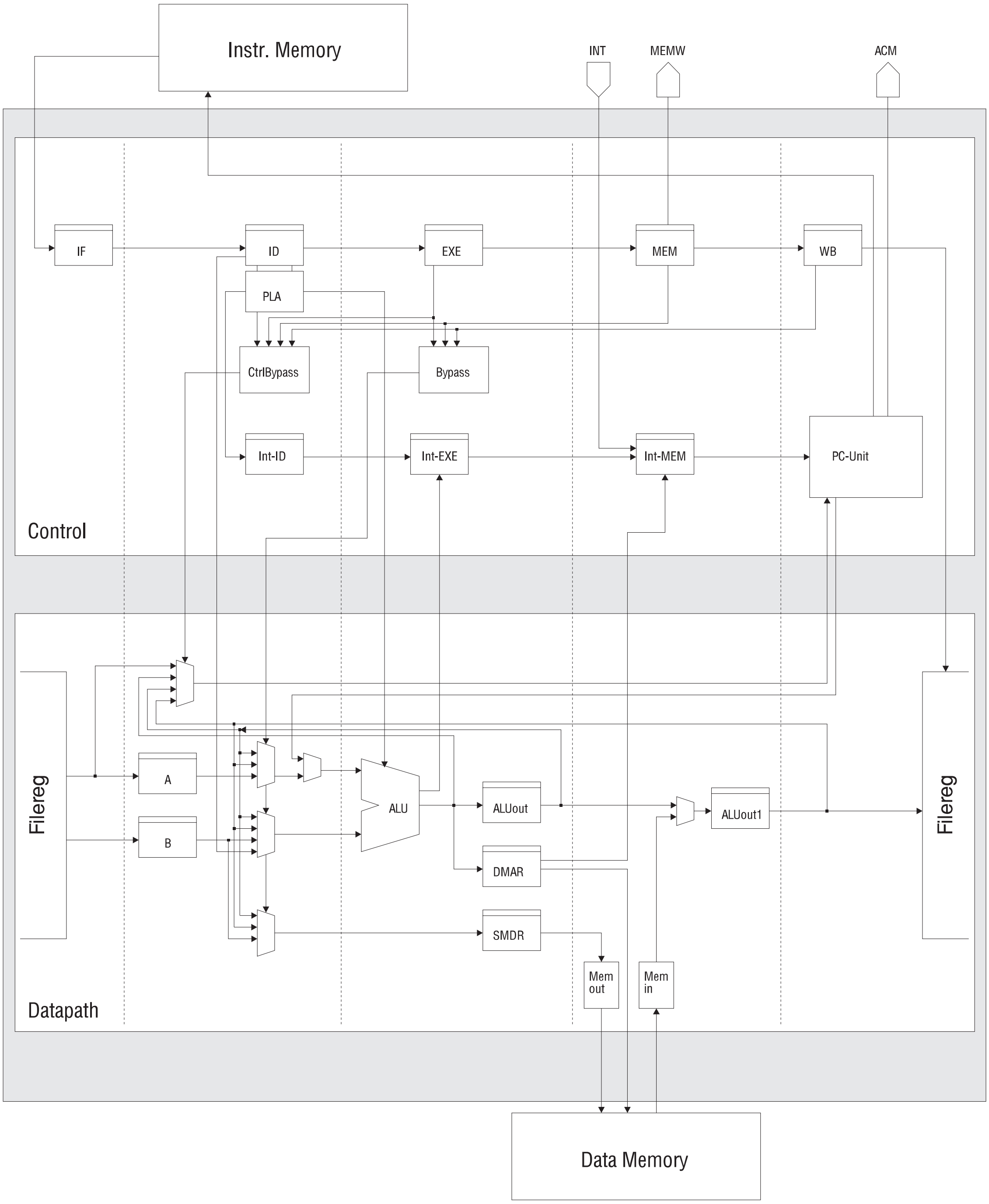
J-type instruction

6	26
OpCode	offset added to PC

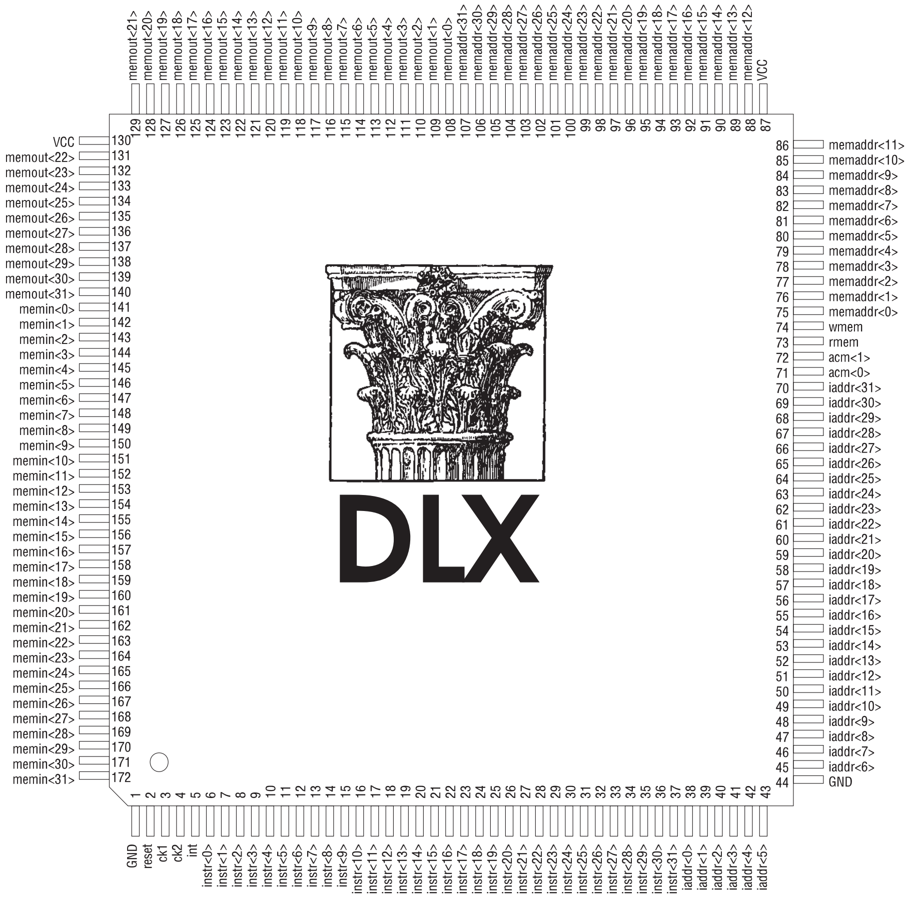
A.3 Pipelinestruktur

	ALU	**ALU_I**	**LOAD**	**STORE**	**CONTROL**
IF	IR ← I–M[PC] PC ← PC+4	IR ← I–M[PC] PC ← PC+4	IR ← I–M[PC] PC ← PC+4	IR ← I–M[PC] PC ← PC+4	IR← I–M[PC] PC ← PC+4
ID	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$BTA \xleftarrow[\phi_1]{} f\ (PC)$ $PC \xleftarrow[\phi_2]{} BTA$
EX	ALUOUT ← A op B	ALUOUT←A op (IR1)	MAR ← A+(IR1)	MAR← A+(IR1) SMDR ← B	
MEM	ALUOUT1←ALUOUT	ALUOUT1←ALUOUT	LMDR←D-M[MAR]	D-M[MAR]←SMDR	
WB	$RF[rd] \xleftarrow[\phi_1]{} ALUOUT1$	$RF[rd] \xleftarrow[\phi_1]{} ALUOUT1$	$RF[rd] \xleftarrow[\phi_1]{} LMDR$		

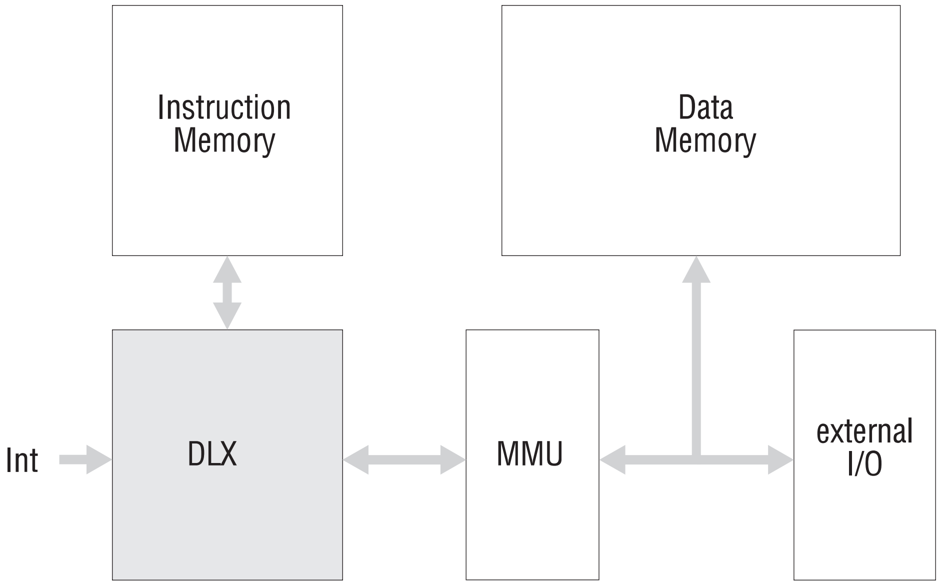
A.4 Mikroarchitektur



A.5 Pinbelegung des Chips



A.6 Systemumgebung



Anhang B: Experimentelle Ergebnisse

B.1 Verifikation der Pipeline-Korrektheit

B.1.1 Ressourcenkonflikte

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Dual_Stage_ Conflict	1691, 64	3.330.880	0 Konflikte
Dual_Phase_Conflict	1688, 32	4.752.706	0 Konflikte
P_Dual ⇒ P_Multiple	1, 03	2.434	-
S_Dual ⇒ S_Multiple	0, 05	81	-
Σ Ressourcenkonflikte	3381, 04	8.086.101	
Mult_Res_Conflict	344, 50	1.069.831	0 Konflikte (ohne Expansion)

B.1.2 Datenkonfilikte

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Dual_RAW_Conflict	450, 76	1.293.950	15 Konflikte in RF (3 slots) und 5 Konflikte in PC (1 slot)
(durch SW-Scheduling)	458, 96	1.344.577	0 Konflikte
(durch Bypassing)	1, 89	5.438	0 Konflikte
Dual_WAW_Conflict	421, 06	1.176.753	0 Konflikte
Dual_WAW_Conflict	428, 87	1.188.762	0 Konflikte
Phase_RAW_Conflict	214, 28	588.525	5 Konflikte by PC (1slot)
Phase_WAR_Conflict	221, 19	588.524	0 Konflikte
Phase_WAW_Conflict	228,15	588.572	0 Konflikte
Dual ⇒ Multiple (RAW)	0, 18	109	-
Dual ⇒ Multiple (WAR)	0, 17	109	-
Dual ⇒ Multiple (WAW)	0, 18	109	-
Σ Datenkonflikte	2425, 69	6.775.424	-
Multiple_RAW_Conflict	18660, 26	50.165.579	21 Konflikte (125 Unterziele)
Multiple_WAR_Conflict	18618, 13	50.165.295	0 Konflikte
Multiple_WAW_Conflict	18512, 27	50.165.316	0 Konflikte

B.1.3 Kontrollkonflikte

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Σ Kontrollkonflikte	35, 16	95.590	5 Konflikte (1 slot)
(durch SW-Scheduling)	32, 44	91.409	0 Konflikte

B.1.4 Hilfsprädikate für Pipeline-Konflikte[1]

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Stage_Used	247, 26	768226	208 Theoreme generiert (13 Sb x 16 Sr)
Phase_Used	1654, 28	3756157	422 Theoreme generiert (26 Pb x 17 Pr)
Stage_Range	151, 00	133963	125 Theoreme generiert (5 Kb x 5 Ps x 5 Kr)
Stage_Domain	151,0 0	133963	125 Theoreme generiert (5 Kb x 5 Ps x 5 Kr)
Phase_Range	133, 35	267141	130 Theoreme generiert (13 Sb x 2 Tp x 5 Kr)
Phase_Domain	133, 35	267141	130 Theoreme generiert (13 Sb x 2 Tp x 5 Kr)
Σ Hilfstheoreme	2470, 24	5.326.591	-

B.2 Semantische Korrektheitsbeweise

Verifikationsziel	Laufzeit (s)	# Inferenzen	Bemerkungen
Befehle der Klassenebene	40, 32000	44.846	5 Theoreme
Befehle der Stufenebene	16, 84000	17.880	13 Theoreme
Befehle der Phasenebene	392, 37000	48.321	26 Theoreme
Befehle der Architekturebene (Instanz.)	4, 06218	5.399	51 Theoreme
Beweisinstanziierung der Klassenebene	4, 06873	5.525	51 Theoreme
Beweisinstanziierung der Stufenebene	0, 00065	13	5 Theoreme
Beweisinstanziierung der Phasenebene	0, 00715	13	5 Theoreme
Σ Semantische Korrektheit	457, 66000	121.997	-

B.3 Beweistaktiken

Taktiken	# Zeilen	# Bytes
Semantische Korrektheit	148	4578
Pipeline-Korrektheit	629	24331
Hilfsbeweise	566	13241
Σ Verifikationstaktiken	1343	42150

---

[1] Die Abkürzungen Kb, Sb und Pb bezeichnen die Klassen-, Stufen- bzw. Phasenbefehle. Kr, Sr, Pr stehen für Klassen, Stufen- bzw. Phasenressourcen. Ps und Tp bezeichnen Pipelinestufen bzw. Taktphasen.

Literaturverzeichnis

[ABDG91] Aharon, A.; Bar-David, A.; Dorfman, B.; Gofman, E.; Leibowitz, M.; Schwartzburd, V.: Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator; IBM Systems Journal, Vol. 30, No. 4, 1991, pp. 527-538.

[AbHL90] Abstract Hardware Limited: LAMBDA - Logic and Mathematics behind Design Automation; User and Reference Manuals, Version 3.1, 1990.

[Adva88] Advanced Mico Devices: Am29000 - 32-Bit Streamlined Instruction Processor, User's Manual; Advanced Micro Devices (AMD), Sunnyvale, CA, 1988.

[Aker78] Akers, S.: Binary Decision Diagrams; IEEE Transactions on Computers, Vol. C-27, No. 6, June 1978, pp. 509-516.

[ALAD92] Aelten, F.; Liao, S.; Allen, J.; Devadas, S.: Automatic Generation and Verification of Sufficient Correctness Properties for Synchronous Processors; Proc. IEEE International Conference on Computer Aided Design (ICCAD92), Santa Clara, CA, November 1992, pp. 183-187.

[Alle93] Allemand, M.: A Rewriting Based Method for the Formal Verification of Microprocessors; Proc. Conference on Hardware Description Languages and their Applications (CHDL93), Ottawa, Canada, April 1993, pp. 103-110, IFIP.

[ALLS94] Arora, T.; Leung, T.; Levitt, K.; Schubert, T.: Report on the UCD Microcoded Viper Verification Project; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 239-252.

[Ance86] Anceau, F.: The Architecture of Microprocessors; Addison-Wesley Publishing Company, 1986.

[Ande91] Anderson, R.: Design and Test Integration, The Reality Behind Fast Time-to-Market; Keynote Address at the 14th Annual IEEE Workshop on Design for Testability, Vail, Colorado, April 1991.

[Andr86] Adrews, P.: An Introduction to mathematical Logic and Type Theory: To Truth through Proof; Academic Press, 1986.

[Aror90] Arora, T.: The Formal Verification of the VIPER Microprocessor: EBM to Phase, Phase to Microcode Level; Master's thesis, University of California, Davis, 1990.

[Ashe93] Ashenden, P.: DLX VHDL Model; Department of Computer Science, University of Adelaide, Australia, November 1993.

[AVCM93] Angelo, C.; Verkest, D.; Claesen, L.; De Man, H.: On the Comparison of the HOL and Boyer-Moore for Formal Hardware Verification; Journal on Formal Methods in System Design, Vol. 2, 1993, pp. 45-72.

[BaFM85] Babiker, S.; Fleming, R.; Milne, R.: A Tutorial for LTS; Standard Telecommunication Laboratories Ltd., London Road, Harlow, Essex, UK, August 1985.

[Bake90] Baker, T.: Headroom and Legroom in the 80960 Architecture; Proc. 35th IEEE Computer Society International Conference (COMPCON90), San Francisco, CA, February, 1990, pp. 299-306.

[BaPa90] Bayol, C.; Paillet, J.: Using TACHE for Proving Circuits; In: Claesen, L. (Ed.), Formal VLSI Correctness Verification, Elsevier Science Publishers B. V. (North-Holland), 1990, pp. 83-88.

[Barb81] Barbaci, M.: Instruction Set Processor Specification (ISPS): The Notation and its Application; IEEE Transactions on Computers, Vol. C-30, No. 1, January 1981, pp. 24-40.

[Barr84] Barrow, H.: VERIFY: A Program for Proving Correctness of Digital Hardware Designs; Artificial Intelligence, Vol. 24, No. 1-3, December 1984, pp. 437-491.

[Barr89] Barret, G.: Formal Methods Applied to a Floating Point Number System; IEEE Transactions on Software Engineering, Vol. 15, No. 5, 1989, pp. 611-621.

[Basi91] Basin, D.: Extracting Circuits From Constructive Proofs; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[BaVe86] Bapat, S.; Venkatesh, G.: Reasoning About Digital Systems Using Temporal Logic; Proc. ACM/IEEE 23rd Design Automation Conference (DAC86), Las Vegas, NV, June1986, pp. 215-219.

[BaWe90] Baeten, J.; Weijland, W.: Process Algebra; Volume 18 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, UK, 1990.

[BCMD91] Burch, J.; Clarke, E.; McMilian, K.; Dill, D.; Hwang, L.: Symbolic Model Checking: 10^20 States and Beyond; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[BCPP88] Borrione, D.; Camurati, P.; Paillet, J.; Prinetto, P.: A Functional Approach to Formal Hardware Verification: The MTI experience; Proc. IEEE International Conference on Computer Design (ICCD88), Rye Brook, NY, October 1988, pp. 592-595.

[BeDM87] Berenbaum, A.; Ditzel, D.; McLellan, H.: Introduction to the CRISP Microprocessor; Proc. 32th IEEE Computer Society International Conference (COMPCON87), San Francisco, CA, February, 1987, pp. 86-90.

[Bemm93] Bemmerl, T.: Technology for TeraFLOPS Architectures; In: Spies, P. (Ed.), Proc. European Informatics Congress Computing Systems Architecture (Euro-ARCH93), Munich, Germany, October 1993, Informatik Aktuell, Springer Verlag, pp. 22-33.

[Benn87] Bennett, J.: A Methodology for Automated Design of Computer Instruction Sets; PhD. Thesis, Computer Laboratory, University of Cambridge, January 1987.

[Bere87] Berebaum, A.: Functional Simulation for the CRISP Microprocessor; Proc. International Conference on Very Large Scale Integration (VLSI87), Vancouver, Canada, August 1987, pp. 323-334.

[Bevi89] Bevier, W.: Kit and the Short Stack; Journal of Automated Reasoning, Vol. 5, 1989, pp. 519-530.

[BGGH92] Boulton, R.; Gordon, A.; Gordon, M.; Harrison, J.; Herbert, J.; Van Tassel, J.: Experience with Embedding Hardware Description Languages in HOL; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 129-156.

[BGHT91 Boulton, R.; Gordon, M.; Herbert, J.; Van Tassel, J.: The HOL Verification of ELLA Designs; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[BHMY89] Bevier, W.; Hunt, W.; Moore, S.; Young, W.: An Approach to System Verification; Journal of Automated Reasoning, Vol. 5, 1989, pp. 411-428.

[BiMa88] Billon, J.; Madre, J.: Original Concepts of Priam, an Industrial Tool for Efficient Formal Verification of Combinational Circuits; In: Milne, G. (Ed.), The Fusion of Hardware Verification and Synthesis, Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 487-501.

[BiMS89] Bickford, M.; Mills, C.; Schneider, E.: Clio: An Applicative Language-Based Verification System; Technical Report TR 15-11, Odyssey Research Associates, Inc., 301A Harris B. Dates Drive, Ithaca, NY 14850, March 1989.

[BJLM86] Birtwistle, G.; Joyce, J.; Liblong, B.; Melham, T.; Schekiwy, R.: Specification and VLSI; In: Milne, G. and Subramanyam, P. (Eds.), Formal Aspects of VLSI Design, Elsevier Science Publishers B.V. (North-Holland), 1986, pp. 83-97.

[Bläs87] Bläsius, K.: Deduktionssysteme; Oldenbourg Verlag, 1987.

[BlKr92] Blanck, G.; Krueger, S.: The SuperSPARC Microprocessor; Proc. 37th IEEE Computer Society International Conference (COMPCON92), San Francisco, CA, February, 1992, pp. 136-141.

[BlNS92] Blomkvist, M.; Nilsson, J.; Sagefalk, W.: A VLSI Implementation of the DLX Microprocessor; Department of Computer Engineering, Lund University, Sweden, September 1992.

[BoCF91] Borrione, D.; Collavizza, H.; Le Faou, C.: uSPEED: a Framework for Specifying and Verifying Microprocessors; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[Bode90] Bode, A.: RISC-Architekturen; BI-Wiss. Verlag, 1990.

[BoMo83] Boyer, R.; Moore, J.: Proof Checking, Theorem-Proving, and Program Verification; Technical Report No. 35, Institute for Computing Science and Computer Applications, University of Texas at Austin, January 1983.

[BoMo88] Boyer, R.; Moore, J.: A Computational Logic Handbook; Academic Press, 1988.

[Bowe87] Bowen, J.: The Formal Specification of a Microprocessor Instruction Set; Technical Monograph PRG-60, Computing Laboratory, Oxford University, 1987.

[BoYu91] Boyer, R.; Yu, Y.: Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor; Technical Report TR-91-33, Computer Sciences and Mathematics Departments, University of Texas at Austin, 1991.

[BrCD85] Browne, M.; Clarke, E.; Dill, D.: Checking the Correctness of Sequential Circuits; Proc. IEEE International Conference on Computer Design (ICCD85), 1985, pp. 545-548.

[BrHY92] Brock, B.; Hunt, W.; Young, W.: Introduction to a Formally Defined Hardware Description Language; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 3-36.

[BrLe90] Brown, M.; Leeser, M.: From Programs to Transistors: Verifying Hardware Synthesis Tools; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer Verlag, 1990, pp. 129-151.

[BrSW93] Brodnax, T.; Schiffli, M.; Watson, F.: The PowerPC 601 Design Methodology; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 248-252.

[Brya81] Bryant, R.: A Switch-Level Simulation Model for Integrated Logic Circuits; PhD. Thesis, Department of Electrical Engineering and Computer Science, Technical Report MIT/LCS/TR-259, Massachusetts Institute of Technology, 1981.

[Brya86] Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation; IEEE Transactions on Computers, Vol. C-35, No.8, August 1986, pp. 677-691.

[Brya90] Bryant, R.: Verification of Synchronous Circuits by Symbolic Simulation; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer Verlag, 1990, pp. 14-24.

[Buck92] Buckow, O.: Formale Spezifikation und (Teil-) Verifikation eines SPARC-kompatiblen Prozessors mit LAMBDA; Diplomarbeit, Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, Oktober 1992.

[BuMS80] Burstall, R.; McQueen, D.; Sannella, D.: HOPE: An Experimental Applicative Language; Proc. LISP Conference, August 1980.

[BuSo88] Bundschuh, B.; Sokolowsky, P.: Rechnerstrukturen und Rechnerarchitekturen; Vieweg Verlag, 1988.

[BuVe91] Busch, H.; Venzl, G.: Proof-Aided Design of Verified Hardware; Proc. 28th ACM/IEEE Design Automation Conference (DAC91), San Francisco, CA, June 1991, pp. 391-396.

[Cade91] Cadence Design Systems Inc.: CADENCE User Manuals; Cadence Design Systems Inc., October 1991.

[CaGM87] Camilleri, A.; Gordon, M.; Melham, T.: Hardware-Verification using Higher-Order Logic; In: Borrione, D. (Ed.), From HDL Description to Guaranteed Correct Circuit Designs, Elsevier Science Publishers B.V. (North-Holland), 1987, pp.43-67.

[CaJB79] Carter, W.; Joyner, W.; Brand, D.: Symbolic Simulation for Correct Machine Design; Proc. ACM/IEEE 16th Design Automation Conference, June 1979, pp.280-286.

[Cami88] Camilleri, A.: Simulation as an Aid to Verification Using the HOL Theorem Prover; Technical Report No. 150, Computer Laboratory, Cambridge University, October 1988.

[Camp90] Camposano, R.: Behavioral-Preserving Transformations for High-Level Synthesis; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer-Verlag, 1990, pp. 106-128.

[CAOV93] CAO-VLSI Team: Implementation of DLX in ALLIANCE; MASI Laboratory, University Pierre et Marie Curie, Jussieu, Paris, France, March 1993.

[CaPr88] Camurati, P.; Prinetto, P.: Formal Verification of Hardware Correctness: Introduction and Survey of Current Research; IEEE Computer, July 1988, pp.8-19.

[CCLO88] Crocker, S.; Cohen, E.; Landauer, S.; Orman, H.: Reverification of a Microprocessor; Proc. IEEE Symposium on Security and Privacy, April 1988, Oakland, California Computer Society Press, Washington, D.C., 1988, pp 166-176.

[CFHM93] Cook, T.; Franzon, P.; Harcourt, E.; Miller, T.: System-Level Specification of Instruction Sets; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 552-557.

[CGED89] Computer General Electronic Design: The ELLA Language Reference Manual, Issue 4.0; Computer General Electronic Design, Chippenham, Wiltshire, U.K., 1989.

[CGLT89] Cohn, R.; Gross, T.; Lam, M.; Tseng, P.: Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor; Proc. third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, 1989.

[ChCo93] Chazarain, J.; Collavizza, H.: Combining Symbolic Evaluation and Object Oriented Approach for Verifying Processor-like Architectures at the RT-level; In: Milne, G. and Pierre, L. (Eds.), Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, Springer Verlag, 1993, pp. 109-121.

[Chin90] Chin, S.: Combining Engineering Vigor with Mathematical Rigor; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer Verlag, 1990, pp. 152-176.

[Chow89] Chow, P.: The MIPS-X RISC Microprocessor; Kluwer Academic Publishers, 1989.

[Chur36] Church, A.: A Note on the Entscheidungsproblem; Journal of Symbolic Computation, Vol. 1, 1936.

[Chur41] Church, A.: The Calculi of lambda-conversion; Annals of Mathematics Studies, No. 6, Princeton University Press, 1941.

[CiGo92] Circello, J.; Coodrich, F.: The Motorola 68060 Microprocessor; Proc. 37th IEEE Computer Society International Conference (COMPCON92), San Francisco, CA, February, 1992, pp. 73-78.

[CiVa87] Ciminiera, L.; Valenzano, A.: Advanced Microprocessor Architecture; Addison-Wesley, 1987.

[ClLM89] Clarke, E.; Long, D.; McMillan, K.: A Language for Compositional Specification and Verification of Finite State Hardware Controllers; In: Darringer, J. and Rammig, J. (Eds.), Computer Hardware Description Language and their Applications (CHDL89), Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 281-295.

[Cloc87] Clocksin, W.: Logic Programming and Digital Circuits Analysis; The Journal of Logic Programming, Vol. 4, No. 1, March 1987, pp. 59-82.

[ClTh93] Cloutier, R.; Thomas, D.: Synthesis of Pipelined Instruction Set Processors; Proc. ACM/IEEE 30th Design Automation Conference (DAC93), Dallas, Texas, June 1993, pp. 583-588.

[Cohn88] Cohn, A.: A Proof of the Viper Microprocessor: The First Level; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988.

[Cohn89a] Cohn, A.: Correctness Properties of the Viper Block Model: The Second Level; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1989, pp. 1-91.

[Cohn89b] Cohn, A.: The Notion of Proof in Hardware Verification; Journal of Automated Reasoning, Vol. 5, 1989, pp. 127-139.

[CoMa91] Coudert, O.; Madre, J.: Symbolic Computation of the Valid States of a Sequential Machine; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[Cons86] Constable, R. et al.: Implementing Mathematics with the Nuprl Proof Development System; Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cook90] Cook, J.: Verification of the C/30 Microcode Using the State Delta Verification System (SDVS); Proc. 13th National Computer Security Conference, Washington, D.C., National Bureau of Standards/National Computer Security Centre, October 1990, pp. 20-31.

[Cook92] Cooke, J.: Formal Methods — Mathematics, Theory, Recipes or what?; The Computer Journal, Vol. 35, No. 5, 1992, pp. 419-423.

[Craw90] Crawford, J.: The Execution Pipeline of the Intel i486 CPU; Proc. 35th IEEE Computer Society International Conference (COMPCON90), San Francisco, CA, February, 1990, pp. 254-263.

[Croc77] Crocker, S.: State Deltas: A Formalism for Representing Segments of Computation; PhD. Thesis, University of California, Los Angeles, 1977.

[Cull86] Cullyer, W.: VIPER: Correspondence Between Specification and Major State Machine; RSRE Report 86004, January 1986.

[Cull88] Cullyer, W.: Implementing Safety-Critical Systems: The VIPER Microprocessor; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988.

[Curz90] Curzon, P.: A Structured approach to the Verification of Low Level Microcode; PhD. Thesis, Computer Laboratory, University of Cambridge, May 1990.

[Deho94] Dehof, M.: Formale Spezifikation und Verifikation des DLX-RISC-Prozessors; Diplomarbeit, Institut für Technik der Informationsverarbeitung, Universität Karlsruhe, August 1994.

[DeMa89] De Man, H.: Transformational Design: A Case Study; In: Claesen, L. (Ed.), Applied Formal Methods for Correct VLSI Design, Elsevier Science Publishers B. V. (North-Holland), 1989, pp. 206-215.

[DeTa94] Dehof, M.; Tahar, S.: Implementierung des DLX RISC-Processors in einer Standardzellen-Entwufsumgebung; Technischer Bericht SFB 358-C2-9/94, Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, März 1994.

[DiAl92] Diefendorff, K.; Allen, M.: The Motorola 88110 Superscalar RISC Microprocessor; Proc. 37th IEEE Computer Society International Conference (COMPCON92), San Francisco, CA, February, 1992, pp. 157-162.

[Digi83] Digital Equipment Corp.: PDP-11 Architecture Handbook; Digital Equipment Corp., Maynard, Massachusetts, 1983.

[Digi86] Digital Equipment Corp.: VAX Hardware Handbook; Digital Equipment Corp., Maynard, Massachusetts, 1986.

[Digi92] Digital Equipment Corp.: Alpha Architecture Handbook; Digital Equipment Corp., Maynard, Massachusetts, Order No. EC-H1689-10, 1992.

[DTSP75] Davidson, E.; Thomas, A.; Shar, L.; Patel, J.: Effective Control for Pipelined Computers; Proc. 10th IEEE Computer Society International Conference (COMPCON75), San Francisco, CA, February, 1975, pp. 181-184.

[Ebci88] Ebcioglu, K.: Some Design Ideas for a VLIW Architecture for Sequential Natured Software; In: Cosnard, Barton, Vanneschi (Eds.), Proc. IFIP WG 10.3 Working Conference on Parallel Processing, North-Holland, 1988, pp. 3-21.

[EDIF87] Electronic Design Interchange Format, Version 2 0 0: EIA Interim Standard No. 44; EDIF Steering Committee, Electronic Industries Association, May 1987.

[Evek87] Eveking, H.: Verification, Synthesis and Correctness-Preserving Transformations - Cooperative Approaches to Correct Hardware Design; In: Borrione, D. (Ed.), From HDL Description to Guaranteed Correct Circuit Designs, Elsevier Science Publishers B.V. (North-Holland), 1987, pp. 229-239.

[Evek91a] Eveking, H.: Experience in Designing Formally Verifiable HDL's; In: Borrione, D. and Waxman, R. (Eds.), Computer Hardware Description Languages an their Applications (CHDL91), Elsevier Science Publishers B.V. (North-Holland), 1991.

[Evek91b] Eveking, H.: Verifikation digitaler Systeme; B.G. Teubner Stuttgart; 1991.

[FGJM85] Futatsugi, K.; Goguen, J.; Jouannaud, J.; Meseguer, J.: Principles of OBJ2; Proc. Symposium on the Principles of Programming Languages, 1985, pp. 52-66.

[Fili91] Filippenko, I.: VHDL Verification in the State Delta Verification System (SDVS); Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[Fitt90] Fitting, M.: First-order Logic and Automated Theorem Proving; Springer Verlag, 1990.

[FKTM86] Fujita, M.; Kono, S.; Tanaka, H.; Moto-Oka, T.: Aid to Hierarchical and Structural Logic Design Using Temporal Logic and Prolog; IEE Proceedings, Part E, Vol. 133, No. 5, September 1986, pp. 283-294.

[FoMa89] Fourman, M.; Mayger, E.: Formally Based System Design - Interactive Hardware Scheduling; In: Musgrave, G. and Lauther, U. (Eds.), Proc. International Conference on Very Large Scale Integration (VLSI89), Munich, Germany, August 1989, Elsevier Science Publishers B.V. (North-Holland), pp. 101-112.

[Four90] Fourman, M.: Formal System Design; In: Staunstrup, J. (Ed.), Formal Methods for VLSI Design, Elsevier Science Publishers B. V. (North-Holland), 1990, pp. 191-236.

[FuFu89] Fujita, M.; Fujisawa, H.: Specification, Verification and Synthesis of Control Circuits with Propositional Temporal Logic; In: Darringer, J. and Rammig, J. (Eds.), Computer Hardware Description Language and their Applications (CHDL89), Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 265-279.

[FuSG89] Fu, B.; Sani, A.; Gelsinger, P.: Performance and Microarchitecture of the i486 Processor; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, MA, September 1989, pp. 182-187.

[FuWS94] Fura, D.; Windley, P.; Somani, A.: Abstraction Techniques for Modeling Real-World Interface Chips; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 267-280.

[GaGu91] Garland, S.; Guttag, J.: A guide to LP, The Larch Prover; Digital Equipment Corporation Systems Research Centre Report 82, December 1991.

[GaKu83] Gajski, D.; Kuhn, R.: Guest Editors' Introduction: New VLSI Tools; Computer, Vol. 16, No. 12, December 1983, pp. 11-14.

[Gese89] Geser, A.: A Specification of the Intel 8085 Microprocessor: A Case Study; In: Wirsing, M. and Berstra, J. (Eds.), Algebraic Methods: Theory, Tools and Applications, Volume 394 of Lecture Notes in Computer Science, 1989, pp. 347-402.

[GFAM89] Gopalakrishnan, G.; Fujimoto, R.; Akella, V.; Mani, N.; Smith, K.: Specification-Driven Design of Custom Hardware in HOP; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1989, pp. 128-170.

[Gilo81] Giloi, W.: Rechnerarchitektur; Springer Verlag, 1981.

[GiMi87] Gimarc, C.; Milutinovic, V.: A Survey of RISC Processors and Computers of the Mid-1980s; Computer, Vol.9, No.9, September 1987.

[Göde30] Gödel, K.: Die Vollständigkeit der Axiome des logischen Funktionenkalküls; Monatshefte Mathematik und Physik, Vol. 37, 1930, pp. 349-360.

[Göde31] Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme; Monatshefte Mathematik und Physik, Vol. 1, 1931, pp. 349-360.

[GoMe93] Gordon, M.; Melham, T.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic; Cambridge, University Press, 1993.

[GoMW79] Gordon, M.; Milner, R.; Wadsworth, C.: Edinburgh LCF: A Mechanical Logic of Computation; Volume 78 of Lecture Notes in Computer Science, Springer Verlag, 1979.

[Goor89] Van De Goor, A.: Computer Architecture and Design; Addison-Wesley, 1989.

[Gord83a] Gordon, M.: LCF_LSM: A System for Specifying and Verifying Hardware; Technical Report No. 41, Computer Laboratory, Cambridge University, 1983.

[Gord83b] Gordon, M.: Proving a Computer Correct using the LCF_LSM Hardware Verification System; Technical Report No. 42, Computer Laboratory, University of Cambridge, September 1983.

[Gord86] Gordon, M.: Why Higher-Order Logic is a good Formalism for Specifying and Verifying Hardware; In: Milne, G. and Subrahmanyam, P. (Eds.), Formal Aspects of VLSI Design, Elsevier Science Publishers B.V. (North-Holland) 1986, pp.153-177.

[GoWi88] Goguen, J.; Winkler, T.: Introducing OBJ3; Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI international, Menlo Park, August 1988.

[GrBi90] Graham, B.; Birtwistle, G.: Formalising the Design of an SECD chip; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer Verlag, 1990, pp. 40-66.

[Gupt92] Gupta, A.: Formal Hardware Verification Methods: A Survey; Journal of Formal Methods in System Design, Vol. 1, No. 2/3, 1992, pp. 151-238.

[GuSt91] Gupta, A.; Stephan, P.: VHDL Design and Analysis of DLX; CS252 Semester Project, University of California at Berkeley, May 1991.

[HaDa85] Hanna, F.; Daeche, N.: Specification and Verification Using Higher-Order Logic; In: Koomen, C. and Moto-oka, T. (Eds.), Hardware Description Languages and Their Applications (CHDL85), Elsevier Science Publishers B.V. (North-Holland), 1985, pp. 418-433.

[HaDa92] Hanna, K.; Daeche, N.: The Veritas Design Logic: A User's View; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 301-310.

[HaHY92] Hamaguchi, K.; Hirashi, H.; Yajima, S.: Design Verification of a Microprocessor Using Branching Time Regular Temporal Logic; Proc. Workshop on Computed Aided Verification (CAV92), Montreal, Canada, June-July 1992, pp. 206-219.

[HaLD89] Hanna, F.; Longley, M.; Daeche, N.: Formal Synthesis of Digital Systems; In: Claesen, L. (Ed.), Applied Formal Methods for Correct VLSI Design, Elsevier Science Publishers B. V. (North-Holland), 1989, pp. 532-548.

[Hall90] Hall, A.: Seven Myths of Formal Methods; IEEE Software, September 1990, pp. 11-22.

[Händ77] Händler, W.: The Impact of Classification Schemes on Computer Architectures; Proc. International Conference on Parallel Processing, Detroit, MI, August 1977, pp. 7-15.

[HaTu93] Harman, N.; Tucker, J.: Algebraic Models and the Correctness of Microprocessors; In: Milne, G. and Pierre, L. (Eds.), Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, Springer Verlag, 1993, pp. 92-108.

[HaVZ90] Hamacher, V.; Vranesic, Z.; Zazy, S.: Computer Organisation; McGraw-Hill International Editions, 1990.

[HaWi93] Hall, K.; Windley, P.: Simulating Microprocessors from Formal Specifications; In: Claesen, L. and Gordon, M. (Eds.), Higher Order Logic Theorem Proving and its Applications, Elsevier Science Publishers B.V. (North-Holland), 1993, pp. 507-526.

[Haye88] Hayes, J.: Computer Architecture and Organization; McGraw-Hill, Second Edition, 1988.

[HCLR88] von Henke, F.; Crow, J.; Lee, R.; Rushby, J.; Whitehurst, R.: The EHDM Verification Environment: An Overview; Proc. 11th National Computer Security Conference, Baltimore, October 1988, pp. 147-155.

[HEKR90] Holden, K.; Eisele, R.; Kobe, M.; Raleigh, J.; Spohrer, T.: 68040 Integer Module; Proc. IEEE International Conference on Computer Design (ICCD90), Cambridge, MA, October 1990, pp. 183-186.

[Henn82] Hennessy, J. et al.: The MIPS Machine; Proc. 24th IEEE Computer Society International Conference (COMPCON82), San Francisco, CA, February, 1982, pp. 2-7.

[Henn84] Hennessy, J.: VLSI Processor Architecture; IEEE Transactions on Computers, Vol. c-33, No.12, December1984.

[HePa90] Hennessy, J.; Patterson, D.: Computer Architecture: A Quantitative Approach; Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Herb88] Herbert, J.: Temporal Abstraction of Digital Designs; In: Milne, G. (Ed.), The Fusion of Hardware Verification and Synthesis, Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 1-25.

[Herb91] Herbert, J.: Dealing With Temporal Complexity in Hardware Verification; Proc. HOL Theorem Proving System and its Applications, Davis, CA, August 1991, IEEE Computer Society Press, pp. 13-21.

[Herb92] Herbert, J.: Incremental Design and Formal Verification of Microcoded Microprocessors; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 157-174.

[Hewl86] Hewlett-Packard: Precision Architecture and Instruction Reference Manual; Hewlett-Packard Company (HP), 1986.

[HMSS87] Homewood, M.; May, D.; Shepherd, D.; Shepher, R.: The IMS T800 Transputer; IEEE Micro, Vol. 7, No. 5, October 1987.

[Hoar81] Hoare, C.: A Calculus of Total Correctness for Communication Processes; Science of Computer Programming, Vol. 1, No. 1, 1881.

[HoKu89] Hofbauer, D.; Kutsche, R.: Grundlagen des maschinellen Beweisens; Vieweg Verlag, 1989.

[HöMS87] Hörbst, E.; Müller-Schloer, C.; Schwärtzel, H.: Design of VLSI Circuits Based on VENUS; Springer Verlag, 1987.

[Hunt85] Hunt, W.: FM8501: A Verified Microprocessor; PhD. Thesis, Institute for Computing Science, University of Texas, Austin, December 1985.

[Hunt89] Hunt, W.: Microprocessor Design Verification; Journal of Automated Reasoning, Vol. 5, 1989, pp. 429-460.

[HuRo91] Huber, J.; Rosneck, M.: Successful ASIC Design the First Time Through; Van Nostrand Reinhold, 1991.

[HuZh93] Hua, G.; Zhang, H.: Formal Semantics of VHDL for Verification of Circuit Designs; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 446-449.

[HwBr84] Hwang, K.; Briggs, F.: Computer Architecture and Parallel Processing; McGraw-Hill, 1984.

[IBMc90] IMB Corporation: The IBM RISC System/6000 processor; IBM Journal of Research Development, Vol. 34, No. 1, January 1990.

[IEEE93] Institute of Electrical and Electronics Engineers: IEEE Standard VHDL Language Reference Manual; IEEE Press, New York, June 1993.

[Inte80] Intel Corporation: The 8080/8085 microprocessor book; Intel Marketing Communications, Chichester Wiley, 1980.

[Inte84] Intel Corporation: IAPX 286 Programmer's Reference Manual; Intel Corporation, Santa Clara, CA, 1984.

[Inte87] Intel Corporation: 80386 Hardware Reference Manual; Intel Corporation, Santa Clara, CA, 1987.

[Inte89a] Intel Corporation: i486 Microprocessor (Data Book); Intel Corporation, Santa Clara, CA, 1989.

[Inte89b] Intel Corporation: i860 64-Bit Microprocessor Programmer’s Reference Manual; Intel Corporation, Santa Clara, CA, 1989.

[Inte89c] Intel Corporation: 80960CA User's Manual; Intel Corporation, Santa Clara, CA, 1989.

[Inte93] Intel Corporation: Pentium Processor User's Manual; Intel Corporation, Santa Clara, CA, 1993.

[JLRS91] Joyce, J.; Liu, E.; Rushby, J.; Shankar, N.; Suaya, R.; von Henke, F.: From Formal Verification to Silicon Compilation; Proc. 36th IEEE Computer Society International Conference (COMPCON91), San Francisco, CA, February, 1991, pp. 450-455.

[JoBG86] Joyce, J.; Birtwistle, G.; Gordon, M.: Proving a Computer Correct in Higher Order Logic; Technical Report No. 100, University of Cambridge, Computer Laboratory, December 1986.

[JoSe94] Joyce, J.; Seger, C.: The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 185-198.

[JoWa89] Jouppi, N.; Wall, D.: Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines; Proc. 3rd international Conference on architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, 1989.

[Joyc88a] Joyce, J.: Formal Verification and Implementation of a Microprocessor; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 129-157.

[Joyc88b] Joyce, J.: Formal Specification and Verification of Microprocessor Systems; Technical Report No. 147, University of Cambridge, Computer Laboratory, September 1988.

[Joyc89] Joyce, J.: Multi-Level Verification of Microprocessor-Based Systems; PhD. Thesis, Computer Laboratory, Cambridge University, December 1989.

[Joyc91] Joyce, J.: More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[JYKH91] Jones, D.; Yang, R.; Kqong, M.; Harper, G.: Verification Techniques for a MIPS Compatible Embedded Control Processor; Proc. IEEE International Conference on Computer Design (ICCD91), Cambridge, Massachusetts, October 1991, pp. 329-332.

[KaAL91] Kalvala, S.; Archer, M.; Levitt, K.: A Methodology for Integrating Hardware Design and Verification; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[KaHe92] Kane, G.; Heinrich, J.: MIPS RISC Architecture; Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Kane88] Kane, G.: MIPS RISC Architecture; Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Kate84] Katevenis, M.: Reduced Instruction Set Computer Architectures for VLSI; MIT Press, Cambridge, Massachusetts, 1984.

[KaZh89] Kapur, D.; Zhang, H.: An Overview of RLL: Rewrite Rule Laboratory; Proc. RTA-89, Volume 355 of Lecture Notes in Computer Science, Springer Verlag, 1989, pp. 513-529.

[Keut91] Keutzer, K.: The Need for Formal Verification in Hardware Design and What Formal Verification Has Not Done for Me Lately; Proc. HOL Theorem Proving System and its Applications, Davis, CA, August 1991, IEEE Computer Society Press, pp. 77-86.

[Kham82] Khambata, A.: Introduction to the Z80 microcomputer; Wiley, New York, 1982.

[Kogg81] Kogge, P.: The Architecture of Pipelined Computers; McGraw-Hill, 1981.

[Koha78] Kohavi, Z.: Switching and Finite Automata Theory; McGraw-Hill, Computer Science Series, Second Edition, 1978.

[KrKS93] Kropf, Th.; Kumar, R.; Schneider, K.: Embedding Hardware Verification within a Commercial Design Framework; Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 93), Lecture Notes in Computer Science, Springer Verlag, 1993.

[KuSK93] Kumar, R.; Schneider, K.; Kropf, Th.: Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment; Journal of Formal Methods in System Design, Vol.2, No. 2, 1993, pp. 165-230.

[Land64] Landin, P.: The Mechanical Evaluation of Expressions; Computer Journal, Vol. 6, No. 4, 1964, pp. 308-320.

[Lang93] Lang, U.: Strukturen zukünftiger kommerzieller Hochleistungs-Rechensysteme; In: Spies, P. (Ed.), Proc. European Informatics Congress Computing Systems Architecture (Euro-ARCH93), Munich, Germany, October 1993, Informatik Aktuell, Springer Verlag, pp. 1-4.

[LCLL92] Lee, M.; Choi, B.; Lee, S.; Lee, K.: ASIC Design of a High Performance RISC; Proc. European Design Automation Conference (EDAC92), Brussels, Belgium, February 1992, pp. 262-265.

[Lesc83] Lescanne, P.: Computer Experiments with REVE Term Rewriting System Generator; Proc. 10th ACM Symposium on the Principles of Programming Languages, Austin, Texas, January 1983.

[Levy91] Levy, B.: An Overview of Hardware Verification Using the State Delta Verification System (SDVS); Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[LFMM92] Levy, B.; Filippenko, I.; Marcus, L.; Menas, T.: Using the State Delta Verification System (SDVS) for Hardware Verification; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B.V. (North-Holland), 1992, pp. 337-360.

[LoDi90] Loewenstein, P.; Dill, D.: Formal Verification of Cache Systems using Refinement Relations; Proc. IEEE International Conference on Computer Design (ICCD90), 1990, pp. 228-233.

[LoDi92] Loewenstein, P.; Dill, D.: Verification of a Multiprocessor Cache Protocol Using Simulation Relations and Higher-Order Logic; Journal of Formal Methods in System Design, Vol.1, No. 4, 1993, pp. 165-230.

[Loew91] Loewenstein, P.: Experiences Using a Theorem Prover for Hardware Verification; Proc. International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

[MaBi88] Madre, J.; Billon, J.: Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behavior; Proc. ACM/IEEE 25th Design Automation Conference (DAC88), Anaheim, CA, June 1988, pp. 205-210.

[MaFo91] Mayger, E.; Fourman, M.: Integration of Formal Methods with System Design; Proc. International Conference on Very Large Scale Integration (VLSI91), Edinburgh, 1991, pp. 3a.2.1-3a.2.11.

[Marc91] Marcus, L.: SDVS 10 Users' Manual; Technical Report ATR-91(6778)-10, The Aerospace Corporation, 1991.

[Marg90] Margulis, N.: i860 Microprocessor Internal Architecture; Microprocessors and Microsystems, Vol.14, No. 2, March 1990.

[MaSh87] May, D.; Shepperd, D.: Formal Verification of the IMS T800 Microprocessor; Internal Report, INMOS Limited, 1987.

[MayD90a] May, D.: Use of Formal Methods by Silicon Manufacturer; In: Hoare, C. (Ed.), Developments in Concurrency and Communications, Addison-Wesley, 1990.

[MayD90b] May, D.: Compiling Occam into Silicon; In: Hoare, C. (Ed.), Developments in Concurrency and Communications, Addison-Wesley, 1990.

[McAl93] McAllister, M.: Machine Abstraction in Microprocessor Specification; In: Claesen, L. and Gordon, M. (Eds.), Higher Order Logic Theorem Proving and its Applications, Elsevier Science Publishers B.V. (North-Holland), 1993, pp. 211-224.

[McCu90] McCune, M.: OTTER 2.0 User's Guide; Technical Report ANL-90/9, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL (USA), March 1990.

[McPC88] McFar,, M.; Parker, A.; Camposano, R.: Tutorial on High-Level Synthesis; Proc. ACM/IEEE 25th Design Automation Conference (DAC88), Anaheim, CA, June 1988, pp. 330-336.

[Mele88] Melear, C.: RISC Architecture of the M88000; Proc. IEEE International Conference on Computer Design (ICCD88), Rye Brook, NY, October 1988, pp. 370-373.

[Melh88] Melham, T.: Abstraction Mechanisms for Hardware Verification; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 129-157.

[Ment89] Mentor Graphics Inc.: GENESIL Designer Manuals; Mentor Graphics Inc., September 1989.

[Miln80] Milner, R.: A Calculus of Communicating Systems; Volume 92 of Lecture Notes in Computer Science, Springer Verlag, New-York, 1980.

[Miln83] Milner, R.: CIRCAL: A Calculus for Circuit Description; Integration, the VLSI Journal, Vol. 1, No. 2 and 3, 1983, pp. 121-160.

[Miln85] Milne, G.: Simulation and Verification: Related Techniques for Hardware Analysis; In: Koomen, C. and Moto-oka, T. (Eds.), Hardware Description Languages and Their Applications (CHDL85), Elsevier Science Publishers B.V. (North-Holland), 1985, pp. 404-417.

[Miln90] Milne, G.: Design for Verifiability; In: Leeser, M. and Brown, G. (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer-Verlag, 1990, pp. 1-13.

[Milu89] Milutinovic, V.: High Level Language Computer Architecture; Computer Science Press, Inc., 1989.

[MiLY90] Minato, S.; Ishiura, N.; Yajima, S.: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation; Proc. ACM/IEEE 27th Design Automation Conference (DAC90), Los Alamitos, CA, June 1990, pp. 52-57.

[MIPS91] MIPS Computer Systems, Inc.: MIPS R4000 Microprocessor User's Manual; MIPS Computer Systems, Inc., Sunnyvale, CA, 1991.

[Moor89] Moore, S.: A Mechanically Verified Language Implementation; Journal of Automated Reasoning, Vol. 5, 1989, pp. 461-492.

[Moor93] Moore, C.: The PowerPC 601 Microprocessor; Proc. 38th IEEE Computer Society International Conference (COMPCON93), San Francisco, CA, February 1993, pp. 109-116.

[Mosz83] Moszkowski, B.: Reasoning about Digital Circuits; PhD. Thesis, Department of Computer Science, Stanford University, 1983.

[Moto75] Motorola Inc.: M6800 Microprocessor Programming Manual; Motorola Semiconductor Products, 2. ed., Geneve Motorola, 1975.

[Moto85] Motorola Inc.: MC68020 32-Bit Microprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall,1985.

[Moto87a] Motorola Inc.: MC68030 Enhanced 32-bit Microprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall, 1987.

[Moto87b] Motorola Inc.: MC688 81/882 Floating-point Coprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall, 1987.

[Moto88] Motorola, Inc.: MC88100 RISC Microprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall, 1988.

[Moto89] Motorola Inc.: MC68040 32-bit third-generation Microprocessor; Motorola Inc., Phoenix, Arizona, 1989.

[MüSc91] Müller-Schloer, C.; Schnitter, E.: RISC-Workstation Architekturen; Springer Verlag, 1991.

[MWBS88] Malik, S.; Wang, A.; Brayton, R.; Sangiovanni-Vincentelli, A.: Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment; Proc. IEEE International Conference on Computer Aided Design (ICCAD88), Santa Clara, CA, November 1988, pp. 6-9.

[Nage75] Nagel, L.: SPICE2: A Computer Program to Simulate Semiconductor Circuits; Memo ERL-M520, University of California, Berkeley, 1975.

[NaSt88] Narendran, P.; Stillman, J.: Formal Verification of the Sobel Image Processing Chip; Proc. 25th ACM/IEEE Design Automation Conference (DAC88), Anaheim, CA, June 1988, pp. 211-217.

[NAUK93] Narita, S.; Arakawa, F.; Uchiyama, K.; Kawasake, I.: Design Methodology for GMICRO/500 TRON Microprocessor; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 253-257.

[OwSR93] Owre, S.; Shankar, N.; Rushby, J.: User Guide for the PVS Specification and Verification System, Language, and Proof Checker; Computer Science Laboratory, SRI International, Melno Park, CA, February 1993.

[PaDo89] Patel, P.; Douglass, D.: Architecture Feature of the i860 - Microprocessor RISC Core and on-Chip Caches; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, MA, September 1989, pp. 385-390.

[Pail89] Paillet, J.: Functional Semantics of Microprocessors at the Machine Instruction Level; In: Darringer, J. and Rammig, J. (Eds.), Computer Hardware Description Language and their Applications (CHDL89), Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 87-101.

[PaMo84] Palmer, J.; Morse, S.: The 8087 (Eight thousand eighty seven) Primer; New York, Willey, 1984.

[PanJ92] Pan, J.: Towards a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit; PhD. Thesis, Division of Computer Science, University of California, Davis, 1992.

[PaSe82] Patterson, D.; Séquin, C.: A VLSI RISC; Computer, September 1982.

[PaSE85] Patel, D.; Schlag, M.; Ercegovac, M.: vFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms; In: Jouannaud, J. (Ed.), Proc. Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985, Springer Verlag, pp. 238-255.

[Patt82] Patterson, D.: A RISCy Approach to Computer Design; Proc. 24th IEEE Computer Society International Conference (COMPCON82), San Francisco, CA, February, 1982, pp.8-14.

[Patt85] Patterson, D.: Reduced Instruction Set Computers; Communications of the ACM, Vol. 28, No. 1, January 1985.

[Paul85] Paulson, L.: The Revised Logic PPLAMBDA: a Reference Manual; Technical Report No. 36, Computer Laboratory, Cambridge University, November 1985.

[Paul91] Paulson, L.: ML for the Working Programmer; Cambridge University Press, 1991.

[Paul94] Paulson, L.: Isabelle: A Generic Theorem Prover; Lecture Notes in Computer Science 828, Springer Verlag, 1994.

[Pnue77] Pnueli, A.: The Temporal Logic of Programs; Proc. 18th. IEEE Symposium on Foundations on Computer Science, 1977, pp. 46-57.

[PuRT87] Pulou, J.; Rainard, J.; Thorel, P.: Microprocesseur à test intégré (MTI), description functionelle et architecture; Technical Report NT/CNS/CCI/59, CNET, Grenoble, January 1987.

[Pygo85] Pygott, C.: Formal Proof of Correspondence Between a Hardware Module and it's Gate-level Implementation; Memo No. 85012, Royal Signals and Radar Establishment, June 1985.

[Pygo88] Pygott, C.: NODEN HDL: An Engineering Approach to Hardware Verification; In: Milne, G. (Ed.), The Fusion of Hardware Verification and Synthesis, Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 211-229.

[Radi83] Radin, G.: The 801 Minicomputer; IBM Journal on Research Development, Vol. 27, No. 3, May 1983.

[RaLi75] Ramamoorthy, C.; Li, H.: Some Problems in Parallel and Pipeline Processing; Proc. 10th IEEE Computer Society International Conference (COMPCON75), San Francisco, CA, February, 1975, pp. 177-180.

[RaLi77] Ramamoorthy, C.; Li, H.: Pipeline Architecture; ACM Computing Survey, Vol.9, No. 1, March 1977, pp. 61-102.

[Ramm89] Rammig, F.: Systematischer Entwurf digitaler Systeme; Teubner Verlag, 1989.

[RoHo86] Roscoe, A.; Hoare, C.: The Laws of Occam Programming; Technical Monograph PRG-53, Computing Laboratory, Oxford University, February 1986.

[Sain93] Saini, A.: Design of the Intel Pentium Processor; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 258-261.

[Schö89] Schöning, U.: Logik für Informatiker; BI Wissenschaftsverlag, Reihe Informatik , Band 56, 1989.

[Schu91a] Schubert, E.: Verification of Integrated Subsystems; Proc. HOL Theorem Proving System and its Applications, Davis, CA, August 1991, IEEE Computer Society Press, pp. 38-51.

[Schu91b] Schubert, E.: Verification of Composed Hardware Systems Using CCS; Proc. HOL Theorem Proving System and its Applications, Davis, CA, August 1991, IEEE Computer Society Press, pp. 88-95.

[ScKK91] Schneider, K.; Kumar, R.; Kropf, Th.: Structuring Hardware Proofs: First steps towards Automation in a Higher-Order Environment; In: Halaas, A. and Denyer, P. (Eds.), Proc. International Conference on Very Large Scale Integration (VLSI91), Elsevier Science Publishers B.V. (North-Holland), 1991, pp. 81-90.

[SeSr89] Sekar, R.; Srivas, M.: Formal Verification of a Microprocessor Using Equational Techniques; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1989, pp. 171- 217.

[SGSE92] Stavridou, V.; Goguen, J.; Stevens, A.; Eker, S.; Aloneftis, S.; Hobley, K.: FUNNEL and 2OBJ: Towards an Integrated Hardware Design Environment; In: Stavridou, V., Melham, T. and Boute, R. (Eds.), Theorem Provers in Circuit Design, Elsevier Science Publishers B.V. (North-Holland), 1992, pp. 197-224.

[Shee84] Sheeran, M.: FP: a Language for VLSI Design; Proc. ACM Conference on Lisp and Functional Programming, August 1984, pp. 104-112.

[SmBH82] Smith, G.; Bahnsen, R.; Halliwell, H.: Boolean Comparision of Hardware and Flowcharts; IBM Journal Research Development, Vol. 26, No. 1, January 1982, pp. 106-116.

[SoVa89] Sohi, G.; Vajapeyam, S.: Tradeoffs in Instruction Format Design for Horizontal Architectures.; Proc. third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, 1989.

[Spiv89] Spivey, J.: The Z Notation: A Reference Manual; Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Spru89] Spruth, W.: The Design of a Microprocessor; Springer Verlag, 1989.

[SrAg86] Srinivas, N.; Agrawal, V.: Prove: Prolog Based Verifier; Proc. IEEE International Conference on Computer-Aided Design (ICCAD86), Santa Clara, CA, Nevember 1986, pp. 306-309.

[SrBi90] Srivas, M.; Bickford, M.: Formal Verification of a Pipelined Microprocessor; IEEE Software, September 1990, pp. 52-64.

[Sriv91] Srivas, M.: Bridging the Formal Methods Gap: A Computer-Aided Verification Tool for Hardware Designs; Proc. 36th IEEE Computer Society International Conference (COMPCON91), San Francisco, CA, February, 1991, pp. 456-461.

[Stal88] Stallings, W.: Reduced Instruction Set Computer Architecture; Proceedings of the IEEE, Vol. 76, No. 1, January 1988, pp. 38-55.

[Stav94] Stavridou, V.: Gordon's Computer: A Hardware Verification Case Study in OBJ3; Formal Methods in System Design, Vol. 4, No. 3, May 1994, pp. 264-310.

[StBE88] Stavridou, V.; Barringer, H.; Edwards, D.: Formal Specification and Verification of Hardware: A Comparative Case Study; Proc. ACM/IEEE 25th Design Automation Conference (DAC88), Anaheim, CA, June 1988, pp. 197-204.

[Ston90] Stone, H.: High-Performance Computer Architecture; Addison-Wesley Publishing Company, 1990.

[SuFr86] Supowit, K.; Friedmann, S.: A New Method for Verifying Sequential Circuits; Proc. ACM/IEEE 23rd Design Automation Conference (DAC86), Las Vegas, NV, June 1986, pp. 200-207.

[SunM89] Sun Microsystems, Inc.: The SPARC Architecture Manual; Sun Microsystems, Inc., USA, Version 8, Part No. 800-1399-09, August 1989.

[TaAS89] Talkhan, E.; Ahmed, A.; Salama, A.: Microprocessors Functional Testing; IEEE Transactions on Computer Aided Design, Vol.8, No. 3, March 1989.

[Taba90] Tabak, D.: RISC Systems; Letchworth, Research Studies Press, 1990.

[TaKu93a] Tahar, S.; Kumar, R.: A Formalization of a Hierarchical Model for RISC Processors; In: Spies, P. (Ed.), Proc. European Informatics Congress Computing Systems Architecture (Euro-ARCH93), Munich, Germany, October 1993, Informatik Aktuell, Springer Verlag, pp. 591-602.

[TaKu93b] Tahar, S.; Kumar, R.: Towards a Methodology for the Formal Hierarchical Verification of RISC Processors; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, IEEE Computer Society Press, pp. 58-62.

[TaKu94a] Tahar, S.; Kumar, R.: Implementing a Methodology for Formally Verifying RISC Processors in HOL; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 281-294.

[TaKu94b] Tahar, S.; Kumar, R.: Formal Verification of Pipeline Conflicts in RISC Processors; Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September 1994, IEEE Computer Society Press, pp. 285-289.

[TaKu94c] Tahar, S.; Kumar, R.: Implementational Issues for Verifying RISC-Pipeline Conflicts in HOL; In: Melham, T. and Camilleri, J. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 854, Springer Verlag, 1994, pp. 424-439.

[Tane90] Tanenbaum, A.: Structured Computer Organization; Prentice-Hall International Editions, 1990.

[Tass93] Van Tassel, J.: Fremto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant; PhD. Thesis, Computer Laboratory, University of Cambridge, July 1993.

[Thom93] Thomas, M.: The Industrial Use of Formal Methods; Microprocessor and Microsystems, Vol. 17, No. 1, 1993, pp. 31-36.

[Tred88] Tredemick, N.: Experiences in Commercial VLSI Microprocessor Design; Microprocessors and Microsystems, Vol. 12, No.8, October 1988.

[TuBH87] Turner, D.; Burns, R.; Hecht, H.: Designing Micro-Based Systems for Fail-Safe Travel; IEEE Spectrum, Vol. 24, No. 2, February 1987, pp.58-63.

[Ueha87] Uehara, T.: Proofs and Synthesis are Cooperative Approaches for Correct Circuit Designs; In: Borrione, D. (Ed.), From HDL Description to Guaranteed Correct Circuit Designs, Elsevier Science Publishers B.V. (North-Holland), 1987, pp. 241-252.

[USDD83] U.S. Department of Defence: Reference Manual for the Ada Programming Language; ANSI/MIL-STD-1815A, January 1983.

[VNMP89] Villarrubia, P.; Nusbaum, G.; Masleid, R.; Patel, P.: IBM RISC Chip Design Methodology; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, Massachusetts, September 1989, pp. 143-147.

[Wagn77] Wagner, T.: Hardware Verification; PhD. Thesis, Department of Computer Science, Stanford University, Semptember 1977.

[WaTh85] Walker, R.; Thomas, D.: A Model of Design Representation and Synthesis; Proc. ACM/IEEE 22th Design Automation Conference (DAC85), Las Vegas, NV, June 1985, pp. 453-457.

[Weis86] Weise, D.: Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits; PhD. Thesis, Massachusetts Institute of Technology, 1986.

[Wind90a] Windley, P.: The Formal Verification of Generic Interpreters; PhD. Thesis, Division of Computer Science, University of California, Davis, July 1990.

[Wind90b] Windley, P.: A Hierarchical Methodology for Verifying Microprogrammed Microprocessors; Proc. IEEE Symposium on Security and Privacy, Oakland, CA, May 1990, pp. 345-357.

[Wint92] Winters, K.: ASIC Design Experience: MDLX; Department of Electrical Engineering, Montana State University, USA, April 1992.

[Wong93] Wong, W.: Formal Verification of VIPER's ALU; University of Cambridge, Computer Laboratory, April, 1993.

[Wong94] Wong, W.: Modelling Bit Vectors in HOL: the word Library; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 371-384.

[WSEB87] Weiser, U.; Sidi, Y.; Epstein, L.; Brian, D.; Kaminker, A.: Design of the NS32532 Microprocessor; Proc. IEEE International Conference on Computer Design (ICCD87), Rye Brook, NY, October 1987, pp. 177-180.

[Wund91] Wunderlich, H.: Rechnergestützte Verfahren für den prüfgerechten Entwurf und Test hochintegrierter Schaltungen; Springer Verlag, 1991.

[Yoel90] Yoeli, M.: Formal Verification of Hardware Design; IEEE Computer Society Press, 1990.

[Youn89] Young, W.: A Mechanically Verified Code Generator; Journal of Automated Reasoning, Vol. 5, 1989, pp. 411-428.

[ZhJS94] Zhu, Z.; Joyce, J.; Seger, C.: Verification of the Tamarack-3 Microprocessor in a Hybrid Verification Environment; In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 253-266.

Tabellarischer Lebenslauf:

Name:	Sofiène Tahar
Geburtsdatum:	5.5.1966
Geburtsort:	Sousse, Tunesien
Eltern:	Dipl.-Ing. Mohamed Tahar Fahima Tahar geb. Akremi
Familienstand:	verheiratet
Staatsbürgerschaft:	tunesisch
Wohnsitz:	Kaiserstr. 45 76131 Karlsruhe
Schulbesuch:	1971 - 1977 Grundschule in Tunis 1977 - 1984 Gymnasium in Tunis Abschluß: Abitur (Baccalauréat)
Studium:	WS 85/86 - SS 90 Studium der Elektrotechnik an der TH Darmstadt Abschluß mit dem Grad "Diplom-Ingenieur"
Berufliche Tätigkeit:	Seit dem 15.1.1991 wissenschaftlicher Angestellter am Institut für Rechnerentwurf und Fehlertoleranz der Universität Karlsruhe bei Prof. Dr. Detlef Schmid

Karlsruhe, den 20.6.1994