Implementierung und Verifikation eines RISC-Prozessors in AVEREST

DANIEL BAUDISCH

PROJEKTARBEIT

eingereicht am
Fachbereich Informatik der Technischen Universität Kaiserslautern

August 2006

Betreuer: Dipl.-Inf. Tobias Schüle
Referent: Prof. Dr. rer. nat. Klaus Schneider

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, am 14. August 2006

Daniel Baudisch

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir bei der Erstellung dieser Arbeit geholfen haben. Dazu zählen in erster Linie meine Betreuer Prof. Dr. rer. nat. Klaus Schneider und Dipl.-Inf. Tobias Schüle. Des Weiteren danke ich den Mitgliedern der Arbeitsgruppe Reaktive Systeme für ihre Unterstützung bei der Erstellung dieser Arbeit.

Inhaltsverzeichnis

Erklärung ii

Danksagung iii

1 Einleitung 1
1.1 Motivation 1
1.2 Ziel dieser Arbeit 1

2 Synchrone Sprachen 2
2.1 QUARTZ 2
2.1.1 Schnittstelle 3
2.1.2 Systemverhalten 3
2.2 Kausalität in synchronen Sprachen 5
2.2.1 Logische Korrektheit 5
2.2.2 Schreibkonflikte 6
2.2.3 Verifikation mit AVEREST 7

3 Temporale Logiken 9
3.1 Beschreibungsprachen für reaktive Systeme 9
3.2 Linear Time Logic (LTL) 9
3.3 Computation Tree Logic (CTL) 10
3.4 CTL* 10

4 Eintaktmodell 12
4.1 Spezifikationen 12
4.2 Implementierung 12
4.2.1 Schnittstelle für Eintaktmodell mit Bitvektoren 12
4.2.2 Schnittstelle für Eintaktmodell mit Ganzzahlen 14
4.2.3 Verhalten 15
4.3 Verifikation 21
4.3.1 Benchmark 23
4.3.2 Auswertung 24
4.4 Synthese 29
4.4.1 Registersatz 30
4.4.2 Befehls- und Datenspeicher 31
4.4.3 Gesamtsystem 31

5 Pipeline-Modell **35**
5.1 Pipelinekonflikte . . . 36
5.1.1 Datenkonflikte . . . 36
5.1.2 Steuerkonflikte . . . 37
5.2 Spezifikationen . . . 37
5.2.1 Auflösung von Konflikten . . . 38
5.3 Implementierung . . . 38
5.3.1 Schnittstelle . . . 38
5.3.2 Verhalten . . . 39
5.4 Verifikation . . . 53
5.4.1 Modularisierung . . . 53
5.4.2 Auswertung . . . 69
5.4.3 Benchmark . . . 72
5.4.4 Verifizierung mit Produktautomaten . . . 73
5.4.5 Synthese . . . 74

6 Zusammenfassung und Ausblick **77**
6.1 Zusammenfassung . . . 77
6.2 Ausblick . . . 78

Abbildungsverzeichnis **80**

Tabellenverzeichnis **82**

Literaturverzeichnis **83**

1 Einleitung

1.1 Motivation

Die Gegenwart von sicherheitskritischen Systemen in unserem alltäglichem Leben, wie z.B. Fahrhilfen und Unfallschutz in Automobilen, oder auch die Steuerung in Flugzeugen, verlangt von diesen Systemen, daß sie fehlerfrei funktionieren, da sie sonst Gesundheit, wenn nicht sogar Leben gefährden würden. Hinzu kommen Ereignisse, wie z.B. der Pentium Bug und dem damit verbundenen Imageverlust, die die Firmen dazu veranlassen, kommerzielle, nicht notwendigerweise sicherheitskritische Systeme zu verifizieren. Zudem kann der durch den Konkurrenzkampf verursachte Zeitdruck dazu führen, daß Änderungen der Spezifikationen innerhalb eines, schon zum Teil implementierten, komplexen Systems Probleme in der Erfüllbarkeit aller Spezifikationen hervorrufen kann. Eine automatische Verifikation kann viel Zeit sparen und zudem sicherstellen, daß das System weiterhin alle Spezifikationen und damit die Wünsche des Kunden erfüllt.

1.2 Ziel dieser Arbeit

Nach einer kurzen Einführung in synchrone Sprachen und temporale Logiken wird mit Hilfe des AVEREST-Paketes ein RISC-Prozessor entworfen und der modellierte Prozessor anschließend mit zwei verschiedenen Model Checkern verifiziert. Der Prozessor wird in zwei Varianten entworfen. Zum einen ein recht triviales Eintakt-Modell und zum anderen eine Variante mit Pipelining und Konfliktauflösung durch Forwarding. Vor allem die letzte Variante sollte die interessantere sein, da gerade Pipelining und Forwarding die Struktur von Prozessoren und damit auch die Verifikation erschweren. Das Eintaktmodell soll nochmals in verschiedenen Varianten implementiert werden. Hierbei soll nicht nur ein Hauptaugenmerk auf die Spezifikationen des Prozessors gesetzt werden, sondern auch auf die Zeiten der Verifikation. Hier wird zwischen einer Strukturvariante und einem Verhaltensmodell verglichen, um Aussagen über das zeitliche Verhalten der Verifikation treffen zu können, wenn Optimierungen auf der Gatter Ebene vorgenommen werden. Weiterhin wird ein Vergleich zwischen einer Ganzzahlvariante und einer Bitvektorvariante gemacht und deren zeitliches Verhalten bei der Verifikation beobachtet.

2 Synchrone Sprachen

Synchrone Sprachen dienen zur Beschreibung reaktiver Systeme [Sch05]. Ziel ist es von der realen Hardware zu abstrahieren und den Entwicklern die Möglichkeit zu geben, physikalische Gegebenheiten, wie Gatterlaufzeiten, zu vernachlässigen. Weiterhin sind diese Sprachen darauf ausgelegt, parallele Prozesse zu erstellen, so daß man sich nicht, wie z.B. in C oder Pascal, um die zusätzlichen Verwaltungsaufgaben für parallele Prozesse kümmern muß.

Als Basis dienen in den synchronen Sprachen Anweisungen, die in sogenannte Mikroschritte und Makroschritte unterteilt werden. Bei den Mikroschritten handelt es sich um Anweisungen, die keine Zeit konsumieren. Makroschritte bestehen aus endlich vielen Mikroschritten und konsumieren genau eine Zeiteinheit. In QUARTZ werden diese Makroschritte durch `pause` Anweisungen gekennzeichnet, wobei es auch viele weitere Anweisungen gibt, die Makroschritte kennzeichnen, jedoch im Falle von QUARTZ durch Kontrollflußanweisungen und `pause` Anweisungen ersetzt werden können. Um sich ein besseres Bild von synchronen Sprachen zu machen, sei auf den ESTEREL Primer [Ber97] und das AVEREST Paket [Ave05] hingewiesen. Im Folgenden wird nur kurz auf die Funktionalität eingegangen, um sich mit den Grundlagen von synchronen Sprachen vertraut machen zu können.

2.1 QUARTZ

QUARTZ ist eine synchrone Sprache, die an ESTEREL anlehnt und zum Paket AVEREST gehört. Programme in QUARTZ werden aus einem oder mehreren Modulen aufgebaut. Diese Module bestehen aus einer Schnittstelle, die die Verbindung zur Umwelt herstellt, und dem Systemverhalten des Moduls.

```
module name
  interface
  description
spec
  specification
end
```

Abbildung 2.1: Aufbau eines Moduls in QUARTZ

name:	Name des Moduls
interface:	Beschreibung der Schnittstelle des Moduls. Dabei wird zwischen Eingabe und Ausgabe unterschieden.
description:	Beschreibung des Verhaltens des Moduls. Damit das Modul reaktiv und deterministisch ist, muß das System für jede Eingabe genau eine Ausgabe haben.
specification	System-Spezifikationen, welche nicht angegeben werden müssen. Sie können mit Hilfe eines Model Checkers verifiziert werden.

In QUARTZ ist es möglich, zu den Modulen, abgesehen von der Beschreibung des Systemverhaltens, zusätzlich Spezifikationen anzugeben. Mit Hilfe eines Model Checkers wird überprüft, ob das implementierte Modell die Spezifikationen erfüllt, wodurch das Verhalten des Moduls verifiziert werden kann. Man beachte, daß das Verhalten nur durch *description*, jedoch nicht durch die Spezifikationen beschrieben wird. Die Spezifikationen können in QUARTZ in LTL, CTL, CTL* (siehe Kapitel 3) oder als μ-Kalkül aufgestellt werden.

2.1.1 Schnittstelle

Wie schon erwähnt besteht die Schnittstelle aus Ein- und Ausgaben. Jeder Ein- oder Ausgabe muß ein Datentyp zugeordnet werden. Ist dieser nicht angegeben, wird die Ein- bzw. Ausgabe als Signal behandelt. Ein Signal kann den Wert *true* oder *false* annehmen. Ansonsten stehen Ganzzahlen und Bitvektoren zur Wahl. Ganzzahlen können beliebig große ganzzahlige Werte annehmen. Bitvektoren hingegen sind in ihrem Wertebereich durch die Anzahl der Bits eingeschränkt.

2.1.2 Systemverhalten

Das Verhalten wird durch Anweisungen beschrieben, wobei die für diese Arbeit wichtigsten Anweisungen hier kurz beschrieben werden. Dazu gehören natürlich die Zuordnungen sowie die schon beschriebene Anweisung `pause`. Bei den Zuordnungen muß man zwischen dem Setzen von Signalen und der Zuweisung von Werten unterscheiden. Jede dieser Zuordnungsmöglichkeiten kann nochmals in zwei Arten unterschieden werden: die sofortigen und die verzögerten. Die Zuweisung `a:=b` sagt aus, daß innerhalb des momentanen Makroschrittes `a` den gleichen Wert wie `b` hat. `next(a):=b` dagegen bedeutet, daß die Zuweisung von `a` zu dem Wert, den `b` im momentanen Taktzyklus hat, erst im nächsten Makroschritt erfolgt. Signale können nicht über die oben beschriebenen Zuweisungen gesetzt werden. Sie werden emittiert. Dafür stellt QUARTZ den Befehl `emit` zur Verfügung. Auch hier muß zwischen sofortigem und verzögertem Senden unterschieden werden. Während `emit z` noch im gleichen Makroschritt das Signal `z` setzt, setzt `emit next(z)` erst im nächsten Taktzyklus das Signal `z`. Ein weitere Eigenschaft der Signale ist, daß sie nur für einen Taktzyklus gesetzt werden. Wird also ein Signal mit `emit z` ausgesendet, so hat es im nächsten Takt, sofern es nicht durch eine andere Anweisung erneut gesendet wird, den Zustand ‚nicht gesetzt'.

Ebenfalls wichtige Anweisungen sind solche, die den Steuerfluß beeinflussen. Dazu gehören unter anderem `loop` und `while` *condition* `do` zu den Schleifenkonstrukten und die `if`- und `case`-Anweisungen zur bedingten Ausführung von Anweisungsblöcken. Die Syntax dieser Anweisungen ist in Abbildung 2.2 widergegeben. Beim Schleifenkörper ist zu beachten, daß dieser Zeit verbrauchen muß, also nicht instantan sein darf. Bei der `loop`-Anweisung fällt auf, daß hier sowohl Schleifen mit Bedingung, als auch ohne Bedingung möglich sind. Letztere sind Endlosschleifen. Der Unterschied zwischen der Schleife ohne Bedingung und der Schleife mit der Bedingung `true` liegt in der Generierung des Code. Da bei der Endlosschleife keine Bedingung notwendig ist, kann der generierte Code kürzer und die Schaltung dementsprechend kleiner ausfallen.

```
loop
  statements
(end|each condition)

while condition do
  statements
end

if condition then
  statements
[else
  statements]
end
oder
if condition else
  statements
end

case
  condition1 do statements1;
  ...
  conditionN do statementsN
else
  statementsDefault
end
```

Abbildung 2.2: Steuerflußanweisungen in QUARTZ

Das ABRO Beispiel

Als nächstes folgt das ABRO-Beispiel, welches durch folgende Eigenschaften spezifiziert wird:

- Schnittstelle
  - Drei Signale als Eingabe: a, b und r
  - Ein Signal als Ausgabe: o
- Verhalten
  - Sobald a **und** b anliegen (nicht zwingenderweise gleichzeitig), soll o auf *true* gesetzt werden.
  - Sobald r gedrückt wird, soll das Programm von vorn beginnen.

```
module ABRO:
input a,b,r;
output o;
  loop
    [await a; await b];
    emit o
  each r
end
```

Abbildung 2.3: Das ABRO Beispiel

Zwei Anweisungen in diesem Modul wurden noch nicht erläutert. Zum einen die Anweisung `await` *signal*, welche die gleiche Bedeutung wie folgender Code hat:

```
while ~signal do
  pause
end
```

Desweiteren ist auch das Konstrukt [*statements*] noch nicht besprochen worden. Hier handelt es sich um einen Ausdruck, der alle Anweisungen zwischen den Klammern synchron ausführt. Im ABRO Beispiel bedeutet das, daß *emit o* erst ausgeführt wird, wenn die Anweisungen `await a` **und** `await b` abgeschlossen sind. An dieser Stelle sei darauf hingewiesen, daß die Mächtigkeit und auch die Komplexität der synchronen Sprachen das hier beschriebene weit übersteigen.

2.2 Kausalität in synchronen Sprachen

2.2.1 Logische Korrektheit

Unter logischer Korrektheit versteht man, daß ein Programm reaktiv und deterministisch ist [Ber97,KJT04], also auf **jede** Eingabe mit **einer** Ausgabe reagiert. Parallelität

von Steuerflußanweisungen und Makroschritten können die semantische Analyse eines Programms sehr schwierig gestalten. Wie in sequentiellen Sprachen ist es möglich, Programme zu schreiben, welche syntaktisch korrekt sind, jedoch keinen Sinn ergeben. In synchronen Sprachen müssen jedoch zusätzliche Fehlerquellen berücksichtigt werden, die erst durch die Parallelität entstehen. Dazu betrachte man die Programme in Abbildung 2.4 und 2.5. Während es für Programm P1 (Abbildung 2.4) mehrere gültige Ausgaben gibt (`o`=0 **und** `o`=1 sind korrekte Ausgaben), hat Programm P2 (Abbildung 2.5) keine gültige Ausgabe, da die `if`-Anweisung widersprüchlich ist. Programmierer von Sprachen wie Basic, Pascal, C, C++, u.a., aber auch Neueinsteiger sollten bei der zeitlichen Ausführung von Befehlen beachten, daß alle Mikroschritte innerhalb eines Makroschrittes parallel ausgeführt werden. Anders ausgedrückt bedeutet das, daß Anweisungen im Quelltext vorangegangene Anweisungen beeinflussen können. Synchrone Programme können bei steigender Größe auch dementsprechend komplex zu verstehen und zu analysieren sein. Daher ist nicht nur die computergestütze Verifikation von synchronen Programmen von großer Bedeutung, sondern auch Kausalitätsanalysen.

```
module P1:
  output o;
  if o then emit o end
end
```

Abbildung 2.4: Ein nicht-reaktives System

```
module P2:
  output o;
  if ~o then emit o end
end
```

Abbildung 2.5: Ein nicht-deterministisches System

2.2.2 Schreibkonflikte

Ein weiteres Problem, mit dem man sich bei synchronen Sprachen auseinandersetzen und beim Erstellen von synchronen Programmen beachten muß, sind Schreibkonflikte, die aufgrund der Parallelität der synchronen Sprachen auftreten können [KJT04]. Hier sind drei Typen zu unterscheiden, die in den Abbildungen 2.6, 2.7 und 2.8 dargestellt sind. Bei allen Typen von Schreibkonflikten liegt das Problem in zwei oder mehreren Anweisungen, die **einer Variable** innerhalb **eines Makroschrittes verschiedene Werte** zuweisen. Der QUARTZ-Compiler bietet die Möglichkeit, Module auf Schreibkonflikte zu testen.

```
module WRCONFLCT1:
  output o:boolean;
  o := #B0;
  o := #B1
end
```

Abbildung 2.6: Schreibkonflikt vom Typ NOW-NOW

```
module WRCONFLCT2:
  output o:boolean;
  next(o) := #B0;
  next(o) := #B1;
  pause
end
```

Abbildung 2.7: Schreibkonflikt vom Typ NEXT-NEXT

```
module WRCONFLCT3:
  output o:boolean;
  next(o) := #B0;
  pause;
  o := #B1
end
```

Abbildung 2.8: Schreibkonflikt vom Typ NEXT-NOW

2.2.3 Verifikation mit AVEREST

In der Einleitung wurde beschrieben, daß es in der Hardwareentwicklung wichtig ist zu zeigen, daß ein System bestimmte Spezifikationen erfüllt. Dazu wird laut [Har03, Wil05] auf Model Checking und Theorem Proving zurückgegriffen.

- *Model Checking* dient dazu, endliche Automaten mit Hilfe von Spezifikationen, die z.B. in Temporaler Logik (3)aufgestellt werden, zu verifizieren [Sch04].

- *Theorem Proving* beweist das Verhalten eines Systems mit Hilfe von mathematischen Theoremen. Dazu werden gegebene Sätze aus gegebenen Regeln hergeleitet oder in Regeln zerlegt, welcher weiter zerlegt werden, bis der Satz bewiesen wurde. Dies kann unter Verwendung von Tools wie z.B. dem Theorem Prover HOL geschehen. Beweise der Korrektheit von Pipeline-Prozessoren und superskalaren Prozessoren gibt es in [AN96, Fox01, Kro01, Man00a, Man00b, Hos00, SSH+99, Kro96, SR94, KPM00, KP01]. In [Kro99] wird die Korrektheit des Tomasulu Algorithmus, welcher häufig in superskalaren Prozessoren eingesetzt wird, bewiesen. Harman

verwendet zum Beweisen von Hardwaresystem [Har00] die Sprache Maude.

In 2.1 wurde erwähnt, daß QUARTZ die Möglichkeit bietet Spezifikationen in ein Modul einzubinden. Diese können als temporale Logiken dargestellt oder als μ-Kalkül beschrieben werden. Um mit Hilfe des AVEREST Paketes ein Modell zu erstellen und zu verifizieren, wird zunächst das Verhalten eines Moduls in der QUARTZ Syntax beschrieben. Nachdem das Verhalten implementiert wurde, müssen noch die Spezifikationen, welche von dem beschriebenen Verhalten erfüllt werden müssen, beschrieben. Diese können, wie erwähnt, in temporaler Logik oder als μ-Kalkül dargestellt werden. Um das System zu verifizieren muß das Verhalten vom Compiler in ein Transitionssystem übersetzt werden. Aus diesem können die Model Checker, also `Beryl` aus dem AVEREST-Paket oder der Cadence `SMV`, einen endlichen Automaten erstellen und diesen darauf Überprüfen, ob er die angegebenen Spezifikationen erfüllt. In [Sch03] wird detaillierter auf die Verifikation reaktiver Systeme eingegangen.

3 Temporale Logiken

3.1 Beschreibungsprachen für reaktive Systeme

Zu den Beschreibungssprachen für reaktive Systeme gehören außer den regulären Ausdrücken, Zustandsdiagrammen, den graphischen Intervall-Logiken und dem modalem μ-Kalkül die temporalen Logiken [Sch03, Ran04a, Ran04b]. Verbreitete temporale Logiken sind LTL (Linear Time Logic), CTL (Computation Tree Logic) und CTL*. Hier wird ein kurzer Überblick über die Sprachen gegeben, da die Spezifikationen des RISC-Prozessors durch temporale Logiken beschrieben werden.

3.2 Linear Time Logic (LTL)

Bei LTL handelt es sich um eine Sprache, in der einzelne Pfade betrachtet werden. Man geht davon aus, daß die Zeit einen Initialzustand ohne Vorgänger besitzt, und diskret unterteilt wird mit unendlich vielen Nachfolgezuständen.

Die BNF für LTL lautet:

$$\varphi ::= \vartheta \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid G\varphi \mid F\varphi \mid X\varphi \mid \varphi U \psi$$

wobei ϑ für eine beliebige atomare Formel steht.

Eine LTL-Formel besteht also aus atomaren Propositionen, booleschen Operatoren und temporalen Verknüpfungen. Temporale Quantoren sind $\mathbf{G}\varphi$ („Es gilt immer φ“), $\mathbf{F}\varphi$ („Es gilt irgendwann φ“ oder „Es gilt mindestens einmal φ“), $\mathbf{X}\varphi$ („Es gilt im nächsten Schritt φ“) oder $\varphi\mathbf{U}\psi$ („Es gilt φ bis ψ gilt“). Abbildung 3.1 zeigt die graphische Interpretation der Quantoren.

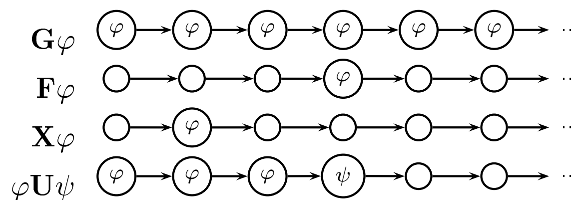


Abbildung 3.1: Beispiele für LTL-Beschreibungen und ihre graphische Interpretation

3.3 Computation Tree Logic (CTL)

Im Gegensatz zu LTL werden in CTL nicht einzelne Pfade betrachtet, sondern Bäume. Daher gibt es hier zusätzlich zu den temporalen Verknüpfungen auch die Pfadquantoren **A** („Auf allen Pfaden") und **E** („Es gibt einen Pfad"). Weiterhin gilt in CTL die Einschränkung, daß es zu jeder temporalen Verknüpfung auch einen Pfadquantoren geben muß.

Die BNF für CTL lautet:

$$\begin{aligned} \varphi &::= \vartheta \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid A\phi \mid E\phi \\ \phi &::= G\varphi \mid F\varphi \mid X\varphi \mid \varphi U \psi \end{aligned}$$

wobei ϑ für eine beliebige atomare Formel steht.

In Abbildung 3.2 sind Beispiele zu CTL abgebildet.

Zur Mächtigkeit muß man sagen, daß keine der beiden Sprachen mächtiger als die andere ist. Es lassen sich mit CTL Formeln darstellen, die mit LTL nicht darstellbar sind, umgekehrt gibt es jedoch auch Formeln in LTL, die mit CTL nicht darstellbar sind.

3.4 CTL*

Wir wissen nun, daß LTL im Gegensatz zu CTL geschachtelte Ausdrücke zuläßt, d.h. in CTL müssen Modalitäten immer paarweise mit einem Pfadquantor (**E** oder **A**) auftreten. In CTL* wird CTL nun so erweitert, daß direkte Verknüpfungen von Modalitäten mit aussagenlogischen Junktoren nun möglich sind, bevor man einen Pfadquantor darauf anwendet.

Die BNF für CTL* lautet:

$$\begin{aligned} \varphi &::= \vartheta \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid A\phi \mid E\phi \\ \phi &::= \theta \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid G\phi \mid F\phi \mid X\phi \mid \phi U \phi \end{aligned}$$

wobei ϑ für eine beliebige atomare Formel steht.

Die Mächtigkeit von CTL* übertrifft die von LTL und CTL. Alle Formeln, die sich mit LTL oder CTL darstellen lassen, lassen sich auch mit CTL* darstellen. Zudem gibt es auch Formeln in CTL*, die weder mit LTL noch mit CTL darstellbar sind.

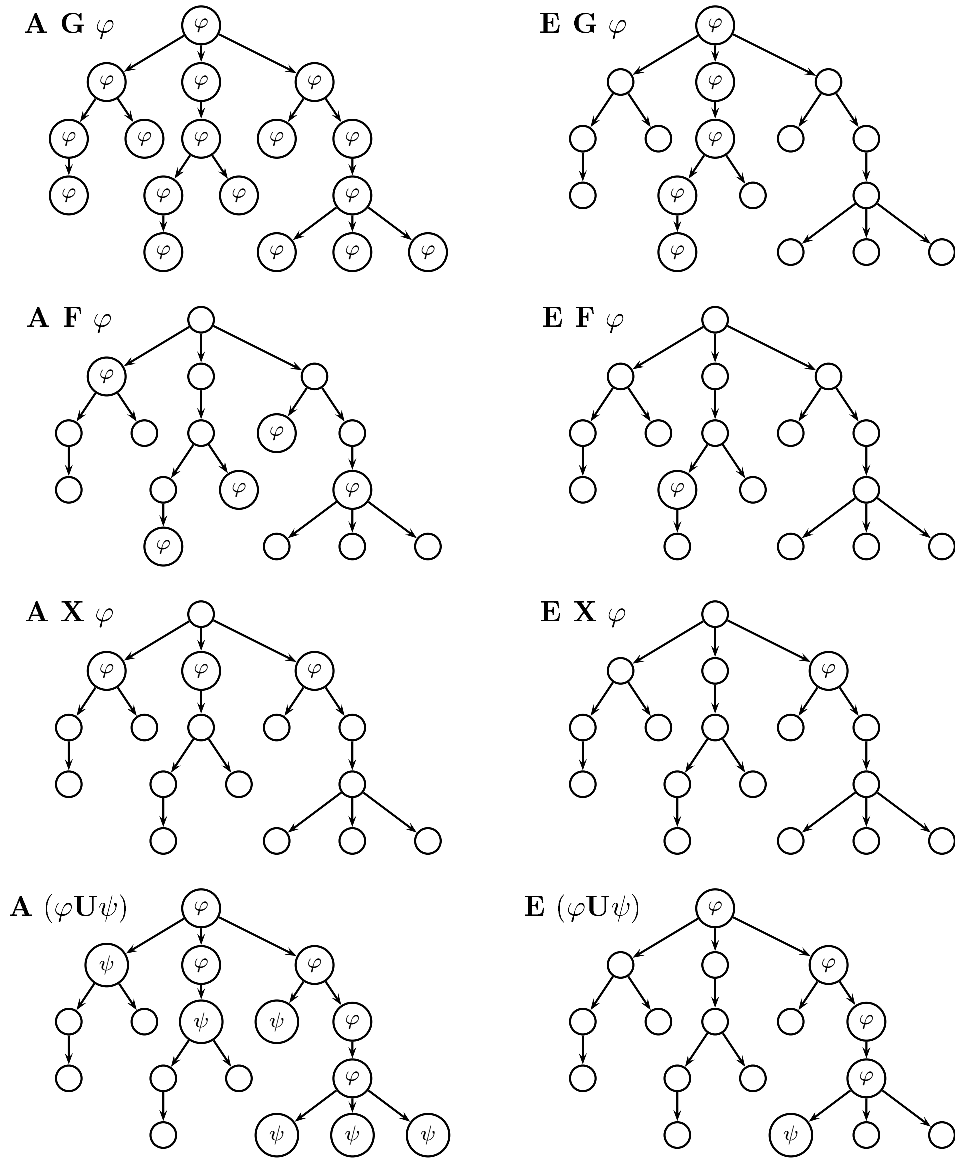


Abbildung 3.2: Beispiele für CTL-Beschreibungen und ihre graphische Interpretation

4 Eintaktmodell

4.1 Spezifikationen

Im Eintaktmodell soll jeder Befehl sequentiell abgearbeitet werden, so daß keine Schreibkonflikte entstehen können. Der Prozessor wird Zugriff auf in Daten- und Befehlsteil getrennten Speicher haben. Um die Speicherbefehle (`SW`, `LW`) in einem Takt bearbeiten zu können, wird angenommen, daß ein Datenwort noch innerhalb des Taktes, in dem die Adresse angelegt wird, gelesen werden kann. Schreibzugriffe erfolgen in der Art, daß die Daten beim Taktsignal geschrieben werden. Der Befehlsspeicher muß das gleiche Leseverhalten aufweisen, wie der Datenspeicher. Weiterhin werden, um die für die Verifikation erzeugten Kripke-Strukturen klein zu halten, die Register nicht in das Modell eingebettet, sondern über die Schnittstelle des Prozessors angesprochen. Später wird das Registerarray implementiert um das Modell zu vervollständigen und die Validierung durch Simulation zu ermöglichen.

QUARTZ bietet zwei Möglichkeiten zur Darstellung von Register- und Speicherinhalten. Zum einen können diese als Bitvektoren mit festgelegter Länge dargestellt werden, zum anderen aber auch als Ganzzahlen, deren Länge nicht festgelegt ist und somit im Wertebereich beliebig groß sein können. Um die Auswirkungen der Darstellungsmöglichkeiten auf die Verifikation und speziell deren Zeiten zu untersuchen, werden zwei Varianten des Eintaktmodells implementiert, die jeweils eine der beiden Darstellungsmöglichkeiten für den Datenpfad verwenden sollen.

4.2 Implementierung

Bevor auf die Implementierung des Verhaltens eingegangen wird, muß die Schnittstelle des Prozessors zu den anderen Hardware Komponenten, wie Datenspeicher, Befehlspeicher oder Registerarray, festgelegt werden. Für die vollständige Schnittstelle des Eintaktmodells gibt es zwei Varianten, die sich in der Darstellung des Datenpfades unterscheiden. Eine Schnittstelle für die Variante mit Bitvektoren und eine Schnittstelle für die Variante mit Ganzzahlen. Die beiden Schnittstellen unterscheiden sich nicht nur in der Repräsentation der Ein- und Ausgänge durch verschiedene Typen, daher werden sie hier getrennt erläutert.

4.2.1 Schnittstelle für Eintaktmodell mit Bitvektoren

Die Schnittstelle für die Bitvektor-Variante (Abbildung 4.1) ist am einfachsten zu erklären. Im folgenden Teil werden die Ein- und Ausgaben der Schnittstelle erklärt. In

```
module EIN_TAKT_RISC:
  input
    MemIn : bitvec[numBitsPerInteger],
    RegInS : bitvec[numBitsPerInteger],
    RegInT : bitvec[numBitsPerInteger],
    InstrReg : bitvec[32];

  output
    MemOut : bitvec[numBitsPerInteger],
    MemAddr : bitvec[numBitsPerInteger],
    MemWrite : boolean,
    RegOut : bitvec[numBitsPerInteger],
    RegIndex : bitvec[5],
    RegWrite : boolean,
    PC : bitvec[numBitsPerInteger-2],
    UNKNOWN_OPCODE;

     [...]

end // module
```

Abbildung 4.1: Schnittstelle der Bitvektor-Variante des Eintaktmodells

Klammern wird angegeben, mit welcher Peripherie die Register bzw. Signale zu verbinden sind.

- *MemIn* (vom Datenspeicher) Datenwort, das an der Adresse *MemAddr* gespeichert ist
- *MemOut* (zum Datenspeicher) Datenwort, das bei gesetztem Signal *MemWrite* bei Taktende in den Datenspeicher an die Adresse *MemAddr* geschrieben werden soll
- *MemAddr* (zum Datenspeicher) Adresse des zu lesenden oder schreibenden Datenworts
- *MemWrite* (zum Datenspeicher) Signal, welches zum Schreiben eines Datenworts gesetzt werden muß
- *RegInS* (vom Registerarray) Inhalt des durch den im eingehenden Befehl indizierten Register RS.
- *RegInT* (vom Registerarray) Inhalt des durch den im eingehenden Befehl indizierten Register RT.
- *RegOut* (zum Registerarray) Datenwort, das bei gesetztem *RegWrite* in das Register, das durch *RegIndex* indiziert wird, am Taktende geschrieben werden soll

- *RegIndex* (zum Registerarray) Index des Registers, das bei gesetztem *RegWrite* geschrieben werden soll
- *RegWrite* (zum Registerarray) signalisiert, ob bei Taktende der Wert aus *RegOut* in das durch *RegIndex* indizierte Register kopiert werden soll
- *PC* (zum Befehlsspeicher) Adresse des Befehlswort das gelesen werden soll.
- *InstrReg* (vom Befehlsspeicher) Das Befehlswort an der Adresse *PC*
- *UNKNOWN_OPCODE* (z.B. zum Interrupt Controller) Signal, das bei einem nicht erkannten bzw. ungültigen Befehl gesetzt wird

Auffällig ist, daß die Inhalte von zwei Registern aus dem Registerarray gelesen werden, deren Indizes jedoch nicht vom Prozessor gesetzt werden. Da die Eingaberegister im Befehlswort eine feste Position haben, können die zugehörigen Bits direkt von dem Befehlswort, das am Prozessor anliegt, mit dem Registerarray verknüpft werden. Daher ist die Extrahierung der Indizes durch den Prozessor nicht notwendig. Die Programme bestehen aus 32 Bit breiten Wörtern und sind dementsprechend ausgerichtet. Die Adresse der Befehle ist daher immer ein Vielfaches von vier, d.h. die zwei niedrigstwertigen Bits sind immer Null und können im Interface eingespart werden. Beim endgültigen Schaltungsentwurf muß am Befehlsspeicher die Adresse lediglich um zwei Nullbits erweitert werden. Wesentlicher Vorteil dieser Vereinfachungen ist die Verkleinerung des Zustandsraumes des Prozessors, was wiederum zu einer Beschleunigung der Verifikation führt. Im Eintaktmodell für die Hardwaresynthese kommt hinzu, daß durch diese Optimierung weniger Schaltelemente benötigt werden.

4.2.2 Schnittstelle für Eintaktmodell mit Ganzzahlen

QUARTZ bietet den Datentyp Integer, mit dem beliebig große ganze Zahlen dargestellt werden können. Die Register werden nun nicht mehr durch Bitvektoren dargestellt, sondern durch Integer. Bei der Schnittstelle für die Ganzzahl-Variante muß man beachten, daß bei Operatoren wie Addition, bitweises UND, ODER, usw., die Typen der Operanden nicht gemischt werden dürfen, diese also übereinstimmen müssen. Es gibt auch keine Möglichkeit, Bitvektoren zu Integern oder Integer zu Bitvektoren zu konvertieren. Da jedoch, z.B. bei Register-Immediate-Befehlen, Daten aus dem Befehlswort, der weiterhin als Bitvektor bestehen bleibt, benötigt werden und diese auf Registerinhalte, die in dieser Variante als Ganzzahlen dargestellt werden, angewendet werden sollen, müssen diese anders „übergeben" werden. Alle Daten des Befehlswort, die potentielle Operanden für Operationen mit Integern sind, müssen deshalb in dieser Schnittstelle als Integer aufgelistet werden. Zu diesen potentiellen Operanden gehört außer dem *Immediate*-Datum für Register-Immediate-Befehle auch das *Target*-Datum, welches das Offset für unbedingte Sprünge enthält. Daher muß die obige Schnittstelle, abgesehen von der Typänderung des Datenpfades, um zwei Einträge erweitert werden. Die endgültige Schnittstelle der Ganzzahl-Variante wird in Abbildung 4.2 gezeigt.

```
module EIN_TAKT_RISC:
  input
    MemIn : integer,
    RegInS : integer,
    RegInT : integer,
    Target : integer,
    ImmediateData : integer,
    InstrReg : bitvec[32];
  output
    MemOut : integer,
    MemAddr : integer,
    MemWrite,
    RegOut : integer,
    RegIndex : bitvec[5],
    RegWrite,
    PC : integer,
    UNKNOWN_OPCODE;

      [...]

end // module
```

Abbildung 4.2: Schnittstelle der Ganzzahlvariante des Eintaktmodells

4.2.3 Verhalten

Das Verhalten des Eintaktmodells wurde in zwei verschiedenen Varianten implementiert. Bei der ersten Variante, der Verhaltensvariante, wird der Prozessor wie ein Interpreter beschrieben. Dabei wird für jeden Befehl ein eigener `if` Block geschrieben, der den auszuführenden Code enthält. Um sich eine bessere Vorstellung zu machen, wie das Ergebnis in Hardware aussieht, kann man davon ausgehen, daß für jeden Befehl eine eigene Schaltung implementiert wird, wobei das endgültige Ergebnis durch einen Multiplexer selektiert wird. Die zweite Variante wird so aufgebaut, wie man ein „echten" Prozessor aufbauen würde. Das heißt, daß die Arithmetik soweit wie möglich zusammengefasst wird, um die Anzahl der Gatter klein zu halten. Im Gegensatz zur ersten Variante ist diese für den Entwickler schwieriger zu lesen und zu warten. Damit kann sich bei nicht erfüllten Spezifikationen die Fehlersuche schwieriger gestalten. Ist ein Interpreter in Software das Ziel, ist die erste Variante die bessere Wahl. Kommt es auf die Größe der Schaltung an, bzw. im Falle der softwareseitigen Verwendung auf geringen Speicherplatzverbrauch, ist der Aufwand für eine Implementierung der zweiten Variante gerechtfertigt.

Eintakt-Verhaltensmodell

Oben wurde erklärt, wie das Verhaltensmodell implementiert wird. Jeder Befehl setzt die Ausgänge und wird von einem `if`-Block umgeben, wodurch das endgültige Ergebnis selektiert wird. Der Code (Abbildung 4.3) sieht dem einer Implementierung eines se-

quentiellen Programms ähnlich, wird aber, im Gegensatz zu sequentiellen Programmen, **parallel** ausgeführt. Da die `if`-Aussagen zudem disjunkt sind, ist es möglich, auf eine Verbindung der `if`-Blöcke durch `else` zu verzichten. Der Compiler kann die Schaltung möglicherweise dadurch verkleinern.

```
module EIN_TAKT_RISC:

    [Deklaration der Schnittstelle]

  let opcode := {InstrReg : 31..26} in
  let funct := {InstrReg : 5..0} in
  loop
    if opcode=#B000000 then
      if funct=INSTR_ADD then
        RegIndex := {InstrReg : 15..11};
        emit RegWrite;
        RegOut := RegInS + RegInT;
        next(PC) := PC + 1
      end;
      if funct=INSTR_AND then
        RegIndex := {InstrReg : 15..11};
        emit RegWrite;
        RegOut := RegInS & RegInT;
        next(PC) := PC + 1
      end;

           [...]

    end;
    if opcode=INSTR_ADDI then
        RegIndex := {InstrReg : 20..16};
        emit RegWrite;
        RegOut := RegInS + ImmediateData;
        next(PC) := PC + 1
      end;

           [...]

    pause
  end // loop
end // module
```

Abbildung 4.3: Ausschnitt aus der Implementierung des Eintakt-Verhaltensmodells

Eintakt-Strukturmodell

Ziel beim Strukturmodell ist es, die Anzahl der Gatter, somit auch die Gesamtgröße des Prozessors, klein zu halten, um diesen für die Hardwaresynthese zu optimieren. Es ist offensichtlich, daß bei der Herstellung kleinerer Prozessoren die Anzahl der Prozessoren pro Wafer erhöht und damit der Gewinn gesteigert werden kann. Zur Verringerung der Gatter wird zuerst das obige Eintakt-Verhaltensmodells als Basis herangezogen. Jeden Befehl getrennt zu behandeln ist selbstverständlich ineffizient und für die Hardwaresynthese von Prozessoren realitätsfremd. Also werden die Anweisungen der Befehle soweit wie möglich zusammengefasst. Dazu gehören zum Beispiel die Signale zum Speichern des Zielregisters oder zum Schreiben in den Datenspeicher (Abbildung 4.4).

```
module EIN_TAKT_RISC:

     [Deklaration des Interface]

  let opcode := {InstrReg : 31..26} in
  let funct := {InstrReg : 5..0} in
  loop

          [Instanziierung des ALU-Moduls]

    if (opcode=#B000000 & (funct=INSTR_ADD | funct=INSTR_AND |
                           funct=INSTR_OR | funct=INSTR_SLT |
                           funct=INSTR_SLL | funct=INSTR_SRA |
                           funct=INSTR_SRL | funct=INSTR_XOR)) |
        opcode=INSTR_ADDI | opcode=INSTR_LW then
      emit RegWrite
    end;
    if opcode=INSTR_SW then
      emit MemWrite
    end;
    if isALUR then
      RegIndex := {InstrReg : 15..11}
    else
      RegIndex := {InstrReg : 20..16}
    end;
    MemAddr := AluResult;
    MemOut := RegInT;

          [...]

    pause
  end // loop
end // module
```

Abbildung 4.4: Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - Setzen der Ausgaben, die die Befehle gemeinsam haben

Eine weitere Möglichkeit zur Optimierung ergibt sich bzgl. der Arithmetik. Die Verhaltensvariante des Eintaktmodells benötigt, abgesehen vom Inkrementierer der Befehlsadresse, nicht weniger als 6 Addierer. Daher werden Berechnungen wie Addition, Logikoperationen, wie *AND*, *OR* und *XOR*, sowie die Shift- und die Vergleichsoperationen durch ein ALU Modul übernommen (Abbildung 4.5). Die Eingänge des ALU Moduls werden durch Multiplexer, in QUARTZ also durch `if`-Anweisungen selektiert. Auch hier werden die Zuordnungen wieder soweit wie möglich zusammengefasst (Abbildung 4.6).

```
module ALUModule:
  input
    in1 : bitvec[numBitsPerInteger],
    in2 : bitvec[numBitsPerInteger],
    op : bitvec[3];
  output
    out : bitvec[numBitsPerInteger];

  case
    op=ALU_ADD do
      out := in1 + in2;
    op=ALU_AND do
      out := in1 & in2;
    op=ALU_OR do
      out := in1 | in2;
    op=ALU_XOR do
      out := ~(in1 kleiner-groesser in2);
    op=ALU_SLL do
      out :=  append(lastn(bvlength(in2)-1, in2), Zero(1));
    op=ALU_SRA do
      out := signext(1, firstn(bvlength(in2)-1, in2));
    op=ALU_SRL do
      out := zeroext(1, firstn(bvlength(in2)-1, in2));
    else // ALU_SLT
      out := ((in1 kleiner in2) ? 1 : 0)
  end
end // module
```

Abbildung 4.5: ALU Modul des Eintaktmodells

[...]

```
if opcode=INSTR_BEQ | opcode=INSTR_BNE then
  Param1 := EXTENDPC(PC)
else
  Param1 := RegInS
end;

if opcode=#B000000 then
  Param2 := RegInT
else
  Param2 := ImmediateData
end;

if isALUR & funct=INSTR_AND then Op := ALU_AND end;
if isALUR & funct=INSTR_OR then  Op := ALU_OR  end;
if isALUR & funct=INSTR_SLT then Op := ALU_SLT end;
if isALUR & funct=INSTR_SLL then Op := ALU_SLL end;
if isALUR & funct=INSTR_SRA then Op := ALU_SRA end;
if isALUR & funct=INSTR_SRL then Op := ALU_SRL end;
if isALUR & funct=INSTR_XOR then Op := ALU_XOR end;
if (isALUR & funct=INSTR_ADD) | ~isALUR then Op := ALU_ADD end;
```

[...]

Abbildung 4.6: Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - Ansteuern der ALU

Zuletzt müssen noch die Ausgänge des Prozessors gesetzt werden, deren Wert vom Ergebnis der ALU abhängt oder abhängen kann (Abbildung 4.7).

[...]

```
if opcode=INSTR_LW then
  RegOut := MemIn
else
  RegOut := AluResult
end;
if ((opcode=INSTR_BEQ) & CompareResult) |
   ((opcode=INSTR_BNE) & ~CompareResult) then
  next(PC) := AluResult
else if opcode=INSTR_J then
  next(PC) := Target
else
  next(PC) := PC + 1
end end;
```

[...]

```
  end // loop
end // module
```

Abbildung 4.7: Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - setzen der, vom Ergebnis der ALU abhängigen Ausgaben

4.3 Verifikation

Nach der Implementierung des Eintaktmodells folgt die Verifikation mit den Model Checkern. Dazu müssen dem Modul die Spezifikationen hinzufügt werden, die es erfüllen muß. Die im folgenden hergeleiteten Spezifikationen werden in QUARTZ im `spec` Abschnitt hinzugefügt. Da die oben erklärten Varianten des Eintaktmodells bis auf die unterschiedlichen Typen (Integer, Bitvektor) semantisch äquivalent sind, sind die Spezifikationen für beide gleichermaßen gültig.

Zur Verifikation des Eintaktmodells wird überprüft, ob es zu jeder Eingabe die gewünschte Ausgabe liefert. Im Klartext bedeutet dies nichts anderes, als für jeden im Prozessor implementierten Befehl die Ausgabe auf Korrektheit zu überprüfen. Im Eintaktmodell gestaltet sich diese Aufgabe recht einfach, da das Verhalten der Befehle lediglich in CTL ausgedrückt werden muß, ohne dabei besondere „Umstände“ berücksichtigen zu müssen. Zu diesen „Umständen“ gehören z.B. Konflikte, die in einem Pipelineprozessor auftreten können. Allerdings ist das Interface des Prozessors zu erweitern. CTL macht einen Vergleich zwischen Variablen und Werten nur zu einem Zeitpunkt möglich, kann jedoch nicht die Variablen innerhalb **eines** Vergleichs zu **zwei** verschiedenen Zeitpunkten betrachten. Aus diesem Grund muß für die Befehlsadresse eine zusätzliche Variable deklariert werden, die den Wert des PC-Registers des jeweils letzten Taktes speichert. Ebenso muß für *ImmediateData* und für *Target* verfahren werden. Abbildung 4.8 zeigt, wie die Spezifikationen im Allgemeinen aussehen.

specInstr: `AG ((Opcode=`*OPCODE*`)` $\rightarrow$ *Verhalten*`)`

Abbildung 4.8: Allgemeine Spezifikation eines Befehls

Am Befehl `ADD` soll gezeigt werden, wie eine Spezifikation für einen Befehl in LTL erstellt wird. Die Konstruktion anderer Befehle verläuft analog.
Der Befehl `ADD` hat folgendes Format [Mip02]:

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 000000	ADD 100000
6	5	5	5	5	6

`ADD` wird durch den im Befehlsformat beschriebenen Opcode(= 000000) und Function Code(= 100000) beschrieben. Weiterhin sind die Indizes der Eingaberegister sowie der Index des Ausgaberegisters enthalten. Zuerst müssen die Register *rs* und *rt* ausgelesen werden. Im obigen Teil wurde erklärt, daß das Setzen der Indizes über den Prozessor nicht notwendig ist. An dieser Stelle kann das Setzen der Indizes nicht beobachtet werden und ist somit nicht verifizierbar. Um fortzufahren wird das korrekte Setzen der Indizes vorausgesetzt. Unter der Voraussetzung, daß die Indizes korrekt gesetzt werden, liegen die Werte der Register zur weiteren „Verarbeitung“ bereit. Die Werte der Eingaberegister müssen addiert und das Ergebnis im Register, das durch *rd* indiziert wird, gespeichert

werden. Der Prozessor muß also den Index des Zielregisters, das Signal zum Schreiben des Registers und das Ergebnis richtig setzen. Bei der Beschreibung der Spezifikation ist zu beachten, daß der Prozessor bei einem Befehl nicht nur das macht, was er machen **muß**, sondern zudem das, was er nicht machen **darf**, nicht macht. In diesem Falle wäre das ein schreibender Speicherzugriff. Also darf das Signal zum Schreiben in den Datenspeicher sowie das Signal UNKNOWN_OPCODE nicht gesetzt sein. Zuletzt ist noch der Befehlszeiger zu erhöhen und abzuspeichern. Das Verhalten des `ADD` Befehls, beschrieben in CTL, zeigt Abbildung 4.9. Da dieses Verhalten nur für den `ADD` Befehl gelten muß, ist für die vollständige Spezifikation des `ADD` Befehls, der obige Ausdruck die Implikation der Bedingung „Befehl ist `ADD`". Abbildung 4.10 zeigt die vollständige Spezifikation in CTL.

$$
\begin{aligned}
&RegIndex = InstrReg.RegRD \wedge \\
&RegOut = RegInS + RegInT \wedge \\
&\qquad \neg UNKNOWN_OPCODE \wedge \\
&\qquad RegWrite \wedge \\
&\qquad \neg MemWrite \wedge \\
&\qquad X(PC = lastPC + 1)
\end{aligned}
$$

Abbildung 4.9: Spezifikation des Verhalten des Prozessors bei anliegendem `ADD`-Befehls

$$
\begin{aligned}
&AG((InstrReg.Opcode = 000000_2 \wedge \\
&\quad InstrReg.Funct = \text{,ADD'}) \rightarrow \\
&\quad (RegIndex = InstrReg.RegRD \wedge \\
&\quad RegOut = RegInS + RegInT \wedge \\
&\quad \neg UNKNOWN_OPCODE \wedge \\
&\quad RegWrite \wedge \\
&\quad \neg MemWrite \wedge \\
&\quad X(PC = lastPC + 1)))
\end{aligned}
$$

Abbildung 4.10: vollständige Spezifikation des Verhalten des Prozessors bei anliegendem `ADD`-Befehl

Analog dazu wird für jeden weiteren Befehl eine Spezifikation aufgebaut. So wird das korrekte Verhalten bei einem anliegendem Befehl gesichert. Damit ist der Prozessor jedoch noch nicht vollständig spezifiziert. Es ist ebenso sicherzustellen, daß bei unbekannten Befehlen die Signale, die ausschlaggebend für Änderungen des Systems sind, nicht gesetzt werden. Zu diesen Signalen gehören `RegWrite` und `MemWrite`, allerdings auch das Signal `UNKNOWN_OPCODE`, welches in der hier implementierten Variante keine direkten Änderungen am System vornimmt. Bei Implementierung eines Interrupt-Controllers können durch Aufruf einer Behandlungsmethode bei einem falsch gesetztem `UNKNOWN_OPCODE` indirekte Änderungen durch die Behandlungsmethode am System hervorgerufen werden. In Abbildung 4.11 sind die dazugehörigen Spezifikationen aufgelistet. Hier wurde auf ein ausführliches Aufzählen aller Opcodes verzichtet und die jeweilige Menge von Befehlen mit Worten umschrieben.

$AG(RegWrite \leftrightarrow$ (*bekannter Befehl liegt an* $\wedge$ *Befehl schreibt Register*))
$AG(MemWrite \leftrightarrow$ *Befehl ist Speicherbefehl*)
$AG(UNKNOWN_OPCODE \leftrightarrow$ *kein bekannter Befehl liegt an*)

Abbildung 4.11: Spezifikationen des Verhaltens von systemverändernden Signalen

4.3.1 Benchmark

Die Spezifikationen des Eintaktmodells werden mit den Model Checkern `SMV` von Cadence Berkeley Laboratories und `Beryl` des AVEREST-Paketes verifiziert. Für das Eintaktmodell sind 17 Spezifikationen zu verifizieren sowie 6 weitere um die Abwesenheit von Schreibkonflikten zu zeigen. Für die ALU sind es 9 Spezifikationen ohne Test auf Schreibkonflikte, mit Test auf Schreibkonflikte 10 Spezifikationen. Als erstes sind die Ergebnisse für die ALU aufgelistet (Tabelle 4.1). Danach folgen die Ergebnisse der Eintaktmodelle (Tabelle 4.2).

Ausgeführt wurde der Benchmark unter Suse Linux 9.3 auf einem PC mit zwei AMD Opteron Prozessoren mit jeweils zwei Gigahertz. Trotz der vier Gigabyte Arbeitsspeicher, mit denen der Rechner ausgestattet ist, wurde jedem Programm maximal anderthalb Gigabyte zugesprochen, um Zugriffe auf die Auslagerungspartion im Parallelbetrieb von zwei Model Checkern auszuschließen. Bei beiden Model Checkern wurde `SIFT` als variable reorder Methode gewählt. Während `Beryl` diese Einstellung standardmäßig verwendet, muß bei `SMV` diese explizit angegeben werden. `beryl` bietet für die Verifizierung verschiedene BDD-Pakete und zwei Pakete zur Behandlung von Presburger-Arithmetik. Letztere sind nur notwendig, wenn mit beliebig großen Zahlen gerechnet wird – mit anderen Worten, falls Variablen vom Datentyp Integer verwendet wurden. Aus den BDD Paketen wurde `CUDD` gewählt. Zur Verifizierung des Ganzzahlmodells wurde `beryl-cudd-dfa` verwendet.

Aus Tabelle 4.1 und Abbildung 4.12 kann entnommen werden, daß die Verifikation selbst für aktuelle Bitbreiten zügig abläuft. Allerdings mit der entscheidenden Einschränkung, daß die ALU keine Ganzzahlmultiplikation beherrscht, geschweige denn Fließkommaarithmetik besitzt. Auch bei den Multiplexern wurde gespart, da diese die Register lediglich um 1 Bit schieben können. Bei Einbettung von komplizierter Arithmetik, wie Fließkommaarithmetik oder Ganzzahl- oder Fließkommamulitplizierer, dürften alle Reserven schnell verbraucht sein und die Verifikation möglicherweise nicht mehr möglich sein. Hier bietet es sich an, die ALU weiter zu modularisieren. Multiplizierer und andere kompliziertere, bzw. aufwändige Rechenarbeiten jeweils in ein separates Modul zu packen und die Korrektheit der Operation im jeweiligen Modul zu überprüfen. Bei Einbindung dieser Module in ein anderes Modul (z.B. die ALU) sollten diese – zur Verifizierung des Moduls – anstatt das Ergebnis zu berechnen, einen eindeutigen Code zurückliefern. Bei der Verifizierung muß, anstatt das Ergebnis mit dem Referenzwert zu vergleichen, die Quelle überprüft werden. Selbstverständlich ist diese Vereinfachung nur dann möglich, falls das Ergebnis nicht für weitere Operationen relevant ist.

4.3.2 Auswertung

ALU

	Zeit		maximale Anzahl an BDD Knoten	
Bitbreite/Datentyp	Beryl	SMV	Beryl	SMV
4 Bit	0.16s	0.11s	13286	1334
5 Bit	0.19s	0.10s	16352	10033
6 Bit	0.23s	0.14s	20440	10075
7 Bit	0.31s	0.17s	24528	10368
8 Bit	0.49s	0.18s	20440	10230
9 Bit	0.44s	0.22s	10220	10296
10 Bit	0.56s	0.28s	22484	10118
11 Bit	0.66s	0.33s	26572	10027
12 Bit	0.79s	0.40s	30660	20276
16 Bit	1.24s	0.86s	39858	44166
20 Bit	3.35s	0.87s	73584	34456
24 Bit	7.34s	1.73s	152278	44645
29 Bit	6.43s	2.25s	122640	57976
30 Bit	7.50s	2.85s	156366	59312
31 Bit	11.71s	2.73s	165564	64319
32 Bit	12.00s	2.99s	142058	65915
33 Bit	10.08s	3.20s	200312	71027
34 Bit	7.49s	>1.5GB	169652	
35 Bit	11.96s	>1.5GB	261632	
60 Bit	140.08s	19.92s	1072078	177604
61 Bit	145.79s	20.72s	890162	185584
62 Bit	257.38s	11.42s	985208	201777
63 Bit	227.40s	21.00s	1216180	320214
64 Bit	117.11s	21.72s	655102	333334
65 Bit	105.89s	19.92s	855414	346665
66 Bit	9724.19s	20.66s	10334464	353692
Integer	3.32s	–	75628	

Tabelle 4.1: Ergebnisse des Benchmarks der Verifizierung des ALU Moduls

Prozessor

Aus der Tabelle 4.2 und der Abbildung 4.14 kann entnommen werden, daß mit der Bitbreite auch die Dauer der Verifikation steigt. Ebenso steigt die maximale Anzahl an BDD-Knoten mit wachsender Bitbreite des Datenpfades (siehe Abbildung 4.15). Die Anzahl der Knoten ist proportional zur Zeit. Daraus kann gefolgert werden, daß, bei Verwendung von weniger Variablen, in der Regel zur Verifizierung weniger Knoten erstellt werden müssen, was sich wiederrum in einer schnelleren Verifikation auswirkt. Bei

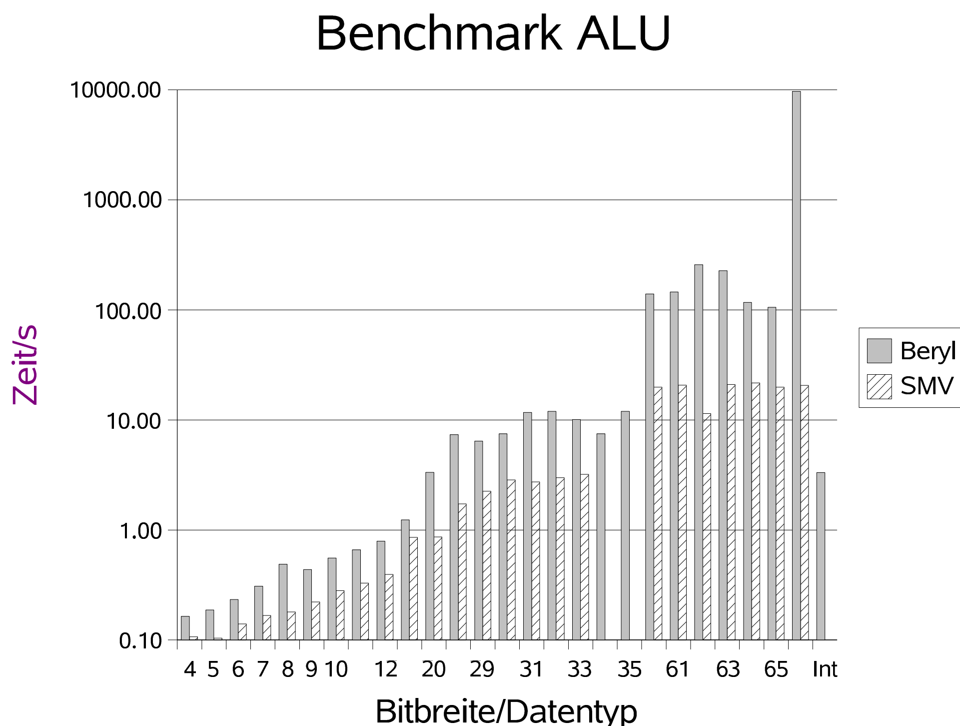


Abbildung 4.12: Zeiten der Benchmarks des ALU Moduls

der 8-Bit- und bei der 9-Bit-Variante des Eintaktmodells ist allerdings zu erkennen, daß der Anstieg nicht stetig ist. Dies ist auf die Reordering Methode der beiden Model Checker zurückzuführen.

Beim Vergleich des Verhaltensmodells mit dem Strukturmodell fällt auf, daß letzteres ein Vielfaches der Zeit benötigt, die zur Verifizierung der Ganzahlvariante aufgebracht werden muß. Ebenso ist die Anzahl der BDD Knoten um ein Vielfaches höher. Während `Beryl` für die Verifizierung der 10-Bit-Variante des Strukturmodells eine maximale Anzahl von 4832016 BDD Knoten angegeben wird, reicht weniger als ein Zehntel (exakt 360766) an BDD Knoten zur Verifizierung des 10-Bit-Variante des Verhaltensmodells. Das Strukturmodell benötigt zur Ansteuerung der ALU lokale Variablen, die selbstverständlich im zu verifizierenden Modell ebenfalls vorhanden sind und damit den Zustandsraum erhöhen. In der Synthese beschreiben diese Variablen jedoch keine Register sondern im Modell zur Hardwaresynthese lediglich Abgriffspunkte an Leiterbahnen. Im Gegensatz zum Modell, welches zur Verifikation betrachtet wird, wird daher die Anzahl der Zustände im realen Modell nicht erhöht. Es wäre sinnvoller gewesen, anstatt lokaler Variablen `let`-Anweisungen zu verwenden. Der Nachteil ist, daß der Code dadurch etwas unleserlicher geworden wäre. Der Extremfall wäre hier die Bestimmung des ALU Opcodes, welche aus 8 Auswertungen besteht, die wiederum in einen Ausdruck gepackt

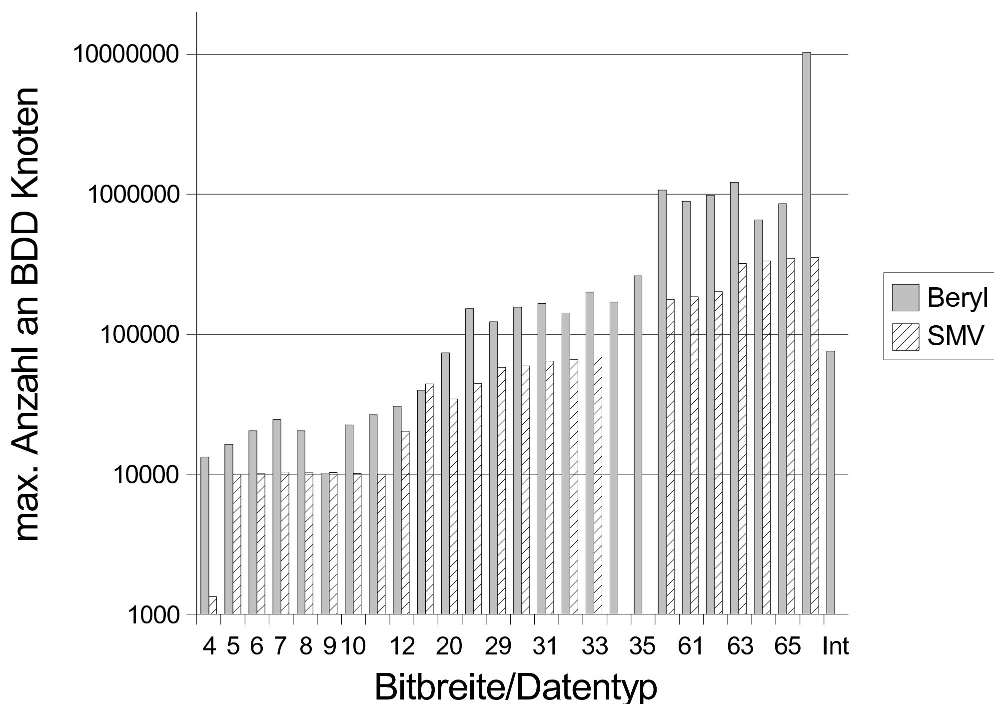


Abbildung 4.13: Speicherbedarf der Benchmarks des ALU Moduls - Angegeben ist der Peaklevel der erstellten BDD-Knoten

werden müssten. Eindeutiger Vorteil bei dieser Vorgehensweise wäre jedoch gewesen, daß der Zustandsraum so groß geblieben wäre wie der des Verhaltensmodells. Die Verifikation wäre vermutlich dementsprechend zügiger vorangegangen. Man muß jedoch beachten, daß eine Beschleunigung der Verifikation nicht erfolgen muß, da das Transitionssystem anders ausfallen kann und ebenfalls Einfluß auf die Dauer der Verifikation hat. Nachträglich wurden diese Änderung noch durchgeführt. Bei nachträglichen Durchläufen hat sich gezeigt, daß die Verifikation des modifizierten Modells und die des Verhaltensmodells für 4 bzw. 5 Bit breite Datenpfade in etwa gleich schnell ist. Die Ergebnisse der Verifikation des modifizierten Modells können der Tabelle 4.3 entnommen werden. Bei einer Breite des Datenpfades von 6 Bit und 8 Bit war die Verifikation des modifizierten Strukturmodells wesentlich schneller als die des nicht modifizierten Strukturmodells. Allerdings dauerte die Verifikation des modifizierten Modells mit `Beryl` trotz gleich großem Zustandsraum immer noch länger als die Verifikation des Verhaltensmodells.

Wie schon erwähnt, steigt bei beiden Modellen (das Verhaltensmodell und das unmodifizierte Strukturmodell; das modifizierte Strutkurmodell wird jetzt nicht mehr betrachtet) mit der Bitbreite des Datenpfades die Zeit, die zur Verifikation benötigt wird. Gerade bei dem Modell für die Hardwaresynthese wäre eine feste Bitgröße interessant, da Hardware

	Verhaltensmodell		Strukturmodell	
Bitbreite/Datentyp	Beryl	SMV	Beryl	SMV
4 Bit	1.3s	1s	3.6s	62s
5 Bit	2.1s	4.6s	107s	276s
6 Bit	3.3s	6.1s	200s	4660s
7 Bit	7.4s	19.3s	60s	>1.5GB
8 Bit	14.5s	30.7s	597s	>1.5GB
9 Bit	21s	71s	1656s	>1.5GB
10 Bit	26s	1146s	4165s	>1.5GB
11 Bit	122s	2152s	1757s	>1.5GB
12 Bit	153s	1217s	>4h	>1.5GB
16 Bit	6642s	>1.5GB	>4h	>1.5GB
Integer	4.7s	-	423.3s	-

Tabelle 4.2: Ergebnisse der Benchmarks der Eintaktvarianten

Bitbreite	Verhaltensmodell	Strukturmodell	modifiziertes Strukturmodell
4 Bit	1.3s	3.6s	1,2s
5 Bit	2.1s	107s	2,6s
6 Bit	3.3s	200s	43s
8 Bit	14.5s	597s	34s

Tabelle 4.3: Vergleich der beiden Modelle mit dem modifizierten Eintakt-Strukturmodell

mit festen Bitbreiten hergestellt wird. Allerdings steigt der Aufwand zur Verifikation gerade bei diesem Modell so weit an, daß es ab einer Größe mit 11 Bit nicht mehr innerhalb von 4 Stunden verifiziert werden kann. Zumindest kommerzielle Prozessoren haben mittlerweile mindestens 32 Bit, der Trend läuft sogar zu 64 Bit Prozessoren. Das Strukturmodell mit einer Bitbreite des Datenpfades in dieser Größenordnung zu verifizieren wird offensichtlich zu viel Zeit in Anspruch nehmen, auch wenn man auf das optimierte Modell zurückgegriffen würde. Hier wäre eine Abstrahierung in der Art, wie sie im Abschnitt 4.3.2 beschrieben wurde, denkbar.

Weniger schnell steigen die Zeiten im Verhaltensmodell. Hier lässt Abbildung 4.14 vermuten, daß sich das Modell noch mit 16 Bit breiten Datenpfaden verifizieren lässt. Allerdings dürfte dies für das Verhaltensmodell weniger von Interesse sein, da dieses eher für die Softwaresynthese gedacht ist. Als Datentyp werden die Ganzzahlen die Präferenz sein. Die Verifikation liegt hier mit ca. 5 Sekunden zwischen der 6 Bit und der 7 Bit Variante des gleichen Modells. Zum Verhaltensmodell ist zu erwähnen, daß diese zwar nicht auf das ALU-Modul, welches in das Strukturmodell eingebettet wurde, zurückgreift, sich jedoch letztendlich die gleichen Probleme stellen, denn das Modell

- hat nur wenige Befehle
- beherrscht keine Multiplikation

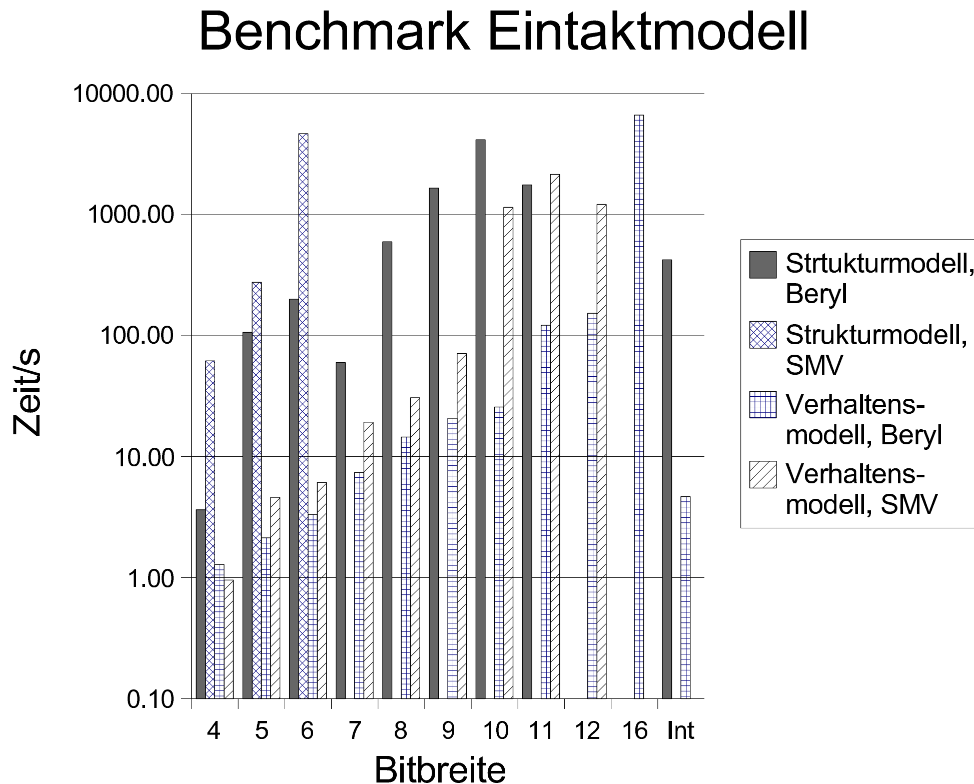


Abbildung 4.14: Zeiten der Benchmarks der Eintaktvarianten

- beherrscht keine Fließkommaarithmetik

Als Lösung bieten sich ähnliche Ansätze wie in 4.3.2 beschrieben. Aufwändige Arithmetik sollte in eigene Module gepackt und verifiziert werden. Zur Verifikation des Prozessors werden diese Module nicht mehr instanziiert. Stattdessen muß dem Ergebnisregister ein eindeutiger Identifikationscode zugeordnet und dieser in den Spezifikationen dementsprechend abgefragt werden.

Fazit

Um die Verifikation zu beschleunigen, kann man allgemein nur den Rat geben, auf größere Module das „Divide and Conquer“-Verfahren anzuwenden. Das heißt, daß diese Module weiter modularisiert werden, bis die einzelnen Teilmodule verifiziert werden können. Allerdings stellt sich die Frage, ab wann ein Modul als größeres Modul zu beschreiben ist. Tabelle 4.3 hat gezeigt, daß die Zeit, die zur Verifikation benötigt wird, nicht allein von der Größe des Zustandsraumes abhängt. Die Reordering-Methode des vom Model Checker verwendeten BDD-Paketes sowie die zum Aufbau der Kripke-Struktur verwendeten Transitionsgleichungen, welche vom Compiler aus dem Code generiert werden, können die Zeit beeinflussen. Somit lässt sich die Dauer der Verifikation nicht vorhersa-

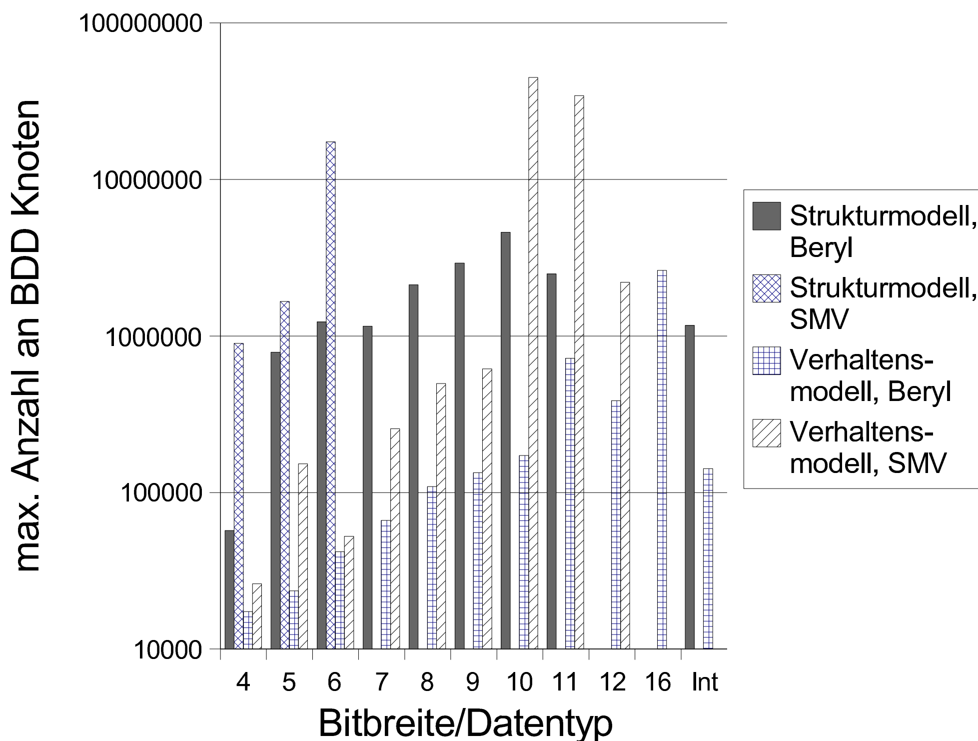


Abbildung 4.15: Speicherbedarf der Benchmarks der Eintaktvarianten - Angegeben ist der Peaklevel der erstellten BDD-Knoten

gen. Zuletzt bleibt lediglich die Möglichkeit, die Optionen der Model Checker auszureizen bzw. deren Parameter zu variieren.

4.4 Synthese

Die Synthese der Prozessoren ist leider ohne weiteres nicht möglich. Zum einen muß noch der Registersatz implementiert werden, was kein Problem darstellt. Zum anderen muß nochmals ein Blick auf die Schnittstelle des Speichers, genauer gesagt auf das vorausgesetzte Speicherverhalten, geworfen werden. Hier zeigt sich, welchen entscheidenden Nachteil die Vereinfachung der Schnittstelle des Speicher hat. Es wurde vorrausgesetzt, daß der Speicher (Daten- sowie Befehlsspeicher) das gleiche Verhalten wie das Registerarray zeigt. Er muß innerhalb eines Taktzyklus, in dem eine Adresse angelegt wird, das Datum an den Eingang des Prozessors legen (Abbildung 4.16).

Die Synthese des Prozessors, wie er momentan vorliegt, ist für FPGA-Boards aus diesem Grund in der Regel nicht möglich, da übliche Speichermodule die Daten erst nach einem

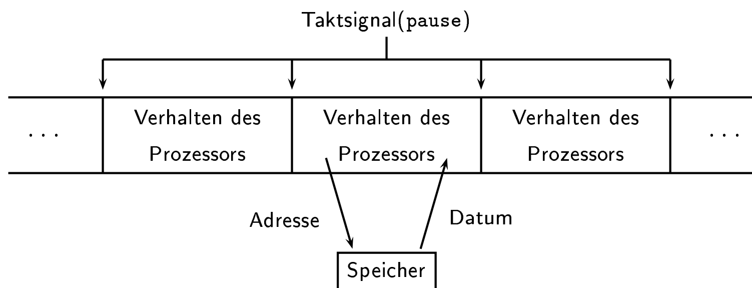


Abbildung 4.16: Verhalten des idealisierten Speichers

oder mehreren Taktzyklen liefern. Synthese auf Basis von Software ist momentan ebenfalls noch nicht möglich, da eine Implementierung des Speichers auf der Zielplattform keinen Sinn macht. Selbstverständlich ist es möglich, auf die Ausgaben des Prozessors mit Eingaben durch Setzen seiner Eingänge zu reagieren. Allerdings ist dies nur bei Taktsignalen möglich (Abbildung 4.17). Aus diesem Grund muß der Speicher ebenfalls in QUARTZ implementiert werden. Für eine Simulation in Software ist dies in Ordnung, für die Synthese in Hardware wäre das jedoch nicht akzeptabel, da dies bedeutet, daß der Speicher auf dem FPGA implementiert werden würde.

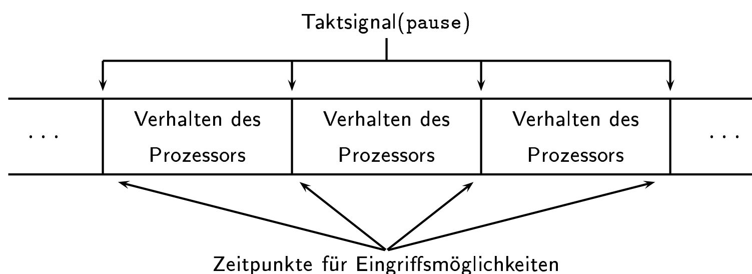


Abbildung 4.17: Eingriffsmöglichkeiten bei Softwaresynthese

Um die Simulation des vorliegenden Prozessors zu ermöglichen, muß außer dem Registersatz auch der Daten- und Befehlsspeicher implementiert werden. Das Verhalten der letzten beiden kann glücklicherweise vom Registersatz abgeleitet werden. Aufgrund der Tatsache, daß der Speicher nun ebenfalls in QUARTZ implementiert wird, stellt sich allerdings noch die Frage, wie dieser initialisiert werden soll. Hier wird im später modellierten Gesamtsystem, welches den Prozessor, das Registerarray und die beiden Speicher umfasst, das Signal `RunRISC` definiert. In Abhängigkeit des Wertes des Signals wird das System als geschlossenes System betrieben oder mit externen Werten initialisiert. Die genauere Funktionsweise wird später beschrieben.

4.4.1 Registersatz

Auf die Implementierung des Registersatzes soll nicht allzu detailiert eingegangen werden. Daher wird zu der Schnittstelle kein QUARTZ-Code angegeben. Die Variablen die

notwendig sind, werden lediglich aufgezählt. Daher ist die Schnittstelle recht schnell beschrieben. Zu den Eingaben gehören die Variablen $RegIndex_{RS}$, $RegIndex_{RT}$, $RegIndex_{RD}$, $RegWrite_{RD}$ und $RegValue_{RD}$. Zu den Ausgaben gehören die Werte der Register RS und RT. Die Typen können der Schnittstelle des Prozessors entnommen werden. Zu beachten ist, daß zwischen der Bitvektor- und der Integervariante des Prozessors unterschieden werden muß und der Datentyp der Register davon abhängt. Bevor das Verhalten des Registerarrays beschrieben werden kann, wird noch ein Array benötigt, um alle Werte der Register speichern zu können. Dieses wird hier *register* genannt und lokal deklariert. Abbildung 4.18 zeigt den QUARTZ-Code des Registerarrays. Ist das Statusflag $RegWrite_{RD}$ gesetzt, muß der anliegende Wert in das entsprechende Register übernommen werden. In jedem Fall müssen die Ausgänge mit den Werten der gewählten Register gesetzt werden.

```
local register:  array 32 of Datentyp in

loop
  if RegWriteRD then
    next(register[RegIndexRD]) := RegValueRD
  end;
  RegValueRS := register[RegIndexRS];
  RegValueRT := register[RegIndexRT];
  pause
end // loop
end // local
```

Abbildung 4.18: Implementierung des Registersatzes

4.4.2 Befehls- und Datenspeicher

Die Schnittstelle und das Verhalten des Befehls- und des Datenspeichers können vom Registersatz abgeleitet werden, da diese sich gleich verhalten. Die einzige Änderung besteht darin, daß die Speicher jeweils nur einen Ausgang haben. Zu jedem Speicher wird eine eigene Implementierung benötigt, da sich beide im Typ des Speichers unterscheiden. Während die Befehle aus Bitvektoren mit jeweils 32 Bit bestehen, können die Daten zum einen ebenfalls als Bitvektoren dargestellt werden - allerdings mit variabler Bitbreite - und zum anderen als Ganzzahlen. Obwohl der Prozessor nur lesenden Zugriff auf den Befehlsspeicher hat, muß dieser trotzdem Befehle speichern können, da dieser erst während der Laufzeit initialisiert werden soll.

4.4.3 Gesamtsystem

Wie auch die anderen Module, benötigt das Gesamtsystem eine Schnittstelle. Diese Schnittstelle muß alle Variablen zur Verfügung stellen, die notwendig sind, um den Prozessor zu initialisieren, sowie starten zu können. Hier wird das System für die Bitvektor-

variante implementiert (Abbildung 4.19). Für das Ganzzahlmodell wäre es notwendig, die Befehle gegebenenfalls zu zerlegen und Daten, wie Sprungadresse oder Immediate-Datum, getrennt zu speichern.

```
input
  ExtInstrAddr: bitvec[numBitsPerInteger],
  ExtInstrValue: bitvec[32],
  ExtInstrWrite,
  ExtMemAddr: bitvec[numBitsPerInteger],
  ExtMemValueIn: bitvec[numBitsPerInteger],
  ExtMemWrite,
  RunRISC: boolean;
output
  ExtMemValueOut: bitvec[numBitsPerInteger],
  UNKNOWN_OPCODE,
  PC: bitvec[numBitsPerInteger];
```

Abbildung 4.19: Implementierung des Gesamtsystems - Schnittstelle

```
local
  MemIn: bitvec[numBitsPerInteger],
  RegValueRS: bitvec[numBitsPerInteger],
  RegValueRT: bitvec[numBitsPerInteger],
  RISCInstrReg: bitvec[32],
  InstrReg: bitvec[32],
  MemOut: bitvec[numBitsPerInteger],
  MemAddr: bitvec[numBitsPerInteger],
  MemWrite,
  RegIndexRD: bitvec[5],
  RegWriteRD,
  RegValueRD: bitvec[numBitsPerInteger],

  InstrMemAddr: bitvec[numBitsPerInteger],
  InstrMemWrite,
  DataMemOut: bitvec[numBitsPerInteger],
  DataMemAddr: bitvec[numBitsPerInteger],
  DataMemWrite
in
```

Abbildung 4.20: Implementierung des Gesamtsystems - lokale Variablen

In Abbildung 4.19 ist zu erkennen, daß es für Befehls- und Datenspeicher jeweils ein Variablen-Tripel gibt, welches zur Initialisierung der Speicher dient. Zu jedem Tripel

gehört die Adresse, das Datum, bzw. der Befehl der geschrieben werden soll, sowie der Status, ob das Datum, bzw. der Befehl geschrieben werden soll. Die letzte Eingabevariable ist das oben erwähnte Flag zur Unterscheidung zwischen dem Initialisierungs- und Laufmodus des Systems. Die Ausgabevariablen dienen zum Auslesen des Speichers und um den Status des Prozessors abzufragen.

Um die Module verbinden zu können, werden weitere Variablen benötigt, welche lokal deklariert werden (Abbildung 4.19. Einige dieser Variablen werden allerdings nicht nur als Verbindung zwischen den Modulen benötigt, sondern auch, um zwischen Initialisierungs- und Laufmodus des Systems unterscheiden zu können. Welche Aufgaben die Variablen genau haben, kann durch Betrachtung der Implementierung des Verhalten (Abbildung 4.21) herausgefunden werden. Im ersten Teil werden die Module instanziiert und gestartet. Der letzte Teil, die Schleife, ist notwendig, um das System in zwei verschiedenen Modi laufen lassen zu können. Hier werden in Abhängigkeit des Modus die Eingänge der Module gesetzt. Zum ersten gehören die Adressen der Speicher. Im Initialisierungsmodus sind sie auf die Eingänge des Systems zu setzen, während sie im Laufmodus vom Prozessor gesetzt werden. Ebenso verhält es sich mit den Schreibflags, welche an den Speicher angelegt werden. Beim Befehlsspeicher sind die Eingänge für den Schreibzugriff etwas einfacher zu setzen, da der Prozessor selbst keinen schreibenden Zugriff auf ihn hat. Zuletzt muß noch dafür gesorgt werden, daß keine Änderung am Registerarray vorgenommen werden und der Prozessor in einem „Ruhezustand“ bleibt. Dies wird dadurch erreicht, indem er im Initialisierungsmodus mit `NOPs`, anstatt mit den im Befehlsspeicher enthaltenen Befehlen bedient wird.

An dieser Stelle soll ein kurzer Vorgriff auf die Pipelinevariante des Prozessors genommen werden. Der Pipelineprozessor könnte aufgrund seines Konzepts nach dem Umschalten vom Laufmodus in den Initialisierungsmodus noch Befehle bearbeiten, welche Schreibzugriffe auf den Datenspeicher versuchen. Diese würden jedoch ignoriert werden. Das *RunRISC*-Flag sollte also wirklich nur zur Initialisierung verwendet werden und nachdem es einmal gesetzt wurde, um das System in den Laufmodus zu schalten, nicht mehr zurückgesetzt werden.

```
RISC: run SEQUENTIAL_RISC
    (ExtMemValueOut,
     RegValueRS, RegValueRT,
     RISCInstrReg)
    (MemOut, MemAddr, MemWrite,
     RegValueRD, RegIndexRD, RegWriteRD,
     PC, UNKNOWN_OPCODE)
||
RegArray: run RegisterArray
    ({RISCInstrReg : 25..21},
     {RISCInstrReg : 20..16},
     RegIndexRD, RegWriteRD,RegValueRD)
    (RegValueRS, RegValueRT)
||
InstrMem: run InstrMemory
    (ExtInstrValue, InstrMemAddr, InstrMemWrite)
    (InstrReg)
||
Mem: run Memory
    (DataMemOut, DataMemAddr, DataMemWrite)
    (ExtMemValueOut)
||
loop
  InstrMemAddr := (RunRISC ? PC : ExtInstrAddr);
  if((~RunRISC) & ExtInstrWrite) then
    emit InstrMemWrite
  end;
  DataMemOut := (RunRISC ? MemOut : ExtMemValueIn);
  DataMemAddr := (RunRISC ? MemAddr : ExtMemAddr);
  if((RunRISC & MemWrite) | (~RunRISC & ExtMemWrite)) then
    emit DataMemWrite
  end;
  RISCInstrReg := (RunRISC ? InstrReg : 0);
  pause
end // loop
end // local
```

Abbildung 4.21: Implementierung des Gesamtsystems - Verhalten

5 Pipeline-Modell

Prozessoren, in denen die Befehle wie im zuvor vorgestellten Eintaktmodell sequentiell abgearbeitet werden, sind heutzutage nicht mehr verbreitet. Wegen der großen Gatterstruktur und der dadurch resultierenden langen Laufzeit von Signalen sind solche Prozessoren gezwungen mit niedrigen Taktraten zu laufen. Nicht nur im kommerziellen, sondern auch in den industriellen Bereichen steigt das Verlangen nach mehr Leistung. Der erste Weg, diese zu steigern ist die Erhöhung der Taktrate. Dies kann durch Teilung der Befehlsabarbeitung in mehrere Schritte erreicht werden. Jeder Schritt wird dabei von einer eigens spezialisierten Stufe erledigt, deren Schaltung durch die Spezialisierung auf den einzelnen Arbeitsschritt kleiner ausfällt, die Signalwege kürzer sind und daher mit einem höherem Takt laufen kann. Die Stufen werden hintereinandergeschaltet, so daß diese parallel arbeiten und nach der Bearbeitung eines Befehls diesen zur weiteren Verarbeitung an die nächste Stufe weitergeben. Trotz der parallelen Abarbeitung ist zu beachten, daß die Befehle *in order* abgearbeitet werden, also die Reihenfolge der Befehle nicht verändert wird. Abbildung 5.1 zeigt das Prinzip der Abarbeitung in einer Pipeline. Das Pipelining bringt trotz in-order-Verarbeitung der Befehle Probleme mit sich. Auf diese Probleme wird in 5.1 eingegangen. Ein Befehl durchläuft während seiner Abarbeitung jede Stufe der Pipeline. Hat eine Stufe einen Befehl verarbeitet, wird dieser an die nächste Stufe weitergereicht. Während sich ein Befehl in einer Stufe befindet, wird in der vorhergehenden Stufe der nächste Befehl verarbeitet. In einer Pipeline mit n Stufen befinden sich somit bis zu n Befehle. Letztendlich wird durch das Pipelining nicht der Abarbeitung eines Befehls beschleunigt, sondern die Abarbeitung einer Befehlskette.

	Stufe				
Takt	`IF`	`ID`	`EX`	`MEM`	`WB`
$C0$	$I0$				
$C1$	$I1$	$I0$			
$C2$	$I2$	$I1$	$I0$		
$C3$	$I3$	$I2$	$I1$	$I0$	
$C3$	$I4$	$I3$	$I2$	$I1$	$I0$
...			...		

Abbildung 5.1: Abarbeitung einer Befehlskette in einem Pipeline Prozessor

5.1 Pipelinekonflikte

Beim Entwurf von Pipeline-Prozessoren ist darauf zu achten, daß die Abhängigkeit der Befehle und ihre Reihenfolge bei der Ausführung von Programmen zu Konflikten führen können. Dabei teilt man die Konflikte in verschiedene Typen und Ursachen ein.

5.1.1 Datenkonflikte

RAW-Konflikte

Bei RAW-Konflikten versucht ein Befehl ein Register zu lesen, das von einem vorhergehenden Befehl verändert wird, aber noch nicht in das Registerarray zurückgeschrieben wurde. Daher würde der neuere Befehl fälschlicherweise mit einem veraltetem Datum rechnen. Zur Lösung dieses Problems verwendet man *Result Forwarding*. Dabei werden berechnete Ergebnisse der Pipeline Stufen nicht nur an ihre nachfolgende Pipeline Stufe weitergegeben, sondern auch an die vorhergehenden, für die das Ergebnis als potentielle Eingabe dient. Nicht immer sind RAW-Konflikte durch Forwarding auflösbar. Wird ein Ergebnis benötigt, bevor es berechnet wurde, müssen die Pipelinestufen, die das Ergebnis erwarten, bzw. sich vor einer Pipelinestufe befinden, die auf das Ergebnis wartet, mit der Befehlsverarbeitung warten, bis das Ergebnis berechnet wurde. Eine andere Möglichkeit der Auflösung dieses Konflikts wäre die Umordnung der Befehle, welche hier nicht behandelt wird. Die folgenden beiden Konflikte betreffen lediglich parallele Prozessoren und werden hier nur zur Vollständigkeit der Datenkonflikte beschrieben.

WAR-Konflikte

Eine weitere Technik zur besseren Nutzung der Parallelität und der daraus folgenden Erhöhung des Befehlsdurchsatz ist das dynamische Scheduling. Hierbei werden die Befehle durch den Prozessor umgeordnet. Durch die Umordnung kann es dazu kommen, daß ein Befehl versucht ein Ergebnis in ein Register zu schreiben, bevor ein ihm in der Befehlskette vorausgehender Befehl, der dieses Register als Eingabe benötigt, gelesen hat.

WAW-Konflikte

Es ist ebenso möglich, daß zwei Befehle, die in das gleiche Register schreiben, umgeordnet werden können. Hier ist darauf zu achten, daß die Ergebnisse in der richtigen Reihenfolge geschrieben werden und neuere Ergebnisse nicht durch alte überschrieben werden.

Wie zuvor erwähnt wurde, betreffen WAR- und WAW-Konflikte nur Prozessoren, die Befehle umordnen. Das hier vorgestellte Pipeline-Modell führt die Befehle „in-order“ aus und ist somit nicht von diesen Konflikten betroffen.

5.1.2 Steuerkonflikte

Aufgrund der Abarbeitung der Befehle in verschiedenen Schritten, kommt es im Steuerfluß zu Verzögerungen. Die Auswertung von bedingten Sprüngen geschieht in der Regel nicht in der ersten Stufe, sondern erst in der sogenannten ausführenden Stufe, die auch die effektive Sprungadresse berechnet. Während ein Sprungbefehl ausgewertet wird, werden daher weitere Befehle in die Pipeline geladen, obwohl diese möglicherweise nicht ausgeführt werden sollen. Im folgenden werden drei Möglichkeiten vorgestellt, wie man mit der Verzögerung der Auswertung von Sprungbefehlen umgehen kann.

- Die erste und ineffizienteste Maßnahme, die Verzögerung in den Griff zu bekommen, ist, die Pipeline solange mit `NOPs` zu füllen, bis das Ergebnis des Sprungs und des korrekten `PCs` feststeht. Das Einfügen der `NOPs` kann vom Compiler übernommen werden oder aber auch durch Implementierung einer entsprechenden Logik im Prozessor realisiert werden.

- Um die Effizienz des Prozessors bei Sprüngen zu verbessern, versucht man eine Vorhersage über den Sprung zu treffen. Die einfachste Vorhersage ist anzunehmen, daß ein bedingter Sprung nie durchgeführt wird, so daß man den PC weiterhin hochzählt und Befehle einliest. Steht das Ergebnis des bedingten Sprungs fest, müssen bei falscher Sprungvorhersage die fälschlicherweise eingelesenen Befehle aus der Pipeline gelöscht werden. Zudem ist auch darauf zu achten, daß Änderungen von solchen spekulativ geladenen Befehle, verzögert werden, bis feststeht, daß sie aufgeführt werden müssen. Wie groß der Leistungsgewinn durch diese Technik ist, wird letztendlich durch die Qualität der Sprungvorhersage bestimmt.

- Die dritte Möglichkeit ist die Nutzung des sogenannten „delayed branch slot“. Alle Befehle, die nach einem Sprungbefehl bis zu dessen Sprungergebnis eingelesen werden, werden auf jeden Fall ausgeführt. Hier liegt es am Compiler diese delayed branch slots sinnvoll zu nutzen. Er kann Befehle, die vor einem bedingten Sprung ausgeführt werden sollen, das Ergebnis des Sprungbefehls jedoch nicht beeinflussen, hinter den Sprungbefehl einordnen. Ist eine Umordnung nicht möglich, muß der delayed branch slot mit `NOPs` aufgefüllt werden. Der Vorteil dieser Lösung ist, daß im Gegensatz zur Lösung mit der Sprungvorhersage kein zusätzlicher Aufwand bei der Implementierung entsteht. Der Nachteil ist, daß dieser Lösungsansatz nur möglich ist, falls man nicht an eine Abwärtskompatibilität gebunden ist, welche die Nutzung des delayed branch slot verhindert.

5.2 Spezifikationen

Das Pipeline-Modell, welches hier implementiert wird, besteht aus fünf Stufen. Es folgt eine Erläuterung der Aufgaben der einzelnen Stufen. Auf eine detailliertere Beschreibung wird im Abschnitt 5.3 „Implementierung“ eingegangen.

`IF`-Stufe	*(Instruction Fetch)* Liest den nächsten Befehl aus dem Befehlsspeicher. Diese Stufe enthält auch den Inkrementer für den `PC`.
`ID`-Stufe	*(Instruction Decode)* Entschlüsselt den Befehl und liest die Operanden.
`EX`-Stufe	*(Execution/Effective Address Calculation)* Führt den arithmetischen, bzw. logischen Teil des Befehls aus. Dazu gehört auch die Berechnung von Sprungadressen und -ergebnissen sowie von Adressen für Speicherzugriffe.
`MEM`-Stufe	*(Memory Read/Write)* Zugriff auf den Datenspeicher.
`WB`-Stufe	*(Write Back)* Schreibt berechnete Ergebnisse in das Registerarray.

5.2.1 Auflösung von Konflikten

Um Datenkonflikte aufzulösen, wird der Pipeline Prozessor mit Forwarding implementiert. Dadurch können sämtliche RAW-Konflikte bis auf einen aufgelöst werden. Folgt direkt auf einen Speicherladebefehl ein Befehl, der das Datum, das aus dem Speicher gelesen werden soll, als Operand benötigt, so befindet sich der Speicherladebefehl erst in der `EX`-Stufe, wenn der Operand gelesen werden soll. Da der Wert des Operanden jedoch erst in der `MEM`-Stufe gelesen wird, muß der folgende Befehl um ein Takt verzögert werden, um den Konflikt aufzulösen. WAR- und WAW-Konflikte existieren im hier implementierten Modell nicht, da der Prozessor die Befehle „in-order“ ausführt.

Somit bleibt die Behandlung von Steuerkonflikten, die durch Sprungbefehle verursacht werden, zu klären. Unbedingte Sprünge werden von der `ID`-Stufe ausgewertet, welche die Sprungadresse direkt an die `IF`-Stufe weiterleitet. Hier entstehen keine Verzögerungen im Steuerfluß und somit auch keine Konflikte. Bei den bedingten Sprüngen ist dies anders, da diese die Sprungbedingung erst in der `EX`-Stufe auswerten. Der Steuerfluß wird hier um einen Takt verzögert. Somit ist der dem bedingten Sprung folgende Befehl in der Pipeline. Nach den Spezifikationen des Eintaktmodells sollte dieser bei durchgeführtem Sprung jedoch nicht ausgeführt werden. In dieser Implementierung wird auf die Abwärtskompatibilität zum Eintaktmodell verzichtet und der Befehl nach einem bedingten Sprungbefehl als delay slot behandelt. Der Befehl direkt nach einem bedingten Sprung wird also immer ausgeführt. Als Bedingung wird an diesen Befehl gestellt, daß er, wie schon beschrieben, das Ergebnis des bedingten Sprungbefehls, sowie den Steuerfluß nicht beeinflusst.

5.3 Implementierung

5.3.1 Schnittstelle

Die Schnittstelle des Pipeline-Modells weicht von dem Eintaktmodell ab. Optimierungen, wie sie im Eintaktmodell vorgenommen wurden sind im Pipeline-Modell nicht mehr möglich. Die Indizes der Register, die eingelesen werden sollen, müssen vom Prozessor gesetzt werden, da das Verbinden der Bits des Befehlsregisters mit dem Registerarray aus folgenden Gründen falsch ist: zum einen muß das Setzen der Indizes verzögert werden,

zum anderen ist es möglich, daß die Indizes durch einen Pipelinestall wiederholt gesetzt werden müssen. Es ist also das Timing, welches das Setzen der Indizes vom Prozessor erzwingt. Es ist jedoch nicht nur die Schnittstelle vom Prozessor mit der Umwelt, die hier geklärt werden muß, sondern auch die Schnittstellen zwischen den einzelnen Pipelinestufen. Diese werden zusammen mit dem Verhalten vorgestellt.

5.3.2 Verhalten

Um auf das Verhalten einzugehen, müssen die Schnittstellen der Pipeline Stufen geklärt werden. Um die Erstellung der Schnittstellen zu erleichtern wird zur besseren Veranschaulichung die oben beschriebene Beziehung der Pipeline Stufen des Prozessors sowohl untereinander als auch mit der Umwelt graphisch dargestellt. Aus der Abbildung 5.2 und den Spezifikationen, lassen sich für die einzelnen Stufen die Schnittstellen herleiten, die in den Grafiken 5.3, 5.5, 5.12, 5.15 und 5.17 beschrieben werden. Daten, die an die jeweils behandelte Stufe weitergeleitet werden, sind ebenfalls in der Grafik eingetragen. Die Stufen, die für die jeweilige Schnittstelle keine Werte zur Verfügung stellen oder von ihnen lesen, sind ausgegraut. Die Implementierung der jeweiligen Stufe folgt direkt im Anschluß ihrer Grafik.

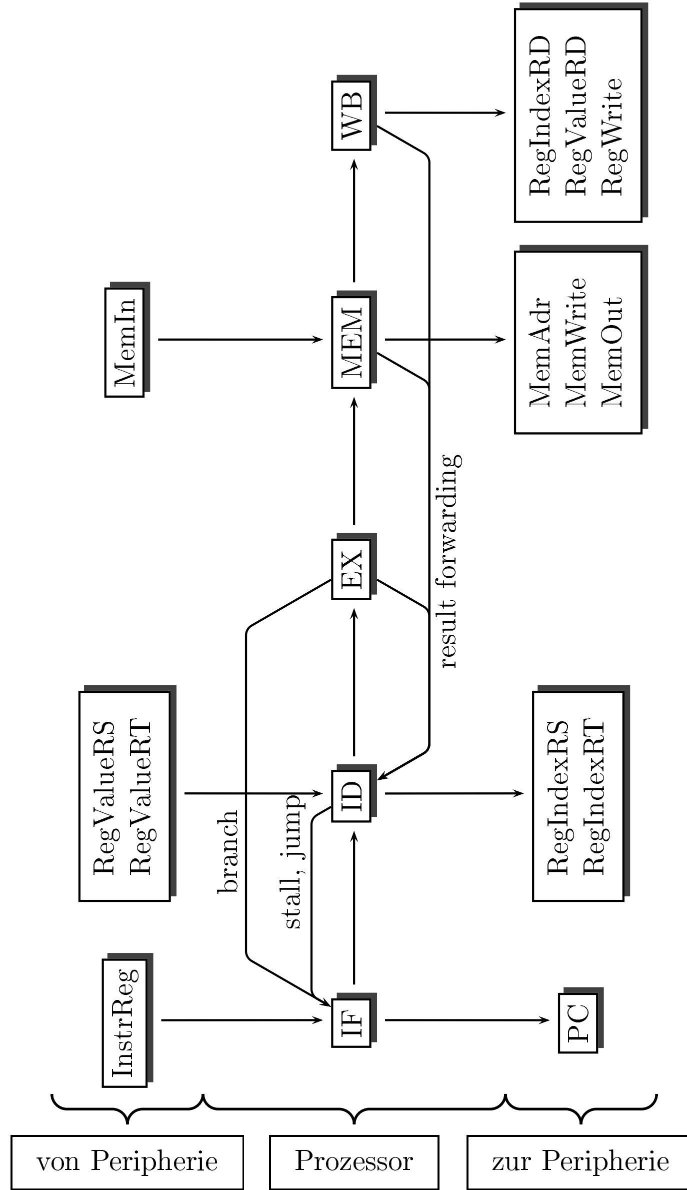


Abbildung 5.2: Beziehung der Pipeline-Stufen untereinander und mit der Umwelt

IF-Stufe

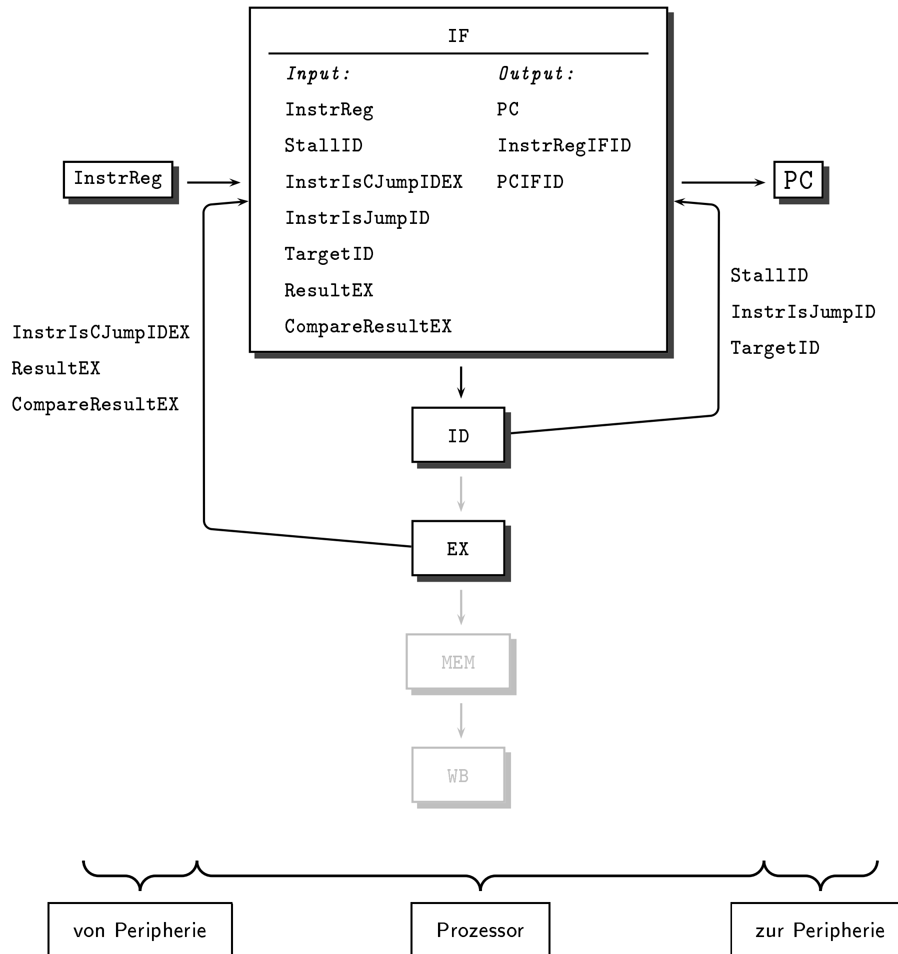


Abbildung 5.3: Schnittstelle der IF-Stufe

Bevor die IF-Stufe einen Befehl liest (siehe Abbildung 5.4), muß überprüft werden, ob ein Stall vorliegt. Ist dies der Fall, werden alle Ausgangswerte beibehalten. Das am Befehlsspeicher anliegende Befehlswort darf nur im Fall „nicht stall" an die ID-Stufe weitergeleitet werden, da hier das Befehlswort nicht zuerst in die IF-Stufe gelesen und dann weitergeleitet, sondern vom Befehlsspeicher direkt an die Eingänge der ID-Stufe angelegt wird. Liegt kein Stall vor, muß die korrekte Befehlsadresse bestimmt werden. Hier gibt es drei Fälle:

- Es liegt ein bedingter Sprung vor. Ist dies der Fall, so liegt die Befehlsadresse in dem von der EX-Stufe berechneten Ergebnis.
- Es liegt ein unbedingter Sprung vor. Die Befehlsadresse liefert die ID-Stufe, die die

Adresse aus dem Befehlswort extrahiert.

- Keiner der anderen Fälle. Die Befehlsadresse wird inkrementiert.

```
if not StallID then
  if InstrClassIDEX=INSTR_CLASS_CJUMP & CompareResultEX then
    PC := ResultEX
  else if InstrIsJumpID then
    PC := TargetID
  else
    PC := PCIFID + 1
  end end
  next(InstrRegIFID) := InstrReg
end
next(PCIFID) := PC
```

Abbildung 5.4: Implementierung der `IF`-Stufe

Zu Beachten ist, daß ein unbedingter Sprung unabhängig von einem anliegendem Stall durchgeführt werden muß. Daher müsste sich der entsprechende ‚`if InstrIsJumpID`'-Block außerhalb des ‚`if not StallID`'-Blocks befinden. Da ein unbedingter Sprung kein Stall hervorrufen kann, ist, sobald die Bedingung ‚`InstrIsJumpID`'erfüllt ist, auch die Bedingung ‚`not StallID`' erfüllt. Deshalb kann der ‚`if InstrIsJumpID`'-Block in den ‚`if not StallID`'-Block hineingezogen werden. Abgesehen davon, stellen die ersten beiden Fälle ein Problem dar, da beide zur selben Zeit auftreten können. Daher wird durch die Spezifikationen festgelegt, daß beide Fälle nie gleichzeitig auftreten dürfen. Im obigen Teil wurde festgelegt, daß nach einem bedingtem Sprungbefehl kein Kontrollflußbefehl folgen darf, also auch kein unbedingter Sprung. Beide Bedingungen können aber nur dann erfüllt sein, wenn ein bedingter Sprung gefolgt von einem unbedingtem Sprung vom Prozessor bearbeitet werden sollen. Allerdings wird dadurch nicht verhindert, daß diese Fälle auftreten können.

Im folgenden wird das Verhalten des Prozessors beschrieben, falls sich in einem delay slot ein Kontrollflußbefehl befindet. Auch wenn das Verhalten möglicherweise zu Optimierungen von Programmen verleiten mag, so ist es keine Einladung dazu. Bei dieser Art von Implementierung des Prozessors ist es ohne Probleme möglich, direkt nach einem bedingten Sprung einen unbedingten Sprungbefehl einzufügen. In diesem Fall wäre die Funktion des delay slots außer Kraft, was heißen soll, daß der Befehl nicht auf jeden Fall ausgeführt wird, sondern, wie man es vom sequentiellen Eintaktmodell erwarten würde, nur wenn der bedingte Sprung nicht ausgeführt wird. Bei zwei oder mehreren bedingten Sprüngen hintereinander lässt sich das Verhalten des Prozessors nicht mehr so einfach beschreiben. Bei disjunkten Bedingungen, wie z.B. bei `switch(var)...case` Anweisungen in C++, könnte man die bedingten Sprungbefehle direkt aneinanderketten, da man weiß, daß maximal einer dieser Sprünge ausgeführt wird. Bei nicht disjunkten Sprungbefehlen wird bei erfüllter Sprungbedingung auf jeden Fall der erste Befehl, der sich an der Sprungadresse befindet ausgeführt. Ob noch die folgenden Befehle ausgeführt werden,

hängt jeweils vom folgenden Sprung ab. Dieses Verhalten wirkt unkontrolliert und dürfte kaum verwendbar sein. Bei Sprungsequenzen mit nicht disjunkten Sprungbedingungen sollten also unbedingt `NOPs` eingefügt werden. Es sei nochmals darauf hingewiesen, daß dies das Verhalten des Prozessors ist, falls sich in einem delay slot ein Steuerflußbefehl befindet, was nach den Spezifikationen nicht erlaubt ist. Ein Grund, warum das obige Verhalten nicht in die Spezifikationen aufgenommen wird, ist der, daß man sich dadurch die Weiterentwicklung des Prozessors, z.B. Weiterentwicklung zu einem superskalarem Prozessor, möglicherweise erheblich erschwert. Die „einfachere“ Spezifikation macht es den Konstrukteuren von Compilern zudem leichter, Programmcode zu erzeugen. Ein weiterer positiver Nebeneffekt ist selbstverständlich die Zeitersparnis bei der Entwicklung des Prozessors, da dieses Verhalten nicht verifiziert werden muß.

ID-Stufe

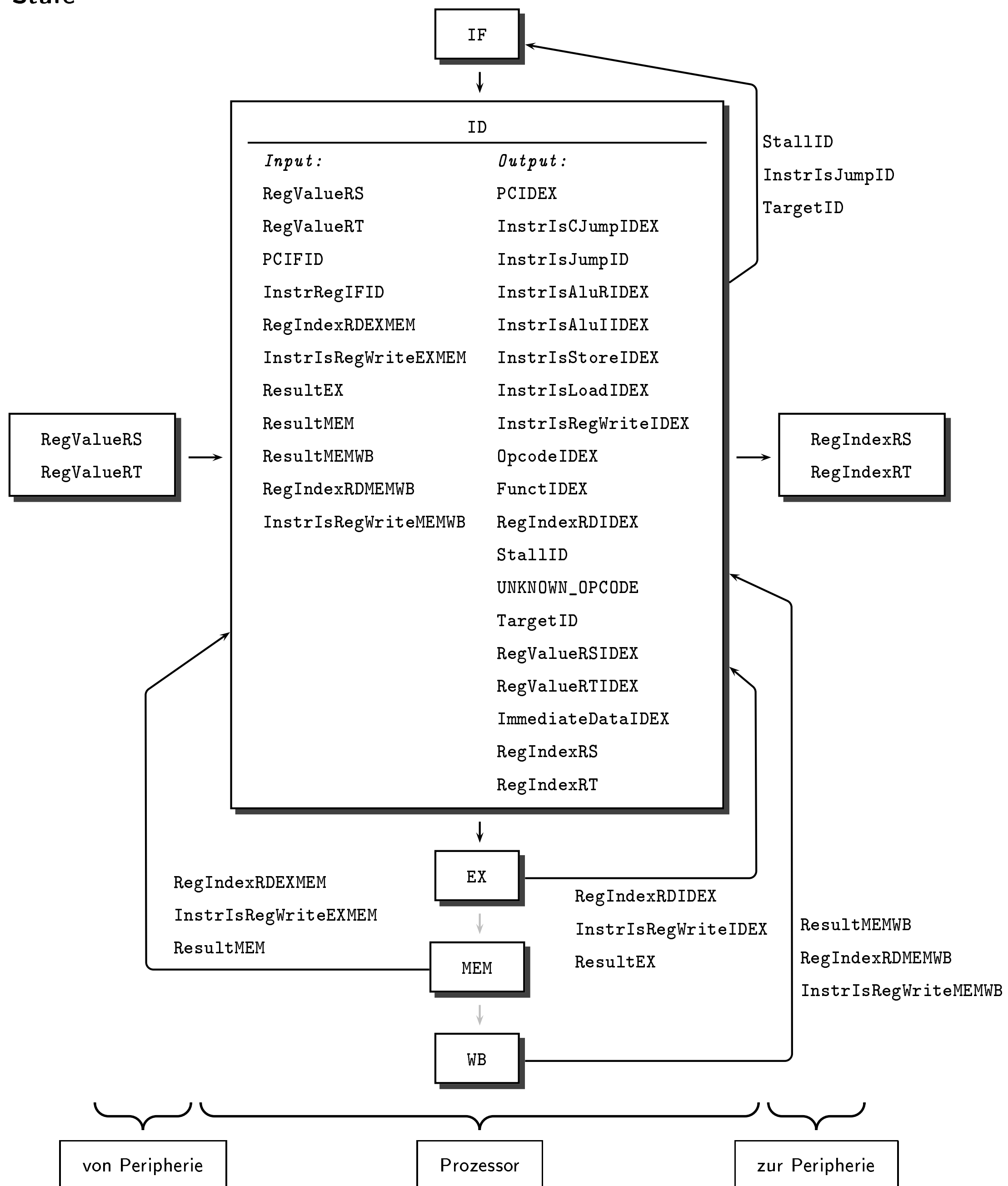


Abbildung 5.5: Schnittstelle der ID-Stufe

In der ID-Stufe werden als erstes die Werte, die später mehrmals gebraucht werden per `let` Anweisung zugeordnet (Abbildung 5.6). Dazu gehören der Opcode und der Funktionscode des Befehls, sowie Auswertungen der Befehlsklasse (Alu Register, Alu Imme-

```
opcode := {InstrRegIFID: 31..26}
funct := {InstrRegIFID: 5..0}
regIndexRS := {InstrRegIFID: 25..21}
regIndexRT := {InstrRegIFID: 20..16}
instrIsAluR := ((opcode=#B000000) &
               ((funct=INSTR_ADD) | (funct=INSTR_AND) |
                (funct=INSTR_OR) | (funct=INSTR_SLL) |
                (funct=INSTR_SLT) | (funct=INSTR_SRA) |
                (funct=INSTR_SRL) | (funct=INSTR_XOR)))
instrIsAluI := (opcode=INSTR_ADDI)
instrIsCJump := (opcode=INSTR_BEQ) | (opcode=INSTR_BNE)
instrIsJump := (opcode=INSTR_J)
instrIsLoad := (opcode=INSTR_LW)
instrIsStore := (opcode=INSTR_SW)
stallCondition := ((~UNKNOWN_OPCODE) &
                  (InstrClassIDEX=INSTR_CLASS_LOAD) &
                  (((regIndexRS=RegIndexRDIDEX) &
                    (~instrIsJump)) |
                   ((regIndexRT=RegIndexRDIDEX) &
                    (instrIsAluR | instrIsCJump | instrIsStore))))
notStallCondition := (not stallCondition)
regIndexRD := (if instrIsAluR then {InstrRegIFID: 15..11} else {InstrRegIFID: 20..16})
instrIsRegWrite := ((~(regIndexRD=#B00000)) &
                   (instrIsAluR | instrIsAluI |instrIsLoad) &
                   notStallCondition) in
```

Abbildung 5.6: Implementierung der ID-Stufe - interne Auswertung des Befehls

```
UNKNOWN\_OPCODE := not (instrIsAluRInstr |
                        instrIsAluIInstr |
                        instrIsCJump |
                        instrIsJump |
                        instrIsLoad |
                        instrIsStore);
```

Abbildung 5.7: Implementierung der ID-Stufe - Setzen der Ausgänge

diate, bedingter oder unbedingter Sprung, Lade-/Speicherbefehl). Desweiteren wird hier auch die stall Bedingung ausgewertet, die, wie schon beschrieben, dann vorliegt, wenn mindestens eines der Register, das geladen werden soll, dem Zielregister des momentan in der EX-Stufe befindlichen Befehls entspricht und dieser Befehl das aktualisierte Datum in der EX-Stufe noch nicht zur Verfügung stellen kann - was lediglich bei einem Ladebefehl der Fall ist. Das Zielregister des von der ID-Stufe behandelten Befehls hängt vom Befehlstyp ab und wird dementsprechend zugewiesen.

Diese Werte gehören lokal zur ID-Stufe und dienen in erster Linie dazu, Bedingungen und Variablen die im Folgenden mehrmals verwendet werden, auszuwerten. Es ist offensichtlich, daß nicht für jede Abfrage einer Auswertung eine eigene Schaltung implementiert

```
RegIndexRS := regIndexRS
RegIndexRT := regIndexRT
InstrIsJumpID := instrIsJump
TargetID := lastn(numBitsPerInteger, InstrRegIFID)
StallID := stallCondition
next(InstrIsCmpEqualIDEX) := (opcode=INSTR_BEQ)
next(PCIDEX) := PCIFID
next(ImmediateDataIDEX) := lastn(numBitsPerInteger, InstrRegIFID);
next(RegIndexRDIDEX) := regIndexRD;
next(InstrIsRegWriteIDEX) := instrIsRegWrite;
```

Abbildung 5.8: Implementierung der ID-Stufe - Setzen der Ausgänge

```
if stallCondition then
  next(InstrClassIDEX) := INSTR_CLASS_NONE
else if instrIsCJump then
  next(InstrClassIDEX) := INSTR_CLASS_CJUMP
else if instrIsRegWrite then
  if instrIsAluR then
    next(InstrClassIDEX) := INSTR_CLASS_ALUR
  else if instrIsAluI then
    next(InstrClassIDEX) := INSTR_CLASS_ALUI
  else if instrIsLoad then
    next(InstrClassIDEX) := INSTR_CLASS_LOAD
  else
    next(InstrClassIDEX) := INSTR_CLASS_NONE
  end end end
else if instrIsStore then
  next(InstrClassIDEX) := INSTR_CLASS_STORE
else
  next(InstrClassIDEX) := INSTR_CLASS_NONE
end end end end;
```

Abbildung 5.9: Implementierung der ID-Stufe - Bestimmung der Befehlsklasse

wird, sondern eine Schaltung, von der jeweils das fertig ausgewertete Signal abgegriffen wird. Es folgt das nach außen sichtbare Verhalten. Als erstes wird ausgewertet, ob ein unbekannter Befehl vorliegt (Abbildung 5.7). Hier könnte der Prozessor so erweitert werden, daß bei einem positiven Ergebnis eine Behandlung durch eine Routine gestartet wird. In Abbildung 5.8 erfolgt das Setzen von Variablen, deren Werte zum größten Teil ausgewertet und in internen Variablen gespeichert wurden und somit lediglich zugewiesen werden müssen. Beim Setzen der Befehlsklasse (Abbildung 5.9) ist darauf zu achten, daß diese nicht nur vom anliegenden Befehl abhängt, sondern auch ob ein Stall anliegt und das zu schreibende Register, falls es geschrieben werden soll, nicht das Register 0 ist. Eine weitere Aufgabe der ID-Stufe ist die Bestimmung des Befehlscode für die ALU (Abbildung 5.10). Dieser ist allein vom Befehl abhängig und daher ohne Probleme zuzuweisen. Das Forwarding der Registerwerte (Abbildung 5.8) gestaltet sich relativ einfach, da lediglich die Stufen in der richtigen Reihenfolge (neueste Ergebnisse zuerst) auf Regi-

```
if instrIsAluR & (funct=INSTR_AND) then next(AluOpIDEX) := ALU_AND end;
if instrIsAluR & (funct=INSTR_OR) then next(AluOpIDEX) := ALU_OR end;
if instrIsAluR & (funct=INSTR_SLT) then next(AluOpIDEX) := ALU_SLT end;
if instrIsAluR & (funct=INSTR_SLL) then next(AluOpIDEX) := ALU_SLL end;
if instrIsAluR & (funct=INSTR_SRA) then next(AluOpIDEX) := ALU_SRA end;
if instrIsAluR & (funct=INSTR_SRL) then next(AluOpIDEX) := ALU_SRL end;
if instrIsAluR & (funct=INSTR_XOR) then next(AluOpIDEX) := ALU_XOR end;
if (instrIsAluR & (funct=INSTR_ADD)) | (~instrIsAluR) then next(AluOpIDEX) := ALU_ADD end;
```

Abbildung 5.10: Implementierung der ID-Stufe - Bestimmung des Opcodes für die ALU

```
if (regIndexRS=RegIndexRDIDEX) & InstrIsRegWriteIDEX then
  next(RegValueRSIDEX) := ResultEX
else if (regIndexRS=RegIndexRDEXMEM) & InstrIsRegWriteEXMEM then
  next(RegValueRSIDEX) := ResultMEM
else if (regIndexRS=RegIndexRDMEMWB) & InstrIsRegWriteMEMWB then
  next(RegValueRSIDEX) := ResultMEMWB
else
  next(RegValueRSIDEX) := RegValueRS
end end end
```

Abbildung 5.11: Implementierung der ID-Stufe - Forwarding - Behandlung von RAW-Konflikten

sterschreibzugriff und passenden Registerindex überprüft werden müssen. Hat keine der Stufen den benötigten Parameter, wird der Wert für das Register `rs` bzw. `rt` aus dem Registerarray gelesen. In Abbildung 5.11 ist das Forwarding für Register `rs` gegeben. Das Forwarding für Register `rt` verläuft analog dazu.

EX-Stufe

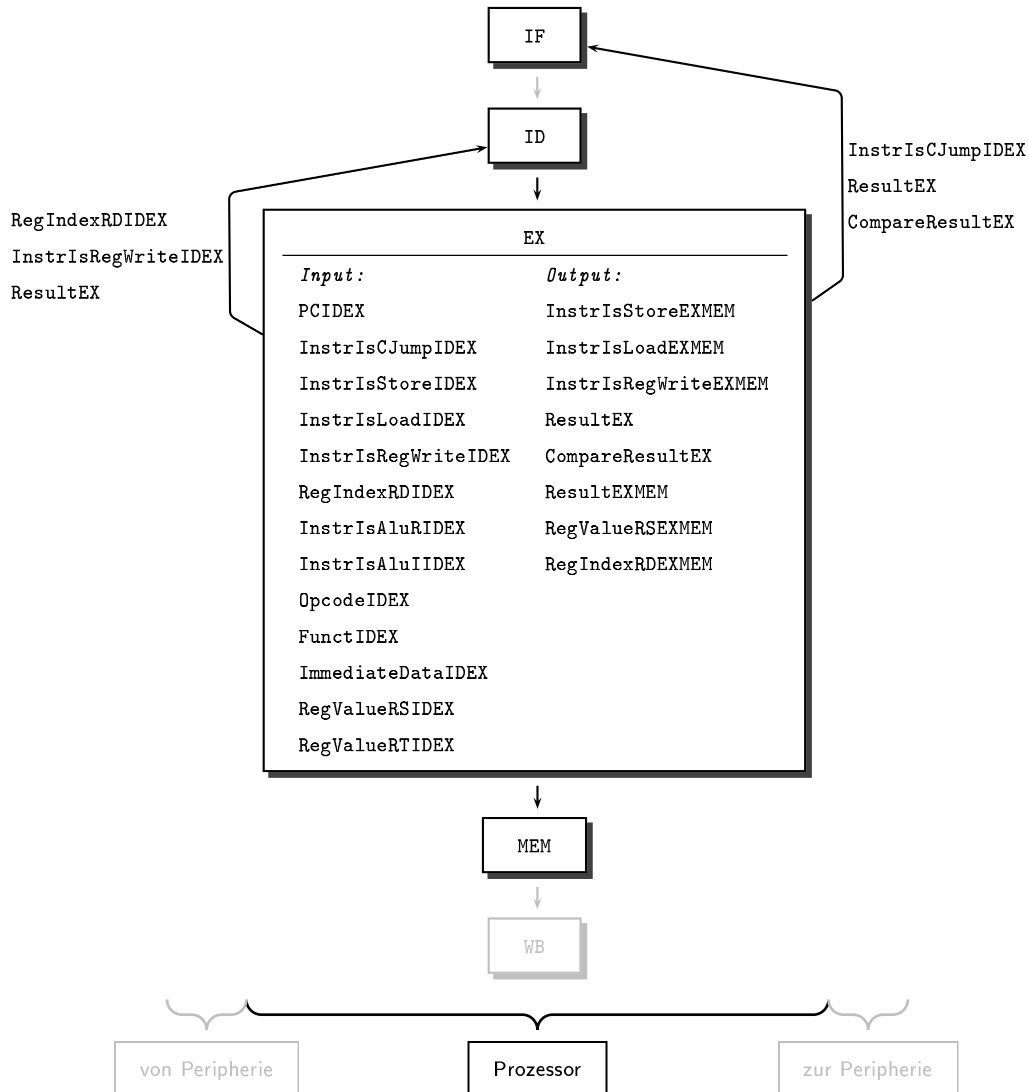


Abbildung 5.12: Schnittstelle der EX-Stufe

Die EX-Stufe des Pipeline-Prozessors dient allein zur Berechnung von arithmetischen und logischen Operationen, sowie von Adressen für bedingte Sprünge, als auch zur Auswertung von Sprungbedingungen. Um die Zahl der Addierer klein zu halten, wird das ALU Modul, welches schon im Eintaktmodell verwendet wurde (Abbildung 4.5), in das Pipelinemodell übernommen. Zur vollständigen Einbindung müssen nur noch die Eingänge zugewiesen werden (Abbildung 5.13).

Zur Vervollständigung der EX-Stufe fehlt noch das Setzen der restlichen Ausgangswerte (Abbildung 5.14). Dazu gehört das Ergebnis der Auswertung der Sprungbedingung, sowie die Zuordnung von Werten der Ausgänge zur MEM-Stufe. Letzteres ist recht einfach, da alle benötigten Werte berechnet wurden und lediglich zugewiesen werden müssen. Die Auswertung des Vergleichs basiert auf der Gleichheit der Eingaberegister und dem Status, ob auf Gleichheit oder Ungleichheit überprüft werden muß.

```
let ALUIn1:= ((InstrClassIDEX=INSTR_CLASS_CJUMP ) ? PCIDEX : RegValueRSIDEX) in
let ALUIn2:= ((InstrClassIDEX=INSTR_CLASS_ALUR ) ? RegValueRTIDEX : ImmediateDataIDEX) in

ALU: run ALUModule(ALUIn1, ALUIn2, AluOp)(ResultEX)
```

Abbildung 5.13: Implementierung der EX-Stufe - Einbindung des ALU Moduls

```
CompareResultEX := (InstrIsCmpEqualIDEX <-> (RegValueRSIDEX=RegValueRTIDEX));
next(RegValueRSEXMEM) := RegValueRSIDEX;
next(ResultEXMEM) := ResultEX;
next(InstrClassEXMEM) := InstrClassIDEX;
next(InstrIsRegWriteEXMEM) := InstrIsRegWriteIDEX;
next(RegIndexRDEXMEM) := RegIndexRDIDEX;
```

Abbildung 5.14: Implementierung der EX-Stufe - Auswertung des Sprungergebnisses und Setzen der Ausgangsvariablen

`MEM`-Stufe

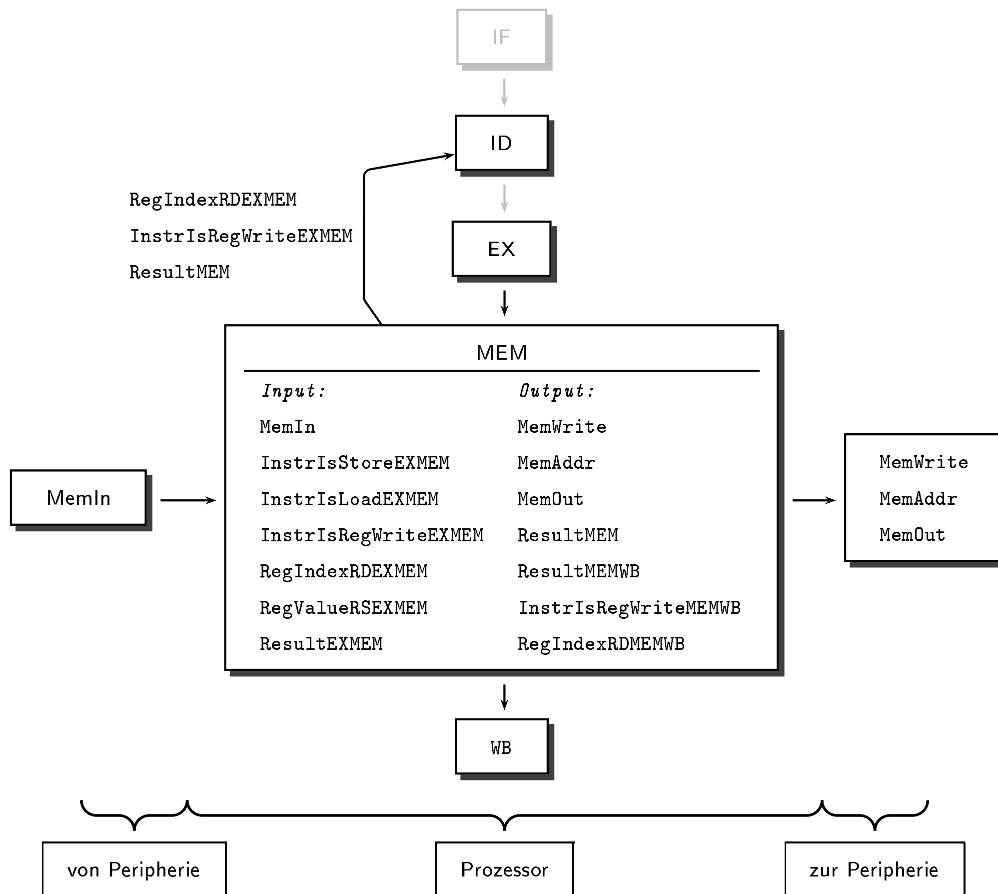


Abbildung 5.15: Schnittstelle der `MEM`-Stufe

Die Implementierung der `MEM`-Stufe ist kurz und einfach (Abbildung 5.16). Da die Werte der zu schreibenden Variablen schon von den vorherigen Stufen berechnet bzw. ausgewertet wurden, müssen diese lediglich weitergeleitet werden. Insgesamt ist in der `MEM`-Stufe lediglich eine Auswertung vorzunehmen, welche das weiterzugebende Ergebnis betrifft. In Abhängigkeit davon, ob ein Ladebefehl anliegt, ist das Ergebnis dem Datenspeicher zu entnehmen oder es wird das, von der `EX`-Stufe berechnete Ergebnis übernommen.

Zuletzt werden die letzten Werte, die von der `WB`-Stufe benötigt werden weitergeleitet.

```
MemWrite := (InstrClassEXMEM=INSTR_CLASS_STORE);
MemAddr := ResultEXMEM;
MemOut := RegValueRSEXMEM;
ResultMEM := ((InstrClassEXMEM=INSTR_CLASS_LOAD) ? MemIn : ResultEXMEM);
next(ResultMEMWB) := ResultMEM;
next(InstrIsRegWriteMEMWB) := InstrIsRegWriteEXMEM;
next(RegIndexRDMEMWB) := RegIndexRDEXMEM;
```

Abbildung 5.16: Implementierung der MEM-Stufe

WB-Stufe

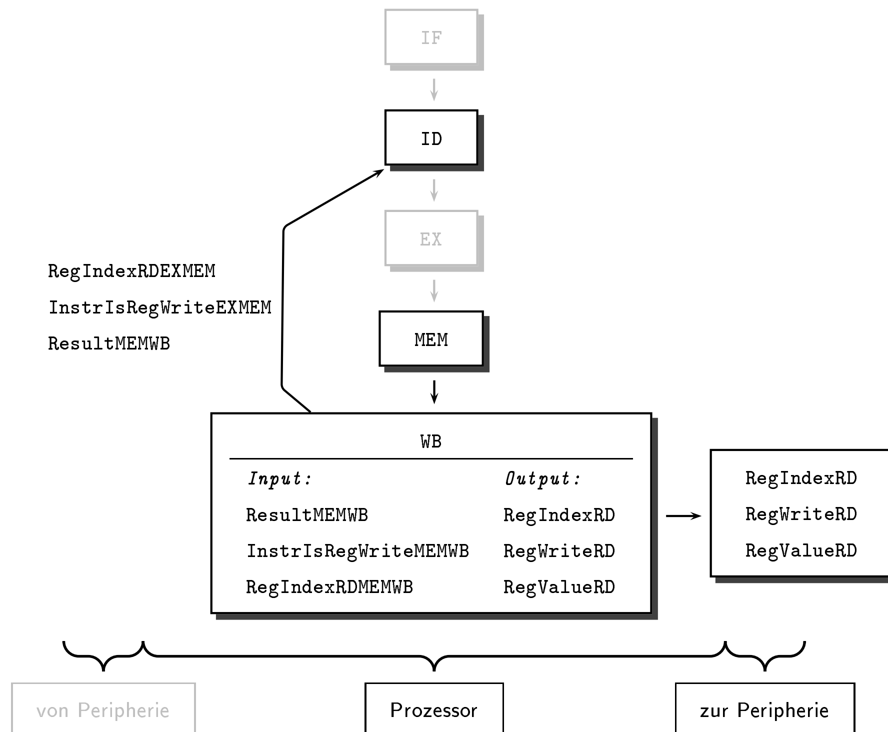


Abbildung 5.17: Schnittstelle der WB-Stufe

Die WB-Stufe beinhaltet die einfachste Implementierung aller Stufen. Sie besteht lediglich darin, die berechneten Werte an die Umgebung, genauer gesagt an das Registerarray, weiterzuleiten.

```
RegValueRD := ResultMEMWB;
RegWriteRD := InstrIsRegWriteMEMWB;
RegIndexRD := RegIndexRDMEMWB
```

Abbildung 5.18: Implementierung der WB-Stufe

5.4 Verifikation

Die Komplexität des Pipeline-Modells ist im Vergleich zum Eintaktmodell höher. Dazu trägt nicht nur die umfangreiche Logik bei. Wie im Eintaktmodell schon erwähnt wurde, wird die Geschwindigkeit der computergestützten Verifizierung stark durch die Größe des Zustandsraumes beeinflußt. Im Pipeline-Modell sind viele zusätzliche Variablen erforderlich, um die Zustände der bearbeiteten Befehle zwischen den Arbeitsschritten, also zwischen den Pipelinestufen, zu speichern. Dies führt zu einem wesentlich größeren Zustandsraum und somit zu einer Verifikation, die entweder sehr langsam oder aufgrund des Speicherbedarfs der Model Checker und der gegebenen Speicherbeschränkungen gar nicht mehr möglich ist. Um das Pipeline-Modell verifizieren zu können kann nicht der gesamte Prozessor auf einmal betrachtet werden.

In [BM96] wenden Börger und Mazzanti das ‚Divide and Conquer' Verfahren zur Verifizierung ihres Pipelineprozessors an. Allerdings wird hier das Modell nicht wirklich verkleinert, sondern vereinfacht. Sie beginnen bei der Verifizierung des Prozessors damit, zu zeigen, daß der Prozessor als sequentielles Modell korrekt arbeitet. Nach und nach wird das Modell um Funktionalität erweitert, wie z.B. Auflösung von Steuer- und Datenkonflikten. Diese Herangehensweise ist hier nicht sinnvoll, da prinzipiell nur die Logik vereinfacht wird, der Zustandsraum jedoch weitestgehend gleich groß bleibt. Das Hauptziel ist im folgenden nicht die Vereinfachung des Modells, sondern, um die Verifikation mit Model Checkern zu ermöglichen, die Verkleinerung des Zustandsraum.

5.4.1 Modularisierung

Da jede Stufe des Pipelineprozessors in einem eigenen Modul implementiert wurde, ist es sinnvoll, die Stufen zunächst so weit es geht, einzeln zu verifizieren. „So weit es geht" bedeutet, die Spezifikationen zu verifizieren, die nicht im direkten Zusammenhang mit dem Verhalten der anderen Stufen stehen.

Spezifikationen der einzelnen Stufen

1. `IF`-Stufe
   Das Verhalten der `IF`-Stufe ist recht einfach, daher sind nicht viele Spezifikationen notwendig, um das Verhalten zu beschreiben.

   - Liegt ein *Stall* vor, so darf die Befehlsadresse nicht verändert werden und im nächsten Taktzyklus muß der momentan anliegende Befehl erneut anliegen.
   - Liegt kein *Stall* vor und in der `EX`-Stufe befindet sich ein bedingter Sprungbefehl, dessen Auswertung positiv ist (der Sprung soll ausgeführt werden), so muß der PC der von der `EX`-Stufe berechneten Adresse entsprechen.
   - Sind die vorherigen Bedingungen nicht erfüllt und in der `ID`-Stufe befindet sich ein unbedingter Sprungbefehl, so ist die Befehlsadresse die entsprechende Sprungadresse.

- Trifft keine der obigen Bedingungen zu, so wird der Befehlszähler lediglich um eins erhöht.

2. ID-Stufe
Die ID-Stufe benötigt deutlich mehr Spezifikationen als die anderen Stufen, da sie die Stufe ist, die die Befehle entschlüsselt. Abgesehen von der EX-Stufe benötigt sie dazu auch mehr Logik als die anderen Stufen. Während die EX-Stufe zum größten Teil aus arithmetischer Logik besteht, die sich sehr gut abstrahieren lässt, benötigt die ID-Stufe die umfangreiche Logik zur Dekodierung der Befehle, welche sich nicht einfach zusammenfassen lässt.

Um die Korrektheit der Dekodierungsstufe zu zeigen, werden die Spezifikationen aus zwei Sichten erstellt. Zum einen wird für jeden bekannten Befehl gezeigt, daß dieser korrekt dekodiert wird und alle notwendigen Signale und Parameter gesetzt werden, zum anderen wird gezeigt, daß alle weiterzuleitenden Signale und Parameter korrekt gesetzt werden, so daß sichergestellt ist, daß keine unnötigen oder sogar falschen Operationen durchgeführt werden, wie zum Beispiel Speicherzugriffe. Letzteres deckt den Fall ab, daß ein unbekannter Befehl anliegt.

- Das Verhalten wird anhand des anliegenden Befehl spezifiziert, unter der Voraussetzung, daß kein *Stall* anliegt. Der *Stall* wird später separat betrachtet.
- Ein bekannter Befehl kann nicht das Signal *UNKNOWN_OPCODE* auslösen.
- Jeder Befehl, außer dem unbedingten Sprungbefehl, darf das Signal *InstrIsJumpID* nicht auslösen.
- Jeder Befehl, der ein Register schreiben will, darf dies nur unter der Bedingung, daß das Zielregister nicht das Register 0 ist. Andernfalls wird er durch einen NOP ersetzt.
- Wird der Befehl durch einen NOP-Befehl ersetzt, darf er seine Klassenzugehörigkeit nicht weitergeben (sie wird durch INSTR_CLASS_NONE ersetzt).
- Wird der Befehl nicht ersetzt, muß er seine Klassenzugehörigkeit weitergeben und festlegen, welche Operation die ALU in der EX-Stufe ausführen muß. Das bedeutet, bei ALU Befehlen die zum Befehl gehörende Operation, bei bedingten Sprung- und bei Speicher-/Ladebefehlen die Operation ‚Addition' um die Sprung-, bzw. Speicheradresse zu berechnen.
- Bei bedingten Sprungbefehlen muß festgelegt werden, um welche Art von Vergleich es sich handelt. Zur Wahl stehen „ist gleich" und „ist ungleich".
- Bedingte Sprungbefehle führen keine schreibenden Zugriffe auf Register aus. Gleiches gilt auch für den Datenspeicherzugriff, welcher jedoch nicht direkt aus der ID-Stufe kontrolliert werden kann, da der Schreibzugriff auf den Datenspeicher durch die Befehlsklasse bestimmt ist. Der Zugriff auf den Datenspeicher kann also nur indirekt durch die Befehlsklasse bestimmt werden.
- Befehle die auf den Datenspeicher zugreifen, führen keine schreibende Zugriffe auf das Registerarray aus.

- Unbedingte Sprünge müssen die Sprungadresse aus dem Befehl extrahieren.
- Da ein unbedingter Sprung in der `ID`-Stufe dekodiert und in dem Moment durchgeführt wird, in dem sich der Befehl in der `ID`-Stufe befindet und folglich in den folgenden Stufen kein Einfluß mehr auf das System hat, bzw. haben darf, muß er ab der `EX`-Stufe wie ein `NOP` behandelt werden. Daher muß der unbedingte Sprungbefehl als NOP-Befehl an die `EX`-Stufe weitergegeben werden.

Im nächsten Abschnitt wird die Dekodierung aus der Sicht der Ausgangsvariablen behandelt. Dadurch wird das korrekte Setzen von Signalen und Zuweisen von Parametern bei nicht bekannten Befehlen sichergestellt. Es folgen die Punkte auf denen die betreffenden Spezifikationen aufbauen.

- Generelle Vorgehensweise ist, das Signale genau dann anliegen, wenn bestimmte Befehle anliegen und eventuell noch gewisse Nebenbedingungen gegeben sein müssen.
- Ein unbekannter Befehl liegt genau dann vor, wenn kein bekannter Befehl anliegt; d.h. wenn der Opcode bzw. Funktionscode nicht zugeordnet werden kann.
- Ein Stall ist genau dann gegeben wenn ein Register gelesen werden soll, dessen Berechnung noch nicht abgeschlossen wurde, bzw. bis zum Ende des Taktzyklus nicht berechnet werden kann. Um die Spezifikation aufzubauen, müssen folgende drei Bedingungen erfüllt sein:
  - Es liegt ein bekannter Befehl an.
  - Der anliegende Befehl benötigt mindestens einen Registerwert als Parameter.
  - In den nachfolgenden Stufen befindet sich ein Befehl, welcher eines der benötigten Register verändern wird, dessen Wert jedoch nicht bis zum Taktende bestimmt werden kann.

  Die erste Bedingung ist erfüllt, sobald das Signal `UNKNOWN_OPCODE` nicht gesetzt ist. Die zweite Bedingung lässt sich umformulieren zu ‚Der anliegende Befehl benötigt nicht keinen Registerwert'. Dies trifft lediglich auf den unbedingten Sprung zu. Zum dritten Punkt gilt allgemein folgendes: es müssen alle, der `ID`-Stufe folgenden Stufen betrachtet werden, bis zu der letzten Stufe, die einen Registerwert berechnen kann. Bei dem hier vorgestelltem Pipeline-Modell müssen daher zwei Stufen, die `EX`-Stufe und die `MEM`-Stufe, betrachtet werden. Da die `EX`-Stufe direkt auf die `ID`-Stufe folgt, kann kein Befehl, der den Registerwert in der `EX`-Stufe berechnet, einen Stall verursachen. Es bleibt die `MEM`-Stufe. Es gibt genau einen Befehl, der dort einen Registerwert laden kann - der Ladebefehl `LW`. Daher gibt es genau einen Befehl, der in der `EX`-Stufe anliegen muß, um einen Stall zu verursachen.

Die Spezifikation, die eine korrekte Auslösung eines Stall sicherstellt lautet also

- es muß ein bekannter Befehl sein und
- in der `EX`-Stufe muß ein Ladebefehl anliegen während
- in der `ID`-Stufe kein Sprungbefehl anliegt (also liegt ein Befehl an, der mindestens ein Register als Operand benötigt) und das zu lesende Register (RS) gleich dem zu schreibenden Register (RD) des Ladebefehls in der `EX`-Stufe entspricht, oder
- kein Sprungbefehl, kein Ladebefehl und kein Befehl vom Typ ALU Immediate liegt in der `ID`-Stufe an (was wiederum bedeutet, daß ein zweites Register gelesen werden muß) und der Index des zu lesenden zweiten Register (RT) stimmt mit dem des zu schreibenden Registers des Ladebefehls in der `EX`-Stufe überein.

Um nun das korrekte Verhalten bei einem anliegendem Stall zu zeigen, muß zur Bedingung des anliegenden Stalls das Verhalten beschrieben werden.

- Liegt ein Stall an, so darf der zu dekodierende Befehl nicht an die nächste Stufe, d.h. `EX`-Stufe, weitergegeben werden und muß durch einen `NOP` ersetzt werden. Dementsprechend gilt auch, daß kein schreibender Zugriff auf das Registerarray erfolgen darf. Weiterhin ist durch den Aufbau der Pipeline gegeben, daß ein Stall maximal einen Takt anliegen kann und somit auf einen Stall ein ‚kein Stall' folgen muß. Dies aus der `ID`-Stufe zu Überprüfen ist zwar in diesem Fall möglich, gehört jedoch generell zur globalen Ansicht der Pipeline und wird erst dort verifiziert. Es ist ebenfalls sicherzustellen, daß bei einem Stall der Befehl in der `ID`-Stufe nicht verändert wird. Dies muß jedoch innerhalb der `IF`-Stufe verifiziert werden.
- Das Signal zur Durchführung eines unbedingten Sprungs wird unverzögert an die `IF`-Stufe weitergeleitet und muß genau dann gesetzt sein, wenn der Sprungbefehl anliegt.
- Liegt ein Sprungsignal an, muß auch die Sprungadresse korrekt aus dem Befehl extrahiert werden.
- Selbstverständlich muß auch eine korrekte Auswertung der Befehlsklasse gegeben sein, damit die folgenden Stufen den Befehl korrekt abarbeiten. Außer der Zuordnung einer Menge von Befehlen zu einer Klasse müssen Nebenbedingungen beachtet werden, wie z.B. Anliegen eines Stalls oder Ersetzung des Befehls durch einen `NOP`, falls Schreibzugriff auf Register 0 stattfinden soll.
  - Ein unbedingter Sprungbefehl wird innerhalb der ersten beiden Stufen behandelt und wird als `NOP` weitergegeben. Dies wurde schon in einer der obigen Spezifikationen angegeben.
  - Die Befehlsklasse ‚bedingter Sprungbefehl' wird genau dann gewählt, wenn ein bedingter Sprung und kein Stall anliegt.

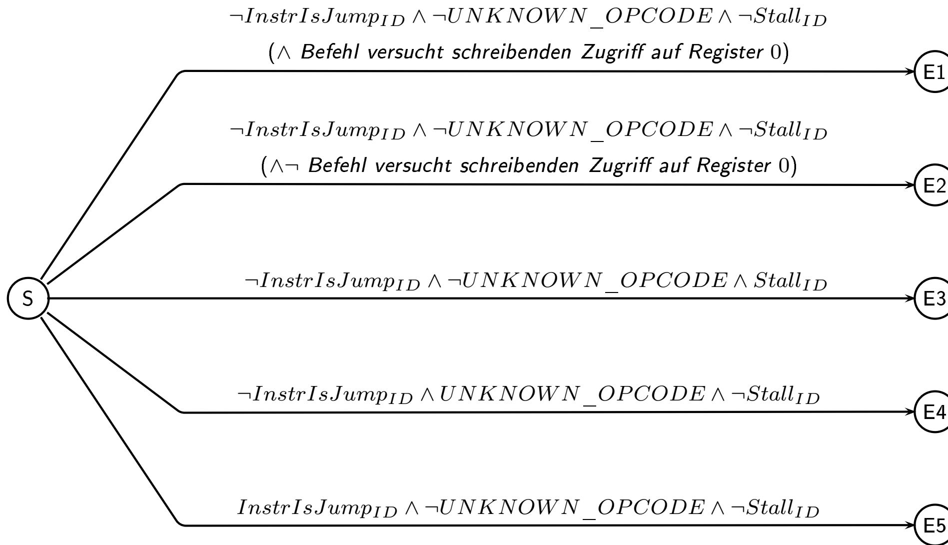
  - Um einen Befehl der ALU Register Klasse handelt es sich genau dann, wenn der Befehlscode einem ALU Register Befehl entspricht, kein Stall anliegt und das Zielregister nicht 0 ist.
  - Analog dazu ist die Spezifikation für die ALU Immediate Klasse.
  - Recht einfach lautet die Bedingung der Speicherklasse: kein Stall und Anliegen des `SW` Befehlscode.
  - Analog dazu die Ladeklasse, jedoch mit der zusätzlichen Bedingung, daß das Register 0 nicht geschrieben werden darf.

- Einige Signale, die von der `ID`-Stufe gesetzt werden und nach außen gehen, sind in ihrer Bedeutung disjunkt (Abbildung 5.19). Das heißt, das Signal für unbedingte Sprünge, einen unbekannten Befehl und einen Stall, sowie ein bestimmter Befehlscode können nur in bestimmten Kombinationen vorkommen.
  - Liegt ein unbedingter Sprung vor, dürfen nicht das Signal ‚unbekannter Befehl' und nicht das Signal ‚Stall' gesetzt sein. Da die Bearbeitung der unbedingten Sprünge in der `ID`-Stufe endet, muß der Befehlscode auf `NOP` gesetzt werden.
  - Analog wird bei einem unbekannten Befehl sowie bei einem Stall verfahren.
  - Liegt kein unbedingter Sprung, jedoch ein bekannter Befehl an, welcher keinen Stall auslöst, muß der Befehl entweder einer Befehlsklasse angehören oder er hat keine Befehlsklasse, weil es sich um einen `NOP` oder ein durch einen `NOP` ersetzter Befehl handelt. Trifft der letztere Fall ein, darf dieser Befehl keinen Schreibzugriff auf das Registerarray bekommen.

- Dem Befehl wird ein Schreibzugriff auf das Registerarray genau dann erlaubt, wenn der Befehl bekannt ist, kein Stall vorliegt und der Befehl nicht versucht das Register 0 zu überschreiben.

- Es ist sicherzustellen, daß der Befehlszähler bei einem Sprungbefehl korrekt an die `EX`-Stufe weitergegeben wird, damit die korrekte Sprungadresse berechnet werden kann.

- Zu den letzten Spezifikationen der `ID`-Stufe gehört das Übernehmen der benötigten Registerwerte von der richtigen Quelle. Das heißt, daß die Befehle, welche sich in den der `ID`-Stufe folgenden Stufen befinden, darauf getestet werden müssen, ob sie das Register schreiben werden, welches der, in der `ID`-Stufe anliegende Befehl, benötigt. Ist dies der Fall, muß es aus der entsprechenden Stufe geladen werden. Die Stufen werden dabei so durchlaufen, daß die jüngsten Befehle zuerst überprüft werden. Das Forwarding geschieht jedoch nur unter der Voraussetzung, daß kein Stall anliegt, da andernfalls das Ergebnis von mindestens einem Parameter fehlt. Die Bedingungen für den Fall, daß ein Stall anliegt, wurde im obigen Teil abgehandelt. Enthält keine der Stufen den benötigten Wert, müssen die Indizes der zu lesenden Register an das Registerarray gelegt und ihre Werte daraus gelesen werden. Anstatt jedoch den



Knoten	Bedingung
E1	$InstrClass_{IDEX} \neq NONE$
E2	$InstrClass_{IDEX} = NONE \wedge \neg InstrIsRegWrite_{IDEX}$
E3,E4,E5	$InstrClass_{IDEX} = NONE$

Abbildung 5.19: Betrachtung der gültigen Kombination von Signalen

Wert jedes Eingaberegisters mit der, in Abhängigkeit vom Forwarding gewählten Quelle zu vergleichen, wird jeder potentiellen Quelle ein Index zugeordnet. In der Spezifikation wird nun, anstatt der Gleichheit der Werte des Eingabeparameter und der Quelle, der korrekte Quellenindex gefordert. Abbildung 5.20 zeigt den Ablauf des Forwarding für Register RS nochmals als Pseudoalgorithmus und Abbildung 5.21 die daraus resultierenden Spezifikationen in CTL. Analog dazu verläuft das Forwarding für Register RT.

3. `EX`-Stufe
   Die Verifikation der `EX`-Stufe beschränkt sich darauf, zu zeigen, daß in Abhängigkeit des ALU Opcodes das richtige Ergebnis berechnet wird und alle Werte, die von der `ID`-Stufe kommen und an die `MEM`-Stufe weitergeleitet werden müssen, weitergeleitet werden. Hierbei muß jedoch folgender Punkt beachtet werden: da die in der `EX`-Stufe verwendete ALU für alle Operationen dient, ist es notwendig die Eingänge für die ALU korrekt zu setzen. Daher ist nicht nur der ALU Opcode selbst entscheidend für das Ergebnis, sondern auch die Befehlsklasse, die die zu wählenden Parameter bestimmt. Aus dem Grund kann keine allgemeine Spezifikation für das berechnete Ergebnis erstellt werden. Stattdessen müssen die Befehle einzeln betrachtet und mehrere Spezifikationen aufgestellt werden. Abbildung 5.22 zeigt die Spezifikation für das korrekte Berechnen einer Addition von zwei Registern und die Spezifikation für die Berechnung einer Adresse für Lade- und Speicherbefehle.

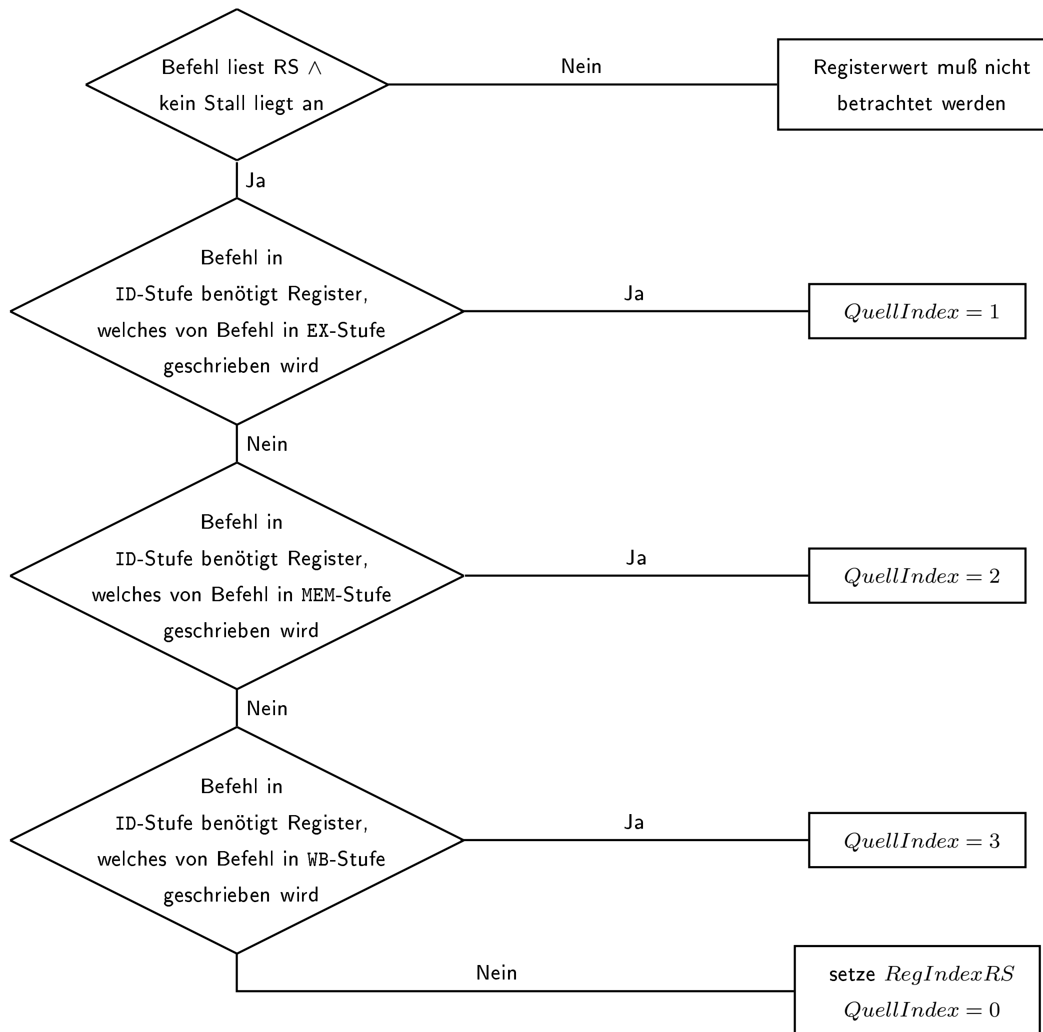


Abbildung 5.20: Pseudoalgorithmus zur Verifikation des Forwarding

Die Spezifikationen der anderen Befehle sind analog dazu aufgebaut.

4. `MEM`-Stufe

- Wie in den vorherigen Stufen, ist auch in dieser Stufe sicherzustellen, daß alle Parameter, die von der nächsten Stufe, hier die `WB`-Stufe, benötigt werden, korrekt weitergeleitet werden. Dabei handelt es sich zuersteinmal um das Signal `InstrIsRegWrite`.
- Da das entsprechende Hintergrundwissen über die Funktionsweise der `WB`-Stufe existiert, reicht es, das korrekte Weiterleiten des Index des zu schreibenden Registers unter der Bedingung, daß es zu schreiben ist, zu fordern. Der Wert

$$
\begin{aligned}
&\mathit{InstrReadsRegRS} := \neg \mathit{Stall}_{ID} \wedge \\
&\quad (\mathit{InstrReg}_{IFID}.\mathit{Opcode} \in \{\texttt{BEQ},\texttt{BNE},\texttt{ADDI},\texttt{SW},\texttt{LW}\} \vee \\
&\quad ((\mathit{InstrReg}_{IFID}.\mathit{Opcode} = 00000_2) \wedge \\
&\quad \mathit{InstrReg}_{IFID}.\mathit{Funct} \in \{\texttt{ADD},\texttt{AND},\texttt{OR},\texttt{SLL},\texttt{SLT},\texttt{SRA},\texttt{SRL},\texttt{XOR}\}))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{AG}\ ((\mathit{InstrReadsRegRS} \wedge \\
&\quad \mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{IDEX} \wedge \\
&\quad \mathit{InstrIsRegWrite}_{IDEX}) \rightarrow \\
&\quad (\texttt{AX}\ \mathit{RegSourceRS}_{IDEX} = 01_2))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{AG}\ ((\mathit{InstrReadsRegRS} \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{IDEX} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{IDEX}) \wedge \\
&\quad \mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{EXMEM} \wedge \\
&\quad \mathit{InstrIsRegWrite}_{EXMEM}) \rightarrow \\
&\quad (\texttt{AX}\ \mathit{RegSourceRS}_{IDEX} = 10_2))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{AG}\ ((\mathit{InstrReadsRegRS} \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{IDEX} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{IDEX}) \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{EXMEM} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{EXMEM}) \wedge \\
&\quad \mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{MEMWB} \wedge \\
&\quad \mathit{InstrIsRegWrite}_{MEMWB}) \rightarrow \\
&\quad (\texttt{AX}\ \mathit{RegSourceRS}_{IDEX} = 11_2))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{AG}\ ((\mathit{InstrReadsRegRS} \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{IDEX} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{IDEX}) \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{EXMEM} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{EXMEM}) \wedge \\
&\quad \neg(\mathit{InstrReg}_{IFID}.\mathit{RegRS} = \mathit{RegIndexRD}_{MEMWB} \wedge \\
&\quad\quad \mathit{InstrIsRegWrite}_{MEMWB})) \rightarrow \\
&\quad (\texttt{AX}\ \mathit{RegSourceRS}_{IDEX} = 00_2))
\end{aligned}
$$

Abbildung 5.21: Spezifikationen für korrektes Forwarding in CTL

specADD: AG $((InstrClass_{IDEX} = AluR \wedge$
$AluOp =$,ADD') $\rightarrow$
$Result_{EX} = RS_{IDEX} + RT_{IDEX}$

specLW_SW: AG $(((InstrClass_{IDEX} = Load \vee$
$InstrClass_{IDEX} = Store) \wedge$
$AluOp =$,ADD') $\rightarrow$
$(Result_{EX} = RS_{IDEX} + ImmediateData_{IDEX}))$

Abbildung 5.22: Beispiele für Spezifikationen der EX-Stufe in CTL

des zu schreibenden Registers und damit auch der Wert, der zwecks Forwarding an die ID-Stufe weiterzugeben ist, hängt von der Befehlsklasse ab. Daher wird die korrekt Zuweisung in getrennten Spezifikationen sichergestellt.

- Liegt ein Ladebefehl an, muß das Ergebnis der MEM-Stufe vom Speichereingang übernommen werden. Zudem muß die Speicheradresse korrekt gesetzt werden, die von der EX-Stufe berechnet wurde und es darf nicht das Signal zum Schreiben eines Speicherregisters gegeben werden.
- Bei einem Speicherbefehl muß ebenfalls die Speicheradresse korrekt gesetzt werden. Weiterhin muß der zu schreibende Wert an den Speicher gelegt und das Signal zum Schreiben gegeben werden. Letzte Bedingung wird dadurch impliziert, daß eine Spezifikation die Beziehung *Signal ,Speicher schreiben' genau dann, wenn Speicherbefehl anliegt* sicherstellt.
- Liegt in der MEM-Stufe weder Speicherbefehl noch Ladebefehl an, so wird das Ergebnis von der EX-Stufe an die ID-Stufe und die WB-Stufe weitergeleitet. Zudem darf kein Signal zum Schreiben an den Datenspeicher gesendet werden.

5. WB-Stufe
   Die Verifikation der WB-Stufe ist aufgrund ihres geringen funktionalen Aufwands einfach. Alle von der WB-Stufe benötigten Daten und Signale wurden von den vorhergehenden Stufen berechnet, bzw. ausgewertet und müssen lediglich korrekt angewendet werden. Aufgabe der WB-Stufe ist es, bei anliegendem Signal zum Schreiben des Registers, die notwendigen Daten an das Registerarray weiterzuleiten.

   - Ein Register ist genau dann zu schreiben, wenn das Signal `InstrIsRegWrite` an der WB-Stufe anliegt.
   - Ist ein Register zu schreiben, müssen der Index und der Wert des zu schreibenden Registers gesetzt werden.

Spezifikationen des gesamten Modells

Im vorherigen Teil wurden die Spezifikationen aufgestellt, die innerhalb der einzelnen Stufen verifiziert werden können. Dazu gehört u.a. der gesamte Datenpfad, welcher einen

großen Teil des Zustandsraum ausmacht. Im folgenden Abschnitt werden die Spezifikationen aufgestellt, welche die Korrektheit des Steuerflußes zeigen, jedoch nicht innerhalb einzelner Stufen verifiziert werden konnten.
Da das Ziel die Verifikation des Steuerflusses ist, besteht die Reduzierung des Pipeline-Modell im wesentlichen aus der Entfernung des Datenpfades und der dazugehörigen Logik, sofern sie keinen Einfluß auf die Steuersignale hat. Zusätzlich kann auf die Teile verzichtet werden, deren Verhalten schon verifiziert wurde. Ein Teil der Spezifikationen entspricht denen von Ramesh in [RB99]. Wie z.B. das Verhalten von Befehlen, falls kein Stall anliegt und das Verhalten der Befehle, falls sie einen Stall verursachen.

1. `IF`-Stufe
   Die `IF`-Stufe weist dem Befehlszeiger die aktuelle Adresse zu und liest das neue Befehlswort ein. Da es keine konkrete Implementierung des Befehlsspeichers gibt, ist für den Prozessor keine Abhängigkeit zwischen dem einzulesenden Befehlswort und der Befehlsadresse vorhanden. Die Befehlsadresse muß daher nicht notwendigerweise gesetzt werden und der Teil, der die Adresse bestimmt, kann eingespart werden. Auch das Weiterleiten der Befehlsadresse an die `ID`-Stufe ist nicht mehr notwendig, da diese am Ende nur dem Zweck dient, Sprungadressen zu berechnen, welche an die `IF`-Stufe zurückgeleitet werden. Der Teil der `IF`-Stufe der diese Adresse benötigt, wurde zuvor eingespart und somit kann der komplette Kreislauf des Befehlszeiger eingespart werden. Da der größte Teil gerade aus der Bestimmung der Befehlsadresse bestand, kann auch die Schnittstelle der `IF`-Stufe entsprechend verkleinert werden. Letztendlich besteht die `IF`-Stufe für die Verifikation lediglich aus der Zuweisung eines Befehlswortes unter der Bedingung, daß kein Stall anliegt. Dazu sollte noch gesagt werden, daß diese Optimierung nur deshalb zulässig ist, da das korrekte Setzen des Befehlszeigers schon in der `ID`-Stufe verifiziert wurde und somit für die Verifizierung des gesamten Modells nicht mehr notwendig ist.

2. `ID`-Stufe
   Soeben wurde der Befehlszeiger aus der `IF`-Stufe entfernt, inklusive dem Weg, den er durch die `ID`-Stufe und die `EX`-Stufe nimmt. Somit fallen in der Schnittstelle der `ID`-Stufe die entsprechenden Variablen, inklusive der Sprungadresse, weg. Als nächstes wird der Datenpfad entfernt. Dazu gehören die Werte, die vom Registerarray geliefert werden und die Ergebnisse der der `ID`-Stufe folgenden Stufen, welche für das Forwarding direkt zur ihr geleitet werden. Da die Werte nicht mehr vom Registerarray eingelesen werden, ist das Setzen der Indizes nicht mehr notwendig. Dadurch werden wiederum die Indizes selber überflüssig und können aus der Schnittstelle entfernt werden. Nicht nur die Eingabewerte, die zur Berechnung des Ergebnisses dienen, werden eingespart. Da das Ergebnis für die Verifikation des abstrahierten Modells nicht von Interesse ist, werden die Steuersignale entfernt, die die Berechnung in der `EX`-Stufe bestimmen. Dazu gehört der ALU Befehlscode, sowie das Steuersignal zum Vergleich (‚ist-gleich‘ oder ‚ist-ungleich‘). Da der Datenpfad für die Verifizierung ausgeschlossen wird, fällt auch der Teil weg, der das Forwarding behandelt, also den Paramtern den aktuellsten Wert zuordnet.

3. `EX`-Stufe
   Die `EX`-Stufe wird durch die Entfernung des Datenpfades fast arbeitslos. Sie dient lediglich noch als Verzögerungsglied der Signale, die an die `MEM`-Stufe weitergeleitet werden.

4. `MEM`-Stufe
   Wie die `EX`-Stufe, so kann auch in der `MEM`-Stufe durch Reduzierung auf ein großen Teil verzichtet werden. Die `MEM`-Stufe muß im reduzierten Modell lediglich das Setzen des Signals zum Schreiben in den Speicher und das Weiterleiten der von der `WB`-Stufe benötigten Variablen sichern.

5. `WB`-Stufe
   Die `WB`-Stufe wird, ähnlich wie die `MEM`-Stufe, darauf reduziert, den Status zum Schreiben eines Registers zu setzen. Der Wert des zu schreibenden Registers gehört zum Datenpfad und ist damit ausgeschlossen. Ebenso kann auf das Setzen des Registerindex verzichtet werden, da dieser Teil in der Verifikation der `WB`-Stufe behandelt wurde. Die korrekte Dekodierung wurde in der `ID`-Stufe und die korrekte Weiterleitung in den ihr folgenden Stufen verifiziert.

Nach der Abstrahierung des Pipeline-Modells sind die Spezifikationen hinzuzufügen. Prinzipiell wurde das gesamte Verhalten des Prozessors schon in den Modulen verifiziert. Die zusammengesetzten Module ermöglichen eine Betrachtung der Abarbeitung eines Befehls über seine gesamte Lebenszeit in der Pipeline des Prozessor. So kann man zu allen Zeitpunkten überprüfen, ob bestimmte Bedingungen erfüllt sind. Dazu gehören z.B. das korrekte Setzen von Signalen (Schreiben von Speicher- oder Registerinhalten). Desweiteren können interne Verhaltensweisen verifiziert werden, wie z.B. das Setzen einer Variablen zu einem anderen Zeitpunkt in einer anderen Stufe ein definiertes Signal auslöst. Zu den Spezifikationen gehören:

- Das Signal zum Schreiben eines Registers muß alle der `ID`-Stufe folgenden Stufen durchlaufen (Abbildung 5.23).

$$\texttt{AG}\ (InstrIsRegWrite_{IDEX} \leftrightarrow ((\texttt{AX}\ InstrIsRegWrite_{EXMEM}) \wedge (\texttt{AX AX}\ InstrIsRegWrite_{MEMWB}) \wedge (\texttt{AX AX}\ RegWriteRD)))$$

Abbildung 5.23: Verfolgen des Signals zum Schreiben des Registers

- Das Schreiben eines Registers wird genau dann ausgelöst, wenn der dekodierte Befehl einer bestimmten Befehlsklasse angehört. Dabei muß das Timing des Auslösens beachtet werden, welches sich durch einen Stall verschieben kann (Abbildung 5.24).

- Ebenso wird das Signal zum Schreiben in den Datenspeicher genau dann ausgelöst, wenn der Befehl der Speicherklasse angehört.

```
AG (RegWriteRD ↔
    ((PSX PSX PSX (¬Stall_ID ∧
                (PSX (¬Stall_ID ∧
                    (((((InstrReg.Opcode = 00000_2) ∧ (
                        (InstrReg.FunctCode = ,ADD‘ ∨
                              ⋮
                         InstrReg.FunctCode = ,SLT‘))) ∨
                       (InstrReg = ,ADDI‘) ∨
                       (InstrReg = ,LW‘)) ∧
                      ¬(InstrReg.RegRD = 00000_2)))
                    ))))
          ∨
     (PSX PSX PSX PSX (Stall_ID ∧ (PSX (¬Stall_ID ∧ (
                      an IF-Stufe anliegender Befehl schreibt Register ∧
                      Befehl schreibt nicht Register 0
                    )))))))
```

Abbildung 5.24: Zusammenhang zwischen dem Signal zum Schreiben des Registers und dem Befehlscode

- Ein Stall hat folgende Verhaltensregeln:
  - Aufgrund der Prozessorarchitektur kann ein Befehl nur einen Stall auslösen. Nach diesem Stall muß das Ergebnis bereit liegen und die Pipeline kann nicht erneut gestallt werden. Daher muß nach einem Stall ein Nicht-Stall folgen.
  - Zwei Befehle können hintereinander einen Stall auslösen. Es muß also ein Stall nach einem Nicht-Stall existieren. Damit wird auch die Existenz eines Stall generell gezeigt.
  - Bei einem Stall darf das Befehlswort in der ID-Stufe nicht überschrieben werden.
  - Liegt ein Stall an, kann im gleichen Takt kein bedingter Sprungbefehl in der EX-Stufe in Bearbeitung sein, da der Stall nur durch einen Ladebefehl in der EX-Stufe ausgelöst werden kann.

- Da der Datenfluß zur Verifikation nicht mehr betrachtet wird, kann für die einzelnen Befehle nicht überprüft werden, ob der Prozessor das richtige Ergebnis liefert. Es ist jedoch möglich für einen anliegenden Befehl zu überprüfen, ob der Steuerfluß sich entsprechend den Vorschriften verhält. Dazu muß beachtet werden, daß durch einen Stall eine zeitliche Verzögerung hervorgerufen werden kann. Unter diesen Umständen müssen folgende Fälle berücksichtigt werden:
  - Wird der vom Befehlsspeicher ankommende Befehl wegen einem Stall verworfen, so wird im nächsten Takt der Befehlszeiger erneut an den Befehlspeicher

$$
\begin{aligned}
&\texttt{AG}\ (Stall_{ID} \leftrightarrow (\neg UNKNOWN_OPCODE \wedge \\
&\qquad InstrClass_{IDEX} = INSTR_CLASS_LOAD \wedge \\
&\qquad ((InstrReg_{IFID}.Opcode \neq \text{,}\texttt{J}\text{`} \wedge \\
&\qquad\quad InstrReg_{IFID}.RegRD = RegIndexRD_{IDEX}) \vee \\
&\qquad ((InstrReg_{IFID}.Opcode \neq \text{,}\texttt{J}\text{`} \vee \\
&\qquad\quad InstrReg_{IFID}.Opcode = \text{,}\texttt{LW}\text{`} \vee \\
&\qquad\quad InstrReg_{IFID}.Opcode = \text{,}\texttt{ADDI}\text{`})) \wedge \\
&\qquad InstrReg_{IFID}.RegRD = RegIndexRD_{IDEX}))) \\
&\texttt{EF}\ (Stall_{ID} \wedge (\texttt{AX AX}\ Stall_{ID})) \\
&\texttt{AG}\ (Stall_{ID} \rightarrow (\texttt{AX}\ \neg Stall_{ID})) \\
&\texttt{AG}\ (Stall_{ID} \rightarrow (\texttt{AX}\ InstrReg_{IFID} = lastInstrReg_{IFID}))
\end{aligned}
$$

Abbildung 5.25: Verhalten eines Stalls in CTL

angelegt, so daß auch der Befehl erneut anliegt. Dieser Fall wurde schon behandelt, als die Spezifikation für das korrekte Verhalten bei einem anliegendem Stall aufgestellt wurde.

– Liegt kein Stall an, so wird der anliegende Befehl im nächsten Takt von der ID-Stufe behandelt. Somit kann der Befehl nur noch durch einen eigens verursachten Stall verzögert werden.

– Wird durch den Befehl kein Stall verursacht, durchläuft der Befehl pro Takt eine Stufe. Es müssen alle Signale beachtet werden, die nach außen gehen, sowie der unbedingte Sprungstatus und die dekodierte Befehlsklasse. Letzteres muß nicht genau bestimmt werden. Es reicht, sicherzustellen, daß kein bedingter Sprung von der EX-Stufe behandelt wird, um auszuschließen, daß Arithmetik-, Logik-, Speicher- oder Ladebefehle einen bedingten Sprung auslösen können (Abbildung 5.26).

$$
\begin{aligned}
&\texttt{AG}\ (((InstrReg.Opcode = 00000_2) \wedge \\
&\quad (InstrReg.Funct = \text{,}\texttt{XOR}\text{`}) \wedge \\
&\quad \neg Stall_{ID} \wedge (\texttt{AX}\ \neg Stall_{ID})) \rightarrow \\
&\quad ((\texttt{AX}\ \neg UNKNOWN_OPCODE) \wedge \\
&\quad\ (\texttt{AX}\ \neg InstrIsJump_{ID}) \wedge \\
&\quad\ (\texttt{AX AX}\ InstrClass_{IDEX} \neq INSTR_CLASS_CJUMP) \wedge \\
&\quad\ (\texttt{AX AX AX}\ \neg MemWrite) \wedge \\
&\quad\ ((InstrReg.RegRD = 00000_2) \leftrightarrow \\
&\quad\quad (\texttt{AX AX AX AX}\ \neg RegWriteRD))))
\end{aligned}
$$

Abbildung 5.26: Verhalten des XOR-Befehls, falls dieser keinen Stall verursacht

– Bei einem anliegenden Stall wird der Befehl exakt um einen Takt verzögert, da, aufgrund der Prozessorarchitektur, nach diesem Takt das Ergebnis für den fehlenden Parameter anliegen muß. Letzteres wurde ebenfalls im obigen Teil

als Spezifikation angegeben. Durch die Verzögerung werden die im letzten Punkt erwähnten Signale um einen Takt verzögert. Abgesehen davon ist noch sicherzustellen, daß der, hervorgerufen durch einen Stall, eingefügte NOP keine Schreibsignale sendet (Abbildung 5.27).

$$
\begin{array}{l}
\texttt{AG} \; (((InstrReg.Opcode = 00000_2) \; \wedge \\
\quad (InstrReg.Funct = \text{,}\texttt{XOR}\text{`}) \; \wedge \\
\quad \neg Stall_{ID} \; \wedge \; (\texttt{AX} \; Stall_{ID})) \; \rightarrow \\
\quad ((\texttt{AX} \; \neg UNKNOWN_OPCODE) \; \wedge \\
\quad (\texttt{AX} \; \neg InstrIsJump_{ID}) \; \wedge \\
\quad (\texttt{AX} \; \texttt{AX} \; \neg UNKNOWN_OPCODE) \; \wedge \\
\quad (\texttt{AX} \; \texttt{AX} \; \neg InstrIsJump_{ID}) \; \wedge \\
\quad (\texttt{AX} \; \texttt{AX} \; \texttt{AX} \; InstrClass_{IDEX} \neq INSTR_CLASS_CJUMP) \; \wedge \\
\quad (\texttt{AX} \; \texttt{AX} \; \texttt{AX} \; \texttt{AX} \; \neg MemWrite) \; \wedge \\
\quad ((InstrReg.RegRD = 00000_2) \; \leftrightarrow \\
\quad\quad (\texttt{AX} \; \texttt{AX} \; \texttt{AX} \; \texttt{AX} \; \texttt{AX} \; \neg RegWriteRD))))
\end{array}
$$

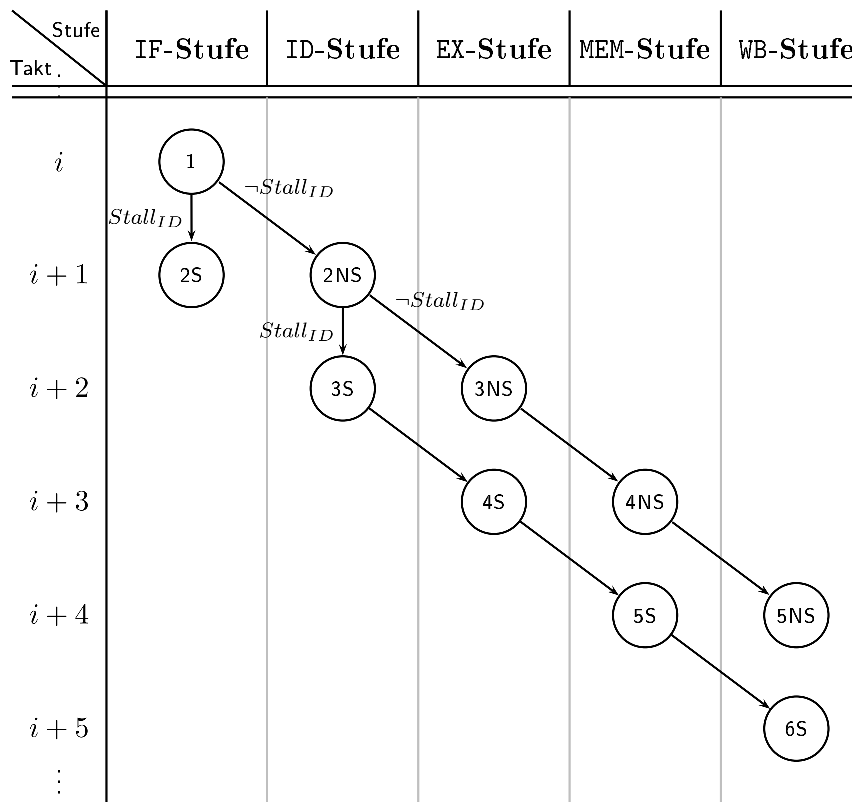
Abbildung 5.27: Verhalten des XOR-Befehls, falls dieser einen Stall verursacht

– *Durch eine graphischen Darstellung des zeitlichen Ablaufs eines Befehl, können solche CTL Beschreibungen leicht abgeleitet werden. Für den obigen Befehl ist der zeitliche Ablauf in Abbildung 5.29 dargestellt. Bis auf die Anfangsbedingung, die in Knoten* 1 *angegeben ist, ist dieser Graph für alle ALU Register Befehle gültig. Aber auch für die anderen Befehle sind die Graphen leicht zu erstellen.*
– Ein unbedingter Sprung kann keinen Stall verursachen (Abbildung 5.28).

$$
\texttt{AG} \; (\neg((InstrReg.Opcode = \text{,}\texttt{J}\text{`}) \; \wedge \; \neg Stall_{ID} \; \wedge \; (\texttt{AX} \; Stall_{ID})))
$$

Abbildung 5.28: Zusammenhang zwischen dem unbedingten Sprungbefehl und dem Stallsignal

- Nachdem unter Betrachtung der Befehle das korrekte Setzen der Signale sichergestellt wurde, wird dies nun in umgekehrter Sicht getan. Diese Vorgehensweise wurde schon in der ID-Stufe angewendet. Das heißt, daß bestimmte Signale genau dann gelten müssen, wenn zuvor ein bestimmter Befehl am Prozessor angelegt wurde. Dazu gehört der Status, ob ein unbekannter Befehl anliegt, die Signale zum Schreiben, sowie die Befehlsklassen. Die Abbildungen 5.30 und 5.31 zeigen die Beschreibungen des Verhaltens der Befehle ADDI und LW in CTL. Dieses ist bis auf die unterschiedlichen Werte der ersten beiden Bedingungen identisch. Diese beiden Spezifikationen sind für die späteren Auswertung der Verifikation, speziell für die zeitliche Dauer der Verifikation, interessant.
- Außerdem wird die Existenz von Schreibzugriffen auf das Registerarray und den



Damit der Befehl korrekt abgearbeitet wird, müssen für die Knoten die Bedingungen erfüllt sein, die in der folgenden Tabelle aufgeführt werden.

Knoten	Bedingung
1	$InstrReg.Opcode =$,`XOR`‘
2S	*erneutes Einlesen des Befehls; wurde in der IF Stufe verifiziert*
2NS	$\neg UNKNOWN_OPCODE \wedge Jump_{ID}$
3S	$\neg UNKNOWN_OPCODE \wedge Jump_{ID}$
3NS	$InstrClass_{IDEX} \neq$,`CJump`‘
4S	$InstrClass_{IDEX} \neq$,`CJump`‘
4NS	$\neg MemWrite$
5S	$\neg MemWrite$
5NS	$RegWrite \leftrightarrow$(`PSX PSX PSX` $InstrReg.RegRD \neq 00000_2$)
6S	$RegWrite \leftrightarrow$(`PSX PSX PSX` $InstrReg.RegRD \neq 00000_2$)

Abbildung 5.29: Beispiel für die graphische Interpretation der Spezifikation aus Abbildung 5.26 und 5.27

AG ($InstrClass_{IDEX} = INSTR_CLASS_ALUI \leftrightarrow$
(((PSX PSX $InstrReg.Opcode =$,ADDI') $\wedge$
(PSX PSX $\neg Stall_{ID}$) $\wedge$ (PSX $\neg Stall_{ID}$) $\wedge$ (PSX PSX $InstrReg.RegRD \neq 00000_2$))
$\vee$ ((PSX PSX PSX $InstrReg.Opcode =$,ADDI') $\wedge$
(PSX PSX PSX $\neg Stall_{ID}$) $\wedge$ (PSX PSX $Stall_{ID}$) $\wedge$ (PSX PSX PSX $InstrReg.RegRD \neq 00000_2$))))

Abbildung 5.30: Korrektes Dekodieren des ADDI Befehls

AG ($InstrClass_{IDEX} = INSTR_CLASS_LOAD \leftrightarrow$
(((PSX PSX $InstrReg.Opcode =$,LW') $\wedge$
(PSX PSX $\neg Stall_{ID}$) $\wedge$ (PSX $\neg Stall_{ID}$) $\wedge$ (PSX PSX $InstrReg.RegRD \neq 00000_2$))
$\vee$ ((PSX PSX PSX $InstrReg.Opcode =$,LW') $\wedge$
(PSX PSX PSX $\neg Stall_{ID}$) $\wedge$ (PSX PSX $Stall_{ID}$) $\wedge$ (PSX PSX PSX $InstrReg.RegRD \neq 00000_2$))))

Abbildung 5.31: Korrektes Dekodieren des LW Befehls

Datenspeicher gezeigt. Es ist offensichtlich, daß der Prozessor in solche „Situationen“ kommen muß.

5.4.2 Auswertung

ALU

Da im Pipeline-Modell das ALU Modul aus dem Eintaktmodell verwendet wurde, können die Ergebnisse der Verifikation des ALU Moduls in Tabelle 4.3.2 nachgelesen werden.

Pipelinestufen

	IF-Stufe			ID-Stufe		
Bitbreite	Beryl	SMV	#Vars	Beryl	SMV	#Vars
2 Bit	0.2s	0.1s	36	76s	2266s	96
3 Bit	0.4s	0.1s	42	263s	1290s	106
4 Bit	0.4s	0.2s	48	143s	2261s	116
6 Bit	0.6s	0.6s	60	2989s	373s	136
8 Bit	1.2s	1.1s	72	343s	386s	156

Tabelle 5.1: Ergebnisse des Benchmark der `IF`-Stufe und `ID`-Stufe

	EX-Stufe			MEM-Stufe		
Bitbreite	Beryl	SMV	#Vars	Beryl	SMV	#Vars
2 Bit	0.3s	0.1s	39	0.1s	0.03s	32
3 Bit	0.6s	0.7s	46	0.1s	0.05s	39
4 Bit	1.5s	1.0s	53	0.2s	0.07s	46
6 Bit	142s	1.6s	67	0.3s	0.14s	60
8 Bit	12s	8.1s	81	0.4s	0.35s	74

Tabelle 5.2: Ergebnisse des Benchmark der `EX`-Stufe und `MEM`-Stufe

	WB-Stufe		
Bitbreite	Beryl	SMV	#Vars
2 Bit	0.05s	0.02s	18
3 Bit	0.05s	0.02s	20
4 Bit	0.06s	0.02s	22
6 Bit	0.07s	0.03s	26
8 Bit	0.09s	0.03s	30

Tabelle 5.3: Ergebnisse des Benchmark der `WB`-Stufe

Der erste Teil der Verifikation behandelte die einzelnen Pipelinestufen, daher werden die Ergebnisse der Verifikation der einzelnen Stufen als erstes betrachtet. In 5.1, 5.2, 5.3 und 5.32 ist deutlich zu sehen, daß, abgesehen von der `ID`-Stufe, alle Stufen zügig verifiziert wurden. Die `IF`-Stufe liegt in etwa gleich auf mit der `EX`-Stufe. Obwohl das Befehlsregister zur Verifikation von 32 Bit auf 4 Bit verkleinert wurde, wird durch die Hilfsvariablen,

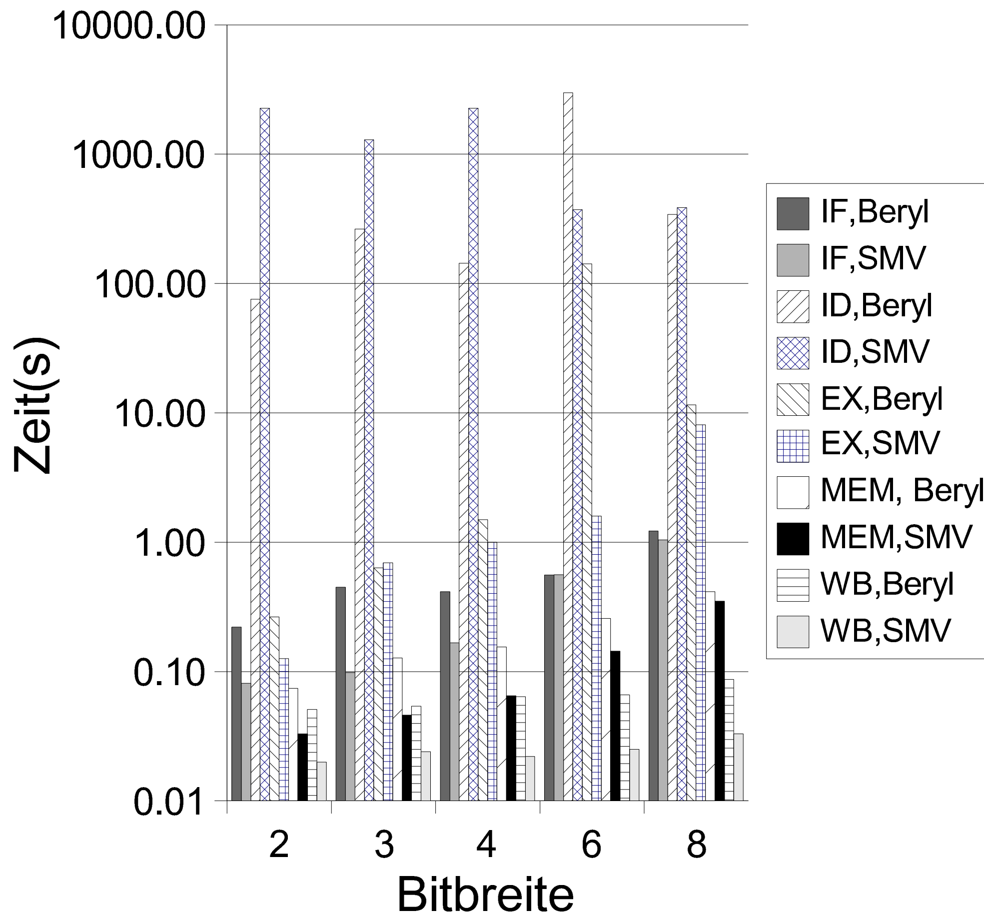


Abbildung 5.32: Ergebnisse der Benchmarks der Pipelinestufen

die zur Verifikation eingeführt wurden, der Zustandsraum vergrößert, wodurch wiederum die Verifikation verlangsamt wird. Insgesamt läuft die Verifikation der `IF`-Stufe bis 8 Bit schnell ab (unter 2 Sekunden), allerdings ist bei steigender Bitzahl mit einem starken Anstieg der Dauer der Verifikation zu rechnen. Hier lässt das Modul nicht viel Raum zur weiteren Abstrahierung. Eine Möglichkeit wäre die Verifizierung durch einzelnes Prüfen der Spezifikationen, wobei alle nicht relevanten Variablen ausgeklammert werden. Eine längere Dauer der Verifikation der `ID`-Stufe war zu erwarten, da diese schon aufgrund des größeren Variablenumfangs einen größeren Zustandsraum hat. Hinzu kommt, daß die `ID`-Stufe im Vergleich zu den anderen Stufen gegen mehr Spezifikationen geprüft werden muß.
Die `EX`-Stufe dagegen wird recht schnell verifiziert und kann zumindest bis 8 Bit in etwa mit der `IF`-Stufe mithalten. Aus den Tabellen 5.1 und 5.2 kann entnommen werden,

daß beide Stufen in etwa die gleiche Anzahl an Variablen haben. Allerdings verläuft die Verifikation der EX-Stufe langsamer als die des ALU Moduls. Dies ist verständlich, da die EX-Stufe das ALU Modul und zusätzliche Variablen und Logik beinhaltet. Ob die EX-Stufe sich bis 64 Bit mit vergleichbaren Aufwand verifizieren lässt, ist daher zweifelhaft. Hier wäre, wie in Abschnitt 4.3.2 beschrieben, eine weitere Abstrahierung möglich, indem das ALU Modul zur Verifikation der EX-Stufe abstrahiert, bzw. ganz ausgeschlossen wird. Die Spezifikationen müssten dann entsprechend angepasst werden, indem z.B. nicht mehr Ergebnisse, sondern die korrekte Ansteuerung des ALU Moduls vorrausgesetzt wird.

Die MEM-Stufe und die WB-Stufe sind in kurzer Zeit verifiziert worden (unter 1 Sekunde). Die MEM-Stufe hat in etwa soviele Variablen wie die IF-Stufe, wird jedoch um Faktor zwei bis drei schneller verifiziert. Ein Grund wird vermutlich die geringere Anzahl an Spezifikationen sein, wodurch jedoch nicht der hohe Geschwindigkeitsfaktor erklärt. Im Wesentlichen wird die Variablenordnung und das Transitionssystem für die schnellere Verifikation verantwortlich sein. Dies wird durch die Ausgabe von Beryl bestätigt, da die Kripke-Struktur der MEM-Stufe um den Faktor zwei schneller erstellt werden konnte. Die schnelle Verifikation der WB-Stufe ist durch deren einfachen Aufbau, der geringen Anzahl an Spezifikationen und den wesentlich kleineren Zustandsraum zu erklären.

Pipelineprozessor

Da zur Abstrahierung des Pipeline-Modells der komplette Datenpfad aus dem Prozessor entfernt wurde, ist das abstrahierte Modell unabhängig von der Bitbreite des Datenpfades. Die Verifikation des abstrahierten Pipelineprozessors in einem Durchlauf war nicht möglich. Während der SMV an der Speicherbegrenzung scheiterte, gelang es Beryl nicht die Kripke-Struktur aufzubauen. Daher wurden die Spezifikationen einzeln geprüft. Einige der Spezifikationen konnten von Beryl nicht innerhalb von 4 Stunden verifiziert werden (siehe Tabelle 5.4 und Abbildung 5.33). Allerdings war bei allen nicht überprüften Spezifikationen der Aufbau der Kripke-Struktur abgeschlossen und somit das Ende der Verifikation der jeweiligen Spezifikation absehbar. An den Spezifikationen 40 (siehe Abbildung 5.30) und 41 (siehe Abbildung 5.31) ist sehr deutlich zu sehen, wie unterschiedlich die Zeiten ausfallen können, obwohl die genannten Spezifikationen sich lediglich in den Konstanten beim Vergleich unterscheiden.

Der SMV schaffte die Überprüfung aller Spezifikationen innerhalb von 3 Stunden. Für den Beryl ist anzunehmen, daß dieser die Verifikation mit einem weniger eingeschränkten Zeitlimit ebenfalls geschafft hätte. Allerdings wäre es auch nicht verkehrt gewesen, das Pipeline-Modell noch weiter zu abstrahieren. Stellt sich hier die Frage, an welcher Stelle weiter gespart werden könnte. Zur Überprüfung des Prozessors auf korrektes Stall-Verhalten wird eine Kopie des Befehlswort, welches an der ID-Stufe anliegt, angelegt. Diese wird jedoch nur für eine Spezifikation verwendet und hätte für alle anderen ausgeklammert werden können. Hier hätte man 32 Bit einsparen können, was mit Sicherheit zu einer schnelleren Verifizierung geführt hätte.

Spez.	Beryl	SMV	Spez.	Beryl	SMV	Spez.	Beryl	SMV
1	>4h	379s	17	1202s	155s	33	1177s	214s
2	>4h	70s	18	1203s	151s	34	1174s	217s
3	1160s	380s	19	1197s	153s	35	1168s	95s
4	1162s	86s	20	1184s	151s	36	1176s	95s
5	1169s	86s	21	1211s	152s	37	1161s	94s
6	1158s	86s	22	1190s	151s	38	1360s	381s
7	1163s	359s	23	1206s	152s	39	>4h	728s
8	1163s	86s	24	1190s	152s	40	8404s	438s
9	1162s	86s	25	1205s	154s	41	>4h	254s
10	1197s	151s	26	1193s	122s	42	>4h	1004s
11	1204s	152s	27	1200s	132s	43	>4h	611s
12	1198s	151s	28	1198s	97s	44	1166s	69s
13	1203s	153s	29	1200s	97s	45	>4h	1193s
14	1177s	153s	30	1164s	235s	46	1165s	70s
15	1217s	155s	31	1167s	94s	47	>4h	307s
16	1192s	153s	32	1171s	99s	$\sum$	>46h	≈3h

Tabelle 5.4: Benchmark der Verifikation der einzelnen Spezifikationen des abstrahierten Pipeline-Modells

Ein weiterer Anreiz wäre allgemein die Verkleinerung, bzw. Kürzung des Befehlsregisters, welches mit 32 Bit einen erheblichen Anteil am Zustandsraum hat. Da der hier implementierte Befehlssatz recht klein ist, könnte man den Opcode um die entsprechende Länge kürzen. Ob sich dies wirklich in einer schnelleren Verifikation bemerkbar macht, muß allerdings erst durch einen Benchmark geklärt werden. Benchmarkergebnisse des Eintaktmodells haben gezeigt, daß ein kleinerer Zustandsraum nicht zwangsweise zu einer schnelleren Verifizierung führt.
Bryant und Velev verifizieren einen Prozessor mit einer fünfstufigen Pipeline mit Hilfe von EVC auf einem 336MHz Sun4 innerhalb von 0,2 Sekunden [MNV01]. Sie beschreiben ihren Prozessor in HDL und verifizieren diesen ebenfalls mit Hilfe von Spezifikationen, welche durch Aussagenlogik beschrieben sind. Allerdings gibt es keine Angaben über den Umfang, bzw. die Anzahl der Spezifikationen. Basis ihrer Spezifikationen ist ein Referenzmodell, welches durch ihr Tool in Aussagenlogik umgewandelt wird. Welche Zeit die Umwandlung in Anspruch nimmt, wurde nicht erwähnt. Evtl. lässt sich beim hier beschriebenen Pipeline-Modell die Verifikation durch andere, äquivalente Beschreibung der Spezifikationen beschleunigen.

5.4.3 Benchmark

Die Benchmarks liefen unter den gleichen Bedingungen wie die des Eintaktmodells. Da lediglich eine Bitvektorvariante des Pipeline-Modells implementiert wurde, gibt es keine Ergebnisse für die Verifikation einer Ganzzahlvariante. Die Verifikation des Gesamtmodells war aufgrund der Komplexität bzw. der Gröle des Zustandsraums nicht in einem

Durchgang möglich. Daher wurde die Verifikation des Gesamtmodells in mehreren Durchläufen mit jeweils einer Spezifikation durchgeführt. Zum SMV ist zu sagen, daß er es aufgrund des enormen Speicherbedarfs nicht geschafft hat. Beryl scheiterte bei abgeschaltetem variable reordering ebenfalls am Speicherbedarf; bei eingeschaltetem variable reordering benötigte der Aufbau der Kripke-Struktur über 4 Stunden und wurde abgebrochen.

5.4.4 Verifizierung mit Produktautomaten

Ein weiterer Ansatz zur Verifikation von endlichen Automaten sind Produktautomaten. Dies setzt zu einem gegebenen Automaten einen vergleichbaren Automaten voraus. Börger und Mazzanti verwenden zur Verifikation eines RISC-Prozessors mit Pipelining einen sequentiellen Prozessor [BM96]. Allerdings wählen auch sie den Weg der Abstrahierung und verwenden formale Beweise. Hier wird kurz auf den Nutzen des Vergleichs von Automaten für die automatische Verifizierung eingegangen.

Ein Produktautomat [Joh04, Chapter 9] liefert für eine Eingabe ein „erfüllt“, wenn beide Automaten, unter Berücksichtigung ihres momentanen Zustands, zu dieser Eingabe die gleiche Ausgabe liefern. Abbildung 5.34 zeigt zwei Automaten und ihren Produktautomaten.
Da der Produktautomat der zwei Automaten als Zustandsraum das Produkt der beiden ursprünglichen Zustandsräume hat, folgt daraus, daß eine Verifikation dementsprechend mehr Speicher und Zeit benötigt, als die Verifikation der einzelnen Automaten.

Da, außer dem Pipeline-Modell, ein Eintaktmodell des Prozessors existiert, welches schon verifiziert wurde, liegt die Verifikation mit Hilfe eines „Vergleichs“ des Pipeline-Modells mit dem Eintaktmodell nahe. Dazu müssten beide Modelle mit denselben Befehlen gespeist werden, während die Ausgabe auf Gleichheit überprüft werden würde. Allerdings ist dies hier aus zwei Gründen nicht möglich. Zum einen weichen die Spezifikationen des Verhaltens des Pipeline-Modells von denen des Eintaktmodells in folgenden Punkten ab:

- Das Pipeline-Modell interpretiert den Befehl nach bedingten Sprüngen als delay slot, führt diesen also im Gegensatz zum Eintaktmodell auch bei erfüllter Sprungbedingung aus.
- Der delay slot darf nicht mit einem Steuerflußbefehl belegt werden.

Zum anderen ist im Benchmark des Pipeline-Modells (siehe Abschnitt 5.4.3) zu sehen, daß selbst die Verifikation des reduzierten Prozessors viel Zeit und Speicher benötigt. Wird eine Verifikation durch einen Produktautomaten angestrebt, bedeutet das, daß, wie zuvor erwähnt, aus den Zustandsräumen der Prozessoren das Produkt der Zustandsräume gebildet werden muß, welcher folglich viel größer ist, als der Zustandsraum des einzelnen Pipeline-Modells. Für die Verifikation mit den Model Checkern bedeutet dies, daß noch mehr Speicher und Zeit benötigt wird. Aus diesem Grund ist es nicht sinnvoll die Verifikation des vollständigen Pipeline-Modells zusammen mit dem Eintaktmodell

in Betracht zu ziehen. Selbst wenn genug Speicher und Zeit vorhanden wäre, würde die Verifikation durch den Vergleich der beiden Modelle dadurch erschwert werden, daß das Pipeline-Modell aufgrund von Stalls selbständig `NOPs` einfügen kann und die Signale des Prozessors dadurch eine immer größer werdende zeitliche Distanz zu den Signalen des Eintaktmodells bekommen. Dies kann zwar dadurch umgangen werden, indem das Eintaktmodell bei einem Stall des Pipeline-Modells einen Takt lang suspendiert wird, zeigt jedoch, daß der Vergleich der beiden Modelle bei weitem nicht so einfach ist, wie es der allgemeine Ansatz für Produktautomaten vermuten lässt. Ein weiterer Einwand, die Gleichheit des Verhaltens einer Implementierung mit dem Verhalten einer anderen über Synchronschaltung zu überprüfen ist folgender: da das Verhalten des Referenzmodells schon verifiziert wurde, müssen auch die Spezifikation, die das Verhalten beschreiben, vorliegen. Somit ist die Synchronschaltung eine unnötige Verkomplizierung des Problems, da es reichen würde, das zu prüfende Modul lediglich gegen die schon vorhandenen Spezifikationen zu testen.

5.4.5 Synthese

Die Synthese des Pipeline-Modells ist, wie die Synthese des Eintaktmodells, nicht ohne Implementierung weiterer Module möglich. Um den Prozessor sinnvoll synthetisieren zu können, fehlt auch hier die Einbettung des Prozessors in ein Systems, welches das Registerfeld, den Befehlsspeicher, sowie den Datenspeicher zur Verfügung stellt. Da das Verhalten dieser Komponenten exakt dem Verhalten der Module des Eintaktsystems entspricht, können diese auch für das Pipeline-Modell verwendet werden. Auch das Modul, welches alle Komponenten instanziiert und zu einem System, zusammenfasst kann bis auf eine kleinere Änderung vollständig übernommen werden. Bei der Ansteuerung des Registerfeldes wurden im Eintaktsystem die Indizes der zu lesenden Register direkt aus dem vom Befehlsspeicher kommenden Befehlswort extrahiert und mit dem Registerfeld verbunden. Im System mit dem Pipeline-Modell stellt der Prozessor die Indizes zur Verfügung. Die Instanziierung des Pipeline-Modells und des Registerfeldes zeigt Abbildung 5.35. Um die Synthetisierung des Pipeline-Modells abzuschließen, müssen die Variablen *RegIndexRS* und *RegIndexRT* noch in die Liste der `local`-Variablen hinzugefügt werden.

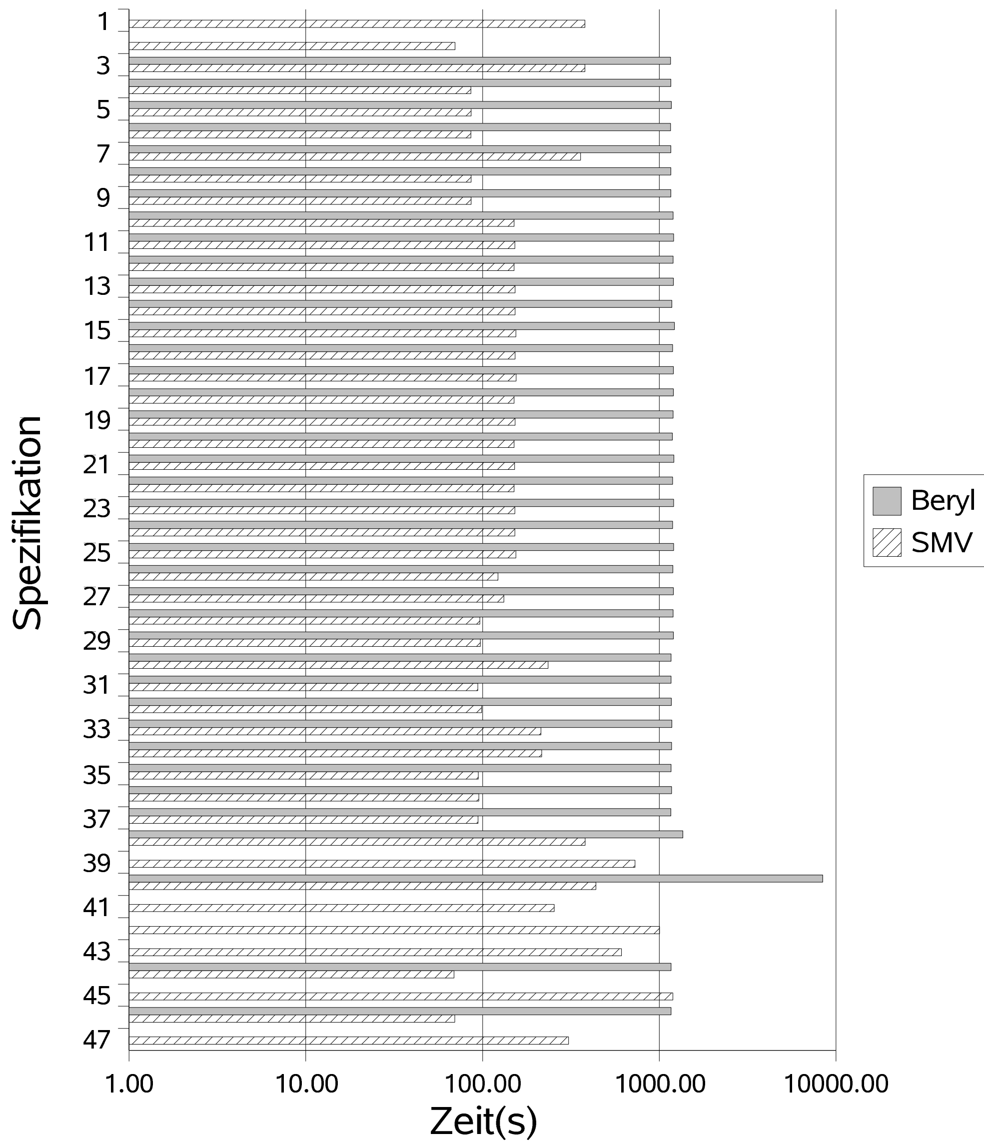


Abbildung 5.33: Ergebnisse der Benchmarks des Pipeline-Modells

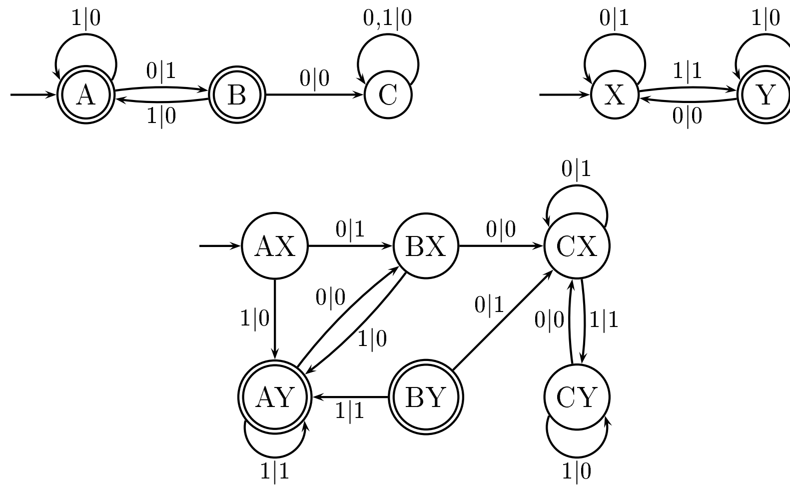


Abbildung 5.34: Zwei Automaten und ihr Produktautomat [Pro]

```
    ⋮

RISC: run Pipeline
     (ExtMemValueOut,
      RegValueRS, RegValueRT,
      RISCInstrReg)
     (MemOut, MemAddr, MemWrite,
      RegIndexRS, RegIndexRT, RegIndexRD,
      RegWriteRD, RegValueRD,
      PC, UNKNOWN_OPCODE)
||
RegArray: run RegisterArray
     (RegIndexRS, RegIndexRT, RegIndexRD,
      RegWriteRD,RegValueRD)
     (RegValueRS, RegValueRT)

    ⋮
```

Abbildung 5.35: Instanziierung des Pipeline-Modells und des Registerfeldes - Modifikationen des Eintaktsystems

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Die Verifikation der Prozessoren mit Hilfe von Model Checkern hat gezeigt, daß selbst schnelle Rechner mit großzügiger Speicherausstattung an ihre Grenzen geführt wurden. Durch geschickte Modularisierung größerer Implementierungen ist die Verifizierung jedoch nicht nur möglich, sondern geht zudem schnell vonstatten. Kommt es während der Implementierungsphase zu Änderungen, können Spezifikationen ohne menschlichen Aufwand erneut geprüft werden. Allerdings muß man sich immer vor Augen halten, daß nur die angegebenen Spezifikationen geprüft werden. Wird ein System nicht vollständig beschrieben, weil eine oder mehrere Spezifikationen falsch (z.B. durch einfache Tippfehler oder falsche Klammersetzung) oder gar nicht angegeben wurden, kann so ein fehlerhaftes System entstehen. Die modulare Verifizierung setzt zudem genauere Kenntnis der Implementierung voraus. Dies sollte jedoch kein Problem sein, da selbst bei einem größerem Projekt mit mehreren Teams die Verifikation der einzelnen Module von den Entwicklern selber durchgenommen werden muß. Problematisch wird es hier erst, wenn das aus den einzelnen Modulen zusammengesetzte Produkt gegen weitere Spezifikationen geprüft werden muß und eine Abstrahierung notwendig ist. Hier ist eine geschickte Vorgehensweise gefragt. Die triviale Lösung wäre selbstverständlich die Umgehung des Problems, indem die Spezifikationen vornherein so aufgeteilt werden, daß das Gesamtmodul nicht mehr verifiziert werden muß.

Die Pipeline des hier implementierten Pipeline-Modells ist sehr einfach gehalten. Jeder Befehl kann, sobald er einmal vom Prozessor „eingelesen“ wurde, oder mit anderen Worten die `ID`-Stufe erreicht hat, maximal einen Takt verzögert werden. Dementsprechend sehen die Spezifkationen recht einfach aus, da es bei der Beobachtung des Verhaltens lediglich zwei Fälle gibt. Heutige Prozessoren sind wesentlich komplexer, und können Befehle weitaus öfter verzögern. Daher müssten auch die Spezifkationen überarbeitet werden. Interessant wäre hier, wie sich die abgewandelten Spezifikationen auf das zeitliche Verhalten des Model Checking auswirken.

Im letzten Abschnitt des Pipelineprozessors wurde über den Ansatz der Verifikation durch Synchronschaltung zweier Implementierungen diskutiert. Dies macht in der Regel keinen Sinn, da dadurch der Zustandsraum enorm erhöht wird. Soll gezeigt werden, daß eine Implementierung das gleiche Verhalten wie eine andere, gegebene Implementierung aufweist, reicht es, die in QUARTZ beschriebenen Module jeweils gegen diegleichen Spezifikationen zu prüfen. Zudem wäre dieses Prinzip nicht mehr brauchbar, sobald es um die Verifikation eines superskalaren Prozessors geht, welcher Befehle out-of-order ausführt.

Dieser Prozessor verrichtet zwar dieselbe Arbeit aber aus Sicht eines Produktautomaten in ganz und gar nicht ähnlicher Arbeitsweise, da die zeitlichen Veränderungen des Prozessors auf das System unter Umständen vollkommen umgordnet werden kann.

Besonders die Implementierung des Pipeline-Modells hat gezeigt, daß speziell bei der Entwicklung von logischen Schaltungen, bei denen Optimierungen zur Verkleinerung der Schaltung üblich sind, die korrekte Funkionsweise gewährleistet wird. Während aus der optimierten Schaltung nicht mehr zwangsläufig die Funktion der Schaltung erkennbar ist, bieten die Spezifikationen eine bessere Möglichkeit zum Verständnis der Schaltung. Hinzu kommt, daß in Spezifikationen keine don't care Fälle betrachtet werden. Spezifikationen beschreiben lediglich das Verhalten, welches gefordert wird, während die Implementierung der Schaltung das vollständige Verhalten beschreibt.

Als Nachteil hat sich gezeigt, daß keine Vorhersage getroffen werden kann, wie lange die Verifikation eines Moduls dauert. Hier müssen entweder die Parameter der Model Checker variiert werden oder das zu verifizierende Modul abstrahiert oder in mehrere Teilmodule zerlegt werden.

6.2 Ausblick

Mit dem Pipeline-Modell wurde bei weitem nicht der heutige Stand der Technik erreicht. Interessant wäre es, folgende Schritte durchzuführen:

- **Berechnung der längsten Pfade.** Die maximale Taktfrequenz wird durch die langsamste Stufe bestimmt. Ist ein Pfad um ein vielfaches länger, als alle anderen Pfade, so richtet sich die Geschwindigkeit nach diesem Pfad, obwohl es nicht notwendig ist, daß jeder Befehl eine Berechnung über diesen Pfad benötigt. Kann man die Berechnung, welche über diesen Pfad läuft, optimieren, so daß der längste Pfad in etwa die Länge der andere Pfade erreicht, kann der Prozessor mit einer höheren Taktfrequenz betrieben und somit eine höhere Leistung erreicht werden.

- **Implementierung einer „realistischen" Speicherschnittstelle mit Steuerung.** Da die Kommunikation des Prozessors mit dem Arbeitsspeicher, also das Laden, bzw. Speichern eines Wortes, in der Regel nicht innerhalb eines Taktes, sondern über mehrere Takte verläuft (siehe Abbildung 4.17), könnte man zunächst damit beginnen, ein alleinstehendes Modul zum Ansprechen des Arbeitsspeicher zu implementieren. Bei der Einbindung des Speichercontrollers in den Prozessor wird mit einem festgelegtem Taktteiler gearbeitet, so daß das bestehende Prozessormodell nicht verändert werden muß. Im nächsten Schritt gilt es, das Modul ohne Taktteiler in den Prozessor einzubinden und die Schnittstelle des Prozessors zum Speicher anzupassen. Durch Verzögerungen beim Lesen und Schreiben ist darauf zu achten, daß die `MEM`-Stufe des Pipelineprozessors evtl. vorhergehende Pipelinestufen anhält um die korrekte Abarbeitung zu gewährleisten. Da FPGA Boards nicht zwangsweise zwei Speicherbausteine besitzen müssen oder diese nicht unbedingt

beide simultan ansprechen können, ist es notwendig Befehls- und Datenwörter aus einem Speicher zu lesen, bzw. zu schreiben. Hier ist folglich auf Zugriffskonflikte und Fairness zu achten.

- **Erweiterung des Befehlssatzes.** Angefangen bei den unvollständigen Shiftern, über Ganzzahlmultiplikation und -division bis zur Fließkommaarithmetik, kann die ALU um viele Befehle erweitert werden. Bei die Implementierung von komplexen Befehlen wäre es sinnvoll, an die ALU nicht mehr die Bedingung zu stellen, daß ein Befehl innerhalb eines Taktes abgearbeitet werden muß. Damit wäre auch die `EX`-Stufe in der Lage, vorhergehende Pipelinestufen durch ein *busy*-Signal anzuhalten. Dies führt wiederrum zu aufwändigeren Spezifikationen, da für Befehle mehr und längere Verzögerungen (Stalls) auftreten können.

- **Implementierung eines Interrupt Controllers.** Hier könnte man damit beginnen, auf das Signal *UNKNOWN_OPCODE* zu reagieren, indem eine festgelegte Behandlungsmethode gestartet wird.

- **Mechanismen zur Synchronisation von Mehrprozessorsystemen**. Die heutigen Prozessoren arbeiten mit enormen Taktfrequenzen im Bereich von mehreren Gigahertz. Durch physikalische Beschränkungen kann mit einer weiteren Erhöhung der Taktfrequenz, wie sie bisher beobachtet wurde nicht mehr gerechnet werden. Der Trend geht momentan eindeutig zu Mehrprozessorsystemen. Damit Prozesse diese Leistung nutzen können, müssen sie die zu erledigende Arbeit auf mehrere Threads verteilen. Damit eine sichere Kommunikation zwischen den Threads gewährleistet und Zugriffe auf gemeinsame Daten koordiniert werden können, muß ein Prozessor Mittel zur Synchronisierung dieser Threads zur Verfügung stellen.

- **Implementierung eines superskalaren Prozessors.** Moderne Prozessoren sind in der Lage den Befehlsstrom zu analysieren und ihn umzuordnen. Zudem können sie mehrere Befehle gleichzeitig abarbeiten, indem die ALU besser ausgelastet wird (z.B. indem eine Addition **und** eine Multiplikation berechnet wird). Dadurch sind diese Prozessoren in der Lage sehr kleine CPI Werte zu erreichen.

Abbildungsverzeichnis

2.1 Aufbau eines Moduls in QUARTZ 2
2.2 Steuerflußanweisungen in QUARTZ 4
2.3 Das ABRO Beispiel 5
2.4 Ein nicht-reaktives System 6
2.5 Ein nicht-deterministisches System 6
2.6 Schreibkonflikt vom Typ NOW-NOW 7
2.7 Schreibkonflikt vom Typ NEXT-NEXT 7
2.8 Schreibkonflikt vom Typ NEXT-NOW 7

3.1 Beispiele für LTL-Beschreibungen und ihre graphische Interpretation . . . 9
3.2 Beispiele für CTL-Beschreibungen und ihre graphische Interpretation . . . 11

4.1 Schnittstelle der Bitvektor-Variante des Eintaktmodells 13
4.2 Schnittstelle der Ganzzahlvariante des Eintaktmodells 15
4.3 Ausschnitt aus der Implementierung des Eintakt-Verhaltensmodells 16
4.4 Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - Setzen der Ausgaben, die die Befehle gemeinsam haben 17
4.5 ALU Modul des Eintaktmodells 18
4.6 Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - Ansteuern der ALU 19
4.7 Ausschnitt aus der Implementierung des Eintakt-Strukturmodells - setzen der, vom Ergebnis der ALU abhängigen Ausgaben 20
4.8 Allgemeine Spezifikation eines Befehls 21
4.9 Spezifikation des Verhalten des Prozessors bei anliegendem ADD-Befehls . . 22
4.10 vollständige Spezifikation des Verhalten des Prozessors bei anliegendem ADD-Befehl 22
4.11 Spezifikationen des Verhaltens von systemverändernden Signalen 23
4.12 Zeiten der Benchmarks des ALU Moduls 25
4.13 Speicherbedarf der Benchmarks des ALU Moduls - Angegeben ist der Peaklevel der erstellten BDD-Knoten 26
4.14 Zeiten der Benchmarks der Eintaktvarianten 28
4.15 Speicherbedarf der Benchmarks der Eintaktvarianten - Angegeben ist der Peaklevel der erstellten BDD-Knoten 29
4.16 Verhalten des idealisierten Speichers 30
4.17 Eingriffsmöglichkeiten bei Softwaresynthese 30
4.18 Implementierung des Registersatzes 31

4.19 Implementierung des Gesamtsystems - Schnittstelle 32
4.20 Implementierung des Gesamtsystems - lokale Variablen 32
4.21 Implementierung des Gesamtsystems - Verhalten 34

5.1 Abarbeitung einer Befehlskette in einem Pipeline Prozessor 35
5.2 Beziehung der Pipeline-Stufen untereinander und mit der Umwelt 40
5.3 Schnittstelle der `IF`-Stufe 41
5.4 Implementierung der `IF`-Stufe 42
5.5 Schnittstelle der `ID`-Stufe 44
5.6 Implementierung der `ID`-Stufe - interne Auswertung des Befehls 45
5.7 Implementierung der `ID`-Stufe - Setzen der Ausgänge 45
5.8 Implementierung der `ID`-Stufe - Setzen der Ausgänge 46
5.9 Implementierung der `ID`-Stufe - Bestimmung der Befehlsklasse 46
5.10 Implementierung der `ID`-Stufe - Bestimmung des Opcodes für die ALU . . 47
5.11 Implementierung der `ID`-Stufe - Forwarding - Behandlung von RAW-Konflikten 47
5.12 Schnittstelle der `EX`-Stufe 48
5.13 Implementierung der `EX`-Stufe - Einbindung des ALU Moduls 49
5.14 Implementierung der `EX`-Stufe - Auswertung des Sprungergebnisses und Setzen der Ausgangsvariablen 49
5.15 Schnittstelle der `MEM`-Stufe 50
5.16 Implementierung der `MEM`-Stufe 51
5.17 Schnittstelle der `WB`-Stufe 52
5.18 Implementierung der `WB`-Stufe 52
5.19 Betrachtung der gültigen Kombination von Signalen 58
5.20 Pseudoalgorithmus zur Verifikation des Forwarding 59
5.21 Spezifikationen für korrektes Forwarding in CTL 60
5.22 Beispiele für Spezifikationen der `EX`-Stufe in CTL 61
5.23 Verfolgen des Signals zum Schreiben des Registers 63
5.24 Zusammenhang zwischen dem Signal zum Schreiben des Registers und dem Befehlscode 64
5.25 Verhalten eines Stalls in CTL 65
5.26 Verhalten des `XOR`-Befehls, falls dieser keinen Stall verursacht 65
5.27 Verhalten des `XOR`-Befehls, falls dieser einen Stall verursacht 66
5.28 Zusammenhang zwischen dem unbedingten Sprungbefehl und dem Stallsignal 66
5.29 Beispiel für die graphische Interpretation der Spezifikation aus Abbildung 5.26 und 5.27 67
5.30 Korrektes Dekodieren des `ADDI` Befehls 68
5.31 Korrektes Dekodieren des `LW` Befehls 68
5.32 Ergebnisse der Benchmarks der Pipelinestufen 70
5.33 Ergebnisse der Benchmarks des Pipeline-Modells 75
5.34 Zwei Automaten und ihr Produktautomat [Pro] 76
5.35 Instanziierung des Pipeline-Modells und des Registerfeldes - Modifikationen des Eintaktsystems 76

Tabellenverzeichnis

4.1 Ergebnisse des Benchmarks der Verifizierung des ALU Moduls 24
4.2 Ergebnisse der Benchmarks der Eintaktvarianten 27
4.3 Vergleich der beiden Modelle mit dem modifizierten Eintakt-Strukturmodell 27

5.1 Ergebnisse des Benchmark der `IF`-Stufe und `ID`-Stufe 69
5.2 Ergebnisse des Benchmark der `EX`-Stufe und `MEM`-Stufe 69
5.3 Ergebnisse des Benchmark der `WB`-Stufe 69
5.4 Benchmark der Verifikation der einzelnen Spezifikationen des abstrahierten Pipeline-Modells 72

Literaturverzeichnis

[AN96] A. C. J. Fox und N. A. Harman: *An algebraic model of correctness for superscalar microprocessors.* In: M. Srivas und A. Camilleri (Herausgeber): *First international conference on formal methods in computer-aided design*, Band 1166, Seiten 346–361, Palo Alto, CA, USA, 1996. Springer Verlag. URL: citeseer.ist.psu.edu/fox96algebraic.html.

[Ave05] *Averest*, 2005. URL: www.averest.org.

[Ber97] Berry, G.: *The Esterel v5 Language Primer.* http://www-sop.inria.fr/esterel.org/, April 1997.

[BM96] Borger, E. und S. Mazzanti: *A Correctness Proof for Pipelining in RISC Architectures*, 1996. URL: citeseer.ist.psu.edu/16386.html.

[Fox01] Fox, A.: *An algebraic framework for modelling and verifying microprocessors using HOL*, 2001. URL: citeseer.ist.psu.edu/fox01algebraic.html.

[Har00] Harman, N.: *Correctness and verification of hardware systems using Maude*, 2000. URL: citeseer.ist.psu.edu/harman00correctness.html.

[Har03] Harrison, John: *Intel's Successes with Formal Methods.* World Forestry Center, Portland OR : Software, Science and Society, 2003.

[Hos00] Hosabettu, R.: *Systematic Verification of Pipelined Microprocessors.* Doktorarbeit, Department of Computer Science, University of Utah, aug 2000. URL: citeseer.ist.psu.edu/hosabettu00systematic.html.

[Joh04] Johann, Weber: *Entwurf eines RISC-Prozessors mit synchronen Sprachen.* Diplomarbeit, TU Kaiserlautern, November 2004.

[KJT04] Klaus Schneider, Jens Brandt und Tobias Schuele: *Causality Analysis of Synchronous Programs with Delayed Actions.* In: *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Seiten 179–189, Washington D.C., USA, September 2004. ACM.

[KP01] Kroening, Daniel und Wolfgang Paul: *Automated Pipeline Design.* In: *Proc. of the 38th Design Automation Conference*, Seiten 810–815. ACM Press, 2001.

[KPM00] KROENING, DANIEL, WOLFGANG PAUL und SILVIA MUELLER: *Proving the Correctness of Pipelined Micro-Architectures.* In: WALDSCHMIDT, KLAUS und CHRISTOPH GRIMM (Herausgeber): *Proc. of ITG/GI/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, Seiten 89–98. VDE Verlag, 2000.

[Kro96] KROPF, THOMAS: *Hardwareverifikation: Verfahren und Werkzeuge zum Entwurf korrekter Schaltungen und Systeme.* Doktorarbeit, Universitaet Karlsruhe, 1996.

[Kro99] KROENING, DANIEL: *Design and Evaluation of a RISC Processor with a Tomasulo Scheduler.* Diplomarbeit, University of Saarland, Computer Science Department, Germany, 1999.

[Kro01] KROENING, DANIEL: *Formal Verification of Pipelined Microprocessors.* Doktorarbeit, Saarland University, 2001.

[Man00a] MANOLIOS, PANAGIOTIS: *Correctness of Pipelined Machines.* In: *Formal Methods in Computer-Aided Design*, Seiten 161–178, 2000. URL: citeseer.ist.psu.edu/426677.html.

[Man00b] MANOLIOS, PANAGIOTIS: *Verification of Pipelined Machines in ACL2*, 2000. URL: citeseer.ist.psu.edu/manolios00verification.html.

[Mip02] *MIPS32 Architecture For Programmers, Volume II: The MIPS32 Instruction Set*, September 2002.

[MNV01] MIROSLAV N. VELEV, RANDAL E. BRYANT: *EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations.* Department of Electrical and Computer Engineering, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A., 2001.

[Pro] *Produktautomatenkonstruktionen.* URL: http://www.brics.dk/~nygaard/dRegAut/product15.html.

[Ran04a] RANGE, BENJAMIN: *Syntax und Semantik von CTL* und LTL.* Proseminar Modale Logik und deren Anwendung – TU Dresden, 2004.

[Ran04b] RANGE, BENJAMIN: *Syntax und Semantik von CTL* und LTL - Proseminar Modale Logik und deren Anwendung.* Proseminar Modale Logik und deren Anwendung, 2004.

[RB99] RAMESH, S. und PURANDAR BHADURI: *Validation of Pipelined Processor Designs Using Esterel Tools: A Case Study.* In: *Computer Aided Verification*, Seiten 84–95, 1999. URL: citeseer.ist.psu.edu/ramesh99validation.html.

[Sch03] SCHNEIDER, K.: *Verification of Reactive Systems – Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[Sch04] SCHWOON, STEFAN: *Model-Checking (SS 2004), Sichere und Zuverlaessige Softwaresysteme.* Institut fuer Formale Methoden der Informatik, Universitaet Stuttgart, 2004.

[Sch05] SCHNEIDER, KLAUS: *System Description Languages.* Presentation of Course, 2005.

[SR94] S. TAHAR und R. KUMAR: *Formal Verification of Pipeline Conflicts in RISC-Processors.* In: *Proc. European Design Automation Conference (EURO-DAC94)*, Seiten 285–289, Grenoble, France, 1994. IEEE Computer Society Press. URL: citeseer.ist.psu.edu/tahar94formal.html.

[SSH+99] SCHMID, DETLEF, KLAUS SCHNEIDER, MICHAELA HUHN, GEORGE LOGOTHETIS und VIKTOR SABELFELD: *Formale Verifikation eingebetteter Systeme*, 1999.

[Wil05] WILLKOMM, DENNIS: *Verifikation der Fliesskomma-Arithmetik bei Intel*, 2005.