Simulation eines MIPS-R3000-kompatiblen Mikroprozessors als HADES-Modell

Claas Rink

Erstbetreuer: Dr. Norman Hendrich

Zweitbetreuer: Prof. Dr. Bernd Wolfinger

17. April 2000

Diplomarbeit am Fachbereich Informatik, Arbeitsbereich TECH, Universität Hamburg

Erklärung

Ich versichere, daß ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und daß ich diese Arbeit in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Hamburg, den 17. April 2000

(Claas Rink)

Zusammenfassung

Das Ziel dieser Diplomarbeit ist die Bereitstellung eines Modells zur Simulation eines MIPS-R3000-kompatiblen Mikroprozessors als HADES-Modell und einer Programmiersprache zur Entwicklung von hochsprachlichen Programmen, die auch auf realen Systemen genutzt werden kann.

Als Ergebnis steht ein in Java implementiertes HADES-Modell zur Systemsimulation des IDT R3051 zur Verfügung. Programme, die innerhalb des Simulationsmodells benutzt werden sollen, können mit dem frei verfügbaren und selbst kompilierbaren GNU-C-Crosscompiler erstellt und vom Simulationsmodell ausgeführt werden. Dieses selbst bildet nicht nur die Schnittstellen, sondern auch das interne Verhalten des zugrunde liegenden Prozessors ab.

Somit steht nun eine Möglichkeit zur Verfügung, mit HADES Hardware zusammen mit davon ausgeführter hochsprachlicher Software zu simulieren. Dabei kann anhand einer Inspektionsschnittstelle der Programmablauf und das interne Verhalten des Prozessors dargestellt werden.

Danksagung

Diese Diplomarbeit ist am Arbeitsbereich Technische Grundlagen der Informatik (TECH) des Fachbereichs Informatik der Universität Hamburg entstanden. Mein besonderer Dank gilt Dr. N. Hendrich für die Unterstützung meiner Arbeit, die vielen Ratschläge und Hilfestellungen und die teilweise wirklich zeitintensive Betreuung. Herrn Prof. Dr. B. Wolfinger danke ich besonders für die Hinweise und Anregungen zum Verfassen dieses Textes, der ohne diese nicht in der vorliegenden Form existieren würde. Mein weiterer Dank gilt meiner Mutter A. Rink-Rogge, deren Kenntnisse der deutschen Sprache weit über meine eigenen hinaus reichen. Ganz besonders möchte ich meiner Frau, M. Rink, danken, die viele Wochenenden ohne mich auskommen mußte und mich trotzdem wunderbar unterstützt hat. Außerdem gilt mein Dank meinem Arbeitgeber, der Netlife GmbH, und meinen dortigen Mitarbeitern, die es mir dank flexibler Arbeitszeiten und kooperativem Umfeld ermöglicht haben, diese Diplomarbeit trotz Termindruck ohne Kompromisse durchzuführen.

Claas Rink

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Schon seit vielen Jahren werden rechnergestützte Simulationen verwendet. Dadurch sollen reale Objekte oder Vorgänge unter bestimmten (meist vereinfachenden) Gesichtspunkten nachgebildet werden. Für den Einsatz von Simulationsexperimenten existieren viele, von Fall zu Fall verschiedene Gründe. Als zu simulierende Objekte oder Vorgänge sind unterschiedlichste Möglichkeiten denkbar. Diese Arbeit wird ein Simulationsmodell für einen Mikroprozessor entwickeln und implementieren.

Für die Simulation eines Mikroprozessors existieren mehrere Gründe: Soll Software für ein System entwickelt werden, das noch nicht existiert, so muß zumindest dessen Kern (das sind in der Regel der Prozessor und ein Speicher) in irgend einer Form vorhanden sein. Häufig ist es vorteilhaft, statt eines Prototypenaufbaus ein hinreichend detailliertes Simulationsmodell zu verwenden. Innerhalb dieses Simulationsmodells kann dann die Software entwickelt werden, die später auf dem zu implementierenden System ausgeführt werden soll. Insbesondere, wenn vor dem Aufbau des Systems dessen exakte Spezifikationen noch nicht bekannt sind, kann durch Simulation der einzelnen Systemkomponenten herausgefunden werden, wie die verschiedenen beteiligten Komponenten dimensioniert sein müssen.

Auch für den Entwurf eingebetteter Systeme sind Simulationen unverzichtbarer Teil des Entwicklungszyklus, denn diese Systeme verfügen nicht über die notwendigen Resourcen, um innerhalb des Systems selbst Software zu entwickeln. So wird oft ein Simulationsmodell als Entwicklungsplattform verwendet.

Einen weiteren Grund für die Simulation eines Mikroprozessors stellt ein erhoffter Erkenntnisgewinn über intern im Mikroprozessor ablaufende Vorgänge dar. Gerade ein Simulationsmodell, das eine Variation interner Komponenten ermöglicht, kann helfen, die Schwächen eines existierenden Mikroprozessors offen zu legen und optimierte Weiterentwicklungen des Prozessors mittels Simulationsexperimenten zu testen. Solche Experimente können auch Lehrzwecken dienen.

Besonders interessant ist die Simulation von Mikroprozessoren, wenn diese nicht nur allein, sondern zusammen mit der sie umgebenden und verschieden konfigurierbaren Peripherie simuliert werden. Diese ziemlich aufwendige Variante wird nur selten angeboten. Sehr flexibel läßt sich diese Aufgabe lösen, indem mehrere, voneinander unabhängige Simulationsmodelle für die Nachbildung einzelner Komponenten durch eine Simulationsumgebung so miteinander verbunden werden, daß als Ergebnis eine Systemsimulation eines komplexen Systems bestehend aus vielen verschiedenen Einzelkomponenten zur Verfügung steht.

1.1 Aufgabenstellung

Für einen real existierenden Mikroprozessor lassen sich verschiedene Schnittstellen definieren, über die der Prozessor mit der ihn umgebenden Welt in Verbindung tritt. Dabei muß zwischen für die gewünschten Simulationszwecke relevanten und irrelevanten Schnittstellen unterschieden werden.

Die wichtigste Schnittstelle ist die *Ein-/Ausgabe-Schnittstelle*, die durch das Verhalten des Prozessors als Reaktion auf seine Eingangssignale definiert wird (folgend wird diese Schnittstelle die *Hardware-Schnittstelle* genannt). Alle weiteren Schnittstellen basieren auf dieser (abstrahieren von der Hardware-Schnittstelle) oder stellen parasitäre Effekte dar, die aufgrund von Umgebungseinflüssen auftreten (zum Beispiel Temperatur, elektromagnetische oder atomare Strahlung).

Eine von der Hardware-Schnittstelle abstrahierende Schnittstelle ist die Maschinensprache des Prozessors: Jedes Kommando der Maschinensprache wird über die Hardware-Schnittstelle an den Prozessor geliefert und bewirkt eine Änderung seines internen Zustands, die der Umgebung wiederum über die Hardware-Schnittstelle mitgeteilt wird. Die Maschinensprache wird im Folgenden auch als *Software-Schnittstelle* des Prozessors bezeichnet.

Sobald die Simulation eines Mikroprozessors statt des Prozessors selbst betrachtet werden soll, kann eine weitere Schnittstelle hinzukommen: Die Interna der Simulation werden gegenüber dem Benutzer offengelegt, damit dieser die innerhalb der simulierten Hardware ablaufenden Vorgänge inspizieren kann. Im Folgenden wird diese Schnittstelle die *Inspektionsschnittstelle* genannt.

Für diese Arbeit soll die Implementation der drei aufgezählten Schnittstellen im Vordergrund stehen.

Ein Simulationsmodell eines Mikroprozessors kann erst dann spezifiziert und implementiert werden, wenn festgelegt wurde, welcher spezielle Mikroprozessor simuliert werden soll. Gerade für eine Systemsimulation bestehend aus einem Mikroprozessor und dessen Umgebung ist die Nachbildung eines möglichst leistungsfähigen Prozessors von Interesse. Denn mit steigendem Leistungsvermögen des Prozessors kann dieser eine zunehmend komplexe Umgebung steuern.

Soll allerdings ein Mikroprozessor simuliert werden, so muß darauf geachtet werden, daß das Simulationsmodell des Prozessors gegenüber dem Original nicht zu langsam wird. Leider ist es generell so, daß mit steigender Leistungsfähigkeit eines Prozessors dessen interner Aufbau und damit auch der Aufbau eines Simulationsmodells wesentlich komplizierter wird. Dadurch steigt der Zeitbedarf für die Simulation eines Mikroprozessors mit dessen Komplexität. Für diese Arbeit muß ein Mikroprozessor gewählt werden, der beide Anforderungen erfüllt: Einerseits muß er leistungsfähig genug sein, um komplex aufgebaute Umgebungen steuern zu können, und andererseits muß der Prozessor einen hinreichend einfachen internen Aufbau besitzen, damit er leistungsfähig simuliert werden kann.

Wie schon eingangs vorgestellt, soll der Mikroprozessor zusammen mit seiner Umgebung durch verschiedene Simulationsmodelle für die einzelnen Komponenten simuliert werden. Als Verbindung zwischen den verschiedenen Simulationsmodellen soll eine Simulationsumgebung zum Einsatz kommen, die die Hardware-Schnittstellen der verschiedenen Simulationsmodelle verwaltet und miteinander verbindet. Konkret wird die Simulationsumgebung HADES (kurz für "Hamburg Design System" [Hendrich 98]) verwendet, die am Fachbereich Informatik der Universität Hamburg entstanden ist.

1.2 Ziele

Das Globalziel dieser Diplomarbeit ist die Entwicklung eines für Systemsimulation geeigneten leistungsfähigen Simulationsmodells eines Mikroprozessors. Aus diesem Globalziel ergeben sich bei näherer Betrachtung mehrere Einzelziele. Werden alle Einzelziele erreicht, so kann auch das Globalziel als bewältigt angesehen werden. Im Folgenden werden die Einzelziele Z_1 bis Z_5 definiert und begründet.

Das erste *Einzelziel* Z_1 fordert eine detailgetreue Hardware-Abbildung. Wie schon im vorigen Abschnitt beschrieben, soll das Simulationsmodell sowohl die Software- als auch die Hardware-Schnittstelle des zu simulierenden Prozessors möglichst genau nachbilden. Einige Abstraktionen und Veränderungen gegenüber dem Vorbild werden notwendig sein, um ein sinnvoll nutzbares Simulationsmodell erstellen zu können. Darauf wird im einzelnen im Kapitel 5 zur Modellbildung eingegangen.

Vorher sollte das Vorbild selbst, der zugrunde liegende Prozessor, eingehend beschrieben werden. Mit diesem Thema beschäftigen sich die Kapitel 2 (Architektur-Beschreibung) und Kapitel 3 (Prozessor-Beschreibung). Zusätzlich zur reinen Schnittstellenkompatibilität des Vorbilds sollen allerdings auch die internen Komponenten (soweit dokumentiert und in Software simulierbar) nachgebildet werden. Da die zugrunde liegende Architektur einige Besonderheiten besitzt, die sie essentiell von anderen weit verbreiteten Architekturen unterscheidet, muß bei diesen Besonderheiten verstärkt auf eine detailgetreue Nachbildung des Vorbildes geachtet werden.

Als *Einzelziel* Z_2 wird die gemeinsame Simulation von Hard- und Software gefordert. Dazu muß die Software-Schnittstelle des Simulationsmodells sinnvoll nutzbar sein. Diese Forderung ergibt letztendlich den Wunsch nach hoher Geschwindigkeit der Simulation. Wünschenswert wäre natürlich der Echtzeitbetrieb, das heißt eine Simulation, die in derselben Geschwindigkeit abläuft wie die zugrunde liegende Hardware. Je geringer der Geschwindigkeitsunterschied zwischen Simulationsmodell und Vorbild ist, desto umfassender läßt sich das Simulationsmodell mit für das Vorbild verfaßter Software benutzen, desto benutzbarer ist die Software-Schnittstelle.

Da das zu entwerfende Simulationsmodell auch zusammen mit anderen innerhalb von HADES simulierten Simulationsmodellen ausführbar sein soll, muß für diese weiteren genügend Rechenzeit zur Verfügung stehen. Erst wenn sowohl das hier entstehende Simulationsmodell als auch die anderen, dieses Simulationsmodell umgebenden als Peripherie dienenden Simulationsmodelle, in Echtzeit simuliert werden können, ist das Ergebnis dieser Arbeit universell, das heißt für nahezu alle real denkbaren Zwecke einsetzbar. Da bei der Forderung nach detailgetreuer Hardware-Nachbildung auch eine möglichst genaue Implementation der internen Vorgänge enthalten war, muß diese auch trotz der teilweise dagegensprechenden Forderung nach hoher Geschwindigkeit beibehalten werden. Für diese Simulation wird das Ziel Z_1 der Originaltreue höher bewertet als das Ziel Z_2 der Geschwindigkeit.

Das *Ziel* Z_3 wünscht eine hohe Flexibilität des Simulationsmodells. Der Benutzer des Simulationsmodells muß die Möglichkeit bekommen, den simulierten Prozessor für unterschiedlichste Zwecke einzusetzen. Dabei soll er sich die Abstraktionsebene der Benutzung und auch der Darstellung der internen Vorgänge aussuchen können. Konkret bedeutet dies, daß das Simulationsmodell eine weitgehende Konfigurierbarkeit der Hardware-Schnittstelle bieten muß, so daß bei hoher Abstraktion die Simulationsgeschwindigkeit gesteigert werden kann. Insofern ist das Ziel Z_3 eng mit dem Ziel Z_2 verbunden. Es muß die Möglichkeit vorhanden sein, Teile der Peripherie auch innerhalb des Simulationsmodells nachzubilden. Den größten Vorteil wird dabei die interne Nachbildung von eigentlich externem Speicher bieten, da viele Prozessoren immer externen Speicher als Programmspeicher benötigen und dieser einfach und leistungsfähig innerhalb des Simulationsmodells zur Verfügung gestellt werden kann.

Das *Ziel* Z_4 fordert hochsprachliche Programmierung sowohl des Simulationsmodells als auch der Programme, die durch das Simulationsmodell ausgeführt werden sollen. Die hochsprachliche Implementation des Simulationsmodells ergibt sich durch die Komplexität der Aufgabe: Bei vertretbarem Aufwand lassen sich die verschiedenen Komponenten des entstehenden Simulationsmodells nur mit den Mitteln einer fortgeschrittenen Hochsprache implementieren. Die in Ziel Z_2 geforderte Geschwindigkeit verbietet dabei den Einsatz einer zu abstrakten und damit langsamen Programmiersprache.

Als Simulationsumgebung kommt wie bereits erwähnt HADES zum Einsatz. Diese Umgebung wurde in der Programmiersprache Java implementiert. Alle Simulationsmodelle, die HADES als Simulationsumgebung nutzen, benutzen normalerweise ebenfalls Java als Implementationssprache, da sonst ein sehr hoher Aufwand für die Programmierung der Schnittstelle zwischen HADES und dem Simulationsmodell getrieben werden muß. Die Forderung nach einer Hochsprache für die Implementation von Programmen, die durch das Simulationsmodell ausgeführt werden sollen, ergibt sich durch die Simulation des Prozessors gemeinsam mit seiner Umgebung. Je komplexer die Umgebung des Prozessors, desto mehr Aufwand muß in die Programmierung des Gesamtsystems gesteckt werden. Dieser Aufwand läßt sich durch den Einsatz einer Hochsprache wesentlich besser strukturieren. Soll ein leistungsfähiger Prozessor direkt über seine Maschinensprache programmiert werden, so ist das im Normalfall ein aufwendiges Unternehmen, da mit steigender Komplexität eines Prozessors auch dessen Software-Schnittstelle an Komplexität gewinnt. Durch die Benutzung einer von der Maschinensprache abstrahierenden Hochsprache und eines entsprechenden Übersetzers kann dieses Problem umgangen werden.

Es muß ein Mechanismus geschaffen werden, mit dem sich hochsprachliche Programme auch für das entstehende Simulationsmodell einfach erstellen und innerhalb des Simulationsmodells benutzen lassen. Es muß ein Übersetzer gefunden und eine Schnittstelle in das Simulationsmodell integriert werden, mit der sich die vom Übersetzer erzeugten Ergebnisse in das Simulationsmodell einlesen lassen. Dabei ist es von Vorteil, wenn sich die gewählte Programmiersprache, deren Übersetzer und auch das Simulationsmodell möglichst einfach benutzen lassen.

Das *Ziel* Z_5 fordert eine Inspektionsschnittstelle. Zur normalen Benutzung, aber auch für spezielle, die Architektur betreffende Untersuchungen, muß eine Inspektionsschnittstelle geboten werden, die dem Benutzer die Interna des Prozessors offenlegt. Diese muß alle innerhalb des Prozessors angebotenen Strukturen und Daten angemessen darstellen. Dadurch soll die Fehlersuche in Maschinenprogrammen vereinfacht werden. Außerdem können so Auswirkungen von Implementationsänderungen sichtbar gemacht werden. Eine besondere Rolle spielt dabei die Darstellung innerhalb des Prozessors vorhandener Komponenten, die für die Befehlsbearbeitung essentiell sind. Der Benutzer sollte jederzeit den genauen internen Zustand ablesen und bewerten können. Moderne Prozessoren enthalten eine Befehls-Pipeline zur Ausführung der Befehle. Existierende Simulationsmodelle für Prozessoren bieten nur selten eine sinnvolle Darstellung der Befehls-Pipeline. Für das hier entstehende Simulationsmodell soll deshalb ein besonderer Wert auf die Darstellung der Pipeline gelegt werden.

Die definierten Ziele Z_1 bis Z_5 sollen zu einem Endprodukt führen, das einige besondere Charakeristika bietet, die so bisher nicht in vergleichbaren Simulationsmodellen zu finden sind. Die in den Vordergrund gestellten Ziele der möglichst detailgetreuen Hardware-Abbildung (Z_1) und der gemeinsamen Simulation von Hard- und Software (Z_2) sind so in dieser Kombination noch nicht realisiert worden. Bei bisher bestehenden Simulationsmodellen wird immer eines dieser beiden Ziele dem anderen vollkommen untergeordnet. Auch eine sich bis zur Pipeline erstreckende Inspektionsschnittstelle (Z_5) ist bisher nur selten anzutreffen.

Das hier entstehende Simulationsmodell soll trotz eines gewissen Gegensatzes die Ziele Z_1 und Z_2 gleichzeitig erfüllen (wobei die detailgetreue Hardware-Abbildung als vorrangig eingestuft wird), so daß eine umfassende Systemsimulation mit allen hier als wichtig vorgestellten Aspekten des zugrunde liegenden Prozessors zur Verfügung gestellt wird.

1.3 Benutzung spezieller Begriffe

Einige hier auftauchende Begriffe besitzen in unterschiedlichen Zusammenhängen ganz verschiedene Bedeutungen. Deshalb ist es wichtig, die in diesem Text benutzten Bedeutungen klarzustellen. Insbesondere die Benutzung von englischsprachigen Fachausdrücken oder deren deutscher Übersetzungen kann zu Problemen führen, weshalb hier mehrheitlich die deutschen Begriffe verwendet werden. Nur bei fest etablierten englischen Begriffen, für die kein allgemein verbreitetes deutsches Gegenstück existiert, wird der englische Ausdruck verwendet.

Begriffe und Abkürzungen, die dem Leser unter Umständen nicht geläufig sein könnten, sind im Anhang G zu finden.

1.4 Aufbau der weiteren Diplomarbeit

Die weiteren Kapitel gliedern sich nach dem Vorgehen.

Das *zweite Kapitel* enthält den allgemeinen Teil der Beschreibung der zugrunde liegenden Hardware. Zuerst wird die zu simulierende Architektur und ein spezieller, diese Architektur implementierender Prozessor ausgewählt. Nach dieser Auswahl kann die Architektur dokumentiert werden.

Im *dritten Kapitel* wird dann speziell der ausgewählte Prozessor mit dessen Besonderheiten in Bezug auf die zugrunde liegende Architektur beschrieben.

Der Inhalt des *vierten Kapitels* stellt die Randbedingungen der Modellierung dar: Hier werden die Simulationsgrundbegriffe geklärt und bereits bestehende Simulationsmodelle mit dem hier entstehenden in Beziehung gesetzt. Außerdem wird HADES zusammen mit seinen Schnittstellen beschrieben.

Aufbauend auf der Beschreibung der Hardware und den Randbedingungen für die Modellierung wird dann im *fünften Kapitel* auf die Modellbildung selbst eingegangen. Hier wird die reale Hardware mit dem entstandenen Simulationsmodell in Bezug gesetzt.

Das *sechste Kapitel* enthält die Beschreibung der für diese Arbeit entwickelten Software-Architektur. Hier werden die Hauptkomponenten des Simulationsmodells vorgestellt und deren Verbindung verdeutlicht.

Im *siebten Kapitel* wird dann auf spezielle Aspekte der Implementation eingegangen. Eine komplette Beschreibung der entstandenen Quelltexte würde den Rahmen dieser Arbeit sprengen (mehr als 120 Java-Klassen mit insgesamt ca. 15.500 Zeilen), deshalb werden hier nur die Klassen beschrieben, die nicht trivial zu implementieren sind und wirklich interessante Mechanismen enthalten oder deren Erklärung zur Vorstellung anderer Klassen notwendig ist.

Die Benutzung sowohl des Simulationsmodells als auch der Hilfsmittel zur Erstellung von hochsprachlichen Programmen für das Simulationsmodell ist Inhalt des *achten Kapitels*. Einige Anwendungsbeispiele runden dieses Kapitel ab.

Ein Fazit zieht das *neunte Kapitel*: Hier wird das Erreichte zusammengefaßt und ein Ausblick auf weitere Möglichkeiten geboten.

Die *Anhänge* enthalten verschiedene Details wie die Beschreibung der Maschinensprache der zugrunde liegenden Architektur, die Erstellung eines Crosscompilers und die Verzeichnisstruktur der entstandenen Software.

2 Auswahl und Beschreibung der Prozessorarchitektur

In diesem Kapitel werden zuerst die zu simulierende Prozessorarchitektur und auch der spezielle, diese Architektur implementierende Prozessor, der nachgebildet werden soll, ausgewählt. Daran schließt sich die genaue Beschreibung der Architektur an. Die Dokumentation des speziellen Prozessors folgt im nächsten Kapitel.

2.1 Auswahl eines geeigneten Prozessors

Teil dieser Diplomarbeit ist vor der eigentlichen Implementation eines Simulationsmodells die Auswahl sowohl einer zur Simulation geeigneten Prozessorarchitektur als auch des konkreten Prozessors, dessen Eigenschaften nachgebildet werden sollen.

2.1.1 Ziele

Vor der Vorstellung der verschiedenen infrage kommenden Architekturen werden zuerst einmal die Ziele für die Architektur- und Prozessorwahl definiert.

- Ein wichtiges Ziel ist die Kompatibilität zu weit verbreiteter Hardware. Je öfter eine Architektur eingesetzt wird, desto mehr Sinn macht die Simulation eines Prozessors, der diese Architektur implementiert.

- Es soll ein 32-Bit Prozessor mit Befehls-Pipeline modelliert werden. Erst Prozessoren dieser Leistungsklasse bieten ausreichende Leistungsfähigkeit für allgemeine Einsetzbarkeit und lassen sich andererseits noch mit vertretbarem Aufwand simulieren.

- Ein möglichst einfacher Aufbau des gewählten Prozessors sorgt für eine leistungsfähige Simulation mit einem nicht zu großen Implementationsaufwand. Zum Beispiel kann im Rahmen von eingebetteten Systemen oft auf komplexe Komponenten wie eine Fließkommaeinheit verzichtet werden.

- Die verfügbare Dokumentation spielt eine wesentliche Rolle. Nur durch eine ausführliche und vollständige Beschreibung des Verhaltens kann ein qualitativ hochwertiges Simulationsmodell erstellt werden.

- Der zu simulierende Prozessor sollte sich durch einen günstigen Preis auszeichnen. Auch dessen Dokumentation und Entwicklungsumgebungen sollten billig in der Anschaffung sein.

- Die Programmierung der Software, die von dem Simulationsmodell ausgeführt werden soll, soll hauptsächlich in einer Hochsprache erfolgen.

- Für den gewählten Prozessor sollten sinnvoll nutzbare Software-Entwicklungsumgebungen existieren. Dabei ist es natürlich von Vorteil, wenn diese nicht nur auf ein Betriebssystem festgelegt sind.

2.1.2 Folgerungen

Da das als Ergebnis vorliegende Simulationsmodell des auszuwählenden Prozessors hauptsächlich Programme ausführen soll, die in einer Hochsprache entstanden sind, ist der Aufbau der Maschinensprache von untergeordneter Wichtigkeit. Wesentlich entscheidender ist das Ergebnis, wie schnell hochsprachlich formulierte Algorithmen innerhalb der Simulation ausgeführt werden können.

Es wird ein möglichst einfacher Aufbau für eine möglichst leistungsfähige Simulation gefordert. Der Prozessor selbst darf dabei nur wenig Peripherie enthalten, da der begrenzte Zeitrahmen und der Aufwand zur Implementation der Peripherie sonst ein Problem darstellen. Unverzichtbar sind natürlich die Speicherschnittstelle und einfach nutzbare Interrupts, während andere Komponenten wie serielle (oder komplexere) Schnittstellen oder gar eine komplette PCI-Bus-Schnittstelle nur für spezielle Zwecke benötigt werden.

Daraus ergibt sich letztendlich, daß der geforderte Baustein eine Mischung zwischen Mikroprozessor und Mikrocontroller darstellt: Ein reiner Mikroprozessor würde nur den Kern und keinerlei Peripherie enthalten; ein Mikrocontroller dagegen liefert meist komplexe Peripherie auf demselben Baustein mit. Da die Grenzen zwischen beiden Begriffen fließend sind, müssen beide Möglichkeiten in Betracht gezogen werden. Zu vielen Mikroprozessoren werden auch kompatible Mikrocontroller angeboten.

2.1.3 Die Kandidaten

Als weit verbreitete Prozessoren bieten sich verschiedene an. Zuerst wird ein Überblick über existierende Architektur-Familien und deren Charakteristika bezogen auf die oben definierten Ziele geboten. Die genannten Merkmale der verschiedenen Familien sind entweder [Tanenbaum 90] oder entsprechenden einzeln genannten Dokumentationen entnommen.

Intel x86 Die in Personal Computern am weitesten verbreitete Prozessorfamilie ist die Intel-x86-Familie. Für diese Prozessoren stehen eine große Anzahl von Betriebssystemen und Entwicklungsumgebungen zur Verfügung. Aufgrund der Anforderung, daß mindestens 32-Bit Arithmetik zur Verfügung stehen soll, müßte ein Modell ab dem 80386 gewählt werden. Leider sind diese Prozessoren jedoch sehr kompliziert aufgebaut. Die Entstehungsgeschichte dieser Prozessorfamilie hat dazu geführt, daß mit jedem neuen Modell aus Kompatibilitätsgründen an veralteten Konzepten festgehalten wurde und trotzdem neue Funktionen eingebaut wurden, die dann entsprechend kompliziert zu benutzen sind. Dieses Manko spricht stark gegen die Möglichkeit, einen Vertreter dieser Prozessorfamilie mit vertretbarem Aufwand und ausreichender Geschwindigkeit simulieren zu können.

Motorola 680x0 Diese Prozessoren sind heute noch weit verbreitet (hauptsächlich als Mikrocontroller) und es existieren geeignete Betriebssysteme und Entwicklungsumgebungen. Auch diese Prozessorfamilie besitzt wie die Intel-x86-Prozessoren eine zu große Komplexität für sinnvolle Simulierbarkeit. Die vielen verschiedenen Adressierungsmodi, die unterschiedliche Länge der Befehlsworte und die Befehlsvielfalt machen die Prozessoren der 680x0er Familie zu typischen CISC-Prozessoren.

Während gute Simulationsmodelle für den einfachen 68000-Prozessor existieren [Palm 99], wäre die Simulation der komplexeren Modelle dieser Prozessorfamilie (68030, 68040) wesentlich aufwendiger.

Motorola PowerPC Als Nachfolger der 680x0er Familie wurde von Apple, IBM und Motorola die PowerPC-Familie eingeführt. Diese Prozessoren sind allerdings nicht so weit verbreitet und besitzen, obwohl sie von den Herstellern als RISC-Prozessoren bezeichnet werden, noch eine relativ hohe Komplexität des Befehlssatzes und daraus folgend des internen Aufbaus.

Intel 960 Die Prozessoren dieser Architekturfamilie sind von Intel entwickelt worden. Hier spricht die geringe Verbreitung dieser Prozessoren gegen die Verwendung dieser Architektur als Simulationsgrundlage. Außerdem enthalten die heute verfügbaren Ausführungen zu viel Peripherie, um ein schlankes und schnelles Simulationsmodell zu gewährleisten [i960 97].

Sun SPARC Die SPARC-Architektur [SPARC 92] ist ein aussichtsreicher Kandidat als Grundlage für ein Simulationsmodell. Diese Prozessoren sind weit verbreitet und gut dokumentiert. Sie werden von verschiedenen Herstellern in vielen Modellvarianten hergestellt. Der interne Aufbau der einfacheren SPARC-Prozessoren ist bis auf einige Details gut zu überblicken und die Software-Schnittstelle, ist klar strukturiert und einfach simulierbar. Viele Betriebssysteme und Entwicklungsumgebungen sind für SPARC verfügbar. Die SPARC-Architektur ist eine der ersten reinen RISC-Architekturen und diente einigen anderen, später entstandenen Architekturen als Vorbild. Als konkrete Implementation dieser Architektur wird der SUN microSPARC IIep in [microSPARC 97] dokumentiert.

MIPS Die MIPS-Architektur ist gleichzeitig mit der SPARC-Architektur entstanden und prägte den RISC-Begriff mit. Sie besitzt dementsprechend einen vergleichbar einfachen Aufbau wie die SPARC-Architektur. Die allgemeine Architekturbeschreibung ist in [Kane 92] zu finden. Heute sind Prozessoren der MIPS-Architektur noch wesentlich weiter verbreitet als SPARC-Prozessoren und werden hauptsächlich in kleinen tragbaren Computern, Laserdruckern, Spielekonsolen und Industrieanwendungen (als eingebettete Prozessoren) eingesetzt. Dementsprechend sind viele verschiedene Ausführungen von unterschiedlichen Herstellern zu sehr günstigen Preisen verfügbar. Auch die Auswahl der Betriebssysteme und der Entwicklungsumgebungen für MIPS-Prozessoren ist sehr gut.

Weitere Andere Prozessoren wurden nicht in die engere Wahl gezogen, da die bisher dokumentierten die am weitesten verbreiteten Modelle darstellen und zu diesen gute Dokumentation im WWW verfügbar ist. Kandidaten wären noch ARM-Prozessoren oder Prozessorkerne, die innerhalb von Komponentenbibliotheken als Teilelemente von Custom-Chip-Designs benutzt werden können. Allerdings sind diese Prozessoren zu wenig verbreitet und zu speziell, so daß hier nicht weiter auf diese eingegangen werden muß.

2.1.4 Ergebnis der Architektur-Wahl: MIPS

Aufgrund des einfachen Aufbaus und dank sehr guter Dokumentation (siehe zum Beispiel auch [Hennessy 96], [Patterson 98], [Sweetman 99]) wurde MIPS als zu simulierende Architektur gewählt. Als eine der ersten echten RISC-Architekturen ist der interne Aufbau einfach und schnell simulierbar. Allerdings muß zur MIPS-Architektur noch erwähnt werden, daß verschiedene Stufen der Architektur existieren. Die ursprünglich definierte Architektur entspricht heute der MIPS ISA 1 (ISA steht für *Instruction Set Architecture*). Weitere definierte Stufen führen zusätzliche Befehle und Worte mit 64 Bit Breite ein. Insofern eignet sich aufgrund der genannten Anforderungen und der geringen Komplexität nur die MIPS ISA 1 für diese Arbeit. Ein Überblick über die verschiedenen ISA-Stufen ist zum Beispiel in [IDT 96] zu finden.

Im Folgenden wird statt von der MIPS ISA 1 immer von der MIPS-R3000-Architektur gesprochen. Dieser Begriff steht direkt für die Implementation eines verbreiteten Bausteins, der der MIPS ISA 1 entspricht und zusätzlich einige, durch die reine Architekturspezifikation nicht abgedeckte Interna enthält.

2.1.5 Ergebnis der Prozessor-Wahl: IDT R3051

Es gibt verschiedene Firmen, die MIPS-kompatible Prozessoren anbieten. Die bekanntesten Vertreter sind Philips, Toshiba, Siemens/Infineon, LSI Logic, NEC und IDT (siehe Übersicht in [MIPS 99]). Die einzelnen Prozessoren unterscheiden sich in der Geschwindigkeit, der internen Ausstattung, dem Stromverbrauch und der implementierten MIPS ISA.

Für diese Arbeit fiel die Wahl des Herstellers auf IDT, da dieser ein breites Produktspektrum von MIPS-kompatiblen Prozessoren anbietet, diese Prozessoren für den Hersteller einen hohen Stellenwert besitzen und ein Modell besonders für die Simulation geeignet ist: Der IDT R3051. Dieser Baustein ist Software-kompatibel zum MIPS R3000 und bietet eine einfache Speicherschnittstelle und statische Adreßumsetzung (statt einer dynamischen Speicherverwaltungseinheit *Memory Managing Unit*, *MMU*) zu einem günstigen Preis. Anders ausgestattete Modelle mit MMU oder Fließkommaeinheit *Floating Point Unit*, *FPU*, und unterschiedlichen Cache-Größen aber mit gleicher Ein- und Ausgangsbelegung sind ebenfalls verfügbar, so daß im Bedarfsfall mit vertretbarem Aufwand auch diese Features unterstützt werden können. Zu dem IDT R3051 [IDT 94] ist wie auch zu allen anderen IDT-Prozessoren [IDT 95] gute Dokumentation direkt im WWW verfügbar, so daß diese Dokumentation zusammen mit [Kane 92] als Grundlage der Hardware-Beschreibung dient.

2.2 Aufbau von MIPS-R3000-kompatiblen Prozessoren

Wie schon in der Gegenüberstellung der zur Auswahl stehenden Prozessoren beschrieben, bietet die MIPS-R3000-Architektur einen klassischen RISC-Aufbau. 32 Register stehen einem verhältnismäßig kleinen, auf compilergenerierte Maschinenspracheprogramme hin optimierten Befehlssatz zur Verfügung. Die Befehlsabarbeitung ist in kleine Schritte aufgeteilt, die durch eine Pipeline parallelisiert werden können. Eine einheitliche Wortbreite von 32 Bit sowohl für Daten- als auch für Befehlsworte vereinfacht den internen Aufbau und die Datenpfade. Die Kodierung der Befehle ist sehr einfach und einheitlich. Datenwerte können zwar auch Viertel- (acht Bit) oder Halbworte (sechzehn Bit) umfassen, diese Datentypen bieten aber gegenüber den 32-Bit Worten keine Geschwindigkeitsvorteile und werden nur durch Lade- und Speicherbefehle unterstützt.

Der Hauptspeicherinhalt wird trotz der generellen Wortbreite von 32 Bit auf Byte-Ebene adressiert, um auch auf Teilworte sinnvoll zugreifen und bestehende Software für andere Architekturen einfacher portieren zu können. Dadurch und durch die Wortbreite von 32 Bit ergibt sich ein Adreßraum von 2^{32} Byte (vier GByte). Die MIPS-R3000-Architektur besitzt eine Registerbank mit 32 Registern. Register R0 ist ein Konstanten-Register und enthält immer den Wert Null, alle anderen 31 Register sind beliebig für variable Inhalte nutzbar. Dazu kommen noch zwei Ergebnisregister für Multiplikations- und Divisionsbefehle. Trotz der freien Verwendbarkeit sind die normalen Register in ihrer Benutzung bestimmten Konventionen unterworfen. So gibt es einige Temporärregister, andere Register dienen der Parameterübergabe zwischen Unterprogrammen und weitere Register halten globale Variablen.

Die Schnittstelle zur Maschinensprache bietet drei verschiedene Befehlsformate, deren Aufbau möglichst einheitlich ist. Diese werden in Abschnitt 2.3 ausführlicher beschrieben. Der Aufbau der Pipeline der MIPS-R3000-Architektur gliedert sich grob in fünf Stufen, so daß bis zu fünf Befehle gleichzeitig bearbeitet werden können. Die genaue Struktur der Pipeline wird in Abschnitt 2.4 beschrieben. Die Benutzung einer Pipeline führt allerdings zu den bekannten Problemen bei der Ausführung einiger voneinander abhängiger direkt aufeinanderfolgender Befehle. Der Abschnitt 2.5 beschreibt diese als Hazards bezeichneten Effekte.

In der MIPS-R3000-Architektur sind für die Beschleunigung der Befehlsbearbeitung ein Befehls- und ein Daten-Cache vorgesehen. Durch den getrennten Zugriff auf Befehle und Daten ergibt sich eine Harvard-Architektur. Die Caches müssen sich nicht innerhalb des Prozessors selbst befinden, sondern können auch extern realisiert sein. Den Caches ist im Folgenden der Abschnitt 2.6 gewidmet. Für eine Erweiterung des Funktionsumfangs stellt die MIPS-R3000-Architektur das Konzept der Koprozessoren zur Verfügung. Durch die Implementation eines Koprozessors kann ein Anbieter eines MIPS-R3000-kompatiblen Bausteins den Befehlssatz um eigene, eventuell herstellerspezifische Charakteristika erweitern, ohne die Regeln der MIPS-R3000-Architektur zu verletzen. Die Koprozessoren werden in Abschnitt 2.7 beschrieben.

2.3 Befehlssatz

2.3.1 Befehlsformate

Die MIPS-R3000-Architektur besitzt drei verschiedene Befehlsformate. Das sogenannte *Immediate*-Befehlsformat kann direkt einen konstanten Parameter und zwei Registernummern aufnehmen, so daß hiermit einerseits Lese- und Schreibbefehle und andererseits arithmetische Befehle mit einem konstanten Operanden kodiert werden können. Das *Jump*-Befehlsformat kann eine konstante Zieladresse aufnehmen und wird für alle Verzweigungsbefehle verwendet. Das dritte Befehlsformat, das *Register*-Format, kommt bei allen Befehlen zum Einsatz, die bis zu drei verschiedene Register benutzen. Die Befehlsformate sind in Abbildung 1 dargestellt. Die in der Abbildung verwendeten Begriffe oder Abkürzungen werden in Tabelle 1 erklärt.

Immediate	Opcode	Source	Target	Immediate		
Jump	Opcode	Jump-Target				
Register	Opcode	Source	Target	Dest.	Shift	Function
	31 26	25 21	20 16	15 11	10 6	5 0

Abbildung 1: Alle Befehlsformate der MIPS-R3000-Architektur

Opcode	Operation code, identifiziert den Befehl (6 Bit)
Source	Source Register, liefert Rechenoperanden (5 Bit)
Target	Target Register, Rechenoperand oder -ergebnis (5 Bit)
Immediate	Immediate Value, Konstanter Wert oder Adressendifferenz (16 Bit)
Jump-Target	Sprungzieladresse (26 Bit)
Dest.	Destination Register, Rechenergebnis (5 Bit)
Shift	Shift amount, so viele Bit wird geschoben (5 Bit)
Function	Function, Erweiterung des Opcode (6 Bit)

Tabelle 1: Abkürzungserklärungen für die Befehlsformate

Alle normalen MIPS-Befehle folgen den drei Befehlsformaten. Es gibt noch einige Ausnahmen, die in den folgenden Abschnitten erwähnt werden. Da für Registerangaben in den Befehlen immer fünf Bit vorgesehen sind, können alle 32 normalen Register der MIPS-R3000-Architektur direkt adressiert werden.

2.3.2 Maschinensprache im Gegensatz zur Assembler-Sprache

Der MIPS-R3000-Befehlsssatz ist zwar sehr einfach gehalten, erfordert aber die Berücksichtigung der Pipeline-Hazards (siehe Abschnitt 2.5), so daß eine direkte Programmierung in Maschinensprache ohne Benutzung eines helfenden Assemblers eher kompliziert ist.

Deshalb wird bei der Benutzung der MIPS-R3000-Architektur wie auch bei einigen anderen RISC-Architekturen zwischen der *Assembler-* und der *Maschinensprache* unterschieden. Die Maschinensprache ist direkt der Befehlssatz, den ein die Architektur implementierender Prozessor ausführen kann. Im Gegensatz dazu ist die Assembler-Sprache eine Obermenge von Befehlen, die durch einen Übersetzer in Maschinensprache umgesetzt wird. Innerhalb der Assembler-Sprache können Makros definiert sein, die dann in mehrere Maschinensprachebefehle überführt werden. Auch können bei der Übersetzung von Assembler-Befehlen in Maschinenbefehle Optimierungen vorgenommen werden.

Ein Beispiel für ein Makro ist der `nop`-Befehl (*No Operation*), dessen Ausführung zu keiner Änderung des Prozessorzustands führt (bis auf den erhöhten Programmzähler). Dieser Befehl wird gezielt für Verzögerungen eingesetzt (siehe Abschnitt 2.5). und existiert nicht in der MIPS-Maschinensprache. Es gibt mehrere Befehle, die sich wie gerade beschrieben verhalten (zum Beispiel eine Addition mit Null). Der `nop`-

Befehl der Assembler-Sprache kann also in einen Maschinensprache-`add`-Befehl übersetzt werden. Ein anderes Beispiel für ein Makro ist der `li`-Befehl (*Load Immediate*). `li` soll eine 32-Bit Konstante in ein Register laden. Da aufgrund der Wortbreite der Befehlsworte von 32 Bit ein derartiger Befehl als Maschinenbefehl nicht möglich ist, wird dieser Assembler-Befehl durch zwei Maschinenbefehle nachgebildet: Ein `lui` (*Load Upper Immediate*) füllt die oberen sechzehn Bit, danach werden mit `ori` (*Or Immediate*) die unteren sechzehn Bit hinzugefügt.

2.3.3 Adressierungsmodi

Als einziger Adressierungsmodus steht Register-indirekt mit Offset zur Verfügung. Für jeden Speicherzugriff wird die Zieladresse aus einem Register geholt und dazu wird ein konstanter Wert addiert. Aus diesem einen Befehlsformat lassen sich allerdings auch zwei weitere Formate ableiten: Register-indirekt ohne Offset (wenn der Offset auf Null gesetzt wird) und direkt (wenn das Register 0 gewählt wird, dann können allerdings nur 64 KByte adressiert werden). Weitere Adressierungsarten können bei Bedarf durch mehrere Maschinenbefehle nachgebildet werden.

2.3.4 Befehlsgruppen

Es gibt fünf verschiedene Befehlsgruppen:

Load/Store Diese Lade- und Speicherbefehle bewegen die Daten zwischen dem Speicher (oder dem Daten-Cache) und den Registern. Alle diese Befehle entsprechen dem *Immediate*-Befehlsformat.

ALU ALU steht für *Arithmetical Logical Unit*. Die ALU-Befehle berechnen arithmetische, logische und Shift-Operationen. Die Operanden sind entweder Register (*Register*-Befehlsformat) oder Register und eine Konstante (*Immediate*-Befehlsformat). Die MIPS-Architektur implementiert eine Drei-Adreß-Maschine. In jedem ALU-Befehl werden sowohl die beiden Operanden als auch das Ergebnisregister getrennt voneinander angegeben, so daß das Ergebnis keinen Operanden überschreiben muß.

Sprung/Verzweigung Die Sprungbefehle benutzen entweder das *Jump*-Format und enthalten das Sprungziel in Form einer absoluten Adresse oder das *Register*-Format, wenn indirekt zu einer in einem Register gespeicherten Adresse gesprungen werden soll. Verzweigungsbefehle werden im *Immediate*-Format angegeben, wobei der *Immediate*-Wert die Zieladresse relativ zum Programmzähler angibt. Für Unterprogrammaufrufe existieren Sprungbefehle, die vor dem eigentlichen Sprung den aktuellen Programmzähler in einem Register hinterlegen. So führt dann ein indirekter Sprung über dieses Register am Ende des Unterprogramms wieder an den Aufrufort zurück.

Koprozessor Für die Benutzung der Koprozessoren (zumindest der Koprozessor 0 ist immer implementiert) steht eine Reihe von Befehlen zur Verfügung. Einige dieser Befehle sind genormt und entsprechen dem *Immediate*-Format, andere Befehle sind nur für spezielle Koprozessoren benutzbar und in einem eigenen Format kodiert.

Spezial Als Spezialbefehle werden alle die Befehle bezeichnet, die in keine der bisherigen Befehlsgruppen passen. So gibt es Befehle für den Zugriff auf die Spezialregister oder Systemaufrufbefehle. Diese Befehle verwenden das *Register*-Format.

Die einzelnen Befehle der MIPS-R3000-Architektur werden im Anhang A in tabellarischer Form vorgestellt. Ausführlich werden diese Befehle und deren Benutzung in [IDT 96] beschrieben.

2.3.5 Asynchrone Befehle

Die MIPS-R3000-Architektur besitzt einige Befehle, die nicht synchron ausgeführt werden. Diese Befehle werden zwar innerhalb der Pipeline-Ausführung gestartet, laufen dann aber parallel zur Ausführung der normalen Befehle ab, bis deren Ergebnis vorliegt. Für diese Befehle sind Spezialregister definiert, die das Ergebnis aufnehmen.

Es existieren insgesamt vier asynchrone Befehle: Zwei Multiplikations- und zwei Divisionsbefehle. Der hinter diesen Instruktionen steckende Berechnungsaufwand läßt es nicht zu, sie ohne erheblichen Aufwand wie alle anderen Befehle synchron auszuführen. Deshalb sind diese Befehle asynchron realisiert. Alle diese Befehle benutzen die beiden Spezialregister `HI` und `LO` als Ergebnisregister. Bei Multiplikationen werden in `HI` die oberen 32 und in `LO` die unteren 32 Bit des Ergebnisses hinterlegt. Divisionen bieten in `LO` das ganzzahlige Divisionsergebnis an, während `HI` den Divisionsrest zugewiesen bekommt.

Für den Zugriff auf die beiden Spezialregister `HI` und `LO` existieren zusätzliche (synchrone) Befehle, die für einen Austausch zwischen den Spezialregistern und den normalen Registern sorgen. Dabei sorgt ein Mechanismus dafür, daß die Pipeline angehalten wird, wenn ein Spezialregister vor dem Ende einer Multiplikation oder Division abgefragt werden soll. Je nach Implementation der MIPS-R3000-Architektur kann die Berechnung der asynchronen Befehle unterschiedlich lang dauern. Die Architektur schreibt keine bestimmten Ausführungszeiten vor.

2.3.6 Registerkonventionen

Die normalen Register der MIPS-R3000-Architektur sind zwar keinen Beschränkungen unterworfen, jedoch existieren bestimmte, zusammen mit der Architektur definierte Konventionen, die festlegen, welche Register für welche Zwecke verwendet werden dürfen. Insbesondere für die Zusammenarbeit von Programmteilen verschiedener Programmiersprachen ist diese Definition der Register eine zwingende Voraussetzung für ein reibungsloses Zusammenspiel der unterschiedlichen Komponenten.

In Tabelle 2 wird die Zuordnung der einzelnen Register genau aufgeschlüsselt. Für die einfachere Benutzung der Register erhalten sie auch symbolische Namen, die innerhalb von Assembler-Programmen statt der Registernummern benutzt werden können.

2.4 Pipeline

Die Befehlsbearbeitung des MIPS-R3000 wird durch eine Pipeline vorgenommen. Die Bearbeitung eines Befehls ist in mehrere Schritte aufgeteilt, die versetzt für verschie-

Register	Name	Bedeutung
0	zero	Liefert immer eine 0, schreiben wird ignoriert
1	at	Assembler-Temporärregister, für einige Makros
2 - 3	v0 - v1	Rückgabewerte von Unterprogrammaufrufen
4 - 7	a0 - a3	Parameter für Unterprogrammaufrufe
8 - 15	t0 - t7	Temporärregister für Unterprogramme
16 - 23	s0 - s7	Registervariablen, müssen vor Benutzung gesichert werden
24 - 25	t8 - t9	Temporärregister für Unterprogramme
26 - 27	k0 - k1	Reserviert für die Ausnahmebehandlung
28	gp	*Global Pointer*: Zeiger auf globale Variablen
29	sp	*Stack Pointer*: Zeiger auf den Stapelspeicher
30	s8 / fp	Neunte Registervariable oder *Frame Pointer*
31	ra	Rücksprungadresse für Unterprogramme

Tabelle 2: Registerkonventionen der MIPS-R3000-Architektur

dene Befehle gleichzeitig ausgeführt werden. Durch die Benutzung einer Pipeline ergeben sich einige Besonderheiten im Gegensatz zu herkömmlichen Prozessoren ohne Pipeline: Es kann vorkommen, daß sich die Ausführung eines Befehls direkt auf die folgenden, auch schon in der Pipeline-Ausführung befindlichen Befehle auswirkt. Diese Fälle werden als Pipeline-Hazards bezeichnet und müssen speziell behandelt und deshalb ausführlich im Abschnitt 2.5 beschrieben werden.

Der grobe Aufbau der Pipeline für einen Befehl wird in Tabelle 3 dargestellt. Dabei werden in der linken Spalte der Tabelle die fünf verschiedenen Pipeline-Schritte aufgezählt. Fast jeder einzelne Pipeline-Schritt ist noch einmal in zwei nacheinander stattfindende und in der rechten Spalte dargestellte Aktionen aufgeteilt.

Pipeline-Schritt	Stattfindende Aktionen
Instruction Fetch	- Virtuelle Instruktionsadresse in physikalische wandeln - Speicherzugriff beginnen
Read and Decode	- Auf Leseergebnis warten - Gelesenen Wert dekodieren - Sprungziel feststellen
ALU Operation	- Berechnung des ALU-Ergebnisses - Virtuelle Datenadresse in physikalische wandeln
Memory Access	- Speicherzugriff beginnen - Auf Leseergebnis warten
Write Back	- Gelesenen oder berechneten Wert ins Zielregister übernehmen

Tabelle 3: Grober Aufbau der einzelnen Schritte der Befehls-Pipeline

Die Ausführung mehrerer Befehle unter Beachtung der verschiedenen Pipeline-Schritte folgt dann in Tabelle 4. In dieser Tabelle stellt jede Zeile die Bearbeitung eines Pipeline-Schrittes dar. Zu jedem Zeitpunkt werden bis zu fünf Befehle gleichzeitig bearbeitet.

Befehl 1	Befehl 2	Befehl 3	Befehl 4	Befehl 5
Instruction Fetch				
Read and Decode	Instruction Fetch			
ALU Operation	Read and Decode	Instruction Fetch		
Memory Access	ALU Operation	Read and Decode	Instruction Fetch	
Write Back	Memory Access	ALU Operation	Read and Decode	Instruction Fetch
	Write Back	Memory Access	ALU Operation	Read and Decode
		Write Back	Memory Access	ALU Operation
			Write Back	Memory Access
				Write Back

Tabelle 4: Abfolge mehrerer Befehle in der Pipeline

Jeder einzelne gerade beschriebene Schritt der Pipeline ist innerhalb der Hardware noch einmal in zwei aufeinanderfolgende Aktionen (Takte $\Phi 1$ und $\Phi 2$) aufgeteilt. Diese Verfeinerung wird in Abbildung 2 dargestellt und entspricht bis auf die für den IDT R3051 uninteressante Benutzung der MMU der Beschreibung in [Hansen 86] und [Riordan 88].

2.5 Hazards

Normalerweise kann sich ein Programmierer, der Maschinensprache benutzt, darauf verlassen, daß jeder einzelne Befehl in sich abgeschlossen ist und sein Ergebnis dem nächsten Befehl zur Verfügung steht. Durch die Benutzung einer Befehls-Pipeline kann diese Erwartung nicht eingehalten werden. Beim Entwurf der MIPS-Architektur wurde dieser Punkt von Beginn an berücksichtigt. Die Abkürzung “MIPS“ zeigt dieses, sie steht für “Microprocessor without interlocked pipeline stages“.

2.5.1 Write-Back-Hazards

Ist zum Beispiel der erste Befehl ein Additionsbefehl, so wird dessen Ergebnis zwar im dritten Schritt der Pipeline berechnet, aber erst im fünften Schritt wieder ins Ergebnisregister geschrieben. Somit dürfte der darauf folgende Befehl eigentlich nicht das

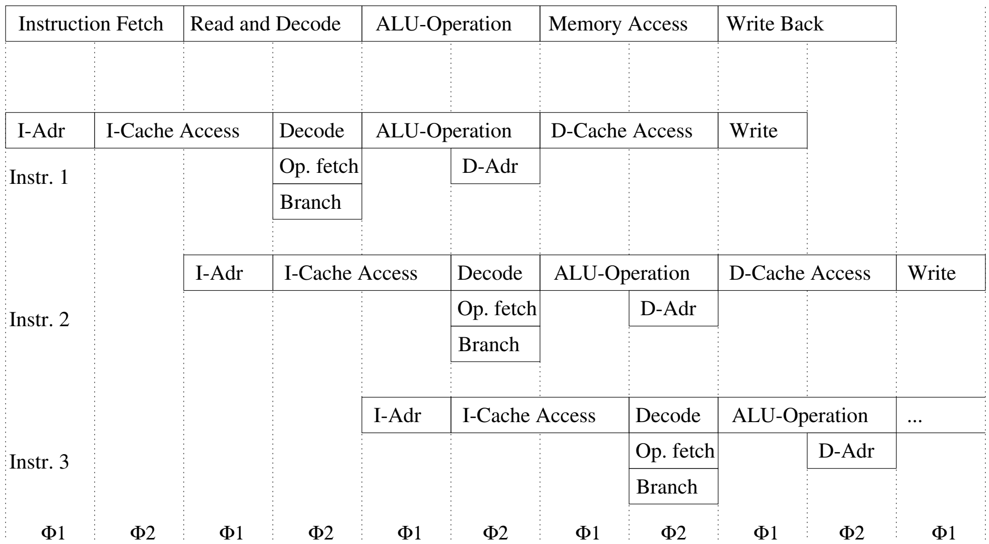

Abbildung 2: Verfeinerte Abfolge mehrerer Befehle in der Pipeline

Ergebnis der Addition für weitere Berechnungen benutzen, da dessen ALU-Operation das Ergebnis zu einem Zeitpunkt benötigt, zu dem dieses noch nicht zurückgeschrieben wurde. Aus diesem Grunde enthält die Pipeline einige Modifikationen, damit das Ergebnis einer ALU-Operation direkt der nachfolgenden ALU-Operation des nächsten Befehls zur Verfügung steht. An dieser Stelle wird ein Pipeline-Hazard durch intelligente Ergebnisweitergabe vermieden.

2.5.2 Lese-Hazards

Anders sieht es bei Lesebefehlen aus. Diese Befehle transportieren ein Datenwort aus dem Speicher (oder Daten-Cache) in den Prozessor. Da das Zeitverhalten des Speichers im Verhältnis zum Prozessorkern relativ langsam ist, steht das Ergebnis erst während des fünften Pipeline-Schrittes zur Verfügung. Das bedeutet, daß sich der direkt nachfolgende Befehl zu diesem Zeitpunkt schon im vierten Pipeline-Schritt befindet und bereits seine ALU-Operation durchgeführt hat, deshalb also nicht auf das Ergebnis des direkt vorausgehenden Lesebefehls Rücksicht nehmen kann.

Die einzige Möglichkeit, dieses Problem zu umgehen, bestände darin, den Pipeline-Ablauf so zu beeinflussen, daß direkt nach der Ausführung eines Lesebefehls kein ALU-Befehl zugelassen wird, der auf das Ergebnisregister des Lesebefehls zugreift. Dieses Verhalten ist auch in der MIPS-R3000-Architektur vorgesehen, allerdings nicht als Hardware-Verhalten, sondern als bei der Übersetzung von Software für diese Architektur notwendiges Software-Verhalten.

Zusammenfassend kann gesagt werden, daß der Lese-Hazard nicht durch den Prozessor selbst verborgen wird. Deshalb muß damit ausgeführte Software darauf Rücksicht nehmen und dieses Verhalten muß von den Übersetzungsprogrammen vor dem Programmierer verborgen werden.

2.5.3 Sprung-Hazards

Sprungbefehle führen zu einem ähnlichen Problem: Das Sprungziel wird berechnet, während der nachfolgende Befehl schon aus dem Speicher gelesen wird. Das Ergebnis der Sprungzielberechnung steht erst vor dem Lesen des übernächsten Befehls zur Verfügung. Erst zu diesem Zeitpunkt kann der Programmzähler auf den neuen Wert umgesetzt und damit der Sprung durchgeführt werden. Auch hier müßte die Ausführung des direkt auf den Sprungbefehl folgenden Befehls vermieden werden, so daß dann erst der übernächste Befehl in der Pipeline, der dann von der richtigen (der Sprungziel-) Adresse geladen wird, zur Ausführung kommt. Änlich wie bei den Lesebefehlen wird dieses Verhalten allerdings der Software (bzw. den Übersetzungsprogrammen) überlassen: Der auf einen Sprungbefehl direkt folgende Befehl wird immer ausgeführt, erst dessen Nachfolger ist der Befehl des Sprungziels.

2.5.4 Auswirkungen

Zu den Hazards muß gesagt werden, daß das Verhalten der MIPS-R3000-Architektur im Vergleich zu anderen Prozessorarchitekturen einzigartig ist: Normalerweise werden Seiteneffekte, die aufgrund einer Pipeline auftreten, vor dem Benutzer der Maschinenspracheschnittstelle verborgen. Nur wenige Architekturen besitzen eine nicht Hazard-freie Schnittstelle, so daß entweder der Programmierer selbst oder die Übersetzungsprogramme zur Erstellung der Maschinensprache auf diese Hazards Rücksicht nehmen müssen.

Dieser Effekt führt auch dazu, daß Programme in MIPS-Maschinensprache schwerer zu lesen sind als normale Maschinensprache anderer Prozessoren. Der Leser muß sich unter anderem immer vor Augen führen, daß der direkt auf einen Sprungbefehl folgende Befehl noch gelesen und ausgeführt wird, bevor der Sprungbefehl ein Umsetzen des Programmzählers bewirkt. Diese Restriktion führt zusammen mit den Lese-Befehl-Hazards zu einem vermehrten Auftreten von `nop`-artigen Befehlen. Solange allerdings nicht direkt die Maschinensprache, sondern die für die MIPS-R3000-Architektur definierte Assembler-Sprache benutzt wird, werden die Hazard-Probleme vollständig vor dem Programmierer verborgen. Entsprechendes gilt selbstverständlich auch für hochsprachliche Programmierung der MIPS-R3000-Architektur.

Zur Verdeutlichung der verschiedenen Hazard-Varianten wird nun noch einmal die verfeinerte Pipeline-Darstellung aus Abschnitt 2.4 aufgegriffen und um die Hazards erweitert. Pfeile stellen die Abhängigkeiten der aufeinander folgenden Befehle dar. Abbildung 3 zeigt das Ergebnis.

2.5.5 Beispiel

Um die Hazards zu verdeutlichen, schließt sich nun ein Maschinensprache-Beispiel an, das die Problematik zeigt und Optimierungsmöglichkeiten darstellt. Tabelle 5 zeigt die Befehle, die ein Assembler-Programmierer auf einem MIPS-R3000-Prozessor ausführen lassen will.

Diese Befehlsabfolge lädt ein Wort aus dem Speicher in das Register $2, addiert eine Konstante hinzu und springt dann zu einem hier nicht genannten Ort. An diese Abfolge schließt sich ein Befehl an, der das Register $2 löscht und in diesem Zusammenhang nicht ausgeführt werden soll.

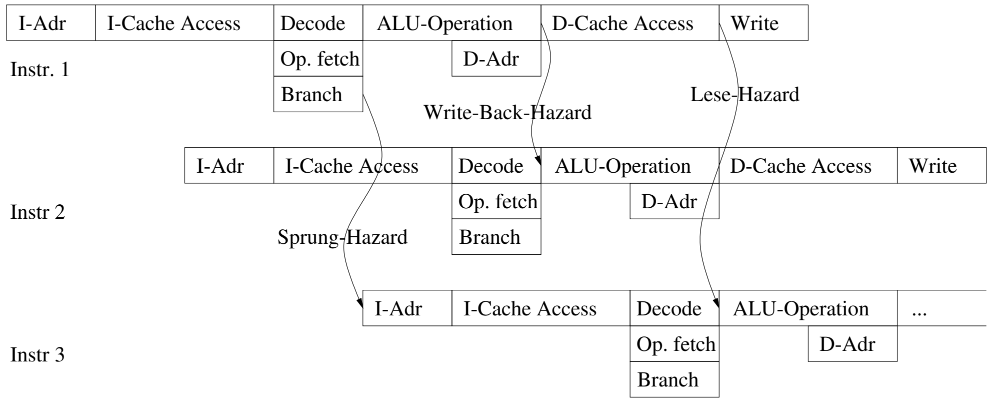

Abbildung 3: Abfolge mehrerer Befehle in der Pipeline mit Darstellung der Hazard-Abhängigkeiten

Zeile	Label	Befehle	Operanden
1		lw	$2, 10($8)
2		addiu	$2, $2, 5
3		j	woanders
4	label	addiu	$2, $0, 0

Tabelle 5: Assembler-Programm ohne Hazard-Berücksichtigung

Ohne weitere Optimierung übersetzt der Assembler diese Befehle in eine Maschinensprachensequenz, die zum Abfangen jedes Hazards einen `nop`-Befehl enthält (der Einfachheit halber wird hier von `nop` geredet, auch wenn dieser Befehl für die MIPS-Architektur nicht wirklich existiert und durch einen anderen, genauso wirkungslosen Befehl ersetzt wird). Das Ergebnis ist in Tabelle 6 zu finden.

Zeile	Label	Befehle	Operanden
1		lw	$2, 10($8)
		nop	
2		addiu	$2, $2, 5
3		j	woanders
		nop	
4	label	addiu	$2, $0, 0

Tabelle 6: Maschinenprogramm mit Hazard-Berücksichtigung

Wenn der Assembler die Maschinensprachensequenz noch optimiert, dann ist es möglich, die ursprüngliche Befehlssequenz ohne Geschwindigkeitsverlust direkt auszuführen und dabei die Hazards zu umgehen. Das Ergebnis befindet sich in Tabelle 7.

Das Ergebnis der Optimierung führt dazu, daß die Befehle der zweiten und der dritten Zeile vertauscht werden. So wird direkt nach dem Ladebefehl zuerst der Sprungbefehl ausgeführt. Danach steht dann das Ergebnis des Ladebefehls für den Additions-

Zeile	Label	Befehle	Operanden
1		lw	$2, 10($8)
3		j	woanders
2		addiu	$2, $2, 5
4	label	addiu	$2, $0, 0

Tabelle 7: Optimiertes Maschinenprogramm mit umsortierten Befehlen

befehl zur Verfügung. Nach dem Additionsbefehl wird dann der eigentliche Sprung ausgeführt, so daß das Verhalten des in Tabelle 5 aufgeführten Assembler-Programms durch dieses Maschinenprogramm exakt nachgebildet wird.

Nicht in jedem Fall kann so effizient optimiert werden. Das bedeutet, daß bei der Übersetzung vom Assembler manchmal `nop`-Befehle eingefügt werden müssen, so daß der effektive Durchsatz der MIPS-R3000-Architektur bei etwas weniger als einem Befehl pro Takt liegt.

2.6 Caches

Heutzutage enthalten fast alle Mikroprozessoren Caches. Sobald ein Prozessor im Vergleich zum ihn umgebenden Speicher relativ schnell getaktet ist, muß ein schneller Zwischenspeicher (ein Cache) dafür sorgen, daß der Prozessor nicht zu viel Zeit mit dem Warten auf den verhältnismäßig langsamen Speicher verschenkt. Ein Cache speichert eine im Gegensatz zum Hauptspeicher kleine Menge von Daten, auf die dann möglichst oft zugegriffen wird, so daß viele langsame Hauptspeicherzugriffe vermieden werden können. Idealerweise ist ein Cache gegenüber dem Prozessorkern vollkommen unsichtbar. Aus der Sicht des Kerns ist die Existenz der Caches nur am verbesserten Zeitverhalten bemerkbar. Normalerweise besitzt der Prozessorkern allerdings Konfigurationsmöglichkeiten, um das Verhalten des Caches zu beeinflussen.

2.6.1 Funktionsweise

Die MIPS-R3000-Architektur definiert zwei Caches: Einen Befehls- (*Instruction Cache*, kurz: I-Cache) und einen Daten-Cache (*Data cache*, kurz: D-Cache). Beide Caches können je nach Implementation der Architektur entweder prozessorintern oder -extern realisiert sein. Heutzutage befinden sich beide Caches aus Kostengründen immer innerhalb des Prozessors.

Beide Caches arbeiten grundsätzlich wie folgt: Vor dem Lesezugriff auf den Speicher werden dem Cache die unteren Adreßbits (die Anzahl der Bits ist abhängig von der Größe des Caches) übergeben. Dieser sucht das in ihm enthaltene Datenwort, die oberen Bits der Adresse und das Gültigkeitszeichen. Wenn die Daten im Cache gültig sind und die oberen Adreßbits übereinstimmen, kann der Wert des Caches direkt übernommen werden. Andernfalls muß der Wert aus dem Hauptspeicher gelesen und in den Cache übernommen werden. In diesem Fall muß der Prozessor auf das Ende des Lesezugriffs warten. Dieser Wartevorgang wird automatisch vollzogen, ohne daß der Prozessorkern darauf Einfluß hat.

2.6.2 Anbindung der Caches an den Kern

Für den Zugriff auf beide Caches wird ein einziger Bus verwendet. Das bedeutet, daß beide Caches nie gleichzeitig angesprochen werden können. Aber das ist auch gar nicht notwendig, da in der R3000-Pipeline die Zugriffe auf beide Caches immer zeitversetzt stattfinden. Während der ersten Hälfte jedes Pipeline-Schrittes ($\Phi 1$) bekommt der D-Cache eine Adresse übergeben und liefert das Ergebnis der vorigen Anfrage. Während der zweiten Hälfte jedes Pipeline-Schrittes ($\Phi 2$) wird der I-Cache angesprochen (siehe Abbildung 2). Der Aufbau der IDT R3051-Cache-Ansteuerung ist in Abbildung 4 dargestellt.

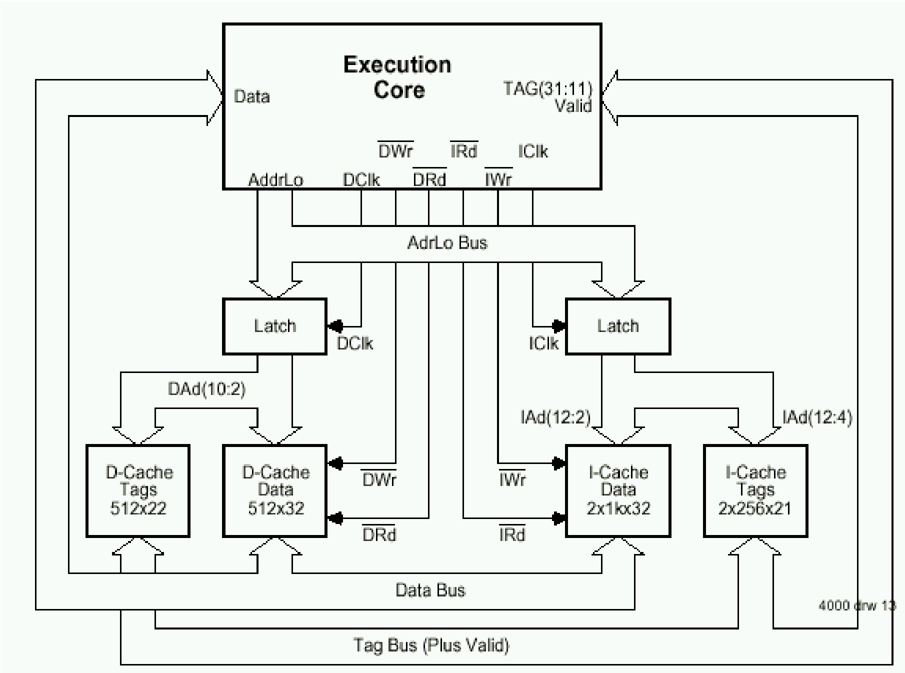

Abbildung 4: Blockschaltbild der Cache-Ansteuerung des IDT R3051 [IDT 94]

2.6.3 Instruktions-Cache

Der I-Cache ist in 32-Bit Worten organisiert und besitzt je nach Implementation zwischen 1 und 32 KByte Umfang. Jeweils vier dieser Worte sind zu einer Cache-Zeile zusammengefaßt, die den Inhalt von vier im Speicher aufeinanderfolgenden Instruktionen speichern können. Wird bei einem Lesezugriff ein Wort nicht im Cache gefunden, wird versucht, eine komplette Cache-Zeile zu füllen und simultan, das heißt sobald das erste angeforderte Datenwort vom Speicher geliefert wurde, die Befehlsbearbeitung fortzusetzen.

2.6.4 Daten-Cache

Auch der D-Cache ist in 32-Bit Worten organisiert und besitzt normalerweise einen Umfang zwischen 512 Byte und 8 KByte. Im Gegensatz zum I-Cache werden diese

Worte alle einzeln behandelt und nicht zu Cache-Zeilen zusammengefaßt, da Datenzugriffe nur selten direkt aufeinanderfolgen. Außerdem vereinfacht sich dadurch die Handhabung von Schreibzugriffen, die beim I-Cache nicht auftreten können (da selbstmodifizierende Maschinenprogramme grundsätzlich nicht erlaubt sind).

2.7 Koprozessoren

Die MIPS-R3000-Architektur sieht bis zu vier Koprozessoren vor, die die allgemeine Funktionalität des Prozessorkerns um spezielle Aufgabenbereiche erweitern. Diese Koprozessoren können entsprechend der MIPS-R3000-Architektur entweder innerhalb des Prozessors oder extern realisiert sein. In der Praxis integrieren die heute angebotenen MIPS-R3000-kompatiblen Prozessoren die Koprozessoren immer intern. Die Koprozessoren stellen hauptsächlich ein Konzept zur Erweiterung des Befehlssatzes und der Kernfunktionalität dar, ohne einen wirklichen zusätzlichen Hardware-Aufwand außerhalb des Prozessors zu bedeuten.

Grundsätzlich können Koprozessoren beliebig gestaltet werden. Es gibt jedoch zwei wichtige Implementationen von Koprozessoren, die durch MIPS vorgegeben wurden: Der Koprozessor 0 zur Speicher- und Exception-Verwaltung und der Koprozessor 1 für Fließkommaberechnungen. Viele Anbieter von MIPS-kompatiblen Prozessoren implementieren diese Koprozessoren. Da hier die allgemeine MIPS-R3000-Architektur beschrieben wird, kann keine spezielle Implementation eines Koprozessors angegeben werden. Im Abschnitt 3.2 wird näher auf die Koprozessoren des IDT R3051 eingegangen.

3 Beschreibung des IDT R3051

3.1 Aufbau

Als direkte Implementation der MIPS-R3000-Architektur besitzt der IDT R3051 alle bereits beschriebenen Elemente. Darüber hinaus kann für diesen speziellen Baustein nun angegeben werden, wie die Koprozessoren implementiert sind und wie der Zugriff auf externen Speicher, also die Hardware-Schnittstelle, realisiert ist. Als Übersicht der Funktionseinheiten des IDT R3051 dient die Abbildung 5, die alle wichtigen Elemente dieses Prozessors in Beziehung setzt.

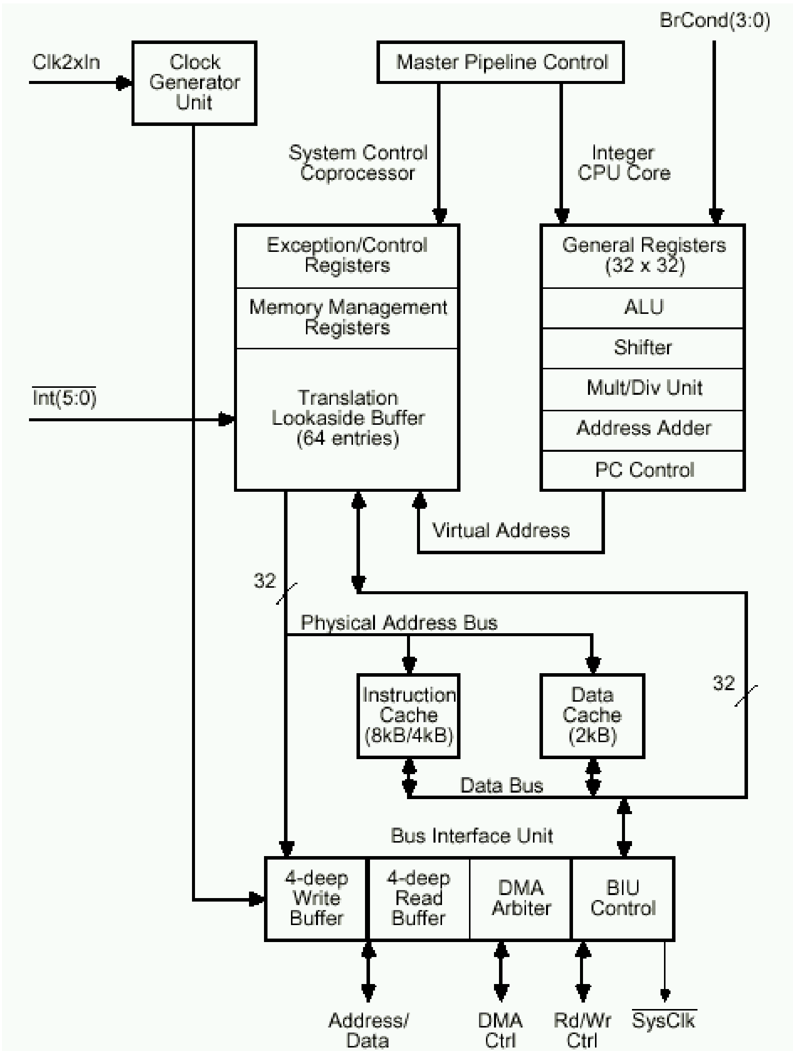

Abbildung 5: Grundlegendes Blockschaltbild des IDT R3051 [IDT 94]

3.2 Koprozessoren

Der IDT R3051 besitzt nur die nötigste Ausstattung an Koprozessoren, nämlich einen Koprozessor 0, der für die Konfiguration des Prozessors zuständig ist. MMU-Funktionen kann dieser Koprozessor nicht ausführen. Auch ist kein Koprozessor 1 für Fließkommaberechnungen vorhanden. Allerdings existieren andere Prozessoren von IDT, die sowohl eine MMU als auch eine FPU enthalten und dank der praktisch identischen Hardware-Schnittstelle anstelle des IDT R3051 eingesetzt werden können.

Bis auf das Fehlen der MMU entspricht der Koprozessor 0 des IDT R3051 dem MIPS-R3000-Koprozessor 0. Aus diesem Grund kann jede normale MIPS-R3000-Software zusammen mit großen Teilen des Betriebssystems direkt auf dem IDT R3051 ausgeführt werden, ohne daß Änderungen vorgenommen werden müssen. Nur die Speicherverwaltung muß an das Fehlen der MMU angepaßt werden.

3.3 Caches

Der IDT R3051 besitzt einen I-Cache mit 4 KByte und einen D-Cache mit 2 KByte Größe. Die Caches verhalten sich wie in Abschnitt 2.6 beschrieben. Dabei ist eine Besonderheit zu beachten, die nicht durch die Architektur, sondern erst durch den konkreten Prozessor vorgegeben wird. Schreibzugriffe werden von den Caches nicht beschleunigt: Jeder Schreibzugriff wird immer in den Cache und in den Speicher geschrieben (Cache mit direktem Schreiben, *Write-Through-Cache*).

Die Benutzung der Caches verläuft, vom Prozessorkern aus gesehen, vollkommen transparent. Nur der Koprozessor 0 enthält einige Möglichkeiten zur Beeinflussung der Arbeit der Caches. Dies ist notwendig, damit zum Beispiel die Cache-Größe, die sich von Modell zu Modell unterscheiden kann, festgestellt werden kann und die Caches vor der eigentlichen Benutzung sinnvoll initialisiert werden können. Im Statusregister des Koprozessor 0 stehen zwei Bit zur Verfügung, die die Arbeitsweise der Caches beeinflussen:

Das Flag IsC (*isolate caches*) trennt die Verbindung zwischen den Caches und der Speicherschnittstelle. Sobald die Caches isoliert sind, kann der Daten-Cache beschrieben und ausgelesen werden, ohne daß der Hauptspeicher beeinflußt wird.

Da die Schreib-/Lesebefehle nur den Daten-Cache beeinflussen, können die Caches auch vertauscht werden. Das Flag SwC (*swap caches*) sorgt dafür, daß der I-Cache die Funktion des D-Cache einnimmt und umgekehrt. Nun kann auch der I-Cache mit den normalen Befehlen gelesen und beschrieben werden.

Die Caches dürfen nur vertauscht oder isoliert werden, wenn das Programm selbst nicht im I-Cache vorhanden ist, sondern direkt aus dem Hauptspeicher heraus ausgeführt wird. Genaueres dazu steht im nächsten Abschnitt.

3.4 Memory Management

Es gibt zwei Versionen des IDT R3051: Der normale IDT R3051 besitzt eine statischer Adreßumsetzung; der IDT R3051E ermöglicht dynamische Adreßumsetzung gleichbedeutend mit einer richtigen MMU. Da dieser Diplomarbeit der IDT R3051 mit statischer Adreßumsetzung zugrunde liegt, muß im Folgenden auch nur darauf eingegangen werden.

Der Prozessor kennt eine Unterscheidung zwischen User- und Kernel-Modus. Teile des Betriebssystem werden im Kernel-Modus, alles andere wird im User-Modus ausgeführt. Die Umschaltung zwischen User- und Kernel-Modus wird im Koprozessor 0 vorgenommen. Der Speicher ist in vier Segmente unterteilt:

kuseg: Das kuseg umfaßt 2 GByte. Befindet sich der Prozessor im User-Modus, dann kann nur auf dieses Speichersegment zugegriffen werden. Alle Zugriffe auf andere Segmente führen zu einer Exception (siehe Abschnitt 3.5). Bei Zugriffen auf das kuseg werden immer beide Caches verwendet.

kseg0: Dieses 512 MByte große Segment enthält ausführbare Betriebssystemteile und Daten. Für den Zugriff auf dieses Segment werden die Caches benutzt.

kseg1: Im Unterschied zu kseg0 werden für den Zugriff auf kseg1 keine Caches benutzt. Deshalb können hier speicherbasierte Ein- und Ausgaben, direkter Speicherzugriff und Cache-Initialisierungen ausgeführt werden. kseg1 umfaßt genau wie kseg0 512 MByte und wird auf denselben physikalischen Adreßraum abgebildet.

kseg2: Dieses Segment verhält sich wie kseg0 (nur bei Prozessoren mit MMU gibt es Unterschiede). Es besitzt 1 GByte Adreßraum.

Zugriffe auf kseg0 und kseg1 werden auf dieselben physikalischen Adressen umgesetzt, jedoch finden Zugriffe entweder mit oder ohne Cache-Benutzung statt. So kann auch mit einem einzigen kleinen Speicher eine Anwendung mit Caches und speicherbasierter Ein- und Ausgabe benutzt werden. Die Umsetzung der Adressen geht aus Tabelle 8 hervor.

Segment	Virtuelle Adresse	Physikalische Adresse	Cache wird benutzt	MMU würde benutzt
kuseg	`0x00000000` - `0x7fffffff`	`0x40000000` - `0xbfffffff`	ja	ja
kseg0	`0x80000000` - `0x9fffffff`	`0x00000000` - `0x1fffffff`	ja	nein
kseg1	`0xa0000000` - `0xbfffffff`	`0x00000000` - `0x1fffffff`	nein	nein
kseg2	`0xc0000000` - `0xffffffff`	`0xc0000000` - `0xffffffff`	ja	ja

Tabelle 8: Statische Umsetzung zwischen virtuellen und physikalischen Adressen

3.5 Exceptions

In der englischen Literatur wird je nach Prozessor oder Architektur entweder von *Exceptions* oder von *Interrupts* und manchmal auch von beidem gesprochen. Die MIPS-Terminologie ist klar definiert: Eine nicht geplante Unterbrechung des Programmablaufs ist immer eine *Exception*, egal ob der Grund der Unterbrechung innerhalb oder außerhalb des Prozessors zu suchen ist. *Interrupts* sind normalerweise externe Unterbrechungen, allerdings gibt es auch die Möglichkeit, intern einen Interrupt zu simulieren. Wenn im Folgenden von Exceptions gesprochen wird, dann sind damit alle möglichen Unterbrechungen, also auch Interrupts gemeint. Interrupts können im Gegensatz zu sonstigen Exceptions aktiviert oder deaktiviert werden. Exceptions lösen immer eine Unterbrechung des Programmablaufs aus und können nicht deaktiviert werden.

Tritt eine Exception auf, so ist immer genau die auslösende Instruktion, deren Kontext und der Grund der Exception bekannt. Deshalb kann die Programmausführung nach Behandlung der Exception direkt fortgeführt werden. Der IDT R3051 implementiert sogenannte präzise Exceptions.

Wird in diesem Text im Folgenden von Exceptions gesprochen, so sind damit immer die hier vorgestellten MIPS-Exceptions gemeint. Diese Exceptions haben nichts zu tun mit denen, die ein Teil der Programmiersprache Java sind.

3.5.1 Exception-Kategorien

Es gibt eine Reihe verschiedener Exception-Kategorien, die die verschiedenen Unterbrechungsgründe widerspiegeln. Tabelle 9 enthält eine Übersicht.

Exception	Nummer/ Abkürzung	Erklärung
Reset	Reset	Reset wurde ausgelöst, Programmzähler wird auf `0xbfc00000` gesetzt.
Bus Error	6/IBE (Instruction), 7/DBE (Data)	Die Speicherschnittstelle meldet einen externen Fehler.
Address Error	4/AdEL (Load), 5/AdES (Store)	Zugriff auf ein kseg im User-Modus oder falsches Alignment.
Overflow	12/Ovf	Überlauf bei Addition oder Subtraktion.
System Call	8/Sys	Wird direkt vom `syscall`-Befehl ausgelöst.
Breakpoint	9/Bp	Wird direkt vom `break`-Befehl ausgelöst.
Reserved Instruction	10/RI	Der Befehl konnte nicht identifiziert werden.
Coprocessor unusable	11/CpU	Ein nicht vorhandener Koprozessor wurde angesprochen.
Interrupt	0/Int	Ein (interner oder externer) Interrupt wurde ausgelöst.

Tabelle 9: Exception-Kategorien und ihre Bedeutung

In dieser Tabelle wurden die Exceptions weggelassen, die allein bei Benutzung einer MMU auftreten können.

3.5.2 Für Exceptions zuständige Koprozessor-Register

Sobald eine Exception auftritt, werden in einigen Koprozessor-0-Registern die Ursache, die Rücksprungadresse und weitere Verwaltungsdaten hinterlegt. Die Exception-Behandlungs-Routine kann dann diese Koprozessor-Register auslesen und aufgrund der dort enthaltenen Information die Ausnahmebehandlung starten. Die für die Exceptions wichtigen Register sind in Tabelle 10 beschrieben.

Nummer	Name	Zweck
4	Context	Nur bei MMU-Benutzung wichtig
8	Bad Virtual Address	Wenn eine Adreß-Exception auftritt, ist hier die auslösende Adresse enthalten
12	Status	Verschiedene Status-Bits
13	Cause	Exception-Kategorie, Interrupt-Nummer, ...
14	Exception PC	Adresse der auslösenden Instruktion

Tabelle 10: Von Exceptions benutzte Register des Koprozessor 0

Die Beschreibung der einzelnen Bits des Status-Registers ist im Anhang B zu finden. Im Cause-Register wird die genaue Ursache einer Exception hinterlegt. Die verschiedenen Bitgruppen des Cause-Registers werden gleichfalls im Anhang B genauer erklärt.

3.5.3 Exception-Vektoren

Es gibt mehrere Adressen, an denen Exception-Behandlungs-Routinen abgelegt sein können, abhängig von der Exception-Kategorie. Außerdem beeinflußt das BEV-Bit im Status-Register die Adressen. Tabelle 11 beinhaltet eine Aufzählung aller Exception-Vektoren.

Exceptionklasse	virtuelle Adresse bei BEV=0	virt. Adr. bei BEV=1
Reset	`0xbfc00000`	`0xbfc00000`
UTLB Miss	`0x80000000`	`0xbfc00100`
Alle anderen	`0x80000080`	`0xbfc00180`

Tabelle 11: Exception-Vektoren

3.5.4 Interrupts

Wie auch im Anhang B beschrieben, zeigen die IP-Bits des Cause-Registers das Auftreten von Interrupts an. Entsprechend der dort realisierten sechs Bits existieren sechs verschiedene Interrupt-Leitungen, die von außen angesprochen werden können. Drei dieser Interrupt-Leitungen werden vom Prozessor automatisch synchronisiert, hier können asynchrone Interrupts gemeldet werden. Die drei anderen Interrupt-Leitungen werden nicht synchronisiert, deshalb muß die Synchronisation außerhalb des Prozessors vorgenommen werden.

3.6 Speicher-Schnittstelle

Als Hardware-Schnittstelle bietet der IDT R3051 eine einfach aufgebaute und trotzdem leistungsfähige Speicher-Schnittstelle. Die Breite des Speicherbusses beträgt immer 32 Bit. Sollen externe Speicher mit geringerer Wortbreite eingesetzt werden, muß die dafür notwendige Schaltung auch extern aufgebaut werden. Der Prozessor ist in der Lage, entweder einzelne 32-Bit Worte (*Single word read*) oder vier 32-Bit Worte (*Quad word read*, *Burst*-Modus) auf einmal zu laden. Gespeichert werden immer einzelne 32-Bit Worte.

Im Folgenden soll nun genauer auf das Verhalten während des Lesens und des Schreibens eingegangen werden. Daran schließen sich die Beschreibung von direktem Speicherzugriff und die Übersicht aller Ein- und Ausgänge an. Abgeschlossen wird die Beschreibung der Speicherschnittstelle mit einer Beschreibung der Einschalt-Initialisierung.

3.6.1 Adreß- und Datenbus

Die Speicher-Schnittstelle besitzt einen gemeinsamen Adreß- und Datenbus (A/D-Bus). Beide Arten von Information müssen daher zeitversetzt über dieselben Anschlüsse übertragen werden. Bei normalen Lese- und Schreibzugriffen ist das kein Problem, denn gängige Speicherbausteine bekommen immer erst die Adresse übergeben, um dann entweder Daten zu liefern (Lesen) oder Daten entgegenzunehmen (Schreiben).

Etwas anders sieht es beim Lesen im Burst-Modus aus: In diesem Fall wird anfänglich eine Adresse übertragen, danach werden nacheinander von aufsteigenden Adressen vier Worte zum Prozessor übertragen. Dazu wäre es eigentlich notwendig, nach der Übertragung des ersten Wortes die nächste Adresse auszugeben und so fort. Würde dieses über den gemeinsamen A/D-Bus erledigt,würde zu viel Zeit verloren gehen, der Burst-Modus wäre zu langsam und damit überflüssig. Deshalb besitzt der IDT R3051 einen zweiten, nur aus zwei Anschlüssen bestehenden Bus, den Addr-Bus. Auf diesem werden nur die Adreßbits zwei und drei übertragen, denn nur diese Bits ändern sich beim Heraufzählen der Adresse für den Burst-Modus. Wird der A/D-Bus mit einem Latch-Register verbunden und die Bits zwei und drei direkt vom Addr-Bus übernommen, so kann der Burst-Modus einfach auf diese Art und Weise implementiert werden.

Der A/D-Bus besitzt noch eine Besonderheit: Werden auf dem Bus Adressen übertragen, so enthalten die untersten vier Bits ein Byte-Enable-Signal, das pro Byte angibt, ob dieses Byte angesprochen werden soll oder nicht. Da ja noch zusätzlich der Addr-Bus die Bits zwei und drei der Adresse liefert und die Bits null und eins dank der Wortbreite von 32 Bit nicht beachtet werden müssen, ist diese Implementation zulässig.

3.6.2 Lesezugriffe

Normale Lesezugriffe ohne Benutzung des Burst-Modus laufen nach einem einfachen Schema ab: Zuerst überträgt der Prozessor die Adresse und trennt sich dann vom Bus. Danach wartet der Prozessor auf die Meldung des Speichers, daß die Daten gleich zur Verfügung stehen (*Acknowledge*, Ack). Mit dem nächsten Taktsignal übernimmt der Prozessor dann die Daten vom A/D-Bus. Damit ist der Lesevorgang beendet. Das entsprechende Impulsdiagram (mit Wartezyklen) ist aus [IDT 94] übernommen und als Abbildung 6 zu sehen.

Falls der Burst-Modus benutzt wird, ist der Ablauf etwas komplizierter. Zu Beginn überträgt der Prozessor wieder die Adresse auf dem A/D-Bus. Dann trennt er

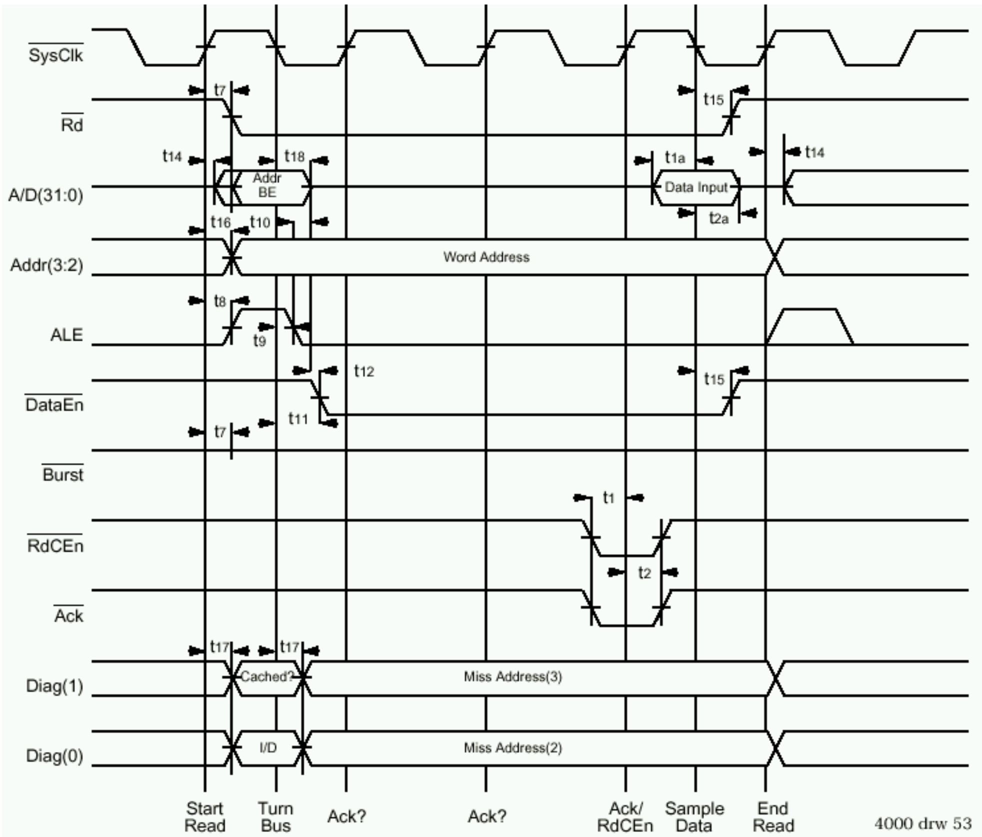

Abbildung 6: Impulsdiagramm des Lese-Zeitverhaltens mit Wartezyklen [IDT 94]

sich vom Adreßbus, läßt aber weiterhin die Bits zwei und drei der Adresse auf dem Addr-Bus erscheinen. Als Acknowledge muß nun vom Speicher das Signal kommen, den Adreßzähler zu erhöhen (*Read Counter Enable*, RdCEn). Daraufhin übernimmt der Prozessor das vom Speicher angelegte Datum als erstes Burst-Wort vom Bus und erhöht die am Addr-Bus anliegende Adresse um eins. Nun kann der Speicher das nächste Wort liefern und wieder durch RdCEn dessen Übergabe einleiten. So werden nacheinander die vier Worte eines Burst-Zugriffs übertragen. Das dazu passende Impulsdiagramm zeigt Abbildung 7.

3.6.3 Schreibzugriffe

Da für Schreibzugriffe kein Burst-Modus existiert, ist deren Implementation weniger kompliziert. Will der Prozessor ein Wort schreiben, so legt er erst die Adresse auf den A/D-Bus und den Addr-Bus. Im nächsten Takt wird dann das zu schreibende Datenwort auf dem A/D-Bus ausgegeben. Der Prozessor wartet nun, bis der Speicher durch das Ack-Signal den Erfolg der Operation bestätigt. Die Abbildung 8 zeigt das genaue Zeitverhalten von Schreibzugriffen.

Die Speicherschnittstelle des IDT R3051 besitzt noch eine wichtige Besonderheit: Es existieren vier Pufferspeicher, die jeweils ein Wort aufnehmen können. Durch diese

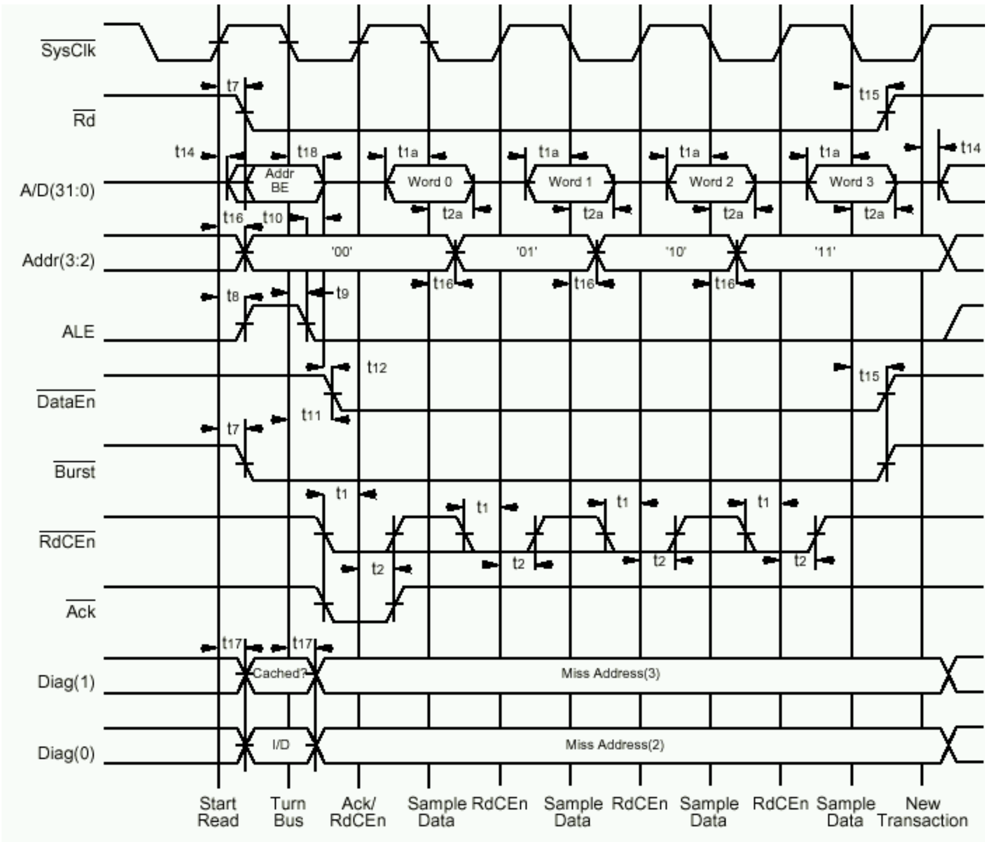

Abbildung 7: Impulsdiagramm des Burst-Zeitverhaltens ohne Wartezyklen [IDT 94]

Puffer ist der Prozessor in der Lage, Schreibzugriffe zu verzögern. Dieses Vorgehen ist sinnvoll, wenn die Speicherschnittstelle aufgrund ungenutzter oder ungefüllter Caches viele Lesezugriffe ausführen muß, so daß zusätzliche Schreibzugriffe die Arbeit des Prozessors durch einzufügende Wartezyklen verlangsamen würden.

Die Logik, die hinter der Benutzung der Schreibpuffer steckt, ist folgende: Sobald ein Schreibwunsch des Prozessors nicht direkt ausgeführt werden kann, da bereits ein Lesewunsch vorliegt oder sich ein Leseauftrag in Bearbeitung befindet, wird der Schreibwunsch in einem der Pufferspeicher gelagert. Solange noch weitere Lesewünsche bearbeitet werden können, bleibt der Schreibwunsch im Pufferspeicher. Sobald die Speicherschnittstelle keine weiteren Lesewünsche mehr bearbeitet, kann sie die Schreibwünsche ausführen, die sich in den Pufferspeichern angesammelt haben. Da nur vier Pufferspeicher vorhanden sind, muß ein Mechanismus vorgesehen werden, der bei einem drohenden Überlauf der Schreibpuffer einen der Schreibaufträge durchführt, auch wenn noch Leseaufträge auf Bearbeitung warten.

3.6.4 Direkter Speicherzugriff

Der Prozessor kann durch ein bestimmtes Signal (*Bus-Request*, BusReq) dazu angewiesen werden, sich vom Bus und allen zur Ansteuerung des Speichers notwendigen

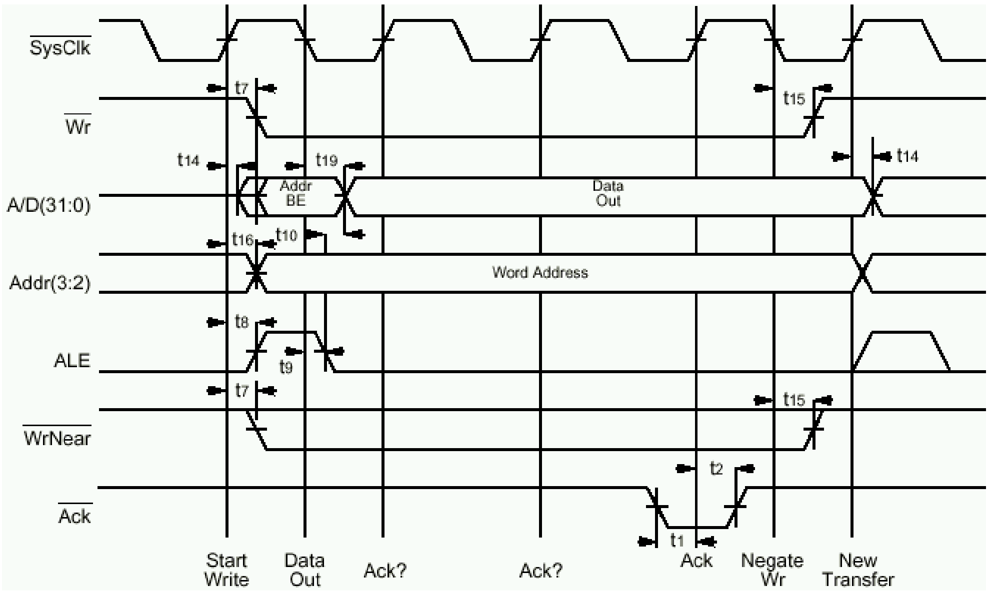

Abbildung 8: Impulsdiagramm des Schreib-Zeitverhaltens mit Wartezyklen [IDT 94]

Signalen zu trennen. Dadurch kann dann ein anderer Baustein diese Infrastruktur für eigene Speichertransfers benutzen. Dieser Modus wird auch *Direct Memory Access* oder *DMA* genannt.

Sobald der Prozessor eine BusReq-Anfrage bekommt, beendet er seine gerade laufende Busoperation und gibt dann den Bus frei. Durch das *Bus-Grant*-Signal (BusGnt) wird dann dem Anfragenden signalisiert, daß er die Speicherschnittstelle für seine Zwecke nutzen kann. Der Prozessor geht dazu über, Wartezyklen auszuführen, bis durch den BusReq-Eingang das Ende des DMA-Vorgangs angezeigt wird. Danach führt der Prozessor seine unterbrochene Arbeit direkt fort.

3.6.5 Alle Ein- und Ausgänge im Überblick

Nachdem die verschiedenen Möglichkeiten der Nutzung der Speicherschnittstelle vorgestellt wurden, sollen nun alle Ein- und Ausgänge im einzelnen vorgestellt werden.

A/D-Bus: Der Adreß-/Datenbus besteht aus 32 Anschlüssen und wird je nach Zustand des Prozessors sowohl als Eingang als auch als Ausgang genutzt.

Addr-Bus: Dieser Bus enthält die Bits zwei und drei der Adresse und wird nur als Ausgang verwendet.

ALE: Das ALE-Signal (*Address latch enable*) signalisiert, daß der A/D-Bus gerade eine Adresse enthält und diese in ein Adreß-Latch übernommen werden kann.

DataEn: Sobald der Prozessor sich vom A/D-Bus trennt, um ein Wort aus dem Speicher zu lesen, signalisiert er durch das DataEn-Signal (*Data enable*), daß der Speicher das Wort auf dem A/D-Bus ausgeben kann.

Burst/WrNear: Dieses Signal hat zwei unterschiedliche Bedeutungen: Wird gelesen, so signalisiert der Prozessor hiermit, daß der Burst-Modus benutzt werden soll. Will der Prozessor schreiben, so soll durch WrNear angegeben werden, ob der letzte Schreibzugriff innerhalb derselben Speicherseite stattfand wie der eben initiierte. Dadurch können spezielle Speichertypen wie *Page-Mode*- oder *Static-Column*-Speicher dabei unterstützt werden, durch schon bestehende Seitenauswahl den Speicherzugriff zu beschleunigen.

Rd: Mit dem *Read*-Signal zeigt der Prozessor, daß ein Lesezugriff stattfinden soll.

Wr: Hier handelt es sich um ein *Write*-Signal zur Indikation von Schreibzugriffen.

Ack: Will der Speicher seine Zustimmung zu einer Operation bekannt geben, so nutzt er das Ack-Signal (*Acknowledge*). Dieses Signal ist ein Eingang.

RdCEn: Für den Burst-Modus benutzt der Speicher statt des Ack-Signals das RdCEn-Signal (*Read counter enable*), um damit gleichzeitig zu signalisieren, daß der Adreßzähler inkrementiert werden soll.

BusError: Falls der Speicher eine Anfrage nicht befriedigen kann, muß er dieses durch das *BusError*-Signal melden.

Diag: Der Prozessor besitzt einen aus zwei Bits bestehenden Diagnose-Bus, der nur als Ausgang verwendet wird. Hier werden je nach Zustand der Speicherschnittstelle verschiedene Werte ausgegeben.

BusReq: Mit dem BusReq-Signal (*Bus request*) meldet ein externer Baustein seinen Wunsch beim Prozessor an, einen DMA-Zyklus durchzuführen. Auch das Ende eines DMA-Zyklus wird dem Prozessor über dieses Signal gemeldet.

BusGnt: Das BusGnt-Signal (*Bus grant*) dient dem Prozessor dazu, den Beginn eines DMA-Zyklus anzuzeigen.

BrCond/SBrCond: Der Koprozessor 0 bekommt durch zwei BrCond- (*Branch condition*) und zwei SBrCond-Signale (*Synchronized branch condition*) mitgeteilt, wie sich der Koprozessor-Sprungbefehl verhalten soll.

Int/SInt: Jeweils drei Int- (*Interrupt*) und SInt-Signale (*Synchronized interrupt*) stehen als Interruptquellen zur Verfügung.

ClkIn: Das ClkIn-Signal (*Clock in*) liefert den Takt für den Prozessor. Dabei ist zu beachten, daß innerhalb fast jedes Pipeline-Schrittes zwei Aktionen durchgeführt werden und dementsprechend immer zwei Takte pro Pipeline-Schritt benötigt werden. Deshalb wird alle zwei Takte die Ausführung eines Befehls beendet.

SysClk: Der Prozessor erzeugt aus dem ClkIn-Signal ein SysClk-Signal (*System clock*). Der Takt dieses Signals entspricht der Hälfte des ClkIn-Signals und wird als Referenztakt für die Speicherzugriffe genutzt.

Reset: Das Reset-Signal initialisiert den Prozessor.

Rsvd: Diese fünf Rsvd-Signale (*Reserved*) werden nicht genutzt und dürfen nicht verbunden werden. Ansonsten Hardware-kompatible Modelle wie der IDT R3041 oder der IDT R3081 benutzen diese Ein- und Ausgänge für spezielle Zwecke.

3.6.6 Einschalt-Initialisierung

Während der Endflanke des Reset-Signals können einige Konfigurationswerte an den Prozessor übergeben werden. Sobald das Reset-Signal beendet wird, übernimmt der Prozessor die an den Interrupt-Eingängen anliegenden Werte in ein internes Register, um damit einige Konfigurationsmöglichkeiten anzubieten. Beachtet werden dabei die SInt-Signale. Die Int-Signale sollten aus Kompatibilitätsgründen während des Reset immer inaktiv sein. Über SInt2 kann gewählt werden, ob der Daten-Cache immer ein Wort einzeln oder vier Worte zugleich im Burst-Modus vom Speicher anfordern soll. SInt1 bietet die Möglichkeit, den Prozessor anzuweisen, bis zum nächsten Reset alle Ausgänge bis auf SysClk offen zu betreiben. Und die Einstellung der Byte-Orientierung kann mit SInt0 vorgenommen werden. Dabei wird die grundlegende Byte-Orientierung eingestellt. Zusätzlich kann über das Statusregister des Koprozessor 0 die gegensätzliche Byte-Orientierung für Benutzerprogramme eingestellt werden.

4 Randbedingungen der Modellierung

In diesem Kapitel wird zuerst der in diesem Zusammenhang verwendete Simulationsbegriff geklärt und das entstehende MIPS-Simulationsmodell eingeordnet und mit bestehenden Ansätzen und deren Abstraktionsebenen verglichen. Danach kann die Simulationumgebung HADES und sowohl dessen Schnittstellen zu Simulationsmodellen im allgemeinen als auch die spezielle Nutzung der Schnittstellen vom MIPS-Simulationsmodell aus genauer beschrieben werden.

4.1 Simulation digitaler Hardware-Komponenten

Eine Simulation stellt immer eine Abstraktion, letztendlich also eine Vereinfachung der zugrunde liegenden realen Vorgänge dar. Mit möglichst wenig Beschreibungs-, Implementations- oder Betriebsmittelaufwand wird versucht, ein komplexes Verhalten so vollständig wie möglich, auf jeden Fall aber für die angestrebten Untersuchungen ausreichend, abzubilden. Es gibt je nach Anspruch an die Wirklichkeitstreue verschiedene Möglichkeiten der Implementation. Dabei müssen immer Kompromisse zwischen Implementations-, Ausführungsaufwand und Realitätsnähe eingegangen werden.

Da sich kontinuierliche Zeitabläufe in einer rechnergestützten Simulation nur schwer nachbilden lassen, bietet sich für die Simulation digitaler Hardware ein ereignisgesteuerter Simulationsablauf an. Das bedeutet, daß als Voraussetzung angenommen wird, daß die einzelnen an der Simulation beteiligten Komponenten auf spezielle Ereignisse warten und, aufgrund dieser, Aktionen ausführen, die wiederum zur Erzeugung von neuen Ereignissen führen. Die Ereignisse einer Simulation müssen sinnvoll verwaltet und an die adressierten Elemente weitergegeben werden. Zentrale Rolle spielt dabei eine Warteschlange, in die die Ereignisse in Abhängigkeit des Auftrittszeitpunktes eingereiht werden. Die einzelnen in der Warteschlange enthaltenen Ereignisse werden der Reihe nach erzeugt und an die von ihnen betroffenen Komponenten weitergeleitet. Dabei entstehen dann wieder neue Ereignisse, die in die Warteschlange übernommen werden. Dieses Verhalten wird als ereignisbasierte Simulation (siehe [Page 91]) bezeichnet. Für die Simulation taktgesteuerter Hardware reicht dieses Modell nicht aus, denn es muß zusätzlich die Möglichkeit vorhanden sein, zu bestimmten (Simulations-) Zeitpunkten neue Ereignisse zu generieren. Erst ein solcher Ereignisgenerator kann eine taktliefernde Hardware-Komponente nachbilden.

Soll im Gegensatz zum komplexen Zusammenspiel mehrerer Hardware-Komponenten nur eine einzige, taktgesteuerte Komponente mit möglichst großer Geschwindigkeit simuliert werden, kann ein anderer Ansatz gewählt werden: In diesem Fall muß nur nacheinander das für jeden Takt auftretende Verhalten nachgebildet werden. Dieser Ansatz wird zyklusbasierte Simulation genannt.

4.2 Vergleich mit bestehenden Simulationsmodellen

Zur Einordnung des während dieser Diplomarbeit entstehenden Simulationsmodells muß zuerst betrachtet werden, welche verschiedenen Klassen von Prozessor-Simulationsmodellen bisher existieren. Diese können dann mit dem MIPS-Simulationsmodell verglichen werden.

Simulation kompletter Endprodukte Einige Simulationsmodelle bieten nicht nur die Simulation eines Prozessors, sondern bilden komplette Endprodukte inklusive ihrer Oberfläche nach. Beispiele dafür wären der Palm-OS-Emulator für den PDA Palm Pilot [Palm 99], siehe Abbildung 9, oder der Playstation Emulator für die Sony Playstation II [Cygnus 98].

Abbildung 9: Oberfläche des Palm OS Emulator [Palm 99]

Da fertige Endprodukte nachgebildet werden, sind keine speziellen Konfigurationsmöglichkeiten vorhanden. Die im Prozessor stattfindenden Vorgänge werden nicht weiter dargestellt und meist vereinfachend simuliert. Vom Benutzer einstellbare Veränderungen an der simulierten Hardware sind nicht vorgesehen. Daraus ergibt sich eine hohe Simulationsgeschwindigkeit, wobei trotz der Simulation aller Peripherie-Komponenten teilweise sogar Echtzeit erreicht wird.

Diese Simulationsmodelle bieten die Möglichkeit, Software für bestimmte Endprodukte zu testen, ohne daß das Endprodukt selbst vorhanden sein muß. Sie werden auch als Entwicklungsplattform von Software für Systeme, auf denen selbst keine Software entwickelt werden kann, genutzt. Die Fehlersuche innerhalb solcher Software wird dabei unterstützt und vereinfacht. Der Palm OS Emulator ist kostenlos verfügbar. Das Simulationsmodell der Sony Playstation kostet fünfstellige Summen [Cygnus 99a].

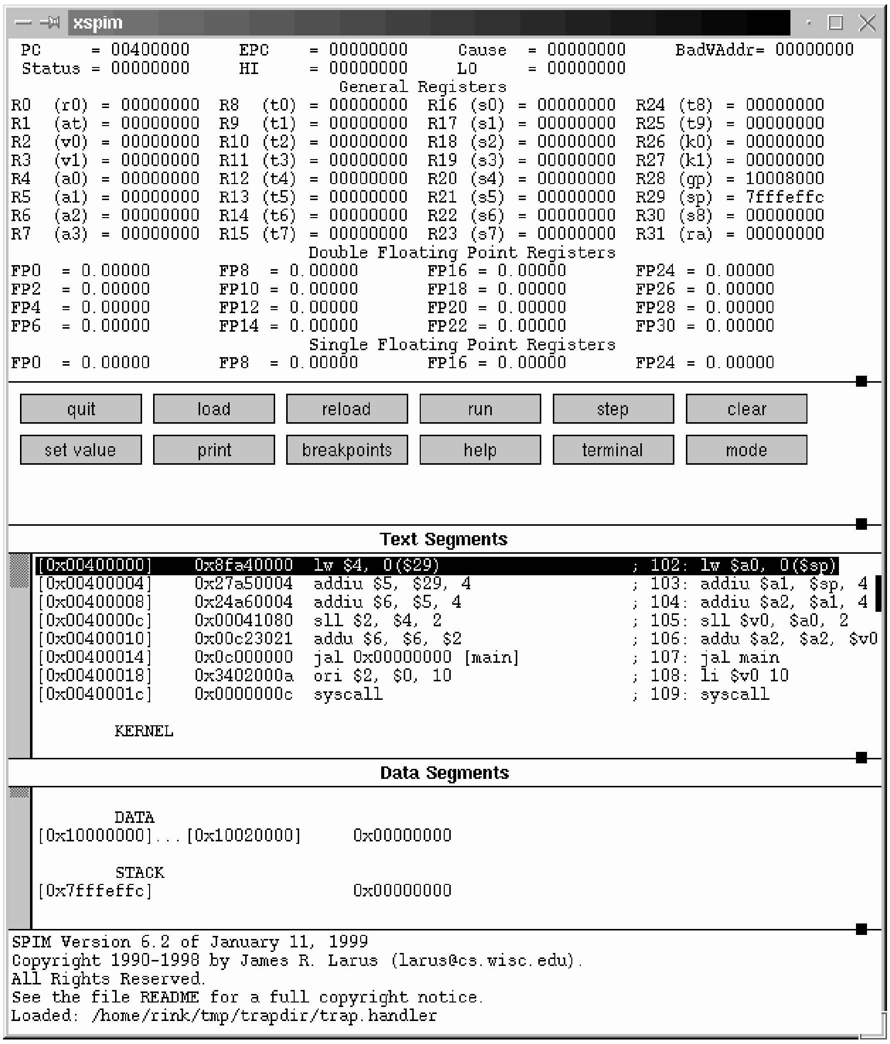

Abbildung 10: Oberfläche von SPIM [Larus 97]

Alleinstehende Prozessorsimulation Für viele Prozessoren existieren oft Simulationsmodelle, die nur die Software-Schnittstelle des Prozessors nachbilden und statt einer Hardware-Schnittstelle nur die nötigste Peripherie anbieten (Speicher, manchmal serielle Schnittstellen).

Ein Beispiel für einen solches Prozessor-Simulationsmodell ist SPIM (siehe Abbildung 10). Dieses Programm simuliert die MIPS-R3000-Architektur zusammen mit dem Arbeitspeicher, der für Programme und Daten genutzt werden kann. Elemente der MIPS-R3000-Architektur, die vor dem Assembler-Programmierer verborgen bleiben (Pipeline und Caches), werden nicht nachgebildet. Nur die Register des Prozessors werden dargestellt, so daß der Benutzer darauf Einluß nehmen kann. Die Simulationsgeschwindigkeit von SPIM ist aufgrund der Vereinfachungen hoch. Das Programm ist kostenlos verfügbar und wird in [Larus 97] dokumentiert.

In-Circuit-Emulation Für einige Prozessoren werden Simulationsmodelle angeboten, die direkt in den Prozessorsockel innerhalb der Hardware-Anwendung gesteckt werden, sich (meistens bis auf die zeitliche Komponente) genauso verhalten wie der zugrunde liegende Prozessor und zusätzlich eine Inspektionsschnittstelle bieten. Solche *In-Circuit*-Simulationsmodelle sind wesentlich teurer als der Original-Prozessor und umständlich in der Handhabung. Sie sind nur zu gebrauchen, wenn die Hardware-Schaltung, innerhalb der der Prozessor eingesetzt werden soll, bereits fertig entwickelt und aufgebaut ist.

Simulation von VHDL- und Verilog-Beschreibungen Zur Simulation digitaler Schaltungen im Rahmen des VLSI-Entwurfs haben sich in den letzten Jahren die beiden Hardware-Beschreibungssprachen VHDL und Verilog durchgesetzt, wobei Verilog in den USA und VHDL in Europa und Japan dominieren. An dieser Stelle soll nur VHDL [IEEE 93a] skizziert werden; in ähnlicher Form gelten alle genannten Eigenschaften auch für Verilog.

VHDL erlaubt eine Kombination von Struktur- und prozessorientierten Verhaltensbeschreibungen. Neben den grundlegenden Datentypen wie Integerzahlen werden spezielle Datentypen zur Logiksimulation von elektrischen Signalen unterstützt. Etabliert ist vor allem das neunwertige Logikmodell nach IEEE 1164 [IEEE 93b], mit dem auch Bussysteme effizient modelliert werden können.

Auch die Systemsimulation wird von VHDL unterstutzt, wobei Modelle aller Abstraktionsebenen kombiniert werden können. Die notwendigen Simulationsmodelle für Prozessoren und Peripheriekomponenten sind allerdings derzeit kaum als freie Software erhältlich und als kommerzielle Modelle sehr teuer. Leider definiert VHDL keine standardisierten Schnittstellen zu anderen Programmiersprachen oder Funktionsbibliotheken (etwa für Grafik). Daher ist es zum Beispiel unmöglich, geeignete Benutzeroberflächen für komplexe Komponenten zu erstellen, etwa zur direkten Anzeige der Daten auf einem Grafikdisplay. Der Umweg über Textausgaben und externe Hilfsprogramme ist mühsam und sehr langsam.

Außerdem sind vollwertige VHDL oder Verilog-Simulatoren derzeit noch extrem teuer und ausschließlich auf die Anforderungen professioneller Schaltungsentwickler ausgerichtet. Die Bedienung ist daher oft sehr komplex und höchste Performance hat Vorrang vor interaktiven Eingriffsmöglichkeiten.

Simulation von Einzelkomponenten Innerhalb einer Simulationsumgebung lassen sich statt eines kompletten Prozessor-Simulationsmodells auch einzelne Komponenten zum Beispiel der Register-Transfer-Ebene wie Registerbank, Rechenwerk und Cache-Speicher simulieren. Dabei wird der interne Aufbau eines Prozessors sehr gut nachgebildet. Allerdings benötigt solch eine Simulation sehr viel Rechenzeit und ist nicht einfach zu bedienen.

Ein Beispiel für eine Simulation von Einzelkomponenten ist die Simulation des D*CORE [Hendrich 2000a], die am Fachbereich Informatik der Universität Hamburg entstanden ist. Die Einzelkomponenten sind HADES-Modelle der Register-Transfer-Ebene und werden zusammen mit externen Komponenten (Speicher) simuliert (siehe Abbildung 11).

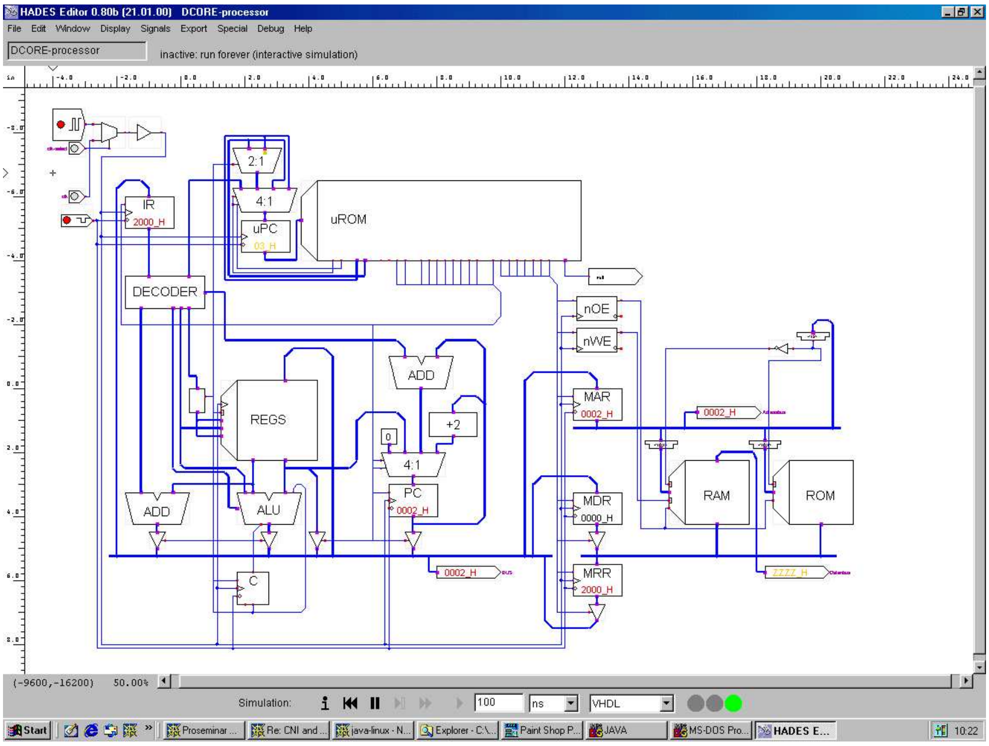

Abbildung 11: D*CORE als HADES-Design

Das MIPS-Simulationsmodell im Vergleich Jede der genannten Simulationsmöglichkeiten für Prozessoren bietet verschiedene typische Vor- und Nachteile. Das MIPS-Simulationsmodell soll die Vorteile verschiedener Möglichkeiten in sich vereinen: Wie bei einem *In-Circuit*-Simulationsmodell soll der Prozessor selbst zusammen mit seiner (allerdings auch simulierten) Umgebung simuliert werden können. Dabei sollen wie bei der Simulation von Einzelkomponenten oder der Simulation durch eine VHDL-Beschreibung auch interne Zustände sichtbar gemacht werden können. Die Ausführungsgeschwindigkeit von Simulationsläufen soll dabei zumindest im Ansatz mit derer für alleinstehende Prozessorsimulation vergleichbar sein.

Das MIPS-Simulationsmodell soll, wie auch die Simulationsumgebung HADES, weitere damit benutzbare Simulationsmodelle und auch die später vorgestellte Entwicklungsumgebung (siehe Abschnitt 8.2) für MIPS-Software, kostenlos verfügbar sein.

4.3 Vorstellung von HADES

Auch wenn für die Logiksimulation beim Entwurf integrierter Schaltungen kaum auf VHDL verzichtet werden kann, bleiben gerade im Hinblick auf Systemsimulation und die Hardware/Software-Cosimulation doch Wünsche offen. Einerseits ist die Interaktion mit externen Programmen und Bibliotheken problematisch und es muß auf maßgeschneiderte Benutzeroberflächen für komplexe Simulationsmodelle verzichtet werden. Andererseits sind die Interaktionsmöglichkeiten während einer laufenden Simulation selbst bei sehr teuren Simulatoren [Synopsys 99] stark eingeschränkt.

Im Gegensatz dazu stehen benutzerfreundliche Simulationsumgebungen wie DigLOG [Gillespie 96], die dem Anwender direkt während des laufenden Simulationsexperiments eine Modifikation der Vorgaben und somit ein "Herumspielen" erlauben. Gerade für Lehrzwecke ist dieses Verhalten wünschenswert. Die bisher angebotenen Simulationsumgebungen wie DigLOG bieten jedoch nur eingeschränkte Abstraktionsfähigkeiten, so daß hauptsächlich Gatter, aber keine komplexeren Komponenten simuliert werden können. Das HADES Framework [Hendrich 98] soll genau diese Lücke schließen.

- HADES bietet ein flexibles und erweiterbares Rahmenwerk.
- Die Software ist frei verfügbar und portabel.
- Zur Benutzung innerhalb eines Web-Browsers ist HADES Applet-kompatibel.
- HADES bietet eine voll interaktive Simulation einschließlich der Möglichkeit zum Editieren während einer laufenden Simulation.
- Alle Abstraktionsebenen von Systemebene hinunter zur Gatterebene werden unterstützt.
- VHDL-basierte Semantik [IEEE 93a] und das Logikmodel nach IEEE 1164 [IEEE 93b] werden angeboten.
- Es existieren Komponenten zur HW/SW-Cosimulation, inklusive Mikroprozessoren und wichtiger Peripherie.
- Die Visualisierung bietet unter anderem einen Glow-Mode, Tool/Tips und die Darstellung von Hazards.

Die grobe Architektur von HADES ist in Abbildung 12 dargestellt. Das jeweilige Simulationsexperiment besteht aus einer beliebigen Anzahl von Simulations-Komponenten (SimObjects), deren logische Verbindung über separate Objekte (Signale) erfolgt. Die Initialisierung und Interaktion der Komponenten erfolgt über einen ereignisbasierten Simulationskern. Das gesamte System ist in Java implementiert. Dies erlaubt es, die Interaktion zwischen Simulator und den Simulationskomponenten zunächst nur über zwei sehr einfache Java-Interfaces zu definieren, die jederzeit durch Definition weiterer Unterklassen verfeinert werden können.

Für den Anwender unmittelbar sichtbar ist die Benutzeroberfläche mit dem Grafikeditor und der Simulationssteuerung. Als Beispiel zeigt Abbildung 11 den Grafikeditor mit dem RT-Modell eines einfachen Mikroprozessors. Bei Bedarf können weitere Fenster geöffnet werden, etwa für die Impulsdiagramme oder die komponentenspezifischen Editoren (vergleiche Abbildung 23 auf Seite 84).

Der Grafikeditor verfügt über eine Referenz auf die zu simulierenden Komponenten, die von einem `Design`-Objekt verwaltet werden. Die Verbindungen zwischen den einzelnen Simulationsmodellen werden über Signale hergestellt.

Einen Ausschnitt aus der Klassenhierarchie der HADES Simulationsmodelle zeigt Abbildung 13. Die Klasse `SimObject` dient als Basisklasse für alle mit HADES simulierbare Kompenten. Die Klasse `Design` enthält Referenzen auf beliebig viele weitere

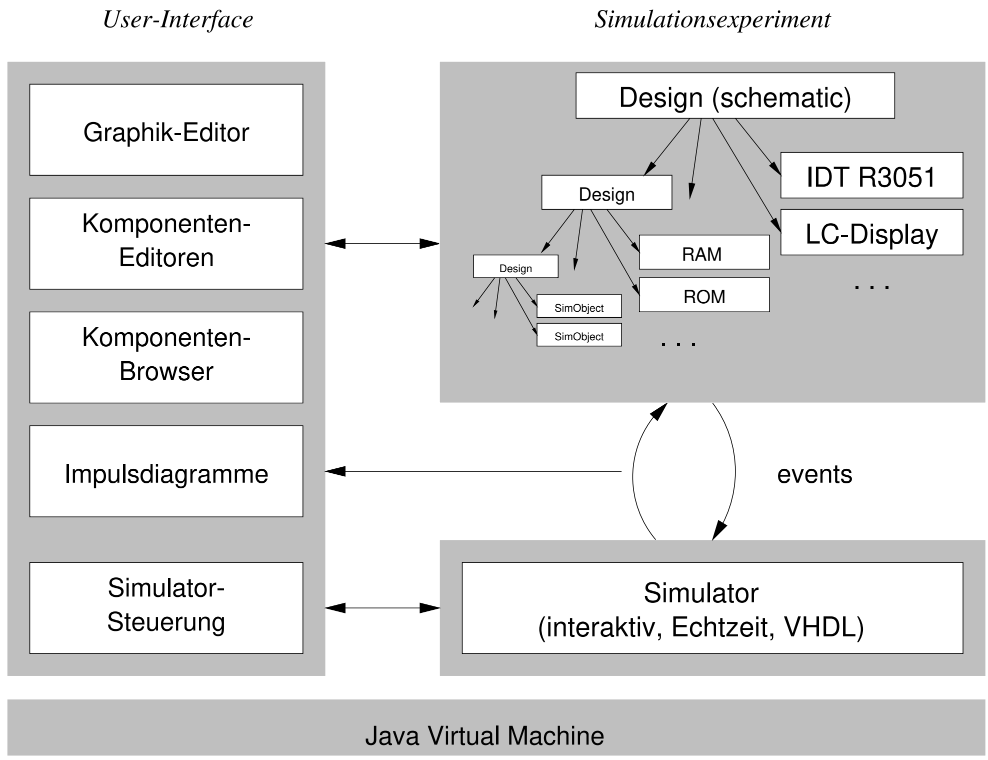

Abbildung 12: Überblick über die Hauptblockebene der HADES-Architektur

`SimObject`-Komponenten und dient als Abstraktion für ein konkretes Simulationsexperiment. Da ein `Design` wiederum andere `Design`-Komponenten enthalten kann, sind hierarchische Modelle beliebiger Tiefe möglich. Konkrete Simulationsmodelle, etwa für ein XOR-Gatter, werden als Unterklassen von `SimObject` realisiert. Dabei sind weitere Zwischenstufen in der Klassenhierarchie möglich. So repräsentiert die Klasse `GenericGate` die gemeinsamen Eigenschaften der einzelnen Logikgatter.

Über die elementaren Gatter und Komplexgatter hinaus umfassen die Modellbibliotheken derzeit auch eine Vielfalt von Modellen für Komponenten auf der Register-Transfer-Ebene, etwa für Addierer, Multiplexer oder Speicher.

Ein weiterer wichtiger Punkt ist die Unterstützung einer voll *interaktiven Simulation*. Simulationsmodelle und Signale können während der Simulation hinzugefügt, gelöscht, oder editiert werden. Zusätzlich erlaubt die Script-Anbindung [Meyer 2000] [Hugunin 97] den vollen interaktiven Zugriff auf alle internen Datenstrukturen der Simulationsumgebung und aller Simulationsmodelle. Eine einfache Bedienungsanleitung zu HADES ist in [Hendrich 97] zu finden.

4.4 Verbindung zwischen HADES und Simulationsmodellen

Alle externen Modelle müssen von `hades.simulator.SimObject` abgeleitet werden und einige der dort definierten Methoden überschreiben. Zusätzlich existiert noch das Java-Interface `hades.simulator.Wakeable`, das einem Modell die zusätzliche Fähigkeit verleiht, sich ohne ein Simulationsereignis aufwecken zu lassen. In einer

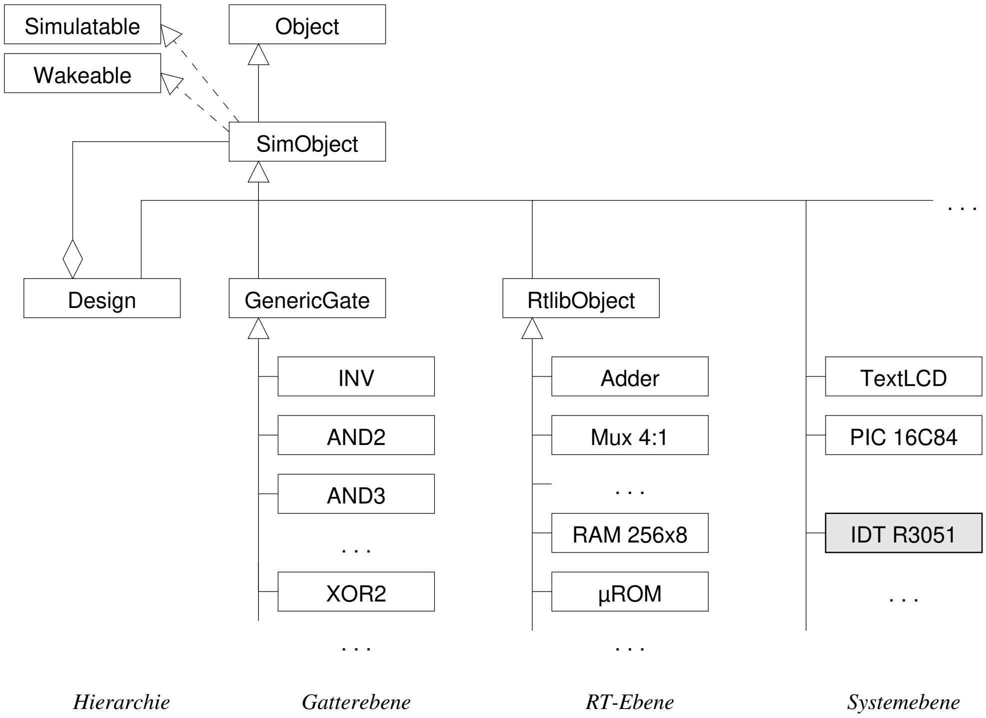

Abbildung 13: Ausschnitt aus der HADES-SimObject Klassenhierarchie

Übersicht folgen jetzt alle für ein externes Modell wichtigen `SimObject`-Methoden, die überschrieben oder speziell behandelt werden müssen.

Konstruktor: Hier muß als erste Methode `super()` (das ist der Konstruktor vom `SimObject`) aufgerufen werden. Im Konstruktor werden alle Ein- und Ausgänge, die Verbindungen zu HADES, definiert. Dazu existiert im `SimObject` ein Array mit Elementen der Klasse `Port` und dem Namen `ports`. Diesem Array werden alle Ports, die das Modell benötigt, zugewiesen.

getSymbolResourceName(): Diese Methode liefert den Namen des Symbols. Im Normalfall ist das der Klassenname mit der Erweiterung „sym".

toString(): Der vollständige Name des Objekts wird von `toString()` geliefert.

write(): Soll ein HADES-Design gespeichert werden, hat jedes Modell die Möglichkeit, eigene spezielle Konfigurationsdaten in die zu schreibende Datei zu integrieren. Diese Konfigurationsdaten können dann bei erneuter Instanziierung von `initialize()` ausgewertet werden.

initialize(): Falls das Objekt eine spezielle Initialisierung benötigt, kann diese hier ausgeführt werden. Hier können zum Beispiel weitere Dateien für die Konfiguration des Objektes eingelesen werden. `initialize()` werden genau die Daten übergeben, die von `write()` erzeugt werden.

elaborate(): Für den Start der Simulation wird `elaborate()` aufgerufen. An dieser Stelle wird das Objekt für den Simulationsablauf selbst vorbereitet. Hier wäre der richtige Ort, initiale `Wakeup`-Ereignisse zu erzeugen (siehe Erklärung des `Wakeable`-Interfaces weiter unten).

evaluate(): Wenn das Simulationsmodell ein Ereignis an einem der Ports des Modells feststellt, dann wird `evaluate()` aufgerufen. `evaluate()` muß dann alle Eingänge abfragen, auf mögliche Veränderungen reagieren und entsprechend interne Vorgänge anstoßen und eventuell neue Ereignisse an den Ausgangsports erzeugen. Zur Erzeugung von neuen Ereignissen existiert die Methode `getSimulator().scheduleEvent()`, die für einen bestimmten Port dessen nächsten Wert für einen bestimmten Zeitpunkt festlegen kann.

configure(): Falls es sich bei dem implementierten Modell um eine komplexere Anwendung handelt, die vom Benutzer konfiguriert werden kann, dann sollte das Modell die Methode `configure()` implementieren und bei dessen Aufruf dafür sorgen, daß eine Benutzerschnittstelle zur Konfiguration des Modells sichtbar wird.

Das `hades.simulator.Wakeable`-Interface definiert nur eine einzige zu implementierende Methode. Diese trägt den Namen `wakeup()` und wird aufgerufen, sobald ein Objekt aufgeweckt werden soll. Falls innerhalb eines zu simulierenden Bauteils interne zeitabhängige Vorgänge ablaufen, die nicht über die Eingänge gesteuert werden, muß diese Methode verwendet werden. Ein Beispiel für ein Bauteil, das mit dem `Wakeable`-Interface ausgestattet werden muß, ist ein Takterzeuger, der zum Generieren eines Taktsignals in definierten Zeitabständen aufgeweckt wird.

5 Modellbildung

5.1 Grundlagen

Die gewählte Sicht der Modellbildung orientiert sich hier unter anderem an der in [Wolfinger 99] für die Modellierung von Kommunikations- und Rechnernetzen vorgeschlagenen Vorgehensweise einer Systemzerlegung und Auftragsmodellierung. In Abbildung 14 wird dargestellt, wie ein reales System in ein modelliertes System zu überführen ist, das sich im Rahmen der Anforderungen für ein MIPS-Simulationsmodell eignet.

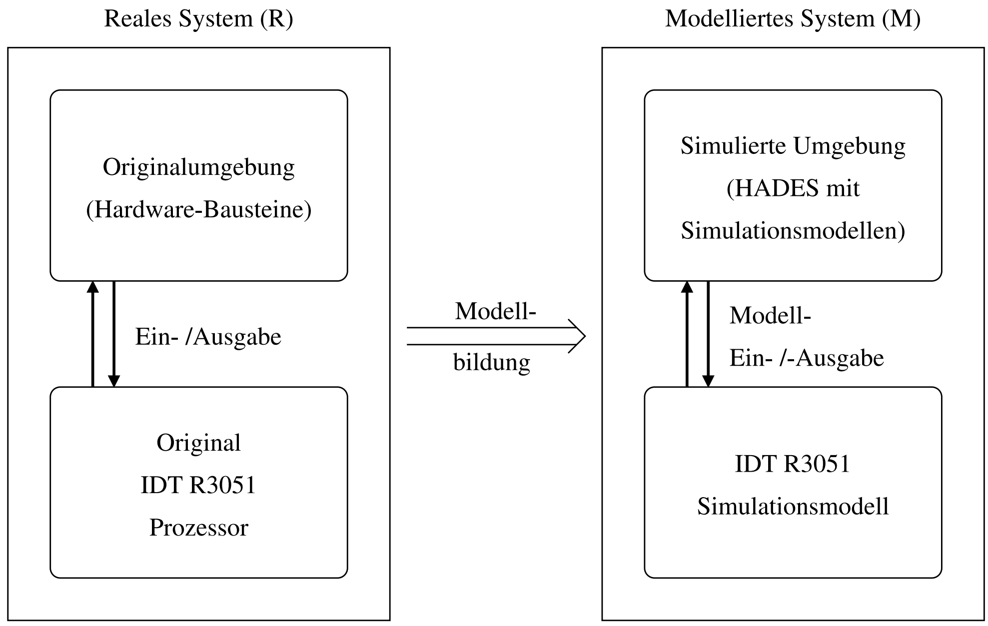

Abbildung 14: Überführung eines realen Systems in ein modelliertes System unter Berücksichtigung von Ein-/Ausgabe

Das reale System besteht aus dem Prozessor und dessen Umgebung. Die Umgebung ist in diesem Falle eine Hardware-Schaltung, die direkt mit dem Prozessor zusammenarbeitet. Der Prozessor besitzt ein zu großen Teilen wohldefiniertes Verhalten. Die Umgebung kann sich für einzelne Simulationen ändern. Die Interaktionen zwischen Prozessor und Umgebung finden auf vielen Wegen statt, deren wichtigste nun im Folgenden vorgestellt werden.

An erster Stelle sei der elektrische Weg genannt, der sich durch die direkt elektrischen Strom leitenden Verbindungen zwischen Prozessor und Umgebung ergibt. In der Abbildung werden diese Verbindungen durch “Ein-/Ausgabe“ gekennzeichnet. Ein weiterer Weg ist die Temperatur: Die Umgebung kann sich so weit erwärmen, daß der Prozessor nur noch eingeschränkt oder gar nicht mehr spezifikationsgemäß arbeitet. Auch der umgekehrte Fall ist denkbar, wenn der Prozessor durch ungenügende Kühlung die Umgebung bis zur Funktionsuntüchtigkeit erwärmt. Innerhalb einer elektronischen Schaltung ist noch der dritte Weg der elektromagnetischen Strahlung als Interaktionsweg zwischen Prozessor und Umgebung denkbar. Auch dieser Weg kann zu unspezifikationsgemäßem Verhalten führen.

Weitere Wege wie zum Beispiel atomare Strahlung sind denkbar und werden hier nicht weiter verfolgt. Alle bis auf den zuerst genannten Weg werden im Folgenden nicht weiter betrachtet, da sie bei einer korrekt aufgebauten Schaltung nicht auftreten sollten und nur mit großem Aufwand simulierbar sind. Der elekrische Weg ist innerhalb einer elektrischen Schaltung im Normalfall der einzige Weg, auf dem sinnvolle Daten zwischen Prozessor und Umgebung ausgetauscht werden.

Das modellierte System bildet alle Elemente des realen Systems durch entsprechende Simulationsmodelle nach. Der Prozessor wird durch das MIPS-Simulationsmodell abgebildet. Die Umgebung wird durch HADES und weitere, über HADES miteinander verbundene Simulationsmodelle dargestellt. Die “Modell-Ein-/-Ausgabe“ (die Verbindung zwischen HADES und dem MIPS-Simulationsmodell) soll sich auf abstrakter Ebene möglichst konform zur Ein-/Ausgabe des realen Vorbilds verhalten. Die Abstraktion von den realen elektrischen Signalen muß noch genauer betrachtet werden und ist Thema des Abschnitts 5.2.

Die Ein-/Ausgabe des Prozessors besteht zu einem Großteil aus den Maschinensprachebefehlen, die der Prozessor ausführen soll. Die Abbildung 15 zeigt die Überführung des realen Systems in ein Modellsystem, diesmal unter besonderer Berücksichtigung der Maschinenprogramme.

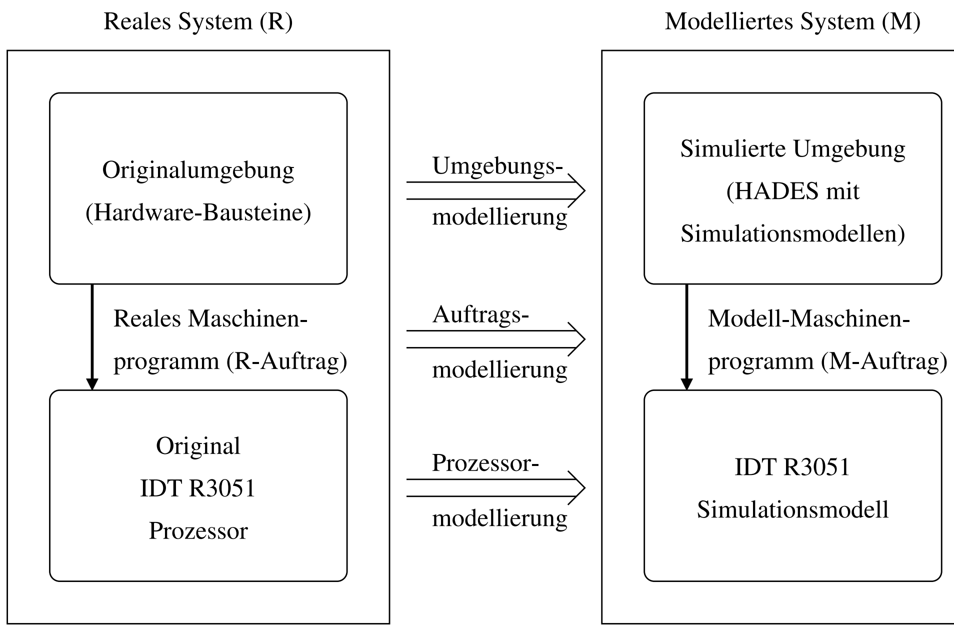

Abbildung 15: Überführung eines realen Systems in ein modelliertes System unter Berücksichtigung der Maschinensprache-Schnittstelle

Im realen System bekommt der Prozessor ein bestimmtes Maschinenprogramm (einen “R-Auftrag“) zur Ausführung übergeben. Das Ziel der Modellbildung muß sein, daß sich das reale Maschinenprogramm in ein Modell-Maschinenprogramm (“M-Auftrag“) überführen läßt. Dieser M-Auftrag kann dann durch das MIPS-Simulationsmodell ausgeführt werden. War die Modellbildung und Implementation erfolgreich, so liefert das Simulationsmodell nach Bearbeitung eines M-Auftrags ein Ergebnis, das mit dem Ergebnis der Bearbeitung eines R-Auftrages durch einen realen Prozessor vergleichbar ist.

Der bestmögliche Fall der Überführungsrelation zwischen R-Auftrag und M-Auftrag ist die Identität, da in diesem Fall das reale Maschinenprgramm auch direkt innerhalb des Simulationsmodells eingesetzt werden kann. Wenn der R-Auftrag direkt innerhalb des Simulationsmodells eingesetzt werden kann, ergibt sich eine einfachere Entwicklungsphase für die Software, die innerhalb des Modells und real eingesetzt werden soll.

5.2 Vorgaben durch HADES

Vorgegeben durch HADES soll das MIPS-Simulationsmodell nur Signale nach dem IEEE-Standard 1164 [IEEE 93b] entgegennehmen können. Diese Signale können entweder die Werte logisch 0 oder logisch 1 oder verschiedene andere Werte zur Repräsentation von Ausnahmezuständen annehmen (siehe Tabelle 12).

Wert	Bedeutung
0	logisch 0
1	logisch 1
U	undefiniert
X	Kurzschluß
Z	offen
W	schwach undefiniert
L	schwach logisch 0
H	schwach logisch 1
D	unbekannt / gleichgültig

Tabelle 12: Von HADES benutzte Logikwerte nach IEEE 1164 [IEEE 93b]

Es werden nur digitale Werte berücksichtigt und von wirklichen Strömen oder Spannungen abstrahiert. Diese Generalisierung stellt eine gängige Vorgehensweise dar, da prozessorintern nur die logischen Werte 0 und 1 interessieren.

5.3 Das Simulationsmodell

Ausgehend von den in den beiden vorigen Abschnitten genannten Voraussetzungen bildet das MIPS-Simulationsmodell die beiden durch die MIPS-R3000-Architektur (Maschinensprache) und den IDT-R3051-Prozessor (Ein- und Ausgabe-Anschlüsse) vorgegebenen Schnittstellen detailgetreu ab. Die Maschinensprache ist mit all ihren Befehlen komplett verfügbar. Das MIPS-Simulationsmodell verhält sich während der Ausführung der Maschinenbefehle so wie die reale Hardware, soweit das aus der vorhandenen Dokumentation ersichtlich ist (die internen Vorgänge sind im Detail teilweise nur zu erraten). Die Ein- und Ausgabe-Anschlüsse werden als Schnittstelle zu HADES möglichst genau nachgebildet.

In Abschnitt 3.1 wird der Aufbau des IDT R3051 beschrieben. Abbildung 5 auf Seite 23 zeigt ein Blockschaltbild, das die wesentlichen Komponenten des Prozessors in einen Zusammenhang setzt. Für die Modellbildung wird nun, ausgehend von dieser

Abbildung, ein abstrakter Aufbau des Simulationsmodells entworfen, der einerseits dem Vorbild so weit wie möglich entsprechen und andererseits einfach implementierbar sein soll. Die Abbildung 16 zeigt eine Vergröberung des Blockschaltbilds, das als Grundlage der Modellbildung dienen soll.

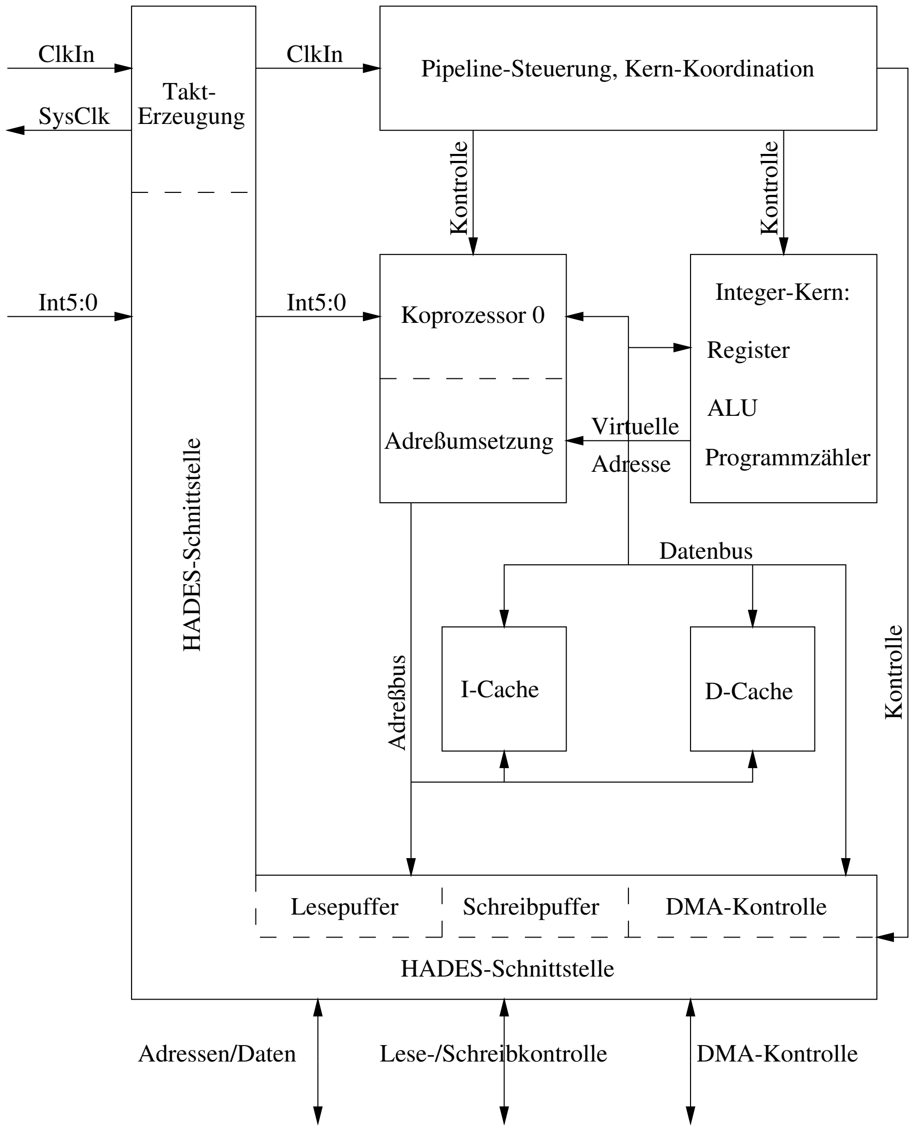

Abbildung 16: Vergröberung des Blockschaltbilds als Grundlage der Modellbildung

Hier werden alle genannten Komponenten des Prozessors durch entsprechende Funktionsblöcke ersetzt. Die sich daraus ergebende Aufgabe wird dann sein, alle diese Funktionsblöcke durch äquivalente Java-Klassen nachzubilden und entsprechend zu verbinden. In Abschnitt 6.2 wird näher darauf eingegangen, dort wird das hier entwickelte Modell als Java-Implementation konkretisiert.

Im Abschnitt 2.4 wird ausführlich auf den Aufbau der im IDT R3051 verwendeten Pipeline eingegangen. Einige der dort beschriebenen Aktionen finden gleichzeitig statt.

Innerhalb des Simulationsmodells sollen solche Vorgänge serialisiert werden, damit die Implementation keine parallelen Prozesse verwenden muß. Tabelle 13 zeigt die resultierende Befehls-Pipeline des Simulationsmodells.

Befehl 1	Befehl 2	Befehl 3	Befehl 4
I-Adresse wandeln			
I-Speicherzugriff beginnen			
Auf I-Ergebnis warten	I-Adresse wandeln		
I-Leseergebnis dekodieren	I-Speicherzugriff beginnen		
ALU-Berechnung	Auf I-Ergebnis warten	I-Adresse wandeln	
D-Adresse wandeln	I-Leseergebnis dekodieren	I-Speicherzugriff beginnen	
D-Speicherzugriff beginnen	ALU-Berechnung	Auf I-Ergebnis warten	I-Adresse wandeln
Auf D-Ergebnis warten	D-Adresse wandeln	I-Leseergebnis dekodieren	I-Speicherzugriff beginnen
D-Leseergebnis in Register	D-Speicherzugriff beginnen	ALU-Berechnung	Auf I-Ergebnis warten
	Auf D-Ergebnis warten	D-Adresse wandeln	I-Leseergebnis dekodieren
	D-Leseergebnis in Register	D-Speicherzugriff beginnen	ALU-Berechnung
		Auf D-Ergebnis warten	D-Adresse wandeln
		D-Leseergebnis in Register	D-Speicherzugriff beginnen
			Auf D-Ergebnis warten
			D-Leseergebnis in Register

Tabelle 13: Befehls-Pipeline des Simulationsmodells

Die in diesem Abschnitt benutzten Methoden der Modellbildung orientieren sich an [Jessen 87].

5.4 Veränderungen gegenüber dem Vorbild

Ausgehend von den Wünschen nach Nachbildung der Maschinensprache- und der Ein-/Ausgabe-Schnittstelle (nach IEEE 1164) werden einige Veränderungen (die meisten sind Vereinfachungen) vorgenommen, um ein hinreichend einfach implementierbares Modell zu erreichen. Im Folgenden werden die verschiedenen vorgenommenen

Veränderungen einzeln aufgezählt und deren Berechtigung begründet.

5.4.1 Asynchrone Befehle

Im IDT-Original werden Multiplikations- und Divisionsbefehle asynchron, das heißt parallel zu anderen Instruktionen ausgeführt, da diese mehrere Takte erfordern. Das MIPS-Simulationsmodell führt solche Befehle stattdessen entsprechend zu den normalen ALU-Befehlen synchron direkt innerhalb der Pipeline aus. Das Ergebnis der Berechnung steht somit direkt nach der ALU-Operation der Pipeline zur Verfügung. Diese Vereinfachung ist zulässig, da die MIPS-Architektur nicht vorgibt, nach welcher Anzahl von Taktsignalen die Ergebnisse vorliegen sollen und je nach Hersteller und Modell sehr unterschiedliche Ausführungszeiten möglich sind. Da die Befehlsausführung blockiert, falls vor dem Ende der Berechnung ein anderer Befehl das Berechnungsergebnis benötigt, kann statt dessen direkt synchron das Ergebnis bereitgestellt werden.

Diese Veränderung führt zu einem einfacheren Aufbau der Befehlsausführungseinheiten, so daß der Komplexitätsgrad gesenkt und damit die Ausführungsgeschwindigkeit des Simulationsmodells erhöht werden kann. An dieser Stelle wird die strenge Kompatibilität zum Hardware-Vorbild IDT R3051 aufgegeben. Die MIPS-R3000-Archtitektur läßt diese Vereinfachung aber zu, so daß die Maschinensprache-Schnittstelle unbeeinflußt bleibt.

5.4.2 nop-Befehl

Wie bereits in Abschnitt 2.3.2 dargestellt, enthält die MIPS-Maschinensprache im Gegensatz zu den meisten anderen Maschinensprachen keinen expliziten `nop`-Befehl. Das MIPS-Simulationsmodell verhält sich an dieser Stelle anders, denn dort ist ein eigenständiger `nop`-Befehl implementiert. Das Befehlswort mit dem Wert 0 wird nicht dem Befehlsobjekt `ShiftLeftLogicalInstr`, sondern dem Befehlsobjekt `NopInstr` zugeordnet. Im MIPS-Befehlssatz gibt es keinen offiziellen `nop`-Befehl, aber die virtuelle Maschine, die von jedem MIPS-Assembler geboten wird, unterstützt diesen Befehl (als `sll $0,$0,$0`, daraus ergibt sich dann das Befehlswort 0).

Der Grund liegt hauptsächlich in der Darstellung: Ohne die Einführung eines `nop`-Befehls würde die Disassemblierung wesentlich schwerer lesbar. Diese Veränderung gegenüber der MIPS-Architektur führt letztendlich nur dazu, daß das Befehlswort 0 einen anderen Namen besitzt. Die Verhaltensweise des Simulationsmodells ändert sich durch diese Befehlsumbennung nicht, deshalb darf diese Veränderung vorgenommen werden.

5.4.3 syscall-Befehl

Für die Fehlersuche und Geschwindigkeitsmessungen besitzt die Implementation des `syscall`-Befehls einige Besonderheiten, die nicht im Original vorhanden sind. Diese geben besondere Werte auf der Standardausgabe aus oder unterbrechen den Programmablauf.

Solche Funktionen sind sinnvoll, solange keine Bibliotheksfunktionen und kein Betriebssystem vorhanden sind. Falls diese Funktionalitäten zu Inkompatibilitäten mit auszuführenden Programmen führen sollten, können sie einfach deaktiviert werden, so daß das Simulationsmodell daraufhin eine zum Original vollkommen vergleichbare Funktion des `syscall`-Befehls bietet.

5.4.4 Koprozessoren

Dem MIPS-Simulationsmodell liegt der IDT R3051 ohne MMU zugrunde. Dementsprechend wird nur der Koprozessor 0 ohne MMU implementiert. Eine FPU ist nicht vorhanden, diese ist nur Teil des (ansonsten kompatiblen) IDT R3081. Hierbei handelt es sich nicht um eine Veränderung gegenüber dem Vorbild. Da jedoch viele andere Implementationen der MIPS-R3000-Architektur eine FPU oder eine MMU enthalten, wird dieser Punkt hier erwähnt.

5.4.5 Instruktions-Cache

Im IDT R3051 ist der Instruktions-Cache mit einer Cache-Zeilengröße von vier Worten implementiert. Normalerweise wird der Instruktions-Cache nur gelesen. Wird der externe Hauptspeicher per Burst-Zugriff gelesen, so werden vier direkt aufeinanderfolgende Worte eingelesen, die gemeinsam in eine Cache-Zeile übertragen werden können. Pro Cache-Zeile wird einmal die Adresse der hier enthaltenen Daten gespeichert, somit vereinfacht sich der Aufbau gegenüber eines Cache, der für jedes einzelne Datenwort die Adresse speichert. Außerdem kann der interne Datenpfad zwischen Burst-Speicherschnittstelle und dem Instruktions-Cache die vierfache Wortbreite betragen.

Im MIPS-Simulationsmodell ist die Struktur von Instruktions- und Daten-Cache gleich, die Cache-Zeilengröße beträgt ein Wort. Diese Vereinfachung führt zu keinen wesentlichen Leistungsunterschieden, nur das Zugriffsverhalten ändert sich durch die unterschiedliche Organisation. Innerhalb des Simulationsmodells werden nur Datenpfade mit 32 Bit Breite benutzt. Durch eine geänderte Implementation des Instruktions-Caches ließe sich dieser Unterschied zwischen Simulationsmodell und Realität aufheben. Dadurch würde sich keine wesentliche Änderung der Ausführungsgeschwindigkeit des Simulationsmodells ergeben.

5.4.6 Byte-Orientierung

Alle Prozessoren der MIPS-Architektur sind in der Lage, mit unterschiedlicher Byte-Orientierung zu arbeiten. Die Simulation beherrscht nur die Byte-Orientierung nach dem *Big-Endian*-Schema und ein Wechsel der Byte-Orientierung zwischen Kernel- und User-Modus ist nicht vorgesehen.

Da eine Änderung der Byte-Orientierung in der Original-Hardware nur aus Kompatibilitätsgründen vorhanden ist, kann die Simulation gut ohne diese Eigenschaft auskommen. Sollte aber die Benutzung von unterschiedlicher Byte-Orientierung einmal für eine Simulationsaufgabe wichtig sein, so läßt sich eine umschaltbare Byte-Orientierung nachrüsten.

5.4.7 Einschalt-Initialisierung

Wie schon im Abschnitt 3.6.6 beschrieben, erhalten die IDT-Prozessoren beim Einschalten einige Konfigurationsmerkmale über bestimmte Eingangssignale mitgeteilt. Da es sich dabei um sehr spezielle Dinge (wie die im Simulationsmodell nicht einstellbare Byte-Orientierung) handelt, reagiert das MIPS-Simulationsmodell nicht auf die Einschalt-Initialisierung. Sollte eine änderbare Byte-Orientierung nachgerüstet werden, so müßte auch die Einschalt-Initialisierung in das Simulationsmodell integriert werden.

5.5 Zukünftige Erweiterungsmöglichkeiten

Der begrenzte Zeitrahmen ließ es nicht zu, alle simulationstechnisch möglichen Merkmale des IDT R3051 wirklich innerhalb des MIPS-Simulationsmodells zu implementieren. Alle der genannten Vereinfachungen lassen sich aber bei Bedarf nachrüsten. Für einige dieser Merkmale sind bereits die Schnittstellen vorhanden (MMU) oder Teilaspekte sind bereits implementiert (Byte-Orientierung).

6 Software-Architektur

6.1 Einbettung in HADES

Als einfacher Schichtenaufbau stellt Abbildung 17 dar, wie das MIPS-Simulationsmodell mit HADES zusammenarbeitet. In der Abbildung werden drei verschiedene Modelle abgebildet, die über HADES miteinander verbunden sind und gemeinsam simuliert werden.

MIPS-Simulationsmodell:

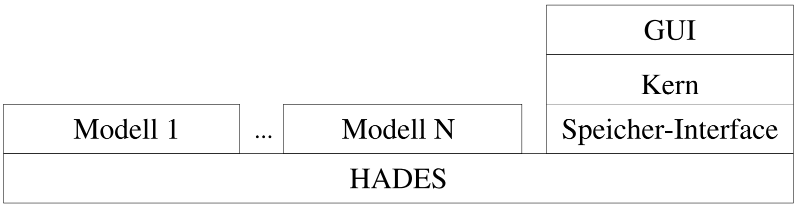

Abbildung 17: Einbettung des Simulationsmodells in HADES

Das MIPS-Simulationsmodell wird in der Abbildung in drei Schichten dargestellt, um die drei beteiligten Funktionsgruppen zu verdeutlichen:

Kern: Im Kern findet die Simulation der MIPS-Maschinensprache statt. Hier wird die Pipeline nachgebildet und hier werden alle Befehle ausgeführt. Auch der Koprozessor 0 ist dieser Funktionsgruppe zuzuordnen. Der Abschnitt 6.3 beschreibt den Aufbau des Kerns genauer.

Speicherschnittstelle: Die Speicherschnittstelle stellt die Simulation der Hardware-Schnittstelle des IDT R3051 dar. Dabei muß insbesondere die Speicherschnittstelle dem anfangs definierten Ziel der Flexibilität genügen, so daß das MIPS-Simulationsmodell an unterschiedliche Anforderungen angepaßt werden kann. Der Aufbau der Speicherschnittstelle muß aufgrund seiner Komplexität ausführlich in Abschnitt 6.4 vorgestellt werden.

GUI: Als *Graphical User Interface*, also als Benutzerschnittstelle stehen einige Klassen zur Verfügung, die die internen Abläufe des MIPS-Simulationsmodells schnell und übersichtlich darstellen. Die Benutzerschnittstelle setzt sich aus verschiedenen Klassen für die einzelnen Darstellungsaspekte zusammen. Architektonisch interessant sind einerseits die Pipeline-Anzeige und andererseits die verschiedenen Möglichkeiten, Speicher darzustellen. Abschnitt 6.5 beschreibt den Aufbau dieser Funktionen.

6.2 Architektur ausgehend vom Blockschaltbild

Im Abschnitt 5.3 wurde auf Seite 48 die Abbildung 16 vorgestellt, die als Grundlage der Modellbildung dienen sollte. Nun wird der dort abstrakt vorgestelle Aufbau des Simulationsmodells konkretisiert. Die Abbildung 18 zeigt die Zusammenarbeit aller wichtigen am Kern beteiligten Java-Klassen und deren Verbindungen untereinander.

Dabei ist der Aufbau der Abbildung weiterhin der Vorlage, Abbildung 5 auf Seite 23, nachempfunden, so daß auf den ersten Blick klar sein sollte, welche Komponenten welche Aufgaben erfüllen.

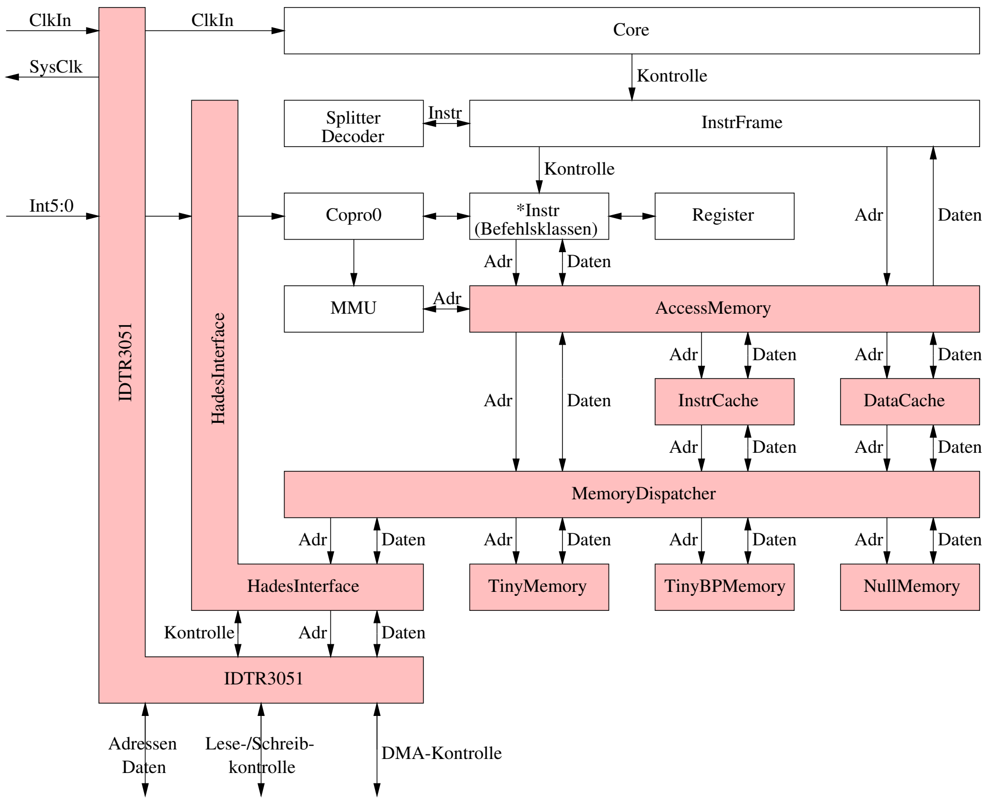

Abbildung 18: Blockschaltbild der Simulationsmodell-Architektur

Im Folgenden werden die drei im vorigen Abschnitt vorgestellten Teilbereiche Kern, Speicherschnittstelle und GUI mit den beteiligten Klassen vorgestellt. Die Kern-Klassen sind in Abbildung 18 weiß dargestellt. Klassen, die der Speicherschnittstelle angehören erscheinen mit grauem Hintergrund. Die GUI-Klassen werden in der Abbildung 18 nicht weiter erwähnt, da sie für die eigentliche Funktionalität des Simulationsmodells keine Bedeutung haben.

6.3 Die wichtigsten Klassen des Kerns

Der Kern des MIPS-Simulationsmodells besteht aus wenigen Hauptkomponenten und vielen Befehlsklassen. Da die Pipeline das Herzstück der Befehlsausführung darstellt, wird diese hier auch als Strukturierung der Klassenvorstellung verwendet.

Die Dachklasse des Kerns ist die Klasse `Core`. Sie enthält die Initialisierung aller zum Kern gehörigen Klassen und ruft die einzelnen Aktionen der Pipeline-Ausführung auf. Für die Bearbeitung eines Befehls ist `InstrFrame` zuständig. Diese Klasse bietet alle Mechanismen, um einen Befehl nacheinander zu laden, zu dekodieren und auszuführen. Dazu benutzt `InstrFrame` verschiedene weitere Klassen, die spezielle Dienste zur Verfügung stellen. Zwei solcher Klassen sind der `Splitter` und der

`Decoder`, die ein Befehlswort in seine Bestandteile aufteilen und dekodieren. Als Ergebnis der Dekodierung steht dann ein Befehlsobjekt zur Verfügung, das eine Instanz einer der Befehlsklassen darstellt. Die verschiedenen Befehlsklassen mit ihrer Hierarchie werden im Abschnitt 6.3.2 vorgestellt. Da alle Befehlsklassen mit der Endung `Instr` versehen sind, wird hier im Folgenden oft von `*Instr` gesprochen, womit alle Befehlsklassen gemeint sein sollen.

Sobald ein Befehlsobjekt mit der Ausführung eines Befehls beginnt, greift es auf die `Registers`, den `Copro0` und die Speicherschnittstelle zu. Muß während der Befehlsausführung eine Exception erzeugt werden, so ruft das Befehlsobjekt den `ExceptionGenerator` auf. Der `ResetHandler` kümmert sich um die Bearbeitung von Resets und setzt alle Kern-Objekte in den Einschaltzustand zurück. Die Zusammenarbeit der für die Befehlsausführung verantwortlichen Klassen wird in Abbildung 19 grafisch dargestellt.

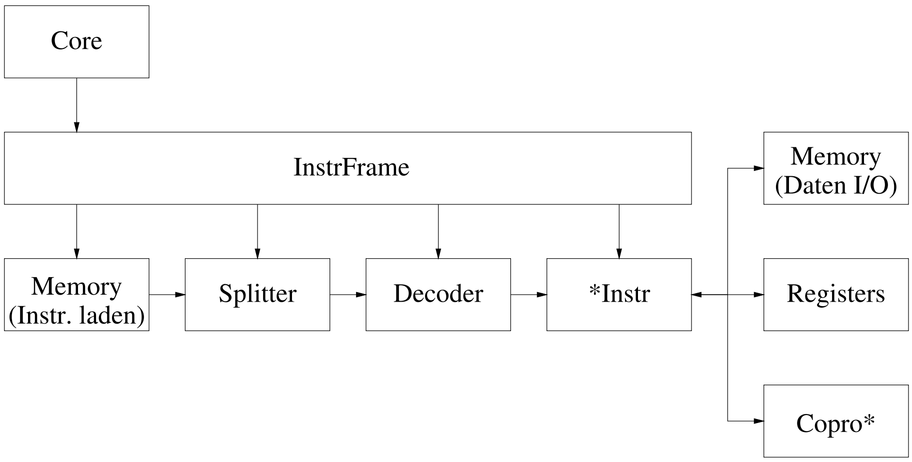

Abbildung 19: Ablauf der Befehlsausführung und daran beteiligte Klassen

6.3.1 Implementation der Pipeline

Ein `InstrFrame`-Objekt entspricht immer einem Befehl während seiner Ausführung. Im Gegensatz zur realen Hardware existiert nicht genau eine Pipeline, die bis zu fünf verschiedene Befehle gleichzeitig aufnimmt, sondern fünf Ausführungseinheiten mit jeweils einem Befehl, die sich, in einander verschachtelt, genauso verhalten wie die reale Pipeline. Die Pipeline enthält, wie schon in Abschnitt 5.3 dargestellt, neun verschiedene Aktionen, von denen immer zwei zu einem Pipeline-Schritt zusammengefaßt werden. Jede Aktion der Pipeline ist innerhalb von `InstrFrame` als Methode angelegt. Diese Aktionen werden dann von der `Core`-Klasse aus verschachtelt aufgerufen.

6.3.2 Befehlsklassenhierarchie

Für jeden MIPS-R3000-Maschinenbefehl gibt es eine entsprechende Befehlsklasse. Um den Aufbau der Klassen zu vereinfachen und oft wiederkehrende Strukturen einmal an zentraler Stelle zu definieren, existiert eine Befehlsklassenhierarchie, die zuerst Basisklassen definiert, von denen dann die eigentlichen Befehlsklassen abgeleitet sind.

So definiert die Klasse `BaseInstr` das Basisverhalten eines Befehls, der nichts bewirkt. Von dieser Klasse sind alle weiteren Befehlsbasis- und Befehlsklassen abgeleitet. So existieren `AluBaseInstr`, `BranchBaseInstr`, `LoadBaseInstr` und weitere Befehlsbasisklassen, die dann dazu führen, daß die eigentlichen Befehlsklassen (deren Namensgebung direkt an die Namen der Maschinenbefehle angelehnt ist) nur genau die Funktionalität enthalten, die sie von anderen, ähnlichen Befehlen unterscheidet. Ein kleiner Ausschnitt der Hierarchie der Befehlsklassenvererbung ist in Abbildung 20 zu erkennen.

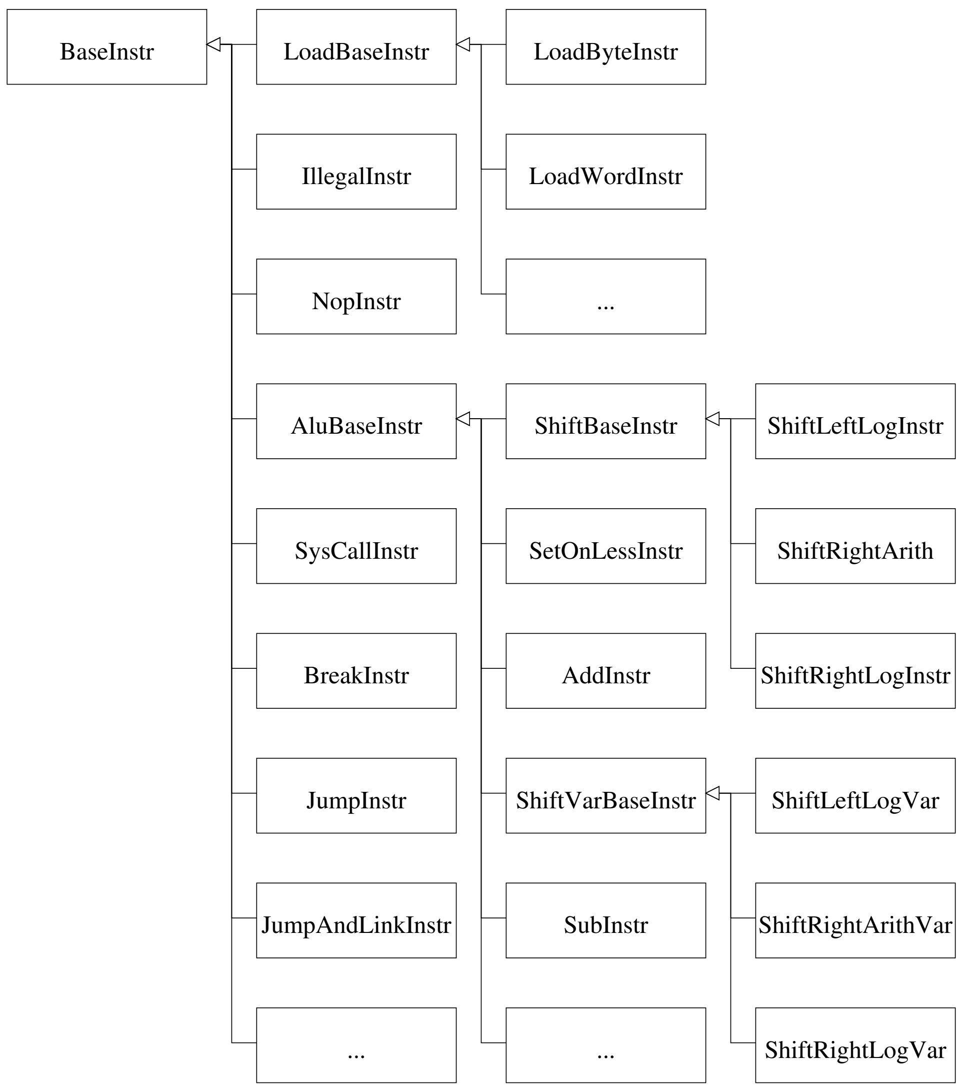

Abbildung 20: Ausschnitt aus der Hierarchie der Befehlsklassenvererbung

6.4 Die Speicherhierarchie

Die Speicherschnittstelle kann als Schichtenmodell verschiedener Klassen dargestellt werden. Abbildung 21 bietet den Überblick über die wichtigsten Klassen der Speicherhierarchie.

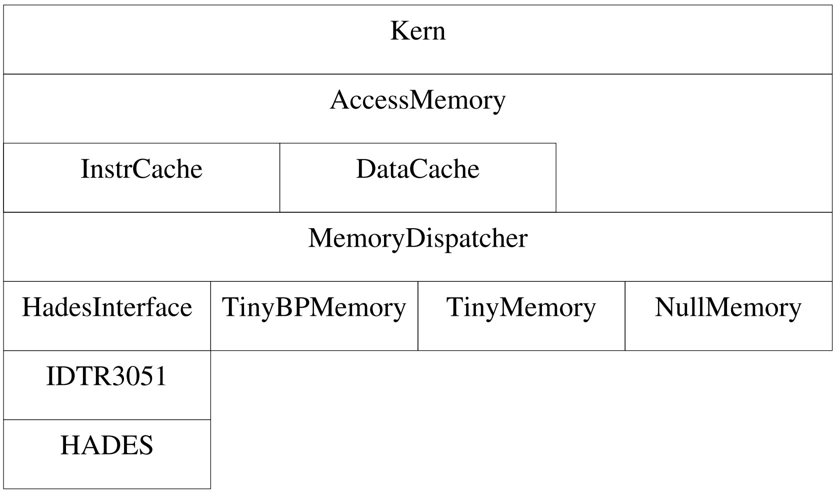

Abbildung 21: Speicherschnittstelle als Schichtenmodell

Eine kurze Übersicht der beteiligten Schichten:

Kern: Diese Schicht stellt genau den im vorigen Abschnitt beschriebenen Kern dar.

AccessMemory: Die Schnittstelle zwischen Kern und der Speicherhierarchie bildet die Klasse `AccessMemory`. Hier werden die Zugriffe auf den Speicher vereinheitlicht und die Konfiguration der darunter liegenden Schichten vorgenommen.

Caches: Diese Schicht beinhaltet die beiden Cache-Klassen `InstrCache` und `DataCache`. In der Abbildung 21 erstreckt sich `AccessMemory` auch über diese Schicht, um zu zeigen, daß AccessMemory zum Teil auch direkt auf die unterhalb der Caches liegende Schicht zugreift.

MemoryDispatcher: In dieser Schicht werden die Speicherzugriffe auf verschiedene Memory-Objekte verteilt.

HadesInterface, Memory-Objekte: Die Memory-Objekte, die einen Speicher implementieren, und das `HadesInterface`, das sich wie ein Memory-Objekt verhält, sind unterhalb des `MemoryDispatcher` angesiedelt und stellen aus der Sicht der oberen Schichten die eigentlichen Speicher dar.

IDTR3051: Zur Verbindung des `HadesInterface` mit HADES selbst dient die Klasse `IDTR3051`. Diese Klasse enthält die Low-Level-Funktionen des `HadesInterface`.

HADES: Darunter ist als unterste Schicht HADES selbst angesiedelt. Diese Schicht ist übrigens, wie später noch genauer erklärt wird, optional. Das MIPS-Simulationsmodell ist auch ohne HADES nur mit seiner restlichen Speicherhierarchie lauffähig.

6.4.1 Flexibilität durch wohldefinierte Interfaces

Alle an der Speicherschnittstelle beteiligten Klassen implementieren ein gemeinsames Java-Interface mit dem Namen `MipsMemory`. So ist eine maximale Flexibilität der Konfiguration der Speicherhierarchie gewährleistet. Das `MipsMemory`-Interface bietet alle notwendigen Methoden, um mit einem Speicher oder Cache innerhalb des Simulationsmodells umzugehen.

Das `MipsMemory`-Interface basiert auf dem allgemeiner formulierten `Memory`-Interface, das nur Basisfunktionalitäten bietet und hauptsächlich als einheitliche Schnittstelle zur Darstellung von Speichern dient. Die Möglichkeiten des `MipsMemory`-Interface werden insbesondere durch den `MemoryDispatcher` ausgeschöpft. Dieser bietet eine flexible Auswahl der verschiedenen Speicherobjekte und kann diese dynamisch in seinen Adreßraum einblenden. Die einzelnen Speicherobjekte sind im Abschnitt 6.4.2 beschrieben. Einen genaueren Überblick über das Konzept des MemoryDispatchers bietet Abschnitt 6.4.3.

Durch den konsequenten Einsatz des `MipsMemory`-Interfaces ist es möglich, die Speicherhierarchie des Simulationsmodells je nach Einsatzzweck neu zu konfigurieren. Durch einfache Änderungen ist es so zum Beispiel möglich, statt der kompletten Speicherhierarchie nur `AccessMemory` zusammen mit einem `TinyMemory`-Speicherobjekt zu benutzen. Daraus resultiert eine höhere Geschwindigkeit, aber auch eine kompliziertere Bedienung und fehlende Flexibilität.

6.4.2 Verschiedene Speicherobjekte

Unterhalb der `MemoryDispatcher`-Schicht sind mehrere verschiedene Speicherobjekte angesiedelt. Diese dienen entweder direkt als Speicher (`TinyMemory`, `TinyBPMemory`), als Hilfe bei der Fehlersuche (`NullMemory`) oder als Verbindung zu HADES (`HadesInterface`). Diese Speicherobjekte können auf beliebige Teile des Adreßraums abgebildet werden und bearbeiten dann alle Anfragen ihres Adreßbereichs. Theoretisch ließen sich an dieser Stelle weitere Caches einbinden (die Architektur läßt das auch auf einfachste Weise zu), da jedoch sowohl die MIPS-R3000-Architektur als auch der IDT R3051 Prozessor an dieser Stelle keine weiteren Zwischenspeicher vorsehen, ist diese Konfigurationsmöglichkeit nicht implementiert.

Jedem Speicherobjekt sind verschiedene Eigenschaften zugeordnet. Alle Speicherobjekte besitzen eine festzulegende Größe (bis auf das `HadesInterface`, dieses bietet immer einen Adreßraum von 2^{32} Bytes) und bilden Zugriffe, die über ihre Grenzen hinweg gehen, mittels einer Modulo-Operation auf den gültigen Raum ab. In der Praxis wird aus Gründen der Geschwindigkeit statt einer Modulo-Operation eine Und-Verknüpfung gewählt. Das setzt dann allerdings voraus, daß die Größe eines Speicherobjekts immer eine Potenz von zwei darstellt.

6.4.3 Das Konzept des MemoryDispatchers

Für die dynamische Einbindung verschiedener Speicherobjekte in den physikalischen Adreßraum steht der `MemoryDispatcher` zur Verfügung. Er unterteilt den Adreßraum in 65536 Segmente mit jeweils 65536 Bytes Größe. Jedem dieser Segmente kann vom Benutzer interaktiv ein Speicherobjekt zugewiesen werden. Wenn dann ein Zugriff auf ein bestimmtes Segment erfolgt, wird die Aktion an das dort eingeblendete Speicherobjekt weitergeleitet. Dabei wird immer die volle 32-Bit Adresse an das Speicherobjekt weitergegeben, damit sich der Speicher auch über mehrere Segmente hinweg erstrecken kann. Auch ist es möglich, daß ein kleiner Speicher einen Teil eines größeren Speichers überlagert.

Der `MemoryDispatcher` ist in der Lage, verschiedene Speicherobjekte mit unterschiedlichem Zeitverhalten zu unterstützen. Je nach gerade zugegriffenem Speicherobjekt und dessen Zugriffsverzögerung kann der `MemoryDispatcher` den `Core` dazu bewegen, Wartezyklen einzulegen. Wird das Simulationsmodell ganz normal zusammen mit HADES eingesetzt, so hat der Benutzer im Normalfall nicht viel mit dem `MemoryDispatcher` zu tun: Dieser ist so voreingestellt, daß der gesamte Adreßraum auf das `HadesInterface` abgebildet wird, nur für geladene Programme werden interne Speicherobjekte angelegt. Das führt zu einfacher Bedienung und schneller Programmausführung. Das detaillierte Verhalten des `MemoryDispatcher` ist in Abschnitt 7.7 beschrieben.

6.5 Pipeline- und Speicherdarstellung

6.5.1 Darstellung der Befehls-Pipeline

Die Klasse `ShowPipe` ist in der Lage, die für eine Pipeline-Darstellung mehrerer `InstrFrame`-Objekte wichtigen Daten nebeneinander anzuzeigen. Dabei wird der aktuelle Bearbeitungszustand jedes `InstrFrame`-Objekts hervorgehoben und eine disassemblierte Darstellung des aktuellen Befehls geboten.

Die für die Darstellung notwendigen Daten werden innerhalb der `InstrFrame`-Objekte gesammelt und von `ShowPipe` im Moment der Anzeige ausgelesen. `InstrFrame` selbst merkt dabei nichts von der Existenz der Pipeline-Darstellung, es kann auch ohne arbeiten.

6.5.2 Darstellung von Speicher-Objekten

Die für das MIPS-Simulationsmodell verwendeten Speicherobjekte implementieren alle das `Memory`-Interface. Über diese Schnittstelle können alle Speicher einheitlich ausgelesen und beschrieben werden, so daß hierüber auch eine Darstellung für den Benutzer realisiert werden kann.

Die verschiedenen Speicher können grob in zwei Gruppen unterteilt werden: Auf der einen Seite stehen die Speicher, die ausführbare Maschinenbefehle enthalten (Programmspeicher, Instruktions-Cache) und auf der anderen Seite stehen Speicher, in denen nur Daten und niemals Maschinenbefehle zu finden sind (Register). Für diese beiden unterschiedlichen Speichervarianten sollen verschiedene Darstellungsmöglichkeiten geboten werden, einerseits möglichst ausführlich mit Anzeige der disassemblierten Maschinenbefehle und andererseits möglichst kompakt mit den nötigsten Daten.

So existieren zwei verschiedene Klassen, die Speicher darstellen können. `MemoryHexView` zeigt ein Speicherobjekt kompakt und platzsparend an. `MemoryDisView` bietet zusätzlich eine Darstellung der disassemblierten Befehle und ist in der Lage, Unterbrechungspunkte innerhalb des dargestellten Speicherobjekts zu setzen (wenn dieses Unterbrechungspunkte unterstützt).

Falls ein Speicherobjekt durch mehr als ein Anzeigeobjekt dargestellt werden soll, gibt es auch die Möglichkeit, statt direkt des Anzeigeobjekts indirekt einen `MemoryViewHandler` zu benutzen. Dieser ermöglicht den Einsatz von mehreren Anzeigen desselben Speichers mit automatischer Auffrischung, wenn sich etwas am Speicherinhalt ändert. Damit `MemoryHexView` und `MemoryDisView` mit dem `MemoryViewHandler` zusammenarbeiten können, müssen beide das `MemoryView`-Interface implementieren, das dem `MemoryViewHandler` in die Lage versetzt, allgemein auf beide zugreifen zu können.

7 Spezielle Aspekte der Implementation

Das MIPS-Simulationsmodell versucht, den Kern einerseits möglichst Hardware-nah und andererseits möglichst leistungsfähig zu simulieren. Deshalb besitzt die Implementation des MIPS-Simulationsmodells einige Funktionsweisen, die nicht trivial nachzuvollziehen sind und genauer erklärt werden müssen. Dieses Kapitel wirft einen Fokus auf solche Details. Auch die Mechanismen, die dieses MIPS-Simulationsmodell von anderen Simulationsmodellen unterscheidet, werden hier eingehend betrachtet. Vor der Beschreibung, wie speziell auf die MIPS-Architektur zugeschnittene Funktionen implementiert sind, werden zuerst einige allgemein nutzbare Komponenten vorgestellt: Bereits im Abschnitt 6.4.1 wurden das `Memory`- und das `MipsMemory`-Interface erwähnt. In Abschnitt 7.1 sollen diese Schnittstellen nun eingehend dargestellt werden, da viele der weiter unten beschriebenen Objekte auf diesen Schnittstellen basieren. Zusätzlich wird noch das `Resetable`-Interface erklärt, da dieses die Grundlage für einen allgemein nutzbaren Reset-Mechanismus darstellt.

Die Beschreibung der zum MIPS-Simulationsmodell gehörigen Klassen soll mit `InstrFrame` und davon benutzten untergeordneten Klassen begonnen werden. Dabei soll das Zeitverhalten (letztendlich die daraus resultierende Implementation der Pipeline) dargestellt werden. Direkt zu `InstrFrame` gehören die Befehlsklassen, die von `InstrFrame` angesteuert werden. Hier werden die Hierarchie der Befehlsklassen sowie Ausschnitte des internen Aufbaus vorgestellt. Die Behandlung der MIPS-Exceptions soll an dieser Stelle auch genauer betrachtet werden, da dieser nicht ganz einfach zu verstehende Mechanismus für eine gelungene Simulation effizient implementiert sein muß. Als Abschluß der Kern-Implementation wird die Klasse `Core` als Dachklasse vorgestellt.

Die Arbeit von `AccessMemory` bei der Benutzung der Caches soll Mittelpunkt des Abschnitts 7.6 sein. Hier werden auch die genauen Mechanismen vorgestellt, die `AccessMemory` benutzt, um die Zugriffe des Kerns auf die darunterliegenden Speicherschichten abzubilden. Daran schließt sich die Betrachtung der Implementation des `MemoryDispatcher` an. Das Konzept dieser Klasse wurde schon in Abschnitt 6.4.3 beschrieben, so daß hier die Interna vorgestellt werden können. Eines der vom `MemoryDispatcher` verwendbaren Speicherobjekte ist `TinyBPMemory`. Dieses Speicherobjekt stellt Unterbrechungspunkte für den MIPS-Maschinenprogrammablauf zur Verfügung. Da hier eine nicht übliche Implementation der Unterbrechungspunkte gewählt wurde, muß auch diese Klasse im Detail vorgestellt werden. Die Arbeit des `HadesInterface` (zusammen mit `IDTR3051`) soll in Abschnitt 7.9 beschrieben werden. Innerhalb dieser Klasse muß auf die Eigenheiten der IDT-R3051-Speicherschnittstelle eingegangen und das richtige Zeitverhalten für alle Ein- und Ausgänge eingehalten werden.

Damit ein MIPS-Maschinenprogramm in den Speicher des Simulationsmodells übernommen werden kann, ist ein Mechanismus notwendig, der ein Speicherobjekt mit dem Inhalt einer ausführbaren Maschinenprogramm-Datei füllt. Der `MemoryLoader` erfüllt diese Aufgabe und wird in Abschnitt 7.10 beschrieben. Ein Ziel dieser Arbeit war die übersichtliche Darstellung der internen Vorgänge. Auf die Visualisierung soll im Abschnitt 7.11 genauer eingegangen werden. Dabei steht nicht die Benutzerschnittstelle, sondern der interne Aufbau im Vordergrund. Die Bedienung des Simulationsmodells wird im Abschnitt 8.1 eingehend beschrieben.

7.1 Global bedeutsame Interfaces

7.1.1 Memory-Interface

Jeder im MIPS-Simulationsmodell enthaltene Speicher implementiert letztendlich das `Memory`-Interface. Die hier enthaltenen Methoden dienen dem einheitlichen Zugriff auf alle Speicher und werden von der Visualisierung und dem `MipsMemory`-Interface benutzt. Die durch dieses Interface beschriebenen Methoden im einzelnen:

getWordWidth(): Jeder Speicher kann eine unterschiedliche Wortbreite der Speicherworte besitzen. Diese wird mit `getWordWidth()` zurückgemeldet. Für die Simulation der MIPS-Architektur, deren Wortbreite 32 Bit beträgt, werden natürlich nur Speicher mit 32 Bit breiten Worten benötigt. Da das `Memory`-Interface jedoch auch für andere Zwecke geeignet ist, wird diese Methode zur Verfügung gestellt.

getMemorySize(): Um die Gesamtgröße des Speichers in Worten zu erfahren, kann `getMemorySize()` aufgerufen werden. Dabei wird die Größe immer in Worten, deren Breite durch `getWordWidth()` festgestellt werden kann, angegeben.

readMemory(): Diese Methode liefert nach Angabe einer Adresse das dazugehörige Speicherwort. Dabei wird davon ausgegangen, daß der Speicher sein Ergebnis ohne Verzögerung ausgeben kann. `readMemory()` liefert direkt den an der angegeben Speicherstelle vorhandenen Wert zurück. Im Gegensatz dazu existieren im `MipsMemory`-Interface Methoden, um den Zugriff auf einen Speicher zu realisieren, dessen Inhalt nur mit Zeitverzögerung ausgelesen werden kann.

writeMemory(): `writeMemory()` dient als Gegenstück zum `readMemory()`, hiermit kann ein Wort in den Speicher geschrieben werden. Auch hier gilt dasselbe, was schon zum Zeitverhalten der `readMemory`-Methode gesagt wurde.

setEntry(), installEntryHandler(): Wird mit `installEntryHandler()` ein Callback-Objekt angegeben, kann danach mit `setEntry()` die `setEntry()`-Methode dieses Objekts aufgerufen werden. Sie dient dazu, nach dem Laden eines neuen `Memory`-Inhalts eine Einsprungadresse zu setzen. Mehr zu diesem Thema wird im Abschnitt 7.10.3 zum `EntryHandler` beschrieben.

Das `Memory`-Interface wird für alle innerhalb der Speicherhierarchie auftretenden Speicherobjekte genutzt. Darüber hinaus werden aber auch die `Register` und die Koprozessoren mit diesem Interface versehen, damit deren Inhalte einfach visualisiert werden können.

7.1.2 MipsMemory-Interface

Das `MipsMemory`-Interface stellt Methodenaufrufe für Ganzwort-Lese- und Schreib-Zugriffe zur Verfügung, die direkt von den verschiedenen zum MIPS-Simulationsmodell gehörigen Speicherobjekten genutzt werden können. Da alle diese Speicherobjekte eine gewisse Zeit für ihre Operationen benötigen, werden außerdem Methoden angeboten, die das Zeitverhalten beeinflussen (`beginCycle()`, `endCycle()`) oder abfragen (`finished()`).

Für die Unterstützung von Caches ist außerdem `getHit()` für die Abfrage, ob die gewünschten Daten im Cache vorhanden sind, integriert.

Ein normaler Lesevorgang geht unter Zuhilfenahme der `MipsMemory`-Methoden vereinfacht so vor sich:

read(adr); Mit der Signalisierung des Lesewunsches wird die Adresse übergeben.

do endCycle(); while (beginCycle()); Da ein Lesewunsch immer innerhalb eines Zyklusses begonnen wird, muß erst einmal der Zyklus beendet werden. Danach wird dann so lange ein Zyklus durchgeführt, bis das Speicherobjekt ein Ergebnis liefern kann (abhängig von dessen Wartezyklen).

result = getReadResult(); Wenn der Speicher seine Arbeit erledigt hat, kann auf das Ergebnis zugegriffen werden.

Schreibvorgänge sind weniger kompliziert, da beim Aufruf von `write()` direkt Adresse und der zu schreibende Wert übergeben werden können. Ein Speicherobjekt, das mit `write()` einen Schreibauftrag bekommen hat, benötigt dann allerdings eine gewisse Anzahl von Zyklen, bis der Schreibvorgang beendet ist und das Speicherobjekt einen neuen Auftrag entgegen nehmen kann. Mit `finished()` kann das Speicherobjekt gefragt werden, ob es bereit ist, einen neuen Auftrag entgegenzunehmen.

Ein das `MipsMemory`-Interface implementierendes Objekt muß immer einen Namen verwalten können. Dieser Name wird über die Methoden `getName()` und `setName()` beeinflußt und dient der Identifikation der `MipsMemory`-Objekte durch den Benutzer.

Das `MipsMemory`-Interface ist abgeleitet vom `Memory`-Interface.

7.1.3 Resetable-Interface

Ein weiteres Interface, das für das MIPS-Simulationsmodell globale Wichtigkeit besitzt, ist das `Resetable`-Interface. Jede Klasse, die dieses Interface implementiert, muß zwei parameterlose Methoden zur Verfügung stellen:

por(): Diese Methode bietet den *Power On Reset* und wird immer dann aufgerufen, wenn ein Kaltstart des betreffenden Objekts durchgeführt werden soll.

reset(): Entsprechend zu `por()` existiert auch eine Methode zur Durchführung eines Warmstarts.

Zur Benutzung des `Resetable`-Interfaces wird die Klasse `ResetHandler` zur Verfügung gestellt. Diese Erweiterung eines `Vector` nimmt alle bei einem Reset zu behandelnden Objekte auf (jedes Objekt sollte sich selbst beim `ResetHandler` anmelden) und leitet `por()`- und `reset()`-Anfragen an diese weiter.

Somit steht ein einfach zu implementierendes und zu benutzendes Instrument zur Verfügung, mit dem sich die Reset-Behandlung übergreifend erledigen läßt.

7.2 Befehlsausführung: InstrFrame

7.2.1 Implementation der Pipeline-Schritte

Die von der MIPS-Architektur verwendete Pipeline wurde bereits in Abschnitt 2.4 in Stichworten beschrieben. Hier sollen die einzelnen Aktionen genauer vorgestellt werden, wobei weniger die Hardware-Vorgabe als die Implementation durch `InstrFrame` und untergeordnete Klassen im Vordergrund stehen. Deshalb wird für jede Aktion angegeben, welcher Methodenaufruf für die Ausführung dieser Aktion steht.

- Instruktionsadresse feststellen (`InstrFrame.translateInstrAdr()`): Der Programmzähler wird gelesen und erhöht. Dann wird der gelesene Wert (die virtuelle Adresse) an die Adreßübersetzungslogik übergeben und in eine reale Adresse umgesetzt.

- Instruktionsadresse auf den Bus legen (`InstrFrame.giveInstrAdr()`): Nun wird dem Cache die Adresse gegeben. Der Cache kümmert sich dann um alles weitere: Entweder ist der Wert vorhanden oder die darunterliegende Schicht muß die Anfrage bearbeiten.

- Instruktion lesen (`InstrFrame.fetchInstr()`): Innerhalb des Kerns wird hier nichts durchgeführt. Entweder wird diese Zeit für den Zugriff auf den Cache genutzt (nicht in der Software, aber in der Realität) oder der externe Speicher wird gelesen (dafür wurde die Adresse ja schon in der vorherigen Aktion auf den Bus gelegt, deshalb muß hier nichts passieren). Eventuelle Wartezyklen des Speichers verlängern die Ausführungszeit dieser Aktion.

- Instruktionsdekodierung (`InstrFrame.decodeInstr()`): Nun wird das gelesene Instruktionswort in seine Bestandteile aufgeteilt (dafür ist der `Splitter` zuständig), so daß die Einzelteile des Befehls für die Dekodierung benutzt werden können (diese Aufgabe erfüllt der `Decoder`). Da (hauptsächlich) die ersten sechs und die letzten sechs Bit den Befehl angeben, wird über diese Teilworte ein Tabellenzugriff ausgeführt, der als Ergebnis das zum Befehl gehörige Instruktionsobjekt liefert. Nachdem diese Aktion durchgeführt wurde, werden die Befehle nicht mehr gemeinsam (in `InstrFrame`) bearbeitet, sondern jeder einzeln von seinem Instruktionsobjekt.

- ALU-Operation (`*Instr.calcAlu()`): Hier berechnet jeder Befehl sein Ergebnis. Sprungbefehle setzen hier den Programmzähler auf die Zieladresse, so daß der übernächste Befehl dann den Sprung ausführen kann, indem dessen Instruktionsadresse den hier gesetzten Wert berücksichtigt.

- Datenadresse feststellen (`*Instr.translateDataAdr()`): Entsprechend der Aktion *Instruktionsadresse feststellen* wird hier bei lesenden oder schreibenden Befehlen die Datenadresse übersetzt.

- Datenadresse auf den Bus legen (`*Instr.giveDataAdr()`): Lesende oder schreibende Befehle arbeiten hier entsprechend *Instruktionsadresse auf den Bus legen*). Falls ein ALU-Befehl durchgeführt wurde, wird ausnahmsweise schon hier dessen Ergebnis in das Zielregister geschrieben, damit der direkt nachfolgende Befehl in seiner ALU-Operation schon mit diesem Wert rechnen kann.

- Daten lesen/schreiben (`*Instr.operateData()`): Hier passiert wieder nichts (entsprechend *Instruktion lesen*).

- Register schreiben (`*Instr.writeRegister()`): Falls ein Lesebefehl vorliegt, wird nun das aus dem Speicher (oder dem Daten-Cache) gelesene Ergebnis in das Zielregister geschrieben.

7.2.2 Splitter

Nachdem ein Befehlswort aus dem Speicher gelesen worden ist und bevor es weiterverarbeitet werden kann, muß es in sinnvolle Portionen aufgeteilt werden, die dann von den nachfolgendenen Instanzen benutzt werden können. Dank des einfachen Befehlsformats der MIPS-Instruktionen kann ein Befehlswort einfach und effizient durch Maskierung und Schiebeoperationen in seine Einzelteile zerlegt werden.

Der Splitter führt für alle Parameter des Befehlswortes diese Maskierungs- und Schiebeoperationen durch, um dadurch zu den Einzelteilen des Befehles zu gelangen. Da der MIPS drei verschiedene Befehlsformate kennt (*Immediate*, *Jump* und *Register*), werden die Parameter für alle drei Formate gleichzeitig zur Verfügung gestellt. Dabei wird der *Immediate*-Wert sowohl mit als auch ohne Vorzeichen angeboten (die Befehlsformate sind in Abschnitt 2.3.1 beschrieben).

Für die Benutzung des Splitter muß zuerst `splitInstruction()` aufgerufen werden. Danach stehen die Einzelteile des Befehls über die `get*()`-Methoden zur Verfügung.

Für den Fall, daß Befehle disassembliert werden sollen, stellt der Splitter eine einfache Methode zur Aufbereitung der Parameter zur Verfügung. Auf diesen Mechanismus wird erst im Abschnitt 7.2.4 zum `InstrFrame` eingegangen, da zur Disassemblierung noch weitere Schritte notwendig sind.

7.2.3 Decoder

Nachdem ein Befehl durch den `Splitter` in seine Einzelteile zerlegt wurde, kann er durch den `Decoder` erkannt werden. Das Erkennen bedeutet in diesem Fall, daß der `Decoder` ein Objekt zurückliefert, in dem alle für die Ausführung (und auch Disassemblierung) des Befehls notwendigen Bestandteile enthalten sind (eine genaue Erklärung der Befehlsklassen ist in Abschnitt 7.3 zu finden).

Für die Ausführung seiner Aufgabe legt der `Decoder` bei seiner Initialisierung für jeden einzelnen Befehl ein Objekt von dessen Befehlsklasse an. Die Referenzen auf die Befehle werden in langen Tabellen gespeichert, über die dann anhand der vom `Splitter` gelieferten Befehlseinzelteile direkt und schnell auf das gewünschte Befehlsobjekt zugegriffen werden kann.

Der `Decoder` besitzt einige Befehlszuordnungen, die in der realen Hardware nicht existieren: Wie im Abschnitt 5.4.2 beschrieben wird der `nop`-Befehl als eigenständiges Befehlsobjekt `NopInstr` angeboten. Außerdem werden alle illegalen Befehlszuordnungen auf die `IllegalInstr` umgeleitet. Diese tut wie die `NopInstr` nichts, sie gibt aber eine Warnung aus. Statt dessen kann von der `IllegalInstr`, wie im realen Vorbild, auch eine Exception ausgelöst werden.

7.2.4 InstrFrame

Der `InstrFrame` ist sozusagen der Rahmen (deshalb der Name) für die Ausführung eines Befehls durch alle Pipeline-Aktionen hindurch.

Für jede Pipeline-Aktion enthält `InstrFrame` je eine Methode, die die entsprechende Aktion ausführt. Die meisten Aktionen werden direkt zu untergeordneten Instanzen weitergereicht, wobei sich `InstrFrame` dann um die Aufbereitung des externen Zustands (das sind dann die Informationen für die Anzeige) kümmert. Dabei wird wirklich nur aufbereitet und nichts angezeigt, damit eine klare Trennung zwischen Funktion und deren Anzeige vorliegt.

Die Methoden `translateInstrAdr()`, `giveInstrAdr()` und `fetchInstr()` werden direkt an `AccessMemory` weitergegeben (siehe unten in Abschnitt 7.6.2). Die dann folgende Dekodierung der Befehle wird wie oben beschrieben vom `Splitter` und dem `Decoder` vorgenommen. Als Ergebnis steht dann ein Befehlsobjekt zur Verfügung, das sich um die weitere Bearbeitung eines Befehls kümmern kann. `calcAlu()`, `translateDataAdr()`, `giveDataAdr()`, `operateData()` und `write-Register()` werden dann an das Befehlsobjekt weitergeleitet, wobei wieder dabei entstandene Informationen für eine eventuelle Ausgabe aufbereitet werden. Die letzte für die Pipeline-Bearbeitung zur Verfügung stehende Methode `finishInstr()` setzt das aktuelle `InstrFrame`-Objekt wieder auf einen definierten Anfangsstatus zurück, um damit zu signalisieren, daß dieses Objekt wieder für die Ausführung eines weiteres Befehls zur Verfügung steht.

`InstrFrame` kann neben der normalen Befehlsbearbeitung noch eine weitere Aufgabe erfüllen, die sozusagen als Abfallprodukt entstanden ist: Die Disassemblierung. Da `InstrFrame` nach Benutzung des `Splitter` und des `Decoder` das identifizierte Befehlsobjekt zur Verfügung steht, können diesem Befehlsobjekt Angaben für die Disassemblierung entnommen werden. In jeder Basis- und Befehlsklasse ist der Befehlsname (`instrName`) und das Parameterformat in kurzer (`instrShortFormat`) und langer (`instrLongFormat`) Form hinterlegt. `InstrFrame` kann auf diese Zeichenketten zugreifen und daraus unter Zuhilfenahme des `Splitter` (`formatOperands()`) eine Assembler-Darstellung des Befehls zusammensetzen.

Damit außenstehende Klassen diesen Mechanismus nutzen können, gibt es die Methoden `beginDisassemble()`, `disassemble()` und `endDisassemble()`. Zwischen Beginn und Ende können beliebig viele Instruktionsworte disassembliert werden. Es muß nur darauf geachtet werden, daß direkt davor `beginAssemble()` und direkt danach `endAssemble()` aufgerufen werden, denn sonst würde der `InstrFrame` auf einer inkonsistenten Datenbasis weiterbenutzt werden.

7.3 Befehlsklassen

Für die Bearbeitung jedes einzelnen Befehls gibt es eine eigene Klasse, die alle Aufrufe dieses Befehls ausführen kann. Jede Pipeline-Aktion nach der Dekodierung des Befehls wird in dieser Befehlsklasse nachgebildet.

Um die vielen Befehle des MIPS-Befehlssatzes besser zu strukturieren und um die Übersicht zu vereinfachen, existieren für Gruppen von gleichartigen Befehlen Basisklassen, die dazu führen, daß die eigentlichen Befehlsklassen nur die wirklich für genau den einzelnen Befehl besonderen Aktionen enthalten. In die Basisklassen sind auch allgemeine Disassemblierungsinformationen zur visuellen Aufbereitung der Befehlsparameter integriert.

7.3.1 BaseInstr

Als Basisklasse für alle Befehle dient `BaseInstr`. Hier wird für alle Pipeline-Aktionen eine Methode vorgegeben, die als Vorgabe nichts tut. Außerdem werden hier die im allgemeinen für die Befehlsklassenbenutzung benötigten Variablen gespeichert. Wichtig ist die `init()`-Methode, mit der das Befehlsobjekt nach der Dekodierung und vor der eigentlichen Ausführung eines Befehls auf einen definierten Zustand zurückgesetzt wird.

Die einzelnen von `BaseInstr` definierten Methoden, die von allgemeinem Interesse sind, werden im Folgenden beschrieben:

por(), reset(): Die Befehlsklassen lassen sich wie alle anderen Kern-Objekte per `ResetHandler` zurücksetzen. Alles weitere zu diesem Thema wird im Abschnitt 7.1.3 beschrieben.

init(): Wichtig ist die `init()`-Methode, mit der nach der Dekodierung und vor der eigentlichen Ausführung eines Befehls das Befehlsobjekt auf einen definierten Zustand zurückgesetzt wird.

getInstrName(): Jede Instruktion besitzt einen Namen (das ist dann der Name des Maschinenbefehls). Dieser Name kann mit `getInstrName()` ausgelesen und für Disassemblierungszwecke genutzt werden.

getInstrLongFormat(), getInstrShortFormat(): Zusätzlich zum Namen besitzt jeder Maschinenbefehl noch ein Instruktionsformat, durch das die Disassemblierung der Befehlsparameter unterstützt wird. Genau genommen definiert jede Instruktion sogar zwei verschiedene Parameterformate, ein kurzes und ein langes. Dieser Unterschied ist notwendig, um zwischen platzsparender und ausführlicher Parameterdarstellung unterscheiden zu können.

getIsJump(): Wenn eine Exception auftritt, muß für jeden Befehl die Information zur Verfügung stehen, ob er einen Sprung verursacht. Ist dem so, dann muß der nachfolgende Befehl aufgrund der Sprung-Hazards (siehe Abschnitt 2.5.3) gesondert behandelt werden.

getBreakPoint(), setBreakPoint(): Diese beiden Methoden unterstützen die Möglichkeiten des `TinyBPMemory`.

calcAlu(): Die erste Aktion der Pipeline, die direkt vom Befehlsobjekt ausgeführt werden kann (vorige Aktionen bearbeitet `InstrFrame`, da das zu verwendende Befehlsobjekt vor Abschluß der Befehlsdekodierung noch nicht feststeht), ist die Berechnung des Befehlsergebnisses in der ALU.

translateDataAdr(): Falls der Befehl einen Speicherzugriff durchführen möchte, wird in dieser Aktion die virtuelle Datenadresse in die physikalische umgewandelt.

giveDataAdr(): Nachdem die physikalische Adresse berechnet wurde, kann der eigentliche Speicherzugriff gestartet werden.

operateData(): Der Kern ist im Gegensatz zu den verwendeten Speichern ziemlich schnell, so daß während dieser Aktion (bis auf Debug- und Visualisierungsaufgaben) nichts getan werden muß. Hier wird einfach darauf gewartet, daß der Speicher die Datenzugriffsoperation beendet.

writeRegister(): Nachdem der Speicherzugriff beendet wurde, kann das Ergebnis in das Zielregister geschrieben werden. Befehle, die statt eines Speicherzugriffs die ALU nutzen, schreiben hier das Ergebnis der ALU-Operation in das Zielregister.

Direkt abgeleitet von `BaseInstr` sind alle weiteren Basis-Befehlsklassen, `NopInstr`, `IllegalInstr`, `SysCallInstr`, `MoveToLoInstr`, `MoveToHiInstr`, `MoveFromLoInstr`, `MoveFromHiInstr`, `JumpRegInstr`, `JumpRegAndLinkInstr`, `CoprocInstr` und `BreakInstr`.

7.3.2 Übersicht über weitere Basisklassen

Im Folgenden werden die verschiedenen Basis-Befehlsklassen erklärt und die davon abgeleiteten Befehlsklassen aufgezählt. Die Benennung der Befehlsklassen stellte machmal aufgrund der langen Befehlsnamen ein Problem dar. Falls in den Befehlsklassennamen ein `U` enthalten ist, bedeutet dieses `Unsigned`. `Imm` steht für `Immediate`.

AluBaseInstr: Für alle ALU-Befehle im *Register*-Format wird `AluBaseInstr` als Basis benutzt. Diese Klasse kümmert sich um das Zurückschreiben des ALU-Ergebnisses, so daß im von `AluBaseInstr` abgeleiteten Befehl nur noch die eigentliche Berechnung (die `calcAlu()`-Aktion der Pipeline) implementiert sein muß.

Von `AluBaseInstr` werden folgende Befehlsklassen abgeleitet: `AddInstr`, `AddUInstr`, `AndInstr`, `NOrInstr`, `OrInstr`, `SetOnLessInstr`, `SetOnLessUInstr`, `ShiftBaseInstr`, `ShiftVarBaseInstr`, `SubInstr`, `SubUInstr` und `XOrInstr`.

AluImmBaseInstr: Diese Klasse bietet Ähnliches wie `AluBaseInstr`, nur werden von dieser Klasse die ALU-Befehle im *Register*-Format abgeleitet.

Von `AluImmBaseInstr` werden folgende Befehlsklassen abgeleitet: `AddImmInstr`, `AddImmUInstr`, `AndImmInstr`, `LoadUpperImmInstr`, `OrImmInstr`, `SetOnLessImmInstr`, `SetOnLessImmUInstr` und `XOrImmInstr`.

AluSpecialBaseInstr: Im Gegensatz zu den normalen ALU-Befehlen stehen die Divisions- und Multiplikationsbefehle. Die Ergebnisse dieser Befehle werden in eigens dafür vorgesehenen Registern gespeichert. Deshalb ist eine spezielle Basisklasse für diese Befehle erforderlich.

Von `AluSpecialBaseInstr` werden folgende Befehlsklassen abgeleitet: `DivInstr`, `DivUInstr`, `MultInstr` und `MultUInstr`.

BranchBaseInstr: Alle Verzweigungsbefehle ohne Speicherung des Ausgangspunktes besitzen `BranchBaseInstr` als Basisklasse, da hier die Methode `branch()` vorhanden ist, die einen Sprung ausführt. Wenn in `calcAlu()` eines Verzweigungsbefehls festgestellt wird, daß die Verzweigung durchgeführt werden muß, reicht der Aufruf von `branch()` aus, um alles in die Wege zu leiten.

Von `BranchBaseInstr` werden folgende Befehlsklassen abgeleitet: `BranchEqualInstr`, `BranchGreaterEqualInstr`, `BranchGreaterInstr`, `BranchLessEqualInstr`, `BranchLessInstr` und `BranchNotEqualInstr`.

BranchLinkBaseInstr: Zusätzlich zur in `BranchBaseInstr` enthaltenen Logik muß hier noch der Ausgangspunkt der Verzweigung gesichert werden.

Von `BranchLinkBaseInstr` werden folgende Befehlsklassen abgeleitet: `BranchGreaterEqualAndLinkInstr` und `BranchLessAndLinkInstr`.

LoadBaseInstr: Alle Befehle, die den Speicher auslesen, benutzen `LoadBaseInstr` als Basisklasse. Hier wird die Adreßumsetzung für den eigentlichen Speicherzugriff ausgeführt.

Von `LoadBaseInstr` werden folgende Befehlsklassen abgeleitet: `LoadByteInstr`, `LoadByteUInstr`, `LoadHalfInstr`, `LoadHalfUInstr`, `LoadWordInstr` und `LoadWordToCoInstr`.

ShiftBaseInstr: `ShiftBaseInstr` benutzt die in `AluBaseInstr` definierten Mechanismen zur Speicherung der ALU-Ergebnisse. Da sich das Befehlsformat der Shift-Befehle von dem der anderen ALU-Befehle unterscheidet, existiert eine eigene Klasse.

Von `ShiftBaseInstr` werden folgende Befehlsklassen abgeleitet: `ShiftLeftLogicalInstr`, `ShiftRightArithInstr` und `ShiftRightLogicalInstr`.

ShiftVarBaseInstr: Für diese Klasse gilt auch das schon zu `ShiftBaseInstr` Gesagte.

Von `ShiftVarBaseInstr` werden folgende Befehlsklassen abgeleitet: `ShiftLeftLogicalVarInstr`, `ShiftRightArithVarInstr` und `ShiftRightLogicalVarInstr`.

StoreBaseInstr: Diese Klasse wird von allen schreibend auf den Speicher zugreifenden Befehlsklassen benutzt. Wie bei `LoadBaseInstr` wird in `StoreBaseInstr` auch die Adreßumsetzung vorgenommen.

Von `StoreBaseInstr` werden folgende Befehlsklassen abgeleitet: `StoreByteInstr`, `StoreHalfInstr`, `StoreWordFromCoInstr` und `StoreWordInstr`.

7.3.3 Aufbau der Befehlsklassen

Für die verschiedenen Befehlsformate und funktional ähnlichen Instruktionen wird nun der generelle Aufbau der Klassen näher dargestellt:

- Die normalen ALU-Befehle (`add`, `and`, `nor`, `or`, `shift`, `sub`, `xor` und `setonless`) berechnen ihr Ergebnis in `calcAlu()` und schreiben dieses in `giveDataAdr()` (zur Vermeidung von Write-Back-Hazards, siehe Abschnitt 2.5.1) und in `writeRegister()` (damit keine Seiteneffekte durch das frühe Schreiben des Ergebnisses auftreten) in das Zielregister.

- Die speziellen ALU-Befehle (`mult`, `div`) berechnen ihr Ergebnis in `calcAlu()` und schreiben dieses in `writeRegister()`. Diese Vereinfachung kann so durchgeführt werden, da die Ausführungsgeschwindigkeit der `mult`- und `div`-Befehle von der Hardware-Geschwindigkeit abhängt (siehe Modellbildung, Abschnitt 5.4.1).

- Die Sprung- und Verzweigungsbefehle berechnen in `calcAlu()` ihr Sprungziel und eventuell die Verzweigungsbedingung, um dann sofort den Programmzähler auf den neuen Wert zu setzen (je nach Bedingung). Die Link-Befehle schreiben zusätzlich in `writeRegister()` den alten Wert des Programmzählers in das Register 31.

- Die `load`- und `store`-Befehle sind ein wenig komplizierter. Hier wird von `translateDataAdr()`, `giveDataAdr()` und `operateData()` das `AccessMemory`-Objekt für den Speicherzugriff verwendet. Die `load`-Befehle schreiben dann in `writeRegister()` das Ergebnis ins Zielregister.

- Alle weiteren Befehle für Spezialaufgaben erfüllen ihre hauptsächliche Arbeit in der `calcAlu()`-Methode.

7.3.4 Beispiel für eine Befehlsklasse: AndInstr

Da hier aufgrund des Umfangs nicht alle Befehlsklassen dargestellt werden können, soll nun beispielhaft die Klasse `AndInstr` als Beleg dafür dienen, daß durch die hierarchische Struktur der Befehlsbasisklassen letztendlich sehr kleine und dadurch einfach implementierbare Befehlsklassen entstehen.

```
package hades.models.mips.instr;

import hades.models.mips.core.PartHandler;
import hades.models.mips.core.Resetable;
import hades.models.mips.core.Splitter;

public class AndInstr extends AluBaseInstr implements Resetable
{
  public AndInstr(PartHandler newPartHandler, Splitter newSplitter)
  {
    super(newPartHandler, newSplitter);
    instrName = new String("AND     ");
  }

  public int calcAlu()
  {
    result = reg.readRegister(splitter.getSourceReg()) &
             reg.readRegister(splitter.getTargetReg());
    return result;
  }
}
```

Wie zu erkennen, muß neben dem Aufruf des Superklassenkonstruktors nur noch der Name der Instruktion und die eigentliche Berechnung des Ergebnisses vorgenommen werden. Alle weiteren Aktionen werden von den Basisklassen ausgeführt und interessieren in diesem Zusammenhang nicht weiter.

7.4 Exception-Behandlung

Für die Benutzung eines Prozessors, der der MIPS-Architektur entspricht, sind Exceptions essentiell. Innerhalb des MIPS-Simulationsmodells können Exceptions an verschiedenen Stellen ausgelöst werden. Einerseits enthalten einige Befehlsklassen die Möglichkeit, eine Exception auszulösen (Überlauf-Exception in `AddInstr` und Ähnlichen, `BreakInstr` und `SysCallInstr` mit speziellen Exceptions), andererseits können `InstrFrame` oder die Speicherhierarchie eine Exception bei Problemen mit der Befehlsausführung auslösen.

Will eine Klasse eine Exception auslösen, so muß sie den `ExceptionGenerator` aufrufen. Dieser sorgt dann dafür, daß die notwendigen Koprozessor-Register mit den richtigen Werten gefüllt werden. Dann wird der Programmzähler auf einen der Exception-Vektoren (siehe Abschnitt 3.5.3) gesetzt und alle gerade bearbeiteten Befehle, die nach dem Exception-auslösenden Befehl gestartet wurden, als ungültig erklärt.

7.5 Core

Alle Kern-Klassen werden vom `Core` zusammengehalten. Dieser stellt die Dachklasse dar, instanziiert alle an der Befehlsausführung beteiligten Objekte und ruft die einzelnen Pipeline-Aktionen von `InstrFrame` auf.

Die Hauptmethode von `Core` ist `cycle()`. Diese Methode ruft für jedes der fünf gleichzeitig aktiven `InstrFrame`-Objekte jeweils eine Pipeline-Aktion auf. Zwei `cylce()`-Aufrufe stellen einen Pipeline-Schritt dar. Innerhalb von `cycle()` wird die Speicherhierarchie gefragt, ob ein Wartezyklus notwendig ist und im Bedarfsfall wird dieser einfügt.

Wird das MIPS-Simulationsmodell ohne HADES benutzt, dann muß pro Takt einfach zwei mal `cycle()` aufgerufen werden, damit MIPS-Maschinenbefehle ausgeführt werden. Wenn statt dessen HADES für einen Takt sorgt, dann muß nur ein Taktgenerator das MIPS-Simulationsmodell mit einem Takt versorgen, der dann dazu führt, daß direkt `cycle()` aufgerufen wird.

7.6 AccessMemory und Cache-Benutzung

`AccessMemory` ist die Verbindung zwischen Kern, MMU und den darunter liegenden Speicherschichten, die per `MipsMemory`-Interface angesprochen werden. Hier werden alle für die Speicherhierarchie benötigten Objekte angelegt und miteinander verbunden. Für Speicherzugriffe notwendige Adreßumsetzungen werden an die MMU übergeben (dynamische Adreßumsetzung) oder direkt ausgeführt (statische Adreßumsetzung).

Speicherhierarchie: Im Konstruktor werden die Variablen `mem`, `icache` und `dcache` initialisiert. Über diese Speicher werden alle Speicherreferenzen abgewickelt. Werden keine Caches gewünscht, so kann einfach jeder dieser Variablen der Wert von `mem` zugewiesen werden.

Adreßumsetzung und Speicherauswahl: Vor einem Speicherzugriff muß entweder mit `translateInstrAdr()` oder mit `translateDataAdr()` die virtuelle in eine reale Adresse umgewandelt werden. Dabei wird zuerst festgestellt, ob die virtelle Adresse den Cache benutzt. Dann wird die statische Adreßumsetzung durchgeführt (die obersten drei Bit der Adressen werden neu zugeordnet). Der Aufruf der `MemoryManagementUnit` schließt die Adreßumsetzung ab.

Instruktionslesebefehle: Da Instruktionen immer genau 32 Bit lang sind, ist dieser Mechanismus einfach: `readInstr()` übergibt die (reale) Adresse an den Speicher und `fetchInstr()` bekommt das Ergebnis aus dem Speicher zurück.

Datenlesebefehle: Das Lesen von Daten funktioniert grundsätzlich nach demselben Prinzip wie das Lesen der Instruktionen (`readWord()` übergibt die Adresse und `getReadResult()` liefert das Ergebnis).

Datenschreibbefehle: Das Schreiben von ganzen Worten wird einfach an den darunter liegenden Speicher weitergegeben (`writeWord()`).

Aus der Sicht des Kerns existiert als Speichersystem nur ein `AccessMemory`-Objekt. Dieses stellt alle Funktionalitäten zur Verfügung, die der Kern für Speicherzugriffe benötigt. Dabei verwendet der Kern virtuelle Adressen. Die Adreßumsetzung wird von `AccessMemory` vorgenommen. Da aufgrund der Zugriffsadressen die Cache-Selektion stattfindet, werden auch die Caches von `AccessMemory` ausgewählt.

7.6.1 Caches

Die beiden Caches werden von `AccessMemory` angelegt und eingebunden. Für die Instruktionen wird der `InstrCache` verwendet, die Daten werden vom `DataCache` verwaltet. Beide sind direkt von der allgemeinen `Cache`-Klasse abgeleitet. Das Verhalten der Caches entspricht der Hardware-Beschreibung bis auf die Einschränkung, daß die interne Struktur des `InstrCache` genauso aufgebaut ist wie die des `DataCache` und keine Größe der Cache-Zeilen ungleich eins unterstützt wird (siehe Abschnitt 5.4.5).

Die Caches bieten eine Implementation des `Memory`-Interfaces, bei der jedes zweite Wort den wirklichen Inhalt des Cache-Speichers darstellt und jedes andere Wort Informationen über die Verwaltungsdaten für das vorige Wort angeben. Die Verwaltungsdaten bestehen aus dem Gültigkeitsbit im höchsten Bit der Ausgabe (wenn dieses Bit nicht gesetzt ist, ist der Cache-Eintrag gültig) und den höherwertigen Adreßbits des gespeicherten Datums in den unteren Bits. Dabei ist zu beachten, daß die Caches nicht wie der Kern mit Byte-Adressen arbeiten, sondern Wort-Adressen benutzen, so daß die hier verwalteten Adressen immer ein Viertel der wirklichen Adressen darstellen. Aus diesem Grunde ist auch immer das oberste Bit der Verwaltungsdaten frei und kann für die Gültigkeitsinformationen verwendet werden.

7.6.2 AccessMemory

Die Initialisierung eines `AccessMemory`-Objekts legt die beiden `Cache`-Objekte und den `MemoryDispatcher` an. Außerdem werden einige statische Tabellen für die Cache-Selektion und die Adreßumsetzung angelegt. Anhand dieser Tabellen ist eine sehr effiziente Arbeit von `AccessMemory` möglich. `AccessMemory` stellt verschiedene Funktionsgruppen zur Verfügung:

- Für die Verwaltung der Caches sind mehrere Methoden notwendig, die das Verhalten der im Koprozessor 0 angesiedelten Steuerungsbits nachbilden und die Caches von darunterliegenden Speicherschichten trennen und tauschen können.
- Die Übersetzung von virtuellen zu physikalischen Adressen wird einerseits anhand statischer Tabellen vorgenommen. Andererseits wird auch die Klasse `MMU` aufgerufen, die eine dynamische Adreßumsetzung entsprechend dem IDT R3051E enthalten kann. Diese Funktion ist bisher nicht implementiert. Für den Kern interessant sind die beiden Methoden `translateInstrAdr()` und `translateDataAdr()`, die vor dem eigentlichen Zugriff die Adresse übersetzen und den zu verwendenden Cache auswählen.
- Mit zwei Methoden kann `InstrFrame` ein Befehlswort aus dem `InstrCache` oder darunter liegendem Speicher lesen: `readInstr()` initiiert den Lesevorgang und `fetchInstr()` holt das gelesene Ergebnis ab.
- Für den lesenden Datenzugriff stehen entsprechend `readWord()` und `getReadResult()` zur Verfügung. Hier wird der `DataCache` oder der darunter liegende Speicher verwendet.
- Das Schreiben von Daten erledigt die Methode `writeWord()`, die den Daten-Cache oder den darunter liegenden Speicher verwendet.

- Zur Abbildung des Zeitverhaltens existieren, wie in allen darunter liegenden Schichten auch, `beginCycle()` und `endCycle()`. In `AccessMemory` werden damit nur die Wartezyklen verwaltet.

7.6.3 Implementation des Memory-Interface

Zusätzlich zu den oben beschriebenen Methoden bildet `AccessMemory` das komplette `Memory`-Interface ab. Dabei werden alle Zugriffe durch die Adreßumsetzungslogik bearbeitet, die angegebenen Adressen decken also den virtuellen Adreßraum ab. Die eigentlichen Zugriffe werden dann an das `Memory`-Interface des `MemoryDispatcher` weitergeleitet (siehe Abschnitt 7.7.3). Damit bietet das `Memory`-Interface von `AccessMemory` eine Möglichkeit, den kompletten virtuellen Adreßraum darzustellen.

7.7 Implementation des MemoryDispatcher

Das Konzept des `MemoryDispatcher` wurde schon in Abschnitt 6.4.3 vorgestellt. Aus diesem Grunde wird an dieser Stelle nur noch einmal kurz das Prinzip erklärt. Der `MemoryDispatcher` bietet die Möglichkeit, den physikalischen Adreßraum durch verschiedene unterschiedliche Speicherobjekte, die dem `Memory`-Interface entsprechen, abzudecken.

Zusätzlich zur Möglichkeit, Speicherzugriffe auf verschiedene Speicherobjekte durchzuführen, implementiert der `MemoryDispatcher` noch die vier Schreibpuffer, mit denen die Schnittstelle nach außen entlastet und der Prozessor beschleunigt werden soll.

7.7.1 Segmentierung des Adreßraums

Der `MemoryDispatcher` teilt den physikalischen Adreßraum in Segmente ein. Die bei jeder Anfrage an den `MemoryDispatcher` übergebene Adresse wird aufgeteilt: Die oberen 16 Bit geben die Segmentnummer an (insgesamt existieren 65536 Segmente). Innerhalb eines Segments sind dann 16384 Worte (entspricht dann 65536 Bytes) adressierbar.

Jedem Segment kann ein Speicherobjekt zugewiesen werden, an das der `MemoryDispatcher` Anfragen weiterleitet. Wenn ein Speicherobjekt kleiner als ein Segment ist, wird das Speicherobjekt einfach mehrmals hintereinander in das Segment eingeblendet (das ist dadurch realisiert, daß Speicherobjekte nur so viele Adreßbits beachten, wie für ihre Größe sinnvoll ist, alle weiteren werden ignoriert). Da ein Speicherobjekt immer die volle Adresse (nicht nur die Adresse innerhalb eines Segments) mitgeteilt bekommt, kann sich ein Speicherobjekt auch über mehrere, nicht zwingend zusammenhängende Segmente erstrecken. Implementiert wird dieses Verhalten durch ein Array, das für jedes Segment ein dem `MipsMemory`-Interface entsprechendes Objekt enthalten kann. Für einen Speicherzugriff werden einfach die 16 höchstwertigsten Bit als Array-Index genommen und das enthaltene Speicherobjekt aufgerufen.

7.7.2 Zustandsdiagramm für den Speicherzugriff

Im `MemoryDispatcher` ist ein Automat enthalten, der das Zeitverhalten der verschiedenen Speicherobjekte und die Benutzung der Schreibpuffer koordiniert. Stellt `AccessMemory` oder einer der Caches eine Anfrage an den `MemoryDispatcher`, so wird diese Anfrage abhängig von der Füllung der Schreibpuffer und letzten Anfrage bearbeitet oder durch Wartezyklen verzögert. Wird eine Anfrage an den `MemoryDispatcher` gestellt, so können drei verschiedene Fälle auftreten:

- Sind alle Schreibpuffer belegt, so muß zuerst der älteste Schreibauftrag ausgeführt werden, damit wieder ein leerer Schreibpuffer zur Verfügung steht. Erst danach kann die eigentliche Anfrage bearbeitet werden.
- Liegt ein Leseauftrag vor und sind nicht alle Schreibpuffer belegt, so wird der Leseauftrag ausgeführt.
- Wenn kein Leseauftrag vorliegt, wird unter der Voraussetzung, daß einer der Schreibpuffer gefüllt ist, ein Schreibauftrag ausgeführt.

7.7.3 Implementation des Memory-Interface

Ähnlich wie die Implementation des `Memory`-Interface für `AccessMemory` bietet auch der `MemoryDispatcher` eine Darstellungsschnittstelle zu seinem kompletten Adreßraum. Auf dieser Ebene werden physikalische und keine virtuellen Adressen mehr verwendet. Alle Zugriffe über dieses Interface werden direkt an die entsprechend den Segmenten zugeordneten Speicher-Objekte weitergeleitet, so daß der komplette Speicherinhalt eingesehen werden kann.

7.8 TinyBPMemory

Ein besonderes Speicherobjekt mit der Unterstützung von Unterbrechungspunkten ist der `TinyBPMemory`. Dieses das `MipsMemory`-Interface implementierende Speicherobjekt bietet die Möglichkeit, zu jeder Adresse Unterbrechungspunkte zu setzen. Wenn dann auf eine solche Adresse zugegriffen wird, kann eine Unterbrechung der Programmausführung innerhalb des MIPS-Simulationsmodells ausgelöst werden.

Für jede Speicherzelle des `TinyBPMemory`-Objektes werden zusätzlich vier verschiedene Werte verwaltet. Der erste Wert beschreibt, ob bei einem lesenden Zugriff eine Unterbrechung ausgelöst wird. Zusätzlich kann dieser Wert entscheiden, ob bei jedem Lesezugriff oder nur bei einem Lesezugriff auf einen bestimmten Wert (das ist der zweite) unterbrochen werden soll. Entsprechende Werte sind für schreibende Zugriffe auf den `TinyBPMemory` vorhanden. Ein `TinyBPMemory`-Objekt kann für zwei verschiedene Zwecke eingesetzt werden: Einerseits kann der Ablauf eines Programms verfolgt und für bestimmte wichtige Punkte innerhalb des Programmablaufs Unterbrechungspunkte gesetzt werden. Andererseits kann auch der Zugriff auf bestimmte Daten überwacht werden.

Die Arbeitsweise des `TinyBPMemory` wird nun durch ein Beispiel verdeutlicht. Für die Speicheradresse 25 kann festgelegt werden, daß alle Schreibzugriffe, die eine 44 schreiben, und alle Lesezugriffe, die eine 45 lesen, zu einer Unterbrechung führen.

7.9 HadesInterface

Die Verbindung zwischen HADES und dem MIPS-Simulationsmodell wird durch die Klassen `HadesInterface` und `IDTR3051` realisiert. Dabei enthält das `HadesInterface` die eigentliche Logik und `IDTR3051` Makros zum Abfragen und Setzen von Signalen für HADES.

7.9.1 Automat für den Speicherzugriff

Der Automat des `HadesInterface` bietet vier verschiedene Abläufe für DMA, Read, Burst-Read und Write. Das (aus Darstellungsgründen etwas vereinfachte) Zustandsdiagramm des Automaten wird in Abbildung 22 dargestellt. Dabei wird der `IDLE`-Zustand für eine übersichtliche Darstellung zweimal abgebildet, ist aber nur einmal vorhanden.

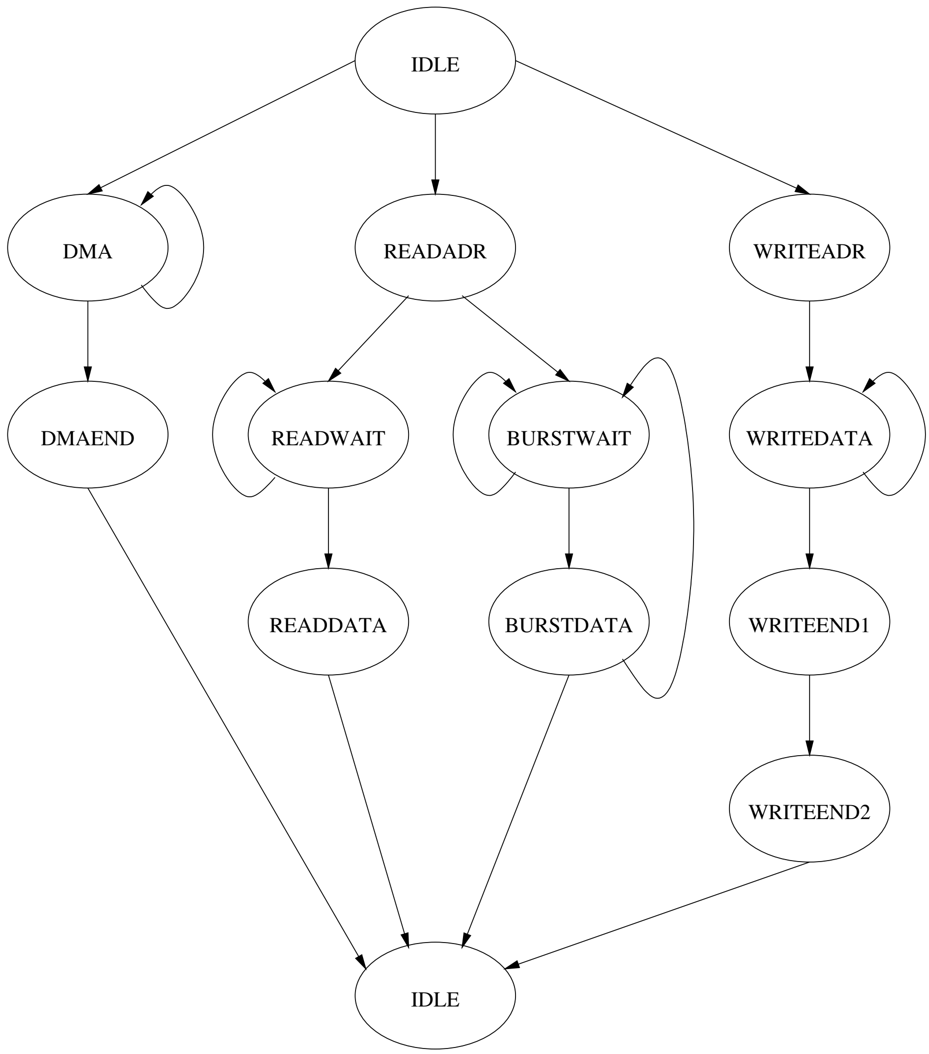

Abbildung 22: Zustandsdiagramm für den Speicherzugriff

Das Ziel dieses Automaten ist die exakte Nachbildung des Zeitverhaltens der IDT-R3051-Speicherschnittstelle.

7.9.2 Implementation des Memory-Interface

Im Gegensatz zur Implementation des `Memory`-Interfaces der darüber liegenden Schichten besitzt das hier implementierte Interface keine weitere Funktionalität. Ab hier benötigen Speicherzugriffe zwingend ein Zeitverhalten und müssen sich an das Zugriffsprotokoll des IDT R3051 halten, so daß direkte Zugriffe nur für die Darstellung der Speicherinhalte ohne gegebenes Taktsignal und eine verstreichende Zeitkomponente nicht möglich sind. Daher werden Schreibzugriffe auf das `Memory`-Interface des `HadesInterface` einfach ignoriert und Lesezugriffe liefern immer eine Null zurück.

7.9.3 IDTR3051

Die Klasse `IDTR3051` ist direkt von `hades.simulator.SimObject` abgeleitet und implementiert alle für ein `hades.simulator.Simulatable` notwendigen Methoden. Hier werden alle Ports zur Verbindung zwischen HADES und dem MIPS-Simulationsmodell angelegt. Die `evaluate()`-Methode reagiert auf Interrupts, das Takt- und das Resetsignal. Von hier aus wird `Core.cycle()` aufgerufen, damit dann der Kern taktgesteuert arbeiten kann.

Außerdem werden von `IDTR3051` diverse Makromethoden zur Verfügung gestellt. Diese bieten dem `HadesInterface` komfortable Möglichkeiten, verschiedene Signale und Busse zu setzen oder abzufragen. Dabei setzt `IDTR3051` die von HADES genutzten `StdLogic1164`-Signale in boolesche Werte um, die dann innerhalb des MIPS-Simulationsmodells Verwendung finden.

7.10 MemoryLoader

Der `MemoryLoader` bietet die Möglichkeit, einem Objekt, das das `Memory`-Interface implementiert, zusätzliche Fähigkeit für Dateizugriffe zu bieten. Dadurch kann ein Speicherobjekt seinen Inhalt in eine Datei schreiben oder den Inhalt einer Datei in das Speicherobjekt einlesen. Da alle am MIPS-Simulationsmodell beteiligten Speicherobjekte dem `Memory`-Interface entsprechen, können alle Speicher mit dem `MemoryLoader` benutzt werden. Sogar der komplette virtuelle Speicher (die Schnittstelle dazu bietet `AccessMemory`) als auch der komplette physikalische Speicher (`MemoryDispatcher`) können theoretisch gespeichert werden, das bedeutet allerdings aufgrund der Größe gewisse Probleme.

Speziell für die Zusammenarbeit mit dem `MemoryDispatcher` besitzt der `MemoryLoader` einige spezielle Fähigkeiten. Der `MemoryLoader` kann den `Memory-Dispatcher` automatisch konfigurieren. Dieses Verhalten wird im Abschnitt 7.10.4 genauer beschrieben.

7.10.1 Dateiformate

Der `MemoryLoader` kann zwar nur ein Format schreiben (das ROM-Format, siehe unten), ist aber in der Lage, einige verschiedene Formate zu lesen. Es folgt die

Aufzählung der unterstützten Dateiformate. Da eine Vorversion des `MemoryLoader` schon bei [Rink 99] eingesetzt wurde, befinden sich unter der Liste der unterstützten Dateiformate einige eher unübliche.

ROM: In [Rink 99] wurden Pic-ROM-Dateien eingeführt. Sie enthalten nacheinander entweder Adressen (diese sind durch einen abschließenden Doppelpunkt zu erkennen) oder Datenwerte, getrennt durch Leerzeichen oder Zeilenvorschübe. Dabei sind sowohl Adressen als auch Daten hexadezimale Werte. Das Einlesen einer Pic-ROM-Datei geht folgendermaßen vor sich: Zuerst wird der Adreßzähler auf 0 gesetzt. Folgend wird in einer Schleife bis zum Dateiende der nächste Wert eingelesen. Wenn es sich bei diesem Wert um eine Adresse handelt, wird der Adreßzähler auf diesen Wert gesetzt. Handelt es sich nicht um eine Adresse, dann wird der Wert an die durch den Adreßzähler angegebene Adresse geschrieben und der Adreßzähler um eins erhöht.

ELF: Das *Executable and Linking Format*, wie es zum Beispiel auch von ausführbaren Linux-Dateien verwendet wird, kann ebenfalls eingelesen werden. Dabei werden allerdings keine Relokationen vorgenommen, da diese insbesondere bei der Nutzung der besonderen Fähigkeiten des `MemoryDispatchers` nicht nötig sind. Da das ELF-Format das wichtigste Dateiformat im Umgang mit dem MIPS-Simulationsmodell ist, folgt im Abschnitt 7.10.2 eine genauere Beschreibung.

ECOFF: ECOFF steht für *Extended Common Object File Format*. Dieses Format wird nur rudimentär unterstützt, da das ELF-Format eigentlich dieselben Zwecke erfüllt und statt dessen genutzt werden sollte. Es wird einfach die Adresse der ausführbaren Maschinenbefehle festgestellt und alles in der Datei Folgende ab Adresse null in den Speicher geschrieben. Für einfach strukturierte Binärdateien führt diese Methode zum Erfolg, es kann aber keine Garantie gegeben werden, daß dieses Vorgehen auch bei komplexeren Dateien anwendbar ist.

RAW: Falls eine Datei nicht erkannt wird, wird sie einfach beginnend ab Adresse null Byte für Byte in den Speicher geschrieben.

7.10.2 Das ELF-Dateiformat

Verschiedene Unix-Varianten, so auch Linux, benutzen das ELF-Dateiformat. ELF steht für *Executable and Linking Format* und bietet ein Format sowohl für Objekt-Dateien als auch für ausführbare Programme. Da der `MemoryLoader` nur ausführbare Programme einlesen kann, werden nur diese hier näher beschrieben.

Ein ELF-Programm besteht aus dem ELF-Header und einigen Segmenten. Zu jedem Segment ist ein eigener Header enthalten. Im globalen ELF-Header sind allgemeine Informationen wie Architektur, Versionsnummer, Byte-Orientierung und Anzahl der Segmente enthalten. Die einzelnen Segmente beinhalten entweder das Maschinenspracheprogramm oder vordefinierte Variablen. Für jedes Segment wird schon vom Linker festgelegt, an welcher Adresse im Speicher sich das Segment idealerweise befinden sollte. Durch zusätzliche Segmente mit entsprechenden Informationen ist aber auch eine Relokation möglich. Das ELF-Format ist insbesondere darauf ausgelegt, daß eine ELF-Datei direkt von einer MMU in den virtuellen Adreßraum eingeblendet und nach wenig Bearbeitung direkt ausgeführt werden kann. Diese Fähigkeit wird vom MIPS-Simulationsmodell nicht genutzt, da hier keine MMU vorhanden ist.

Dank seiner weiten Verbreitung sind viele Entwicklungsumgebungen erhältlich, die ihre Ergebnisse im ELF-Format schreiben können. So unterstützt das MIPS-Simulationsmodell durch die Implementation des ELF-Formats praktisch beliebige Entwicklungswerkzeuge für alle wichtigen Programmiersprachen. Das ELF-Format wird in [Rusling 99] beschrieben.

7.10.3 EntryHandler-Interface

Leider stellt Java keine Prozedur- (oder im Java-Umfeld besser zu bezeichnen als Methoden-) Variablen zur Verfügung. Um einen Rückrufmechanismus zu implementieren, muß statt dessen ein Interface definiert werden, das genau die gewünschte Methode enthält. Alle Klassen, die eine Rückrufmethode anbieten, müssen dieses Interface implementieren, damit das die Methoden-Variable aufrufende Objekt eine Referenz auf das die Rückrufmethode enthaltende Objekt verwalten und dessen Rückrufmethode aufrufen kann. Der `EntryHandler` ist ein solches Rückruf-Interface. Er stellt nur eine Methode, `setEntry()`, zur Verfügung.

Im MIPS-Simulationsmodell gibt es das Problem, daß, sobald mit dem `MemoryLoader` ein neues Programm in den Programmspeicher geladen wird, die Einsprungadresse in das Programmzähler-Register geschrieben werden muß. Der `MemoryLoader` weiß aber nichts von irgendwelchen Registern, er hat nur Zugriff auf das `Memory`-Objekt, das er füllen soll. Deshalb wird im `Memory`-Objekt mit `installEntryHandler()` eine Referenz auf den `Core` hinterlegt. Ruft nun der `MemoryLoader` für seinen `Memory` (dem Programmspeicher), `setEntry()` auf, wird dieser Methodenaufruf an den `Core` weitergeleitet, der dann Zugriff auf die `Register` hat und den Programmzähler auf den neuen Wert setzen kann.

7.10.4 Zusammenarbeit mit dem MemoryDispatcher

Der `MemoryLoader` kann sich speziell verhalten, wenn er den `MemoryDispatcher` (den physikalischen Adreßraum) mit einem ELF-File füllen soll. Das ELF-File ist in mehrere verschiedene Segmente aufgeteilt, die nicht zusammenhängend beliebig über den virtuellen Adreßraum verteilt sind. Der `MemoryLoader` kann für jedes Segment, das er aus einem ELF-File liest, ein eigenes `TinyBPMemory`-Objekt anlegen und dieses mit dem entsprechenden Segment im `MemoryDispatcher` verknüpfen. Solche vom `MemoryLoader` im `MemoryDispatcher` angelegten Speicherobjekte bekommen den Namen *ELF* mit fortlaufender Nummerierung.

Die Einschaltkonfiguration des MIPS-Simulationsmodells in Zusammenarbeit mit HADES blendet das `HadesInterface` auf dem gesamten Adreßraum ein. Wenn direkt nach dem Start des MIPS-Simulationsmodells eine ELF-Datei mit dem `MemoryLoader` in den `MemoryDispatcher` eingelesen wird, legt der `MemoryLoader` automatisch neue Segmente für die verschiedenen ELF-Segmente an, so daß alle aus der ELF-Datei geladenen Segmente innerhalb des MIPS-Simulationsmodells als schnelle interne Speicher zur Verfügung stehen und der restliche Adreßraum weiterhin das `HadesInterface` benutzt.

7.11 Visualisierung

Die Visualisierung ist vom eigentlichen Kern strikt getrennt, damit der Kern auch ohne sie lauffähig ist und sich eine bessere Trennung zwischen der eigentlichen Funktion der Anwendung und deren Anzeige ergibt. Die für eine Visualisierung wichtigen Komponenten des MIPS-Simulationsmodells sind die verschiedenen Speicherobjekte und die Pipeline. Für die Konfiguration des `MemoryDispatcher` muß zusätzlich noch ein Mechanismus geboten werden.

Grundsätzlich soll das MIPS-Simulationsmodell in der Lage sein, sowohl mit HADES zusammen als auch eigenständig arbeiten zu können. Daraus ergeben sich einige Unterschiede in der Anzeige bei den beiden verschiedenen Möglichkeiten.

7.11.1 MemoryView-Interface

Es gibt die Möglichkeit, alle Klassen, die das `Memory`-Interface implementieren, auf verschiedene Art und Weise darzustellen. Als einheitliche Schnittstelle zur Darstellung eines `Memory` gibt es das `MemoryView`-Interface, das dann von allen `Memory`-darstellenden Klassen implementiert werden muß. Das `MemoryView`-Interface enthält (unter anderen) folgende wichtige Methoden:

init(): Diese Methode setzt die Anzeige zurück. Normalerweise wird hier wieder der Anfang des Speichers dargestellt, unabhängig davon, welchen Ausschnitt des Speichers der Benutzer vorher angesehen hat. Diese Methode ist zum Beispiel nach einem Reset oder dem Laden eines neuen Programms sinnvoll.

update(): Soll der Inhalt eines `MemoryView` komplett neu gezeichnet werden, bietet sich `update()` an.

smartUpdate(): Falls sich nur Einzelteile des sichtbaren Speicherausschnitts geändert haben, wird mit `smartUpdate()` ein leistungsfähigerer Mechanismus als `update()` geliefert. Diese Methode kann bei einzelnen `MemoryView` auch direkt `update()` aufrufen.

7.11.2 MemoryViewHandler

Für die Verwaltung von mehreren `MemoryView`, die gleichzeitig denselben `Memory` darstellen, steht der `MemoryViewHandler` zur Verfügung. Für jedes `Memory`-Objekt muß aus diesem Grunde ein `MemoryHandler`-Objekt angelegt werden, das dann den einzelnen `MemoryViews` übergeben werden kann. Der `MemoryViewHandler` besitzt als Member-Variable einen `MemoryLoader`. Die Methoden des `MemoryViewHandlers`:

addView(): Mit dieser Methode kann sich ein `MemoryView` beim `MemoryViewHandler` anmelden.

removeView(): Falls ein `MemoryView` wieder geschlossen werden soll, kann er sich beim `MemoryViewHandler` auch wieder abmelden.

getMemory(): Diese Methode liefert den `Memory` zurück, für den der `MemoryViewHandler` die `MemoryViews` verwaltet.

init(): Alle angemeldeten `MemoryView` führen ihr `init()` aus.

update(): `update()` für alle beteiligten `MemoryView`.

smartUpdate(): Rundruf an alle dazugehörigen `MemoryView` für ein `smartUpdate()`.

load(): Wenn der Benutzer in einem `MemoryView` den Load-Knopf gedrückt hat, muß diese Methode aufgerufen werden. Sie benutzt dann den `MemoryLoader` für das Einlesen der Datei. Danach wird mit `init()` und `update()` dafür gesorgt, daß der neue Inhalt auch angezeigt wird.

save(): Entsprechend zum `load()` gibt es auch ein `save()`.

7.11.3 MemoryHexView

Als erste das `MemoryView`-Interface implementierende Klasse wird nun der `MemoryHexView` vorgestellt. Diese Klasse stellt einen `Memory` für den Benutzer in Form einer verschiebbaren Liste mit Hexadezimalwerten dar. Der Benutzer kann einzelne Zeilen dieser Liste mit der Maus auswählen und in einem `TextField` ändern.

Der `MemoryHexView` bietet sich besonders für die Anzeige von Inhalten an, die nur hexadezimal dargestellt werden müssen. Die beiden dem `Memory`-Interface entsprechenden Klassen, die durch einen `MemoryHexView` dargestellt werden, sind die `Register` und der `Copro0`.

7.11.4 MemoryDisView

Eine weitere Implementation des `MemoryView`-Interface ist der `MemoryDisView`. Er stellt den Inhalt eines, das `Memory`-Interface implementierenden, Speicherobjektes als disassemblierte Liste dar. Dabei wird zeilenweise zuerst die Adresse, dann der Inhalt als Hexadezimalwert und zuletzt der disassemblierte Inhalt angezeigt.

Wie der `MemoryHexView` bietet auch der `MemoryDisView` die Knöpfe `Load` und `Save`, die sich entsprechend dem `MemoryHexView` verhalten.

Die unter Zuhilfenahme des `InstrFrame` disassemblierten Speicherworte (siehe Abschnitt 7.2.4) werden direkt in einen `Canvas` geschrieben. Neben diesem erscheint ein `Scrollbar`, mit dem die Ausgabe bewegt werden kann.

Falls der dargestellte Speicher nicht nur das `Memory`-Interface implementiert, sondern vom `BreakPointMemory` abgeleitet ist (das ist der `TinyBPMemory`), bewirkt ein Mausklick auf die Adresse einer Zeile, daß ein neues Fenster der Klasse `ShowDisBPFrame` geöffnet wird, in dem die Unterbrechungspunkte für diese Speicherstelle beeinflußt werden können.

7.11.5 ShowPipe

Ein schwieriges Problem ist die Darstellung einer Pipeline während der Ausführung mehrerer Befehle. Aus Geschwindigkeitsgründen sollte die Darstellung möglichst schnell sein, andererseits müssen aber alle relevanten Informationen angezeigt werden. Zusätzlich ist es natürlich noch notwendig, den Zustand der Pipeline auf den ersten Blick identifizieren zu können.

`ShowPipe` ist der Versuch, die in der Simulation enthaltene Pipeline einerseits übersichtlich und andererseits möglichst kompakt (schnell) und informativ darzustellen. Als Ergebnis wird nun für jeden `InstrFrame` eine Spalte angezeigt, die für jede Aktion der Pipeline eine Zeile mit zur Aktion gehörigen Informationen enthält. Dabei wird die Zeile der Aktion, die zuletzt ausgeführt wurde, farblich hervorgehoben.

Die Informationen, die `ShowPipe` darstellt, werden von `InstrFrame` und den einzelnen `*Instr` zur Verfügung gestellt. Dabei kümmert sich `ShowPipe` um die Aufbereitung der einzelnen Parameter für die Darstellung.

8 Benutzung des MIPS-Simulationsmodells

Die Benutzung des Simulationsmodells gliedert sich in drei verschiedene Bereiche: Einerseits muß die Bedienung der Oberfläche ausführlich erklärt werden. Außerdem muß auf die Benutzung einer Hochsprache zur Erstellung von durch das Simulationsmodell ausführbaren Programmen eingegangen werden. Diesem Bereich läßt sich noch die Frage nach einem Betriebssystem, das vom Simulationsmodell ausgeführt werden soll, zuordnen. Zusätzlich soll die Vorstellung von einigen Anwendungsbeispielen die Benutzung verdeutlichen und gleichzeitig die Leistungsfähigkeit des MIPS-Simulationsmodells zeigen.

8.1 Bedienung der Oberfläche

Das MIPS-Simulationsmodell besitzt eine grafische Benutzeroberfläche, die einen einfachen Zugang zu allen wesentlichen internen Komponenten bietet. Hier können die Inhalte aller Speicher dargestellt und durch den Benutzer geändert werden. Auch der Zustand der Pipeline wird grafisch ausgegeben.

Die Oberfläche gliedert sich in mehrere Fenster, die für die verschiedenen Schichten des MIPS-Simulationsmodells zuständig sind. Das Hauptfenster enthält die Bedienungselemente für die Kern- und Koprozessorkomponenten Register und Pipeline. Ein weiteres Fenster ermöglicht die Konfiguration des `MemoryDispatcher`. Für jedes einzelne Speicherobjekt (auch für gesamte Adreßräume und Caches) können Fenster zur Darstellung und Änderung des Speicherinhalts geöffnet werden.

Das Hauptfenster enthält ein Menu, über das die Konfiguration des MIPS-Simulationsmodells vorgenommen werden kann. Außerdem werden alle weiteren Fenster über das Menu geöffnet.

8.1.1 Das Hauptfenster

Im Hauptfenster des MIPS-Simulationsmodells wird der Inhalt der normalen und der Koprozessor-Register als `MemoryHexView` dargestellt. Jeder der beiden `MemoryHexView` besteht aus einer Liste zur Darstellung des Inhalts, einigen Knöpfen und einem `TextField` zur Änderung des Inhalts. Jede Zeile der Liste enthält die Nummer des ersten in der Zeile dargestellten Registers und den hexadezimalen Inhalt von vier Registern. Wird mit der Maus auf eine Zeile der Liste geklickt, dann erscheint der Inhalt dieser Zeile im `TextField` unterhalb und kann geändert werden. Sobald die Änderung mit der Enter-Taste beendet wurde, wird der Inhalt des `TextField` eingelesen und in den `Memory` übernommen. Das Parsing der Eingabe wird dabei ähnlich wie das in Abschnitt 7.10 beschriebene Einlesen von `ROM`-Dateien gehandhabt: Werte mit folgendem Doppelpunkt sind Adressen für nachfolgende Werte ohne Doppelpunkt, die dann aufsteigend an die entsprechenden Adressen geschrieben werden.

Soll der Inhalt des zum `MemoryHexView` gehörenden `Memory` aus einer Datei gelesen oder in eine Datei geschrieben werden, so kann der Benutzer die Knöpfe `Load` und `Save` benutzen. Beide Methoden öffnen einen `FileDialog`, um vom Benutzer den Namen der Datei zu erfahren, und rufen dann die entsprechenden Methoden des `MemoryViewHandlers` auf. Als weitere Bedienelemente stehen für die Benutzung eines `MemoryHexView` noch die Knöpfe `Up` und `Down` zur Verfügung, mit denen die

Anzeige der Liste im `Memory` bewegt werden kann.

Unterhalb der Registerdarstellung erscheint die Pipeline. Die Zeilen dieser Darstellung sind die verschiedenen Pipeline-Aktionen (sie werden am linken Rand benannt). In Spalten erscheinen die Ergebniswerte der einzelnen `InstrFrame`-Methoden, die die Pipeline bilden, für jedes instanziierte `InstrFrame`-Objekt. Das Ergebnis der zuletzt ausgeführten Methode jedes `InstrFrame` wird dabei in rot ausgegeben. Falls ein `InstrFrame` inaktiv ist, wird dessen aktuellstes Ergebnis statt dessen blau ausgegeben.

Abbildung 23 zeigt das Hauptfenster nachdem schon einige Befehle bearbeitet wurden.

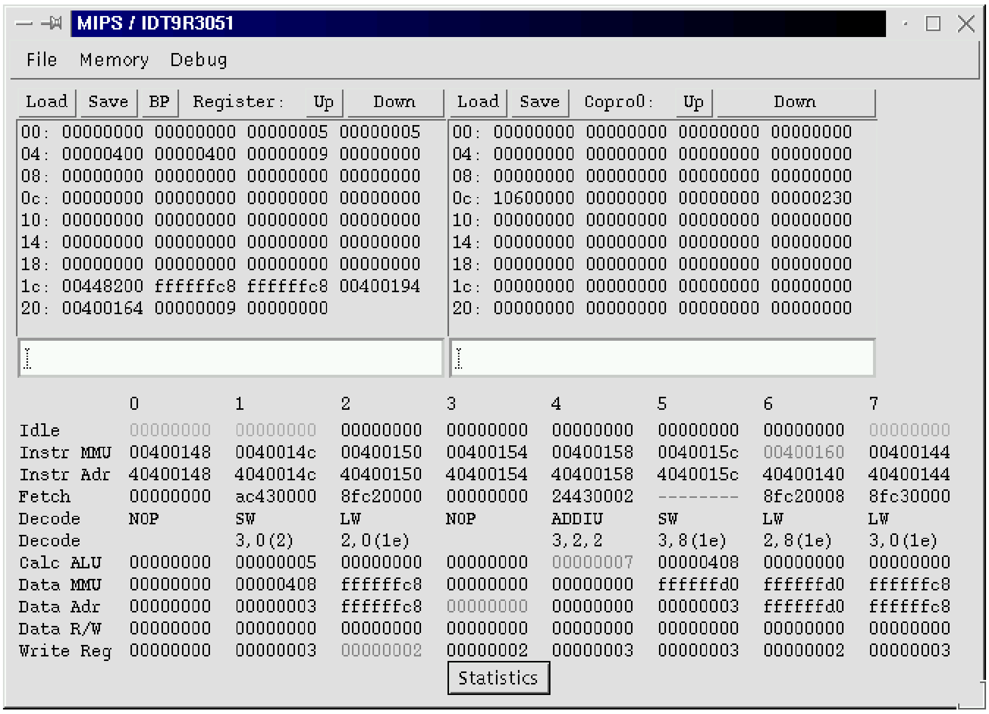

Abbildung 23: Das Hauptfenster des MIPS-Simulationsmodells

8.1.2 Das Menu

Im Hauptfenster ist das Menu zur Konfiguration und zum Öffnen der weiteren Fenster enthalten. Es besteht aus den drei Hauptmenupunkten `File`, `Memory` und `Debug`. Der Hauptmenupunkt `File` besteht aus den folgenden Untermenupunkten:

New: Das MIPS-Simulationsmodell wird in seinen Anfangszustand zurückgesetzt.

Open...: Nach der Auswahl einer ELF-Datei kann diese eingelesen werden. Dabei werden für die verschiedenen ELF-Segmente neue Speicherobjekte angelegt und der `MemoryDispatcher` wird automatisch konfiguriert.

Close: Wurde das MIPS-Simulationsmodell von HADES aus aufgerufen, so wird durch die Close-Funktion einfach das Hauptfenster geschlossen und kann jederzeit durch HADES wieder geöffnet werden. Läuft das MIPS-Simulationsmodell alleinstehend, so schließt die Close-Funktion das Fenster und beendet das Simulationsmodell.

Der Hauptmenupunkt `Memory` faßt alle Aktionen zur Visualisierung der verschiedenen an der Simulation beteiligten Speicher und deren Konfiguration zusammen. Er besteht aus den folgenden Untermenupunkten:

Configuration...: Durch diese Funktion wird ein Fenster zur Konfiguration des `MemoryDispatcher` geöffnet.

Show virtual address space...: Das `Memory`-Interface von `AccessMemory` kann den Inhalt des virtuellen Adreßraums zur Verfügung stellen. Der Aufruf dieses Menupunktes öffnet ein `MemoryDisView` zur disassemblierten Darstellung des virtuellen Adreßraums.

Show physical address space...: Entsprechend zum vorigen Punkt kann auch das `Memory`-Interface des `MemoryDispatcher` durch einen `MemoryDisView` dargestellt werden.

Invalidate caches: Manchmal kann es sinnvoll sein, die Caches zu leeren (bzw. alle Cache-Einträge ungültig zu machen).

Show instruction cache...: Wenn der Inhalt des `InstrCache` angezeigt werden soll, öffnet dieser Menu-Punkt einen `MemoryDisView` zur Darstellung des Inhalts und der Verwaltungsinformationen des Caches.

Show data cache...: Entsprechend kann auch der `DataCache` angezeigt werden.

Unterhalb des Hauptmenupunktes `Debug` sind alle Funktionen zusammengefaßt, die dem Benutzer eine detailliertere, aber auch technischere Einsicht in den internen Ablauf des MIPS-Simulationsmodells ermöglichen. In diesem Menu kann durch die folgenden Schalter beeinflußt werden, welche internen Komponenten Debug-Informationen liefern sollen:

Output to standard out: Alle durch folgende Schalter einzustellenden Debug-Ausgaben werden auf der Standard-Ausgabe angezeigt.

Output to './showmips.dbg': Entsprechend können alle Debug-Ausgaben auch in die Datei `showmips.dbg` im aktuellen Verzeichnis umgeleitet werden.

Core: Wird dieser Schalter aktiviert, dann meldet das `Core`-Objekt alle Methodenaufrufe mit Parametern. Hier kann erkannt werden, ob die Simulation normal arbeitet und wieviele Takte seit dem letzten Reset vergangen sind.

Instruction frame: Dieser Schalter sorgt für Debug-Ausgaben aller `InstrFrame`-Objekte. Jede Pipeline-Aktion wird dokumentiert.

Instruction Decoder: Der `Decoder` meldet sich nach Aktivierung dieses Schalters immer, wenn er ein Befehlswort identifiziert hat.

Exception handler: Falls eine Exception auftritt, kann der `ExceptionHandler` für detaillierte Debug-Ausgaben sorgen.

Access memory: Die Arbeitsweise von `AccessMemory` wird durch Aktivierung dieses Schalters ausgegeben.

Memory dispatcher: Die Debug-Ausgaben des `MemoryDispatcher` sind besonders ergiebig: Diese Klasse zeigt sowohl das wirklich benutzte Speicherobjekt als auch den Zustand seines internen Automaten.

HADES interface: Auch die Debug-Ausgaben des `HadesInterface` enthalten den Zustand des Automaten. Zusätzlich werden alle Zugriffe auf Signale von HADES protokolliert.

Instruction cache: Alle Zugriffe auf den `InstrCache` können als Debug-Ausgaben dargestellt werden.

Data cache: Auch der `DataCache` kann seine Benutzung protokollieren.

Alle unter dem Hauptmenupunkt `Debug` angeordneten Schalter beeinflussen die Debug-Ausgaben. Dabei werden die Debug-Informationen in einem speziellen Format aufbereitet, damit die Übersichtlichkeit nicht gänzlich verloren geht. Dieses Format wird in Abschnitt 8.1.6 dokumentiert.

8.1.3 Speicherdarstellung

Für die Darstellung der meisten `MipsMemory`-Speicherobjekte wird der `MemoryDisView` mit Ausgabe der disassemblierten Befehle verwendet. In jeder Zeile wird die Adresse eines Speicherwortes, der hexadezimale Inhalt des Wortes und dessen disassemblierte Befehlsbeschreibung ausgegeben. Wenn der hexadezimale Inhalt eines Befehlswortes angeklickt wird, kann dieser geändert werden. Dabei wird der Wert rot hervorgehoben. Erst wenn nach der Änderung des Wertes die Enter-Taste gedrückt wird, wird der neue Wert in den Speicher übernommen. Durch ein Drücken der Esc-Taste wird der Änderungsvorgang abgebrochen.

Oberhalb der Speicherliste bieten die Knöpfe `Load` und `Save` die Möglichkeit, den Inhalt des Speichers aus einer Datei zu laden oder diesen in einer Datei zu speichern (gespeichert werden können allerdings nur Speicher bis zu einer Größe von 10 Millionen Worten). Daneben erscheint ein Knopf `Debug`, der die Debug-Ausgaben für diesen Speicher ein- oder ausschaltet. Rechts von der Speicherliste erscheint normalerweise ein Scrollbalken mit dem sich im Speicher bewegt werden kann. Bei sehr großen Speichern (wie zum Beispiel bei der Darstellung des virtuellen oder des physikalischen Adreßraums) werden zwei Scrollbalken nebeneinander dargestellt. Der linke Scrollbalken verändert dann die unteren 16 Bit der Adresse, der rechte die oberen 16 Bit.

Obwohl das MIPS-Simulationsmodell unterhalb von `AccessMemory` nicht mehr mit Byte-Adressen, sondern mit Wort-Adressen arbeitet, werden bei der Speicherdarstellung durchgängig Byte-Adressen verwendet. Debug-Ausgaben stehen dazu im Gegensatz, hier werden je nach Schicht oberhalb von `AccessMemory` Byte-Adressen und unterhalb davon Wort-Adressen verwendet. Abbildung 24 zeigt beispielhaft eine Speicherdarstellung.

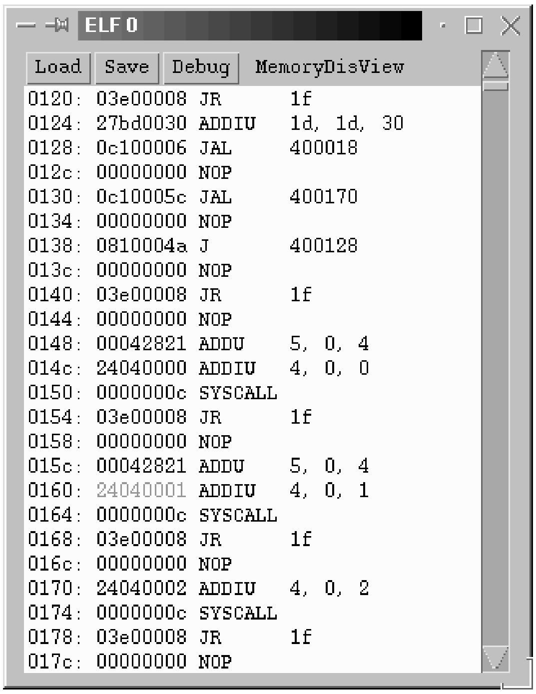

Abbildung 24: Speicherdarstellung: MemoryDisView

Wird ein Speicher von `BreakPointMemory` abgeleitet, so bietet der `MemoryDisView` zusätzliche Möglichkeiten zur Einstellung von Unterbrechungspunkten. In diesem Fall öffnet ein Mausklick auf eine Adresse ein zusätzliches Fenster, mit dem sich die Unterbrechungspunkte für diese Adresse einstellen lassen.

Die Unterbrechungsmöglichkeiten, die das MIPS-Simulationsmodell bietet, gehen über die von normalen Debuggern bekannte Ausstattung hinaus, deshalb muß hier genauer darauf eingegangen werden. Für jede Adresse kann sowohl ein Unterbrechungspunkt für Lese- als auch einer für Schreibzugriffe gesetzt werden. Dabei können diese beiden verschiedenen Unterbrechungsmöglichkeiten entweder bei jedem Wert eine Unterbrechung auslösen oder nur, wenn jeweils ein spezieller Wert aus der Speicher gelesen oder in den Speicher geschrieben wird. So kann zum Beispiel für eine Speicherzelle festgelegt werden, daß alle Schreibzugriffe, die eine sieben schreiben, und alle Lesezugriffe, die eine drei als Ergebnis liefern, eine Unterbrechung auslösen.

Abbildung 25 zeigt Fenster zur Konfiguration der Unterbrechungspunkte.

8.1.4 Konfiguration des MemoryDispatcher

Der `MemoryDispatcher` erlaubt die freie Abbildung von verschiedenen Speicherobjekten auf den Adreßraum. Die Benutzerschnittstelle für den Konfigurationsmechanismus muß die Möglichkeit bieten, neue Speicherobjekte anzulegen, vorhandene zu löschen und die Speicherobjekte in bestimmte Segmente einzublenden.

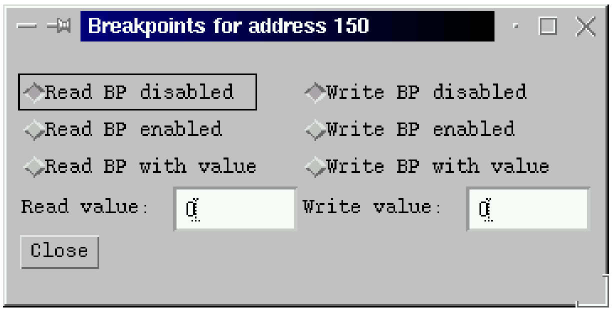

Abbildung 25: Einstellung von Unterbrechungspunkten

Das Fenster für die `MemoryDispatcher`-Konfiguration ist in drei Bereiche unterteilt. Rechts können neue Speicherobjekte angelegt werden, die Mitte zeigt alle vorhandenen Speicherobjekte in einer Liste und links erscheinen alle Segmente des Adreßraums zusammen mit den ihnen zugeordneten Speicherobjekten. Soll ein Speicherobjekt neu angelegt werden, so muß die Größe, der Name und der Typ des Speicherobjekts festgelegt werden. Die verschiedenen Typen:

1: External/HADES: Hier verbirgt sich das `HadesInterface`. Dieses Objekt kann nur einmal angelegt werden.

2: Internal memory: Dieser Speichertyp legt ein `TinyMemory`-Objekt an. Dieser Speichertyp ist schnell und verbraucht verhältnismäßig wenig Speicher, so daß er für normale Simulationszwecke am geeignetsten ist.

3: Internal with breakpoints: Die hierfür verantwortliche Klasse ist `TinyBPMemory`. Das ist ein interner Speicher, der Unterbrechungspunkte zur Verfügung stellt. Dieser Speichertyp verbraucht ungefähr die fünffache Speichermenge wie der `TinyMemory` und sollte deshalb nur dann eingesetzt werden, wenn die Unterbrechungsfunktionalität wirklich benötigt wird.

4: Null, ignore access: Dieser Speichertyp ignoriert Schreibzugriffe vollständig und liefert als Leseergebnis immer eine Null. Hinter diesem Speichertyp verbirgt sich `NullMemory`.

5: Null, report access: Es gibt auch die Möglichkeit, einen `NullMemory` zu instanzieren, der alle Zugriffe in den Debug-Ausgaben protokolliert.

In der Liste der schon existierenden Speicherobjekte werden die Größe, die Nummer des Speichertyps und der Name des Speicherobjekts angezeigt. Hier kann ein Speicherobjekt ausgewählt und angezeigt (ein `MemoryDisView` stellt den Speicherinhalt dar) oder gelöscht werden. Ein Speicherobjekt wird allerdings erst dann wirklich gelöscht, wenn es keinem `MemoryDispatcher`-Segment mehr zugeordnet ist.

Die links erscheinende Übersicht über alle Segmente des `MemoryDispatcher` enthält die virtuelle und die physikalische Adresse des Segments sowie den Namen des zugeordneten Speicherobjekts. Soll einem Segment ein neues Speicherobjekt zugeordnet werden, muß dieses Speicherobjekt aus der Liste der vorhandenen Speicherobjekte ausgewählt, danach auf das gewünschte Segment geklickt und der `Register`-Knopf gedrückt werden. Nun erscheint der neu zugeordnete Speicherobjektname hinter der Segmentadresse.

Abbildung 26 zeigt das Fenster zur Konfiguration des `MemoryDispatcher`.

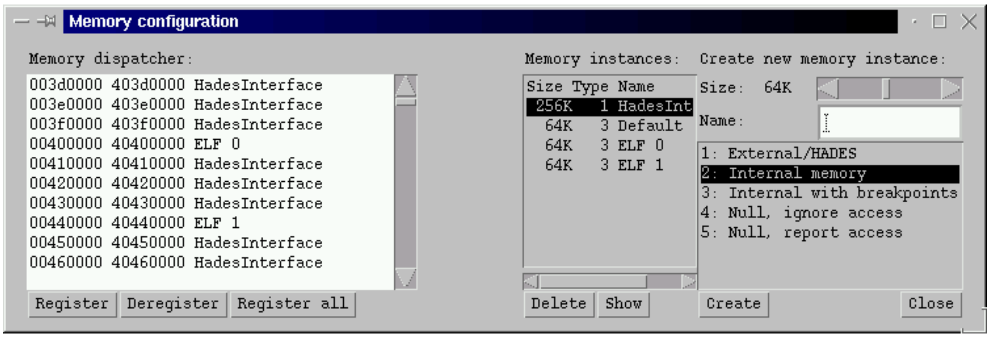

Abbildung 26: MemoryDispatcher-Konfiguration

8.1.5 Benutzung des Simulationsmodells

Von HADES aus wird das Simulationsmodell durch die Schritte `create` -> `by name` -> `hades.models.mips.mipsmemory.IDTR3051` gestartet. Nun kann das MIPS-Hauptfenster geöffnet und ein Programm geladen werden. Sobald HADES eine Simulation durchführt, wird die volle Geschwindigkeit erst erreicht, wenn alle zum MIPS-Simulationsmodell gehörigen Fenster geschlossen sind. Die grafische Darstellung von Speicherinhalten oder der Pipeline kostet wesentlich mehr Rechenleistung als die Simulation des MIPS-Prozessors.

Zur Speicherkonfiguration bei der Benutzung des MIPS-Simulationsmodells zusammen mit HADES ist noch folgendes zu sagen: Direkt nach dem Starten des MIPS-Simulationsmodells werden alle `MemoryDispatcher`-Segmente mit dem `HadesInterface` vorbelegt. Wenn der Benutzer dann ein ELF-Programm lädt und dadurch die für die Programmausführung notwendigen ELF-Segmente automatisch in den Adreßraum eingeblendet werden, ist normalerweise keine weitere Konfiguration notwendig, um das geladene Programm zur Ausführung zu bringen. Natürlich muß bei der Verwendung von speicherbasierter Ein- und Ausgabe darauf geachtet werden, daß die Ein-/Ausgabe-Adressen nicht mit den ELF-Segment-Adressen kollidieren. Eventuell muß der GNU-Linker angewiesen werden, seine Segmente an andere Adressen zu legen.

Anders verhält sich das MIPS-Simulationsmodell, wenn es ohne HADES gestartet wird. In diesem Fall muß `java hades.models.mips.gui.ShowMips` aufgerufen werden. Dadurch erscheint dann das Hauptfenster, in dem in der untersten Zeile sieben Knöpfe die Möglichkeit bieten, eine unterschiedliche Anzahl von Zyklen durchzuführen. Jetzt verhält sich die Einschaltkonfiguration des `MemoryDispatcher` anders: Es existiert kein `HadesInterface`. Statt dessen wird ein sogenanntes `Default` `TinyBPMemory`-Objekt mit 16 KByte Größe angelegt und in alle Segmente eingeblendet. Wird nun ein Programm geladen, das auch ohne HADES ausführbar ist, werden alle benötigten ELF-Segmente in den Speicher eingeblendet und das Programm ist direkt lauffähig.

8.1.6 Format der Debug-Ausgaben

Das Debug-Menu des Hauptfensters erlaubt die Einstellung der verschiedenen Debug-Ausgaben. Normalerweise sind alle Punkte deaktiviert und es werden keine Debug-Ausgaben generiert. Allerdings sind innerhalb des MIPS-Simulationsmodells einige Plausibilitätstests insbesondere für Speicherzugriffe und den Ablauf der Automaten enthalten. Sollte ein interner Fehler für inkonsistente Zustände verantwortlich sein, dann werden unabhängig von den Einstellungen der Debug-Ausgaben Fehlermeldungen auf der Standard-Ausgabe angezeigt. Diese sind daran zu erkennen, daß die ersten zwei Zeichen einer Zeile `E:` enthalten.

Normale Debug-Ausgaben benutzen folgendes Format: Jede Methode, die genau eine Zeile Debug-Informationen ausgeben möchte, gibt zuerst den Namen seiner Klasse und der Methode selbst an. Danach folgen die eigentlichen Informationen. Einige Klassen geben statt ihres Namens auch den Objektnamen aus. So können Speicherobjekte und die verschiedenen `InstrFrame`-Objekte besser unterschieden werden. Will eine Methode mehrere Zeilen ausgeben, so leitet eine Zeile mit einem `Begin:`, gefolgt von Klassen- und Methodennamen, die Ausgaben ein. Alle nun folgenden Zeilen werden um zwei Leerzeichen eingerückt, bis dann eine `End`-Zeile die Debug-Ausgaben dieser Methode beendet. Jede dazwischenliegende Debug-Ausgabe der Methode wird dabei nicht extra durch Klassen- und Methoden-Name gekennzeichnet, da durch die Blockstruktur die Zuordnung klar ist. Ein Beispiel für eine Debug-Ausgabe:

```
Begin: Core.cycle(): cycle = 1009
  Begin: AccessMemory.beginCycle
    Begin: MemoryDispatcher.beginCycle: current state=WRITENOWAIT,
           readBufUsed=0, writeBufUsed=0
      MemoryDispatcher.beginCycle: lowerFinished=false
    End: MemoryDispatcher.beginCycle: new state=WRITENOWAIT,
         waitstate=false
    wait=false, dwait=false, iwait=false
  End
  InstrFrame6.finishInstr
  InstrFrame7.operateData
  InstrFrame0.translateDataAdr
  Begin: AccessMemory.translateDataAdr(ffffffd4)
    selectCache: ffffffd4) -> Data cache
  End: result = ffffffd4
  InstrFrame1.decodeInstr
  Begin: AccessMemory.fetchInstr
  End: result = 8fc20004
  InstrFrame2.giveInstrAdr
  Begin: AccessMemory.readInstr(): physical address=4040007c
  End
  Begin: MemoryDispatcher.endCycle: current state=WRITENOWAIT,
         readBufUsed=0, writeBufUsed=0
  End: MemoryDispatcher.endCycle: new state=WRITENOWAIT
End
```

Für die hier erzeugte Ausgabe waren die Debug-Ausgaben für `Core`, `InstrFrame`, `AccessMemory` und den `MemoryDispatcher` aktiviert. In diesem Beispiel ist ein Durchlauf der `cycle()`-Methode des `Core` zu erkennen. Dazu werden zuerst die `beginCycle()`-Methoden aller am Speichersystem beteiligten Objekte aufgerufen. Danach bearbeiten die verschiedenen `InstrFrame`-Objekte ihre jeweils nächste Pipeline-Aktion. Zuletzt beendet der `endCycle()`-Aufruf quer durch das Speichersystem die Ausführung von `cycle()`.

8.2 Hochsprachliche Programmierung

Das MIPS-Simulationsmodell kann erst als wirklich benutzbar bezeichnet werden, wenn dargestellt wurde, wie sich hochsprachliche Programme in MIPS-Maschinensprache übersetzen lassen. Ein Ziel sollte dabei sein, die entstehende Software sowohl auf dem MIPS-Simulationsmodell innerhalb eines HADES-Designs als auch auf dem realen Nachbau dieses HADES-Designs in Hardware ablaufen lassen zu können.

Zur Benutzung eines Prozessors ist im Normalfall immer irgendeine Art von Betriebssystem notwendig. Deshalb muß in Abschnitt 8.2.1 zuerst betrachtet werden, welche Möglichkeiten sich für das MIPS-Simulationsmodell bieten, ein Betriebssystem zu benutzen.

Für die hochsprachliche Programmierung wurde die Sprache C gewählt. Diese Programmiersprache ist standardisiert, gerade im Bereich der eingebetteten Prozessoren sehr weit verbreitet und bietet gute Möglichkeiten zur Erstellung von leistungsfähigen, portablen und kompakten Programmen. Die Benutzung eines C-Übersetzers setzt normalerweise eine Sammlung von ebenfalls standardisierten Basisfunktionalitäten, genannt die `libc`, voraus. Der Abschnitt 8.2.2 geht genauer auf die `libc` ein.

Nach der Klärung dieser Voraussetzungen muß ein konkreter Übersetzer (zusammen mit einem Assembler, der den vom Übersetzer generierten Assembler-Quelltext in die Maschinensprache übersetzt) vorgestellt werden, mit dem sich C-Programme für das MIPS-Simulationsmodell übersetzen lassen. Hierbei wird aus praktischen Gründen ein Crosscompiler eingesetzt, der auf einer anderen als der MIPS-Plattform C-Programme in die MIPS-Maschinensprache übersetzt und das Ergebnis direkt dem MIPS-Simulationsmodell zur Verfügung stellt.

Der gewählte Crossassembler ist der *GNU-Assembler* (GAS), der in den *GNU-Binutils* enthalten ist. Als Crosscompiler leistet der *GNU-C-Compiler* (GCC) gute Dienste. Diese beiden Übersetzer wurden gewählt, da die GNU-Software frei verfüg- und benutzbar und gut dokumentiert ist (für weitere Informationen zum GCC und zu den Binutils siehe [GCC 98]). Der Anhang D beschreiben die Erstellung von Crossassembler und Crosscompiler.

Zuletzt muß noch die Benutzung des Crosscompilers zur Übersetzung von C-Quelltexten in MIPS-Maschinensprache behandelt werden. Abschnitt 8.2.3 beschreibt den Umgang mit dem GNU-C-Compiler, so daß der Leser in die Lage versetzt wird, selbst Programme für das MIPS-Simulationsmodell zu erstellen und die Anwendungsbeispiele im Abschnitt 8.3 zu übersetzen.

8.2.1 Betriebssystem

Als Vermittler zwischen Hard- und Software steht normalerweise ein Betriebssystem zur Verfügung. Dessen Dienste umfassen je nach Aufgabe des Gesamtprodukts aus Hard- und Software verschiedene Bereiche. Im Folgenden wird eine (willkürliche) Aufteilung der Betriebssysteme in verschiedene Komplexitätsstufen vorgenommen, um anhand dieser verschiedenen Stufen den Bezug zum MIPS-Simulationsmodell herzustellen.

- Kleine Mikrocontroller zur Steuerung einfacher Aufgaben benötigen nur eine simple Verwaltung der Hardware-Ressourcen. Diese Verwaltung wird normalerweise in Form eines zum Programm linkbaren Moduls geliefert oder direkt für die konkrete Aufgabenstellung neu implementiert und kann nicht wirklich als Betriebssystem bezeichnet werden.

- Für komplexere Aufgabestellungen (beziehungsweise Prozessoren, die diese bearbeiten) müssen verschiedene Dienste wie Unterbrechungs-, Speicher- und teilweise auch Prozeßverwaltung hinzukommen. Ein Beispiel für ein solches Betriebssystem ist ECOS, das folgend noch eingehender betrachtet wird.

- Zu den wirklich alles umfassenden Betriebssystemen gehören dann die Systeme, die auf große Speichermedien zugreifen und dafür Dateisysteme benutzen. Außerdem bieten diese Betriebssysteme heute auch die Implementation von Kommunikationsprotokollen für den Netzwerkbetrieb. Beispiele für diese Betriebssysteme sind die bekanntesten wie Unix oder Windows.

Der dieser Diplomarbeit zugrunde liegende MIPS-Prozessor bietet ausreichende Leistungsfähigkeit für alle drei Stufen von Betriebssystemen. Aufgrund seiner Komplexität im Bereich der Unterbrechungsbehandlung und der Cache-Verwaltung machen Anwendungen, die ein “Betriebssystem“ der ersten Stufe benutzen, wenig Sinn, da hiermit nur sehr einfache Aufgaben gelöst werden können.

Als guter Kompromiß zwischen Leistungsfähigkeit und Kompaktheit bietet sich ein Betriebssystem der zweiten Stufe an. Diese Gruppe von Betriebssystemen bietet eine ausreichende Abstraktion von der Hardware, so daß damit (zumindest zum Teil) portable Anwendungen implementiert werden können. Aufgrund des nicht zu komplexen Aufbaus dieser Betriebssysteme können sie einfach auf neue Plattformen portiert und an neu hinzukommende Peripherie angepaßt werden.

Es existieren auch Betriebssysteme der dritten Stufe für die MIPS-R3000-Architektur. Als prominenteste Vertreter seien hier einerseits Windows NT (wird für MIPS nicht mehr weiterentwickelt) und andererseits Linux (durch allgemein zugängliche Quelltexte immer weiterentwickelbar) genannt. Die Anpassung dieser Betriebssysteme an einen konkreten Prozessor und dessen Peripherie ist im allgemeinen aufwendig.

Das schon erwähnte Betriebssystem ECOS von Cygnus (siehe [Cygnus 99b]) ist, wie auch Linux, im Quelltext verfügbar. Es wurde bereits an einen der MIPS-R3000-Architektur entsprechenden Prozessor von Toshiba angepaßt. ECOS bietet Speicher- und Prozeßverwaltung in Form von zum auszuführenden Programm linkbare Bibliotheken. Der Umfang der zum Betriebssystem gehörigen Quelltexte ist überschaubar, so daß sich hier ein aussichtsreicher Kandidat anbietet, der mit verhältnismäßig wenig

Aufwand an die Benutzung mit dem MIPS-Simulationsmodell angepaßt werden kann. Die konkrete Bewältigung dieser Portierung übersteigt den Rahmen dieser Diplomarbeit und kann als Aufgabenstellung für eine nachfolgende Arbeit gewählt werden.

Technisch reizvoll, allerdings aufgrund der Simulationsgeschwindigkeit des MIPS-Simulationsmodells (siehe Abschnitt 9.1.6) nur eingeschränkt nutzbar, wäre eine Anpassung von Linux an das MIPS-Simulationsmodell. Zumindest die Basisfunktionalitäten des Kerns dürften sich einfach portieren lassen. Für die Benutzung von Dateisystemen müßten zuerst Simulationsmodelle für entsprechende Peripherie erstellt werden.

8.2.2 libc

Normalerweise wird zur Nutzung eines C-Übersetzers zwingend die `libc` vorausgesetzt. Diese Sammlung von Basisfunktionalitäten ist standardisiert und bietet dem Programmierer über die Sprache C hinaus portablen Umgang mit Ein- und Ausgaben, Zeichenkettenoperationen, Betriebssystemschnittstellen und vielem mehr. Dabei sind die Grenzen zwischen `libc` und Betriebssystem fließend: Je mehr Funktionalität das Betriebssystem bietet, desto weniger muß die `libc` zur Unterstützung des Betriebssystems unternehmen.

Bei der Verwendung von Betriebssystemen, die keine Dateisysteme anbieten, können auch einige Teile der `libc` so schwer zu implementieren sein, daß sie weggelassen werden. In dem Fall steht dem Programmier dann nur ein Teil der `libc` zur Verfügung. Gerade für den eingebettenen Einsatz von Mikrocontrollern werden oftmals die normalen Operationen für Dateien und Ein-/Ausgabe eingespart, da sie nicht unbedingt notwendig sind. Teile der `libc` lassen sich unabhängig vom darunter liegenden Betriebssystem implementieren (als Beispiel seien die Zeichenkettenoperationen genannt). Hier sind die geringsten Probleme bei der Portierung auf eine neue Umgebung zu erwarten.

Die Bereitstellung einer für das MIPS-Simulationsmodell geeigneten `libc` hängt zwingend mit der Bereitstellung eines Betriebssystems zusammen und kann deshalb auch nicht Aufgabe dieser Diplomarbeit sein. Wie im Abschnitt 8.3 dargestellt wird, kann für einfache Aufgaben auch auf den Einsatz der `libc` verzichtet werden.

8.2.3 Benutzung des Crosscompilers

Solange keine `libc` existiert (siehe dazu Abschnitt 8.2.2) und kein Laufzeitsystem mit den Programmen verbunden werden kann, muß eine eigene Initialisierungssequenz dafür sorgen, daß sich Programme überhaupt linken lassen. Als minimale Version reicht die folgende MIPS-Assembler-Sequenz:

```
        .file   1 "initcodeelf.s"

        .text
        .align  2
        .globl  _start
        .ent    _start
_start:
        j       main
```

```
        j       _start
        .end    _start

        .globl  __main
        .ent    __main
__main:
        j       $31
        .end    __main
```

Diese Datei kann mit folgendem Aufruf übersetzt werden:

```
/opt/mips/bin/mips-idt-elf-gcc initcodeelf.s -c
```

Danach sind alle Komponenten für eigene C-Programme vorhanden. Bei der Programmierung müssen allerdings einige Einschränkungen beachtet werden:

- Da keine `libc` vorhanden ist und keine FPU simuliert wird, kann nur Ganzzahlarithmetik verwendet werden.
- Dynamische Speicherverwaltung findet nicht statt (kann aber mit verhältnismäßig wenig Aufwand in C implementiert werden).
- Es gibt bisher keine weiteren unterstützenden Libraries (bis auf ein etwas umfangreicheres `initcodeelf.s`, das mit `print` die Ausgabe von Ganzzahlen ermöglicht).

`initcodeelf.s` und die dazugehörige Header-Datei `initcode.h` werden genauer im Anhang E beschrieben.

Ist ein Programm fertig erstellt und soll es übersetzt werden, wird der Compiler folgendermaßen aufgerufen:

```
/opt/mips/bin/mips-idt-elf-gcc <prg>.c initcodeelf.o \
                               -nostdlib -o <prg>
```

Natürlich können dabei noch weitere Objektfiles angegeben werden. Das entstehende `<prg>` im ELF-Format kann dann dank des `MemoryLoader` (siehe Abschnitt 7.10) direkt vom MIPS-Simulationsmodell eingelesen und ausgeführt werden.

8.3 Anwendungsbeispiele

Drei verschiedene Anwendungsbeispiele zeigen die Möglichkeiten des MIPS-Simulationsmodells. Zuerst werden Primzahlen berechnet und direkt in den Speicher geschrieben. Als zweites Beispiel wird ein Burst-fähiges HADES-Design zur Ansteuerung eines externen Speichers vorgestellt. Im dritten Beispiel wird als komplexe Ein-/Ausgabe-Aufgabe ein das Simulationsmodell eines grafischen LCD-Displays ansteuerndes HADES-Design zusammen mit der steuernden Software vorgestellt.

8.3.1 Primzahlen als einfacher Test

Der erste Test und damit auch das erste Beispiel der Benutzung des MIPS-Simulationsmodells ist eine einfache Primzahlberechnung durch einen Divisionsrestalgorithmus, der im Gegensatz zum Sieb des Eratosthenes wenig Speicher benötigt, aber auch langsamer ist. Da für diesen Test weder die `libc` noch ein Betriebssystem zur Verwaltung der Ausgabe zur Verfügung steht, werden alle berechneten Primzahlen nacheinander in den Speicher geschrieben.

Zum Zeitpunkt der Entstehung dieses Primzahltests wurde gerade der Kern des MIPS-Simulationsmodells fertig gestellt. Die Speicherschnittstelle stand noch nicht zur Verfügung. Deshalb ist die Primzahlenberechnung auch ohne HADES lauffähig. Im Folgenden erst einmal das Primzahlenprogramm im C-Quelltext:

```
#include "initcode.h"

void prim(int max, int startadr)
{
  int prim, teiler, *adr, teilt;

  adr = startadr;
  *adr = 2;

  for (prim = 3; prim < max; prim += 2) {
    teiler = 3;
    teilt = 0;
    while (teiler * teiler <= prim) {
      if (prim % teiler == 0) {
        teilt = 1;
        teiler = prim;
      } else {
        teiler += 2;
      }
    }
    if (teilt == 0) {
      adr++;
      *adr = prim;
    }
  }
}

int main(void)
{
  prim(0x400, 0x400);
}
```

Das Programm berechnet alle Primzahlen bis 1024 und schreibt die Ergebnisse ab Speicherstelle 1024 direkt in den Speicher. Die verwendeten Operationen decken alle Grundrechenarten und Programmstrukturen ab.

Als Beispiel für die Übersetzung durch den GCC wird nun die im Programm enthaltene `while`-Schleife als durch den GCC erzeugter Assembler-Quelltext angegeben.

```
$L6:
        lw      $2,4($fp)       // teiler
        lw      $3,4($fp)       // teiler
        mult    $2,$3           // teiler^2
        mflo    $6
        sw      $6,20($fp)
        lw      $2,0($fp)       // prim
        lw      $6,20($fp)      // teiler^2
        slt     $3,$2,$6        // prim < teiler^2
        beq     $3,$0,$L8       // wenn nicht, dann $L8
        j       $L7             // sonst $L7, ende von while
$L8:
        lw      $2,0($fp)       // prim
        lw      $3,4($fp)       // teiler
        rem     $2,$2,$3        // Divisionsrest von prim / teiler
        bne     $2,$0,$L9       // wenn das nicht null, dann $L9
        li      $2,0x00000001
        sw      $2,12($fp)      // teilt = 1
        lw      $2,0($fp)       // prim
        sw      $2,4($fp)       // teiler = prim
        j       $L10            // weiter mit while
$L9:
        lw      $2,4($fp)       // teiler
        addu    $3,$2,2         // teiler = teiler + 2
        sw      $3,4($fp)
$L10:
        j       $L6             // weiter mit while
$L7:
```

Dieses mit Kommentaren versehene Beispiel zeigt, wie Ausdrücke der Programmiersprache C in Assembler übersetzt werden. Dabei muß beachtet werden, daß der Übersetzer in diesem Fall das Programm ohne Optimierung übersetzt hat. Wird die Optimierung eingeschaltet, so ergibt sich ein besseres Ergebnis, das mehr Register für lokale Variablen verwendet, aber sehr viel schwerer lesbar ist.

Für einen vollständigen Test der Funktionalität des MIPS-Simulationsmodells ist dieses Primzahlenprogramm allerdings nicht ausreichend geeignet, da hier nur ein einziger Datentyp und keine Array-Operationen, keine globalen Variablen und keine Prozedur-Rückgabewerte verwendet werden. Außerdem ist dieser Test noch zu wenig umfangreich, um wirklich fundierte Rückschlüsse auf die Funktionalität des MIPS-Simulationsmodells ziehen zu können.

Wird das MIPS-Simulationsmodell ohne HADES gestartet, kann das übersetzte Primzahlenprogramm direkt geladen und dann durch die Betätigung einer der Cycle-Knöpfe ausgeführt werden. Die Darstellung des `Default TinyBPMemory` zeigt die berechneten Ergebnisse an.

8.3.2 HADES-Design für externen Speicherzugriff

Einen Schritt weiter geht das zweite Anwendungsbeispiel. Hierbei soll kein konkreter C-Quelltext vorgestellt werden, sondern ein HADES-Design, das die simulierten Ein- und Ausgabe-Anschlüsse des `IDTR3051` für die Ansteuerung eines Speichers verwendet. Dabei soll erst einmal der gesamte Adreßraum auf einen Speicherbaustein abgebildet werden. Statt dessen könnten noch Adreßdekoder den Adreßraum in mehrere Teile aufteilen und verschiedenen Speicher- oder Peripherie-Komponenten zuordnen. Die Abbildung 27 zeigt das entstandene HADES-Design.

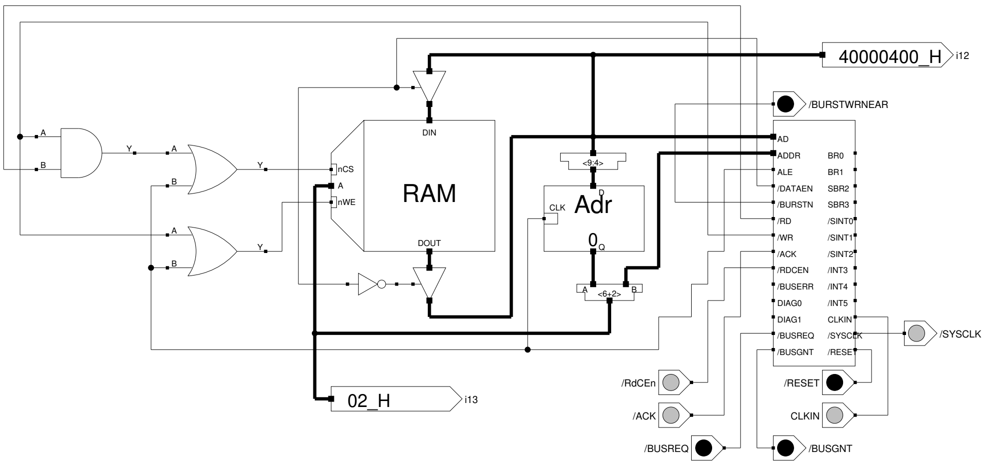

Abbildung 27: HADES-Design für externen Speicherzugriff

Das Hauptproblem, das bei diesem Design gelöst werden muß, ist der gemeinsame Adreß- und Datenbus. Die beiden zeitversetzt gelieferten Angaben müssen voneinander getrennt und dann dem RAM-Baustein zugeführt werden. Dabei ist noch zu beachten, daß nicht alle Adreßbits auf dem einen Bus sichtbar sind. Die beiden Adreßbits zwei und drei sind noch von einem separaten Bus abzugreifen.

In diesem Design besteht der RAM-Baustein aus 256 Worten mit 32 Bit Breite. Deshalb werden die Bits neun bis vier des Adreß- und Datenbusses abgetrennt und einem Latch zugeleitet. Dieses Latch dient dann als Quelle für einen permanenten Adreßbus. Die Bits zwei und drei der Adresse werden direkt von dem dafür vorgesehenen Bus geholt. Damit steht dem RAM eine vollständige aus acht Bits bestehende Adresse zur Verfügung. Da die untersten beiden Bits dieser Adresse nicht über das Latch geleitet wurden, ist dieser Adreßbus auch Burst-fähig.

Das Acknowledge- (bzw. das Read-Counter-Enable-) Signal wird in diesem Design für Versuchzwecke von Hand geschaltet. Hier müßte in einem automatisiert ablaufenden Design die richtige Verzögerung für das entsprechende Taktsignal generiert werden. Für Testzwecke reicht es, beide Schalter auszuschalten, dann wird der Speicher ohne Wartezyklen betrieben.

Ein möglicher Test für dieses Design ist die schon vorgestellte Primzahlenberechnung. Da durch einen Start per HADES das `HadesInterface` den gesamten Adreßraum belegt und danach das Laden des Primzahlenprogramms nur die ELF-Segmente in den Adreßraum einblendet, ist das Programm sofort lauffähig und schreibt alle berechneten Werte direkt in den RAM-Baustein. Außerdem werden noch die Informationen des Stapelspeichers an das obere Ende des RAM geschrieben.

8.3.3 Ein-/Ausgabe-Beispiel: GraphicsLCD

Das letzte und komplexeste Beispiel ist die Ansteuerung einer grafischen LCD-Anzeige. Für HADES existiert das Modell eines NLC-128x64-LCD-Displays, das durch zwei Samsung-KS0108-Display-Controller verwaltet wird. Dieses Display soll nun per speicherbasierter Ausgabe angesprochen werden. Dabei steht erst einmal die Ausgabe von Text im Vordergrund, da das MIPS-Simulationsmodell bisher keine Möglichkeit besitzt, direkt Text auszugeben.

Die Display-Controller besitzen einen aus acht Bits bestehenden Bus und einige weitere Steuerleitungen zur Selektion des Kommandos und des zu verwendenden Controllers. Das hier vorgestellte Design benutzt dabei nur die Fähigkeit, Daten in den Display-Speicher zu schreiben, da für die Textausgabe der Speicher nicht ausgelesen werden muß.

Die Display-Controller bieten keine eingebaute Möglichkeit Zeichen auszugeben, deshalb muß jedes Zeichen pixelweise aufgebaut werden. Der Speicher des Displays bündelt immer acht senkrechte Pixel zu einem Byte, das an die Display-Controller übertragen wird. Werden mehrere Bytes an die Display-Controller übergeben, dann erscheinen diese von links nach rechts auf dem Display. Dabei muß beachtet werden, daß die linken 64 Spalten vom ersten und die rechten 64 Spalten vom zweiten Display-Controller verwaltet werden. Deshalb muß in der Mitte der Controller umgeschaltet werden.

Als Realisierung der Schnittstelle wird ein aus 32 Bits bestehendes und direkt mit dem MIPS-Datenbus verbundenes Latch gewählt, das mit den einzelnen Eingabeleitungen des Display-Controllers verbunden wird. Dieses Latch wird in den MIPS-Adreßraum an Adresse `0xa0000000` eingeblendet – das ist die physikalische Adresse 0, wobei für den Zugriff keine Caches verwendet werden sollen. Wenn nun das MIPS-Simulationsmodell ein Wort an diese Adresse schreibt, werden dessen untere dreizehn Bits direkt an die Display-Controller weitergeleitet. Das Enable-Signal, das die Display-Controller zusätzlich benötigen, wird automatisch generiert, wenn ein neues Wort in das Latch geschrieben wird.

Die Abbildung 28 zeigt das HADES-Design, das in der Lage ist, die Samsung-KS0108-Display-Controller anzusprechen.

Zu diesem Design gehört noch eine Sammlung von C-Prozeduren, die die Steuerung des Displays kapseln sollen. Dazu dient die Datei `graphics.c`. Sie enthält eine vollständige Definition des ASCII-Zeichensatzes (also insgesamt 128 Zeichen), für den die Zeichen als Proportionalfont mit acht Pixel Höhe und drei bis sechs Pixel Breite vorliegen. Außerdem enthält `graphics.c` die Ausgabe-Prozedur, die in der Lage ist, mehrere Buchstaben Spalte für Spalte auszugeben, dabei die Grenzen der beiden Display-Controller zu beachten und automatisch zwischen ihnen umzuschalten und am Zeilenende die nächste Zeile zu benutzen. Die Schnittstelle `graphics.h` bietet die Aufrufe `text()` und `bigtext()`, die eine Zeichenkette normal oder doppelt breit ausgeben, ohne daß der Programmierer bei der Benutzung weiteres bedenken muß. Im Anhang F sind `graphics.h` und `graphics.c` zu finden. Ein Beispielprogramm zur Nutzung des Displays kann ganz einfach aufgebaut sein:

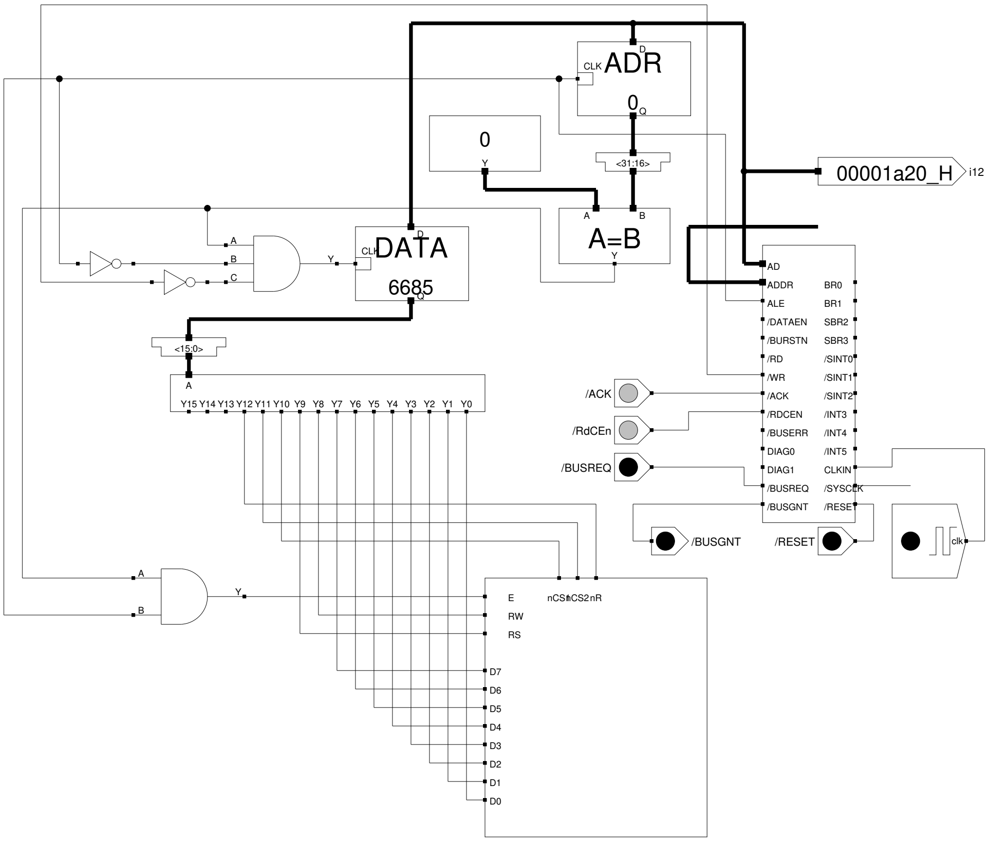

Abbildung 28: HADES-Design mit speicherbasierter grafischer Ausgabe

```
#include "initcode.h"
#include "graphics.h"

int main(void)
{
  enableDisplay(1);

  setTextPos(3, 1);
  bigtext("** Mips **");

  setTextPos(0, 3);
  text("Textausgabe mit der Mips-Simulation einfach gemacht: ");
  text("graphics.h bietet eine Ansteuerung des Samsung KS0108 ");
  text("Displaycontrollers.");
}
```

Dieses Programm aktiviert das Display, setzt die Position und gibt Text aus. Das Ausgabeergebnis ist in Abbildung 29 zu erkennen. Das hier abgebildete Fenster wurde vom Simulationsmodell des Displays erzeugt.

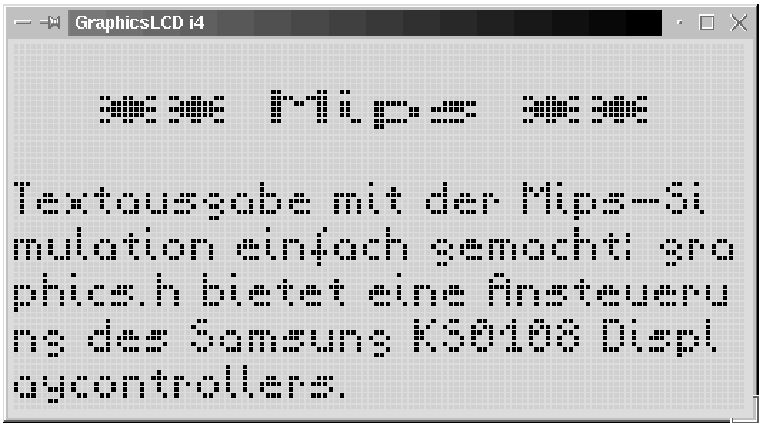

Abbildung 29: Ausgabe des GraphicsLCD

9 Zusammenfassung und Ausblick

9.1 Ergebnisse der Arbeit

Das entstandene MIPS-Simulationsmodell soll nun unter dem Aspekt betrachtet werden, ob die im Abschnitt 1.2 genannten Einzelziele Z_1 bis Z_5 erreicht wurden. Daran schließt sich direkt eine Betrachtung bezüglich der erreichten Geschwindigkeit an.

9.1.1 Detailgetreue Hardware-Abbildung (Z_1)

Das Ziel der detailgetreuen Hardware-Abbildung wird im Rahmen der Vorgaben, die durch die gewünschte Zusammenarbeit mit HADES gegebenen waren, erreicht. Das MIPS-Simulationsmodell simuliert nicht nur ein getreues Verhalten der Schnittstellen, sondern auch interne Vorgänge, soweit sie aus der Dokumentation ersichtlich sind. So werden die Pipeline mit all ihren Feinheiten, die Caches, die Schreibpuffer und das genaue Zeitverhalten aller nach außen sichtbaren Signale nachgebildet.

Dabei muß einschränkend erwähnt werden, daß diese Aussage für einige Spezialitäten nicht zutrifft. Wie schon im Abschnitt 5.4 (Veränderungen gegenüber dem Vorbild) beschrieben wurde, sind einige spezielle Vorgaben der Realität nicht im Simulationsmodell umgesetzt. Diese Punkte betreffen allerdings nur verhältnismäßig unwichtige Vorgaben und können bei Bedarf nachgerüstet werden.

9.1.2 Gemeinsame Simulation von Hard- und Software (Z_2)

Innerhalb des MIPS-Simulationsmodells wurden vielfältige Anstrengungen unternommen, die Simulation möglichst effizient ablaufen zu lassen. Trotz anderer, dagegensprechender Ziele wie der möglichst detailgetreuen Hardware-Abbildung (Z_1) und einer hohen Flexibilität (Z_3) erreicht das Simulationsmodell eine Geschwindigkeit, mit der sich auch komplexe Programme sinnvoll durch das MIPS-Simulationsmodell ausführen lassen. Die erreichte Geschwindigkeit kann noch nicht als Echtzeitgeschwindigkeit bezeichnet werden, reicht aber für viele Anwendungen vollkommen aus und läßt weiteren, gleichzeitig ablaufenden über HADES verbundenen Simulationsmodellen genügend Raum. Genauere Geschwindigkeitsbetrachtungen werden im Abschnitt 9.1.6 geboten.

Da HADES eine in sich geschlossene Welt bildet und für alle simulierten Modelle dieselben Zeitbedingungen gelten, ist die nicht vorhandene Echtzeitfähigkeit des MIPS-Simulationsmodells kein wirkliches Problem: Wenn das MIPS-Simulationsmodell nicht so schnell ausgeführt werden kann wie es der Takt eigentlich vorgibt, werden alle anderen an der Simulation beteiligten Modelle entsprechend gebremst, so daß die innerhalb der Simulation verstrichene Zeit für alle Modelle gleich ist. Deshalb ergibt sich als einzige Einschränkung der erhöhte Zeitbedarf für die Simulation von Aufgaben, die eigentlich eine Echtzeit-Simulation benötigen. Die Ergebnisse der Simulation werden dadurch nicht verfälscht.

Abschließend kann gesagt werden, daß das MIPS-Simulationsmodell eine Geschwindigkeit erreicht, die ausreichend ist, um Hard- und Software sehr gut gemeinsam betrachten zu können. Somit ist das MIPS-Simulationsmodell ausgezeichnet nutzbar und für viele Zwecke als Systemsimulation einsetzbar.

9.1.3 Flexibilität (Z_3)

Die anfangs geforderte Flexibilität in der Speicherkonfiguration wird umfassend zur Verfügung gestellt. Der `MemoryDispatcher` kann verschiedene Speicherobjekte in seinen Adreßraum einblenden und diese einheitlich verwenden. Kommt das `HadesInterface` zum Einsatz, so benutzt die Simulation die simulierte Ein-/Ausgabe-Schnittstelle des IDT R3051 und bedient sich eines externen Speichermodells. Werden statt dessen interne Speicher verwendet, so gibt es komfortable Möglichkeiten zur Verfolgung des Programmablaufs (Unterbrechungspunkte). Außerdem steigt die Simulationsgeschwindigkeit.

Für normale Simulationsaufgaben hat sich eine Konfiguration bewährt, bei der Programm und Daten in innerhalb des Simulationsmodells zur Verfügung gestellten Speicherobjekten abgelegt sind. Für die Ansteuerung von Peripherie wird dann das `HadesInterface` benutzt, damit der Kontakt zu anderen Simulationsmodellen aufgenommen werden kann. Der `MemoryLoader` unterstützt dieses Vorgehen optimal, da er automatisch für die Segmente von ausführbaren ELF-Dateien Speicherobjekte innerhalb des MIPS-Simulationsmodells anlegt.

Vorstellbar wäre noch eine weitergehende Flexibilität, so daß nicht nur ein spezieller Prozessor, sondern verschiedene Prozessoren einer Baureihe simuliert werden könnten. Das hier vorgestellte MIPS-Simulationsmodell kann, wie vorgegeben, genau einen Prozessor, den IDT R3051, simulieren. Die von anderen verwandten Bausteinen angebotenen Zusatzausstattungen wie eine MMU oder eine FPU werden nicht nachgebildet, können aber über teilweise schon vorhandene Schnittstellen nachgerüstet werden. Entsprechend vorstellbar wäre eine vom Benutzer einstellbare Cache-Konfiguration. Bisher läßt sich die Größe der Caches einzig durch (allerdings sehr einfache) Modifikationen am Quelltext von `AccessMemory` verändern.

9.1.4 Hochsprachliche Programmierung (Z_4)

Durch seinen `MemoryLoader` ist das MIPS-Simulationsmodell in der Lage, ausführbare ELF-Dateien einzulesen und direkt auszuführen. Deshalb können alle Hochsprachenübersetzer verwendet werden, die MIPS-Maschinensprache-Dateien im ELF-Format erzeugen können. Ein besonderer Fokus wird in diesem Text auf den GNU-C-Compiler geworfen, da dieser allgemein anerkannt gut optimierte Maschinenprogramme für viele verschiedene Architekturen liefert und kostenlos verfügbar ist. Gerade dessen Fähigkeiten als Crosscompiler sind in diesem Zusammenhang sehr wichtig.

Außer C wären weitere Programmiersprachen wie C++, Fortran, Pascal u.s.w. denkbar. Für alle diese Sprachen existieren Anpassungen zur Benutzung des GCC-Compilers, so daß nicht weiter darauf eingegangen werden muß. Und auch ein Java-Frontend für den GCC befindet sich in der Entwicklung, so daß Java-Programme mit allen dort vorhandenen Vorteilen (insbesondere Debugger) auf anderen Maschinen entwickelt werden können, um dann durch den GCC in MIPS-Maschinencode übersetzt und durch das MIPS-Simulationsmodell ausgeführt werden zu können.

Wie schon in Abschnitt 8.2 beschrieben, existieren keine `libc` und auch kein Betriebssystem. Das Erstellen einer umfangreichen Anwendung für das MIPS-Simulationsmodell ist deshalb eher kompliziert. Insbesondere vor der Portierung eines Programms von einer anderen Architektur sollten Schritte zur Portierung der `libc` und zumindest Teilen eines Betriebssystems vorgenommen werden.

Durch die vom MIPS-Simulationsmodell zur Verfügung stehenden Mechanismen wird der Programmierer bei der Fehlersuche durch Unterbrechungspunkte innerhalb des Maschinenspracheprogramms und durch die umfassende Inspektionsschnittstelle unterstützt.

9.1.5 Inspektionsschnittstelle (Z_5)

Alle am MIPS-Simulationsmodell beteiligten Komponenten können für den Benutzer sinnvoll angezeigt werden. So wird der Benutzer in die Lage versetzt, den aktuellen Stand der Pipeline direkt abzulesen, den Inhalt der Caches zu inspizieren oder sogar den genauen Zustand des Automaten der Speicherschnittstelle herauszufinden. Alle diese Informationen sollen einerseits helfen, die internen Abläufe des simulierten Prozessors zu verstehen und andererseits die Fehlersuche unterstützen.

Gerade die Darstellung der Pipeline ist etwas besonderes: Hier wird dem Benutzer genau Auskunft darüber gegeben, welche Befehle sich gerade in welchem Bearbeitungszustand befinden. Eine vergleichbare Pipeline-Darstellung innerhalb von Software zur Simulation von Hardware ist dem Autor nicht bekannt. Alle entsprechenden Simulationmodelle müssen ohne eine Darstellung der Pipeline-Aktivitäten auskommen.

9.1.6 Geschwindigkeitsmessungen

Die Geschwindigkeit des MIPS-Simulationsmodells hängt von verschiedenen Faktoren ab:

- Je seltener das `HadesInterface` benutzt wird, desto schneller wird die Simulation ausgeführt. Daraus folgt, daß die Benutzung von internen Speichern für Programm und Daten wünschenswert ist.
- Die zusammen mit dem MIPS-Simulationsmodell verwendeten anderen Simulationsmodelle benötigen auch Rechenzeit – je mehr, desto weniger bleibt für das MIPS-Simulationsmodell übrig.
- Je nach verwendeter JDK-Version, *Just-In-Time-Compiler* (JITC) und Betriebssystem unterscheidet sich die Geschwindigkeit erheblich.

Die besten Geschwindigkeitsergebnisse werden folglich dann gemessen, wenn das MIPS-Simulationsmodell ohne HADES ausgeführt wird oder wenn nur wenige, einfache andere Simulationsmodelle neben dem MIPS-Simulationsmodell ausgeführt werden. Als beste Java-Version hat sich Java 2.0 gemeinsam mit dem Hotspot-JITC [Javasoft 99] unter Windows 95 bewährt. Die für die Entwicklung verwendete Plattform (Linux, JDK 1.1.7, kein JITC) hat sich dagegen als wesentlich langsamer herausgestellt: Das oben beschriebene Beispiel der Primzahlberechnung (siehe Abschnitt 8.3.1) wird hier um den Faktor 40 langsamer ausgeführt.

Die nun folgenden Messungen wurden durch einem Computer ausgeführt, der mit einem Intel Celeron Prozessor mit 466 MHz Taktfrequenz, 128 MByte RAM und der oben beschriebenen optimalen Software-Konfiguration ausgestattet war. Bei einem nur innerhalb des MIPS-Simulationsmodells ablaufendem Programm (wie zum Beispiel die Primzahlenberechnung) ließen sich ungefähr 350.000 MIPS-Maschinenbefehle pro Sekunde ausführen. Das ist also die maximale Leistungsfähigkeit.

Bei starker Nutzung des `HadesInterface` für die Ansteuerung des grafischen LCD (siehe Abschnitt 8.3.3), aber internem Programm, werden ca. 60.000 Befehle pro Sekunde ausgeführt. Leistungsmessungen für Programme, die außerhalb des MIPS-Simulationsmodell in einem RAM gespeichert sind, wurden nicht vorgenommen, da sich ein solches Design nicht für geschwindigkeitsorientierte Simulation eignet.

Der IDT R3051 wird vom Hersteller mit einer maximalen Taktrate von 40 MHz vertrieben. Daraus ergibt sich eine Befehlsrate von maximal 20 Millionen Befehlen pro Sekunde. Das MIPS-Simulationsmodell ist also unter Einsatz der oben beschriebenen Plattform und optimalen Bedingungen ungefähr 60 mal langsamer als der zu simulierende Prozessor. Durch den Einsatz eines optimal ausgestatteten PC dürfte zum jetzigen Zeitpunkt (April 2000) ungefähr ein Verhältnis von 1:30 erreicht werden.

Als weiterer Geschwindigkeitstest wurde `AccessMemory` durch die Klasse `FastAccessMemory` ersetzt, die keine Speicherhierarchie anbietet, sondern direkt einen einfachen Speicher implementiert und keine weiteren Konfigurationsmöglichkeiten zuläßt (wird `ShowMips` direkt mit dem Parameter `fast` aufgerufen, dann startet das MIPS-Simulationsmodell in dieser Konfiguration). Dadurch wurde die Primzahlenberechnung um den Faktor 1,9 beschleunigt.

9.1.7 Erreichen des Globalziels

Als Ergebnis dieser Diplomarbeit steht ein leistungsfähiges MIPS-Simulationsmodell zur Verfügung, das vielen Aufgaben der Systemsimulation gewachsen ist. Sofern keine Echtzeitprobleme oder Schaltungen mit vielen verschiedenen Komponenten nachgebildet werden sollen, kann das MIPS-Simulationsmodell als vollwertiger Ersatz der realen Hardware eingesetzt werden. Allerdings müssen dabei einige Einschränkungen der Modellbildung berücksichtigt werden.

Die Programmierung des Simulationsmodells kann durch den GNU-C-Crosscompiler vorgenommen werden und bietet dadurch die Möglichkeit, Software für das MIPS-Simulationsmodell einfach zu erstellen.

Dem Benutzer des MIPS-Simulationsmodells wird eine umfangreiche Inspektionsschnittstelle zusammen mit komfortabler Bedienung zur Verfügung gestellt.

9.2 Möglichkeiten der Weiterentwicklung

9.2.1 Verfeinerung der Modellbildung

Die Modellbildung (siehe Kapitel 5) zählt einige Einschränkungen gegenüber der realen Hardware auf. Durch zusätzlichen Programmieraufwand dürften sich die Einschränkungen beheben lassen, ohne daß sich dadurch andere Nachteile ergeben.

Eine Einschränkung wird allerdings immer bestehen bleiben: Das MIPS-Simulationsmodell ist für eine Zusammenarbeit mit HADES konzipiert und wird deshalb immer dessen Grundmodell für Signale entsprechen. Deshalb läßt sich nur das digitale Verhalten des IDT-Prozessors ohne die Berücksichtigung weiterer äußerer Faktoren, wie zum Beispiel der Umgebungstemperatur, simulieren.

9.2.2 Validation der Schnittstellenkonformität

Das im Rahmen dieser Diplomarbeit entstandene MIPS-Simulationsmodell soll auf den Ebenen der Maschinensprache-Schnittstelle und der Ein-/Ausgabe-Schnittstelle nach IEEE 1164 dem Vorbild IDT R3051 entsprechen. Diese Kompatibilität muß durch Validation überprüft werden. Dabei ist die Validation nicht Teil dieser Arbeit – hier werden nur die Wege aufgezeigt, wie eine Validation durchgeführt werden kann.

Ein erster Ansatz wäre der theoretische Beweis der Korrektheit der verwendeten Algorithmen. Die Beweisbarkeit von Algorithmen und Programmen ist allerdings (zumindest für so komplexe Aufgaben wie hier) praktisch unmöglich. Entsprechende Probleme bereiten Korrektheitsbeweise für Prozessoren (Ansätze hierzu siehe [Hunt 94] oder [Bose 99]), deshalb ist dieser Weg nicht benutzbar.

Eine andere Möglichkeit ergibt sich durch massives zufälliges Testen mit direktem Vergleich zum Original. Ein möglicher Versuchsaufbau bestände aus einem 4 GByte großem Speicher, der mit zufälligen, aber bekannten Werten gefüllt wird. Ein Original-IDT-R3051-Prozessor wird dann mit dem Speicher verbunden und führt eine fest definierte, große Anzahl von Befehlen aus. Die danach im Speicher erscheinenden Werte werden protokolliert. Sobald dieser erste Teil des Tests durchgeführt wurde, wird dann das MIPS-Simulationsmodell mit einem 4 GByte großem Speicher, der dieselben Werte wie im ersten Testteil enthält, versehen und wieder wird die gleiche, fest definierte Befehlsmenge ausgeführt. Wenn nach dem Ende des zweiten Teils beide Ergebnisse übereinstimmen, ist die Wahrscheinlichkeit für die Kompatibilität des MIPS-Simulationsmodells hoch – aber nichts ist bewiesen.

Leider läßt sich dieser Test nur mit verhältnismäßig großem Aufwand durchführen: Ein IDT R3051 Prozessor muß vorhanden sein und mit dem Speicher verbunden werden, es muß eine Möglichkeit vorhanden sein, den Speicherinhalt von einem Hintergrundspeicher einzulesen und dort wieder zu speichern und das MIPS-Simulationsmodell muß zusammen mit 4 GByte simuliertem Speicher ausgeführt werden können.

Der gerade beschriebene Test läßt sich noch vereinfachen, indem nicht der volle Adreßraum durch Speicher abgedeckt wird, sondern nur ein begrenzter Speicher von beispielsweise 1 MByte Größe benutzt wird und bei allen Speicherzugriffen nur die unteren 20 Adreßbits beachtet werden. So wäre der Test auch mit heutigen Mitteln praktisch im universitären Rahmen durchführbar.

Wenn keine Möglichkeiten zur Verfügung stehen, durch einen Hardware-Aufbau einen direkten Test des Originalprozessors durchzuführen, kann statt dessen auch ein anderes Simulationsmodell eingesetzt werden. Dazu muß dieses natürlich bereits entsprechend gegenüber dem Originalprozessor überprüft worden sein. So ließe sich die bereits in Abschnitt 4.2 vorgestellte Simulations-Software SPIM einsetzen, um mit dem im Rahmen dieser Arbeit entstandenen MIPS-Simulationsmodell verglichen zu werden. Allerdings simuliert SPIM nicht den IDT R3051, sondern einen MIPS R3000-Prozessor zusammen mit FPU und MMU, so daß das bestehende Simulationsmodell nicht verglichen werden kann.

Die für diese Arbeit genutzte Validation beschränkte sich auf die Benutzung des GCC zur Übersetzung von C-Algorithmen in ausführbare MIPS-Maschinenprogramme, die dann, auf dem MIPS-Simulationsmodell ausgeführt, die erwarteten Ergebnisse lieferten.

9.2.3 Vereinfachung der Programmierung

Die Funktionalität zur Verwendung von ausführbaren ELF-Dateien bietet dem Benutzer des MIPS-Simulationsmodells grundsätzlich alle Möglichkeiten für hochsprachliche Programmierung.

Allerdings muß noch für eine Vereinfachung der Programmierung gesorgt werden, damit wirklich sinnvoll mit dem Simulationsmodell umgegangen werden kann. Für den C-Compiler fehlt noch eine Implementation von (den für diese Zwecke sinnvollen) Teilen der `libc`. Dazu wäre dann noch eine ausgefeiltere Laufzeitumgebung oder sogar ein Betriebssystem notwendig. Alles das sprengt den Rahmen dieser Diplomarbeit und müßte daher in zukünftigen Arbeiten weiter verfolgt werden.

Auch eine weitergehende Unterstützung des Programmierers durch den Einsatz eines Debuggers wäre sinnvoll. Für die Benutzung des *GNU-Debugger* müßten umfangreiche Änderungen am Simulationsmodell vorgenommen werden, so daß sich dieser Weg nur eingeschränkt empfiehlt.

Vom GCC ist seit kurzer Zeit ein "Ableger" erhältlich, der Java-Programme direkt in Maschinensprache übersetzt. Sobald dieser fehlerfrei und zuverlässig arbeitet, wird es hiermit einfach möglich sein, auch Java-Programme mit dem MIPS-Simulationsmodell zu nutzen.

9.2.4 Steigerung der Geschwindigkeit

Innerhalb des MIPS-Simulationsmodells wurden vielfältige Mechanismen zur Steigerung der Ausführungsgeschwindigkeit genutzt. Bestimmt ließe sich die Leistung noch ein wenig verbessern, wenn komplexe Mechanismen, die nicht direkt zur Kernfunktionalität des Prozessors gehören (als Beispiel sei hier der `MemoryDispatcher` genannt), weggelassen würden. Dadurch würde das MIPS-Simulationsmodell allerdings einen Großteil seiner Flexibilität und damit seiner Benutzungsmöglichkeiten verlieren.

Pro simuliertem MIPS-Maschinenbefehl werden, wie aus den Geschwindigkeitsbetrachtungen ersichtlich, ungefähr 1000 Intel-Maschinenbefehle ausgeführt. Pro Pipeline-Aktion werden also etwas mehr als 100 Intel-Maschinenbefehle benötigt.

Wenn man bedenkt, daß es sich bei der Implementationssprache Java um eine maschinenferne Sprache mit zum Teil sehr abstrakten Mechanismen handelt, muß eingesehen werden, daß keine wesentlichen Geschwindigkeitssteigerungen durch Optimierung möglich sein werden. Umgekehrt ausgedrückt: Die Implementation ist so gut gelungen, daß weitere Optimierungen keinen großen Erfolg zeigen werden. Verschiedene Profiling-Tests haben diese Aussage bestätigt.

Literatur

[Bryant 81] Randall E. Bryant, *MOSSIM: A Switch-Level Simulator for MOS-LSI*, Proceedings 18. Design Automation Conference, 1981

[Bose 99] Pradip Bose, Jacob A. Abraham, *Research Report: Performance and Functional Verification of Microprocessors*, IBM Research Division, T.J. Watson Research Center, New York, 1999

[Cygnus 98] *Cygnus Next Generation Playstation Simulation Environment*, Cygnus Solutions, Sunnyvale, California, 1998
`www.redhat.com/support/wpapers/cygnus_risk/playstation.html`

[Cygnus 99a] *Cygnus GNUPro PLUS* Cygnus Solutions, Sunnyvale, California, 1999
`www.redhat.com/services/gnupro/gnupro_plus.html`

[Cygnus 99b] `Embedded Configurable Operating System`, Cygnus Solutions, Sunnyvale, California, 1999
`www.redhat.com/services/ecos/index.html`

[GCC 98] GCC Team, *GNU Compiler Collection Homepage*, 1998
`gcc.gnu.org/`

[Gillespie 96] Gillespie, G., Lazzaro, J., *DigLOG, Caltech VLSI CAD Tools*, California Institute of Technology, 1996
`ftp://ftp.pcmp.caltech.edu/pub/chipmunk`

[Hansen 86] Craig Hansen, Dan Freitas, Ed Hudson, John Moussouris, Steve Przybylski, Tom Riordan and Chris Rowen, *A RISC microprocessor with integral MMU and cache interface*, Proc. IEEE International Conference on Computer Design: VLSI in Computers, New York, 1986

[Hendrich 97] N. Hendrich, *HADES Tutorial 0.43*, Universität Hamburg, FB Informatik, 1997

[Hendrich 98] N. Hendrich, *HADES homepage*, Universität Hamburg, FB Informatik, 1998
`tech-www.informatik.uni-hamburg.de/applets/hades`

[Hendrich 2000a] N. Hendrich, M. Grove, *Praktikum Technische Informatik T3*, Universität Hamburg, FB Informatik, 2000
`tech-www.informatik.uni-hamburg.de/Dokumentation/`
`epra/t3-1.ps.gz`

[Hendrich 2000b] N. Hendrich, *A Java-based framework for simulation and teaching*, Proc. EWME 2000, Aix-en-Provence, to appear, 2000

[Hennessy 96] John L. Hennessy, David A. Patterson, *Computer architecture: A quantitative approach, 2nd Edition*, Morgan Kaufmann Publishers, San Francisco, California, 1996

[Hugunin 97] J. Hugunin, *Python and Java: the Best of both worlds*, Proc. 6th International Python Conference, San Jose, California, 1997

[Hunt 94] Warren a. Hunt Junior, *FM 8501: A verified Microprocessor*, Springer Verlag, Heidelberg, 1994

[i960 97] *i960 JX Microprocessor Developer's Manual*, Intel Corporation, Santa Clara, California, 1997
`www.intel.com/design/i960/manuals/272483.htm`

[IDT 94] *The IDTR3051, R3052 RISController Hardware User's Reference Manual*, Integrated Device Technology, Santa Clara, California, 1994
`www.idt.com/docs/79R3051_MA_65105.pdf`

[IDT 95] *IDT79R3041 Hardware User's Reference Manual*, Integrated Device Technology, Santa Clara, California, 1995
`www.idt.com/docs/79R3041_MA_41469.pdf`

[IDT 96] *IDT MIPS Microprocessor Family Software Reference Manual*, Integrated Device Technology, Santa Clara, California, 1996
`www.idt.com/docs/79R3051_MA_24473.pdf`

[IEEE 93a] IEEE Computer Society, *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1993, 1993

[IEEE 93b] IEEE Computer Society, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (std_logic_1164)*, IEEE Std. 1164-1993, 1993

[Javasoft 99] Sun Microsystems, *The Java Hotspot performance engine*, Sun Microelectronics, Mountain View, California, 1999
`www.javasoft.com/products/hotspot`

[Jessen 87] E. Jessen, R. Valk, *Rechensysteme - Grundlagen der Modellbildung*, Springer Verlag, Berlin, 1987

[Kane 92] Gerry Kane, Joe Heinrich, *Mips Risc Architecture*, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[Larus 97] James R. Larus, *SPIM S20: A MIPS R2000 simulator*, Computer Science Department, University of Wisconsin-Madison, 1997
`www.cs.wisc.edu/~larus/spim.html`

[Meyer 2000] M. Meyer, *Scripting für das HADES-Simulations-Framework*, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, to appear, 2000
`tech-www.informatik.uni-hamburg.de/applets/hades/`

[microSPARC 97] *microSPARC - IIep User's Manual*, Sun Microelectronics, Mountain View, California, 1997
`www.sun.com/microelectronics/manuals/`
`microSPARC-IIep/802-7100-01.pdf`

[MIPS 99] *MIPS product line card*, MIPS Technologies Inc., Mountain View, California, 1999
`www.mips.com/Documentation/LineCard.pdf`

[Page 91] Bernd Page, *Diskrete Simulation*, Springer Verlag, Berlin, 1991

[Palm 99] Palm Inc., *Palm OS Emulator*, Santa Clara, California, 1999
`www.palm.com/devzone/pose/pose.html`
`www.hewgill.com/pilot/copilot/index-old.html`

[Patterson 98] David A. Patterson, John L. Hennessy, *Computer organization and design: The hardware/software interface, 2nd Edition*, Morgan Kaufmann Publishers, San Francisco, California, 1998

[Plauger 92] P. J. Plauger, *The standard C library*, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[Rink 99] C. Rink, *Ein Simulationsmodell für den PIC16C84-Microcontroller*, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1999
`tech-www.informatik.uni-hamburg.de/applets/hades/`

[Riordan 88] Tom Riordan, G.P. Grewal, Simon Hsu, John Kinsel, Jeff Libby, Roger March, Marvin Mills, Paul Ries, Randy Scofield, *The MIPS M2000 System*, Proc. IEEE International Conference on Computer Design: VLSI in Computers, New York, 1988

[Rusling 99] David A. Rusling, *The Linux Kernel*, Linux Documentation Project, 1999
`www.linux.ncsu.edu/LDP/LDP/tlk/kernel/processes.html`

[Simonovich 95] D. Simonovich, *Entwurf und Bewertung konservativer Verfahren in der verteilten Simulation*, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1995

[SPARC 92] *The SPARC architecture manual, 7th Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[Sweetman 99] Dominic Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, San Francisco, California, 1999

[Synopsys 99] *VHDL Simulation Reference Manual*, Synopsys Online Documentation v1999.10, Synopsys Inc., Mountain View, California, 1999

[Tanenbaum 90] Andrew S. Tanenbaum, *Structured Computer Organization, 3rd Edition*, Prentice Hall, London, 1990

[Verilog 99] Cadence Design Systems Inc., *Verilog/VHDL Simulation*, Sunnyvale, California, 1999
`www.cadence.com/datasheets/affirma_nc_verilog_sim.html`

[Wolfinger 99] Bernd E. Wolfinger, *Characterization of Mixed Traffic Load in Service-integrated Networks*, System Science Journal, Vol. 25, No. 2, 1999, S. 65-86

A Übersicht der MIPS-Maschinensprache

Dieser Anhang bietet eine Übersicht über alle MIPS-Maschinenbefehle in tabellarischer Form.

Befehl	Parameter	Beschreibung
LB	rt, offset(base)	Lese Byte mit Vorzeichenbeachtung
LBU	rt, offset(base)	Lese Byte ohne Vorzeichenbeachtung
LH	rt, offset(base)	Lese Halbwort mit Vorzeichenbeachtung
LHU	rt, offset(base)	Lese Halbwort ohne Vorzeichenbeachtung
LW	rt, offset(base)	Lese Wort
LWL	rt, offset(base)	Lese linkes Teilwort
LWR	rt, offset(base)	Lese rechtes Teilwort
SB	rt, offset(base)	Schreibe Byte
SH	rt, offset(base)	Schreibe Halbwort
SW	rt, offset(base)	Schreibe Wort
SWL	rt, offset(base)	Schreibe linkes Teilwort
SWR	rt, offset(base)	Schreibe rechtes Teilwort

Tabelle 14: Lese- und Schreibbefehle

Befehl	Parameter	Beschreibung
ADD	rd, rs, rt	Addiere zwei Register
ADDU	rd, rs, rt	Addiere zwei Register ohne Überlaufbeachtung
SUB	rd, rs, rt	Subtrahiere zwei Register
SUBU	rd, rs, rt	Subtrahiere zwei Reg. ohne Überlaufbeachtung
SLT	rd, rs, rt	Vergleiche zwei Reg., wenn Bedingung, dann 1
SLTU	rd, rs, rt	Vergleich ohne Vorzeichen
AND	rd, rs, rt	Logische Und-Verknüpfung zweier Register
OR	rd, rs, rt	Logisch Oder zwischen zwei Registern
XOR	rd, rs, rt	Exklusive Oder-Verknüpfung zweiter Register
NOR	rd, rs, rt	Logisch Nicht-Oder zwischen zwei Registern

Tabelle 15: Normale arithmetische Befehle

Befehl	Parameter	Beschreibung
SLL	rd, rt, shamt	Schiebe Register links um konstante Zahl Bits
SRL	rd, rt, shamt	Schiebe rechts um konstante Zahl Bits
SRA	rd, rt, shamt	Schiebe vorzeichenerhaltend rechts
SLLV	rd, rt, rs	Schiebe Reg. links um variable Zahl Bits
SRLV	rd, rt, rs	Schiebe Reg. rechts um variable Zahl Bits
SRAV	rd, rt, rs	Schiebe Reg. vorzeichenerhaltend rechts

Tabelle 16: Schiebebefehle

Befehl	Parameter	Beschreibung
ADDI	rt, rs, immediate	Addiere Register und Konstante
ADDIU	rt, rs, immediate	Addiere Reg. und Konst. ohne Überlaufbeachtung
SLTI	rt, rs, immediate	Vergleich von Register und Konstante
SLTIU	rt, rs, immediate	Vergleich von Reg. und Konst. ohne Vorzeichen
ANDI	rt, rs, immediate	Logisch Und zwischen Register und Konstante
ORI	rt, rs, immediate	Logisch Oder zwischen Register und Konstante
XORI	rt, rs, immediate	Exklusiv Oder zwischen Register und Konstante
LUI	rt, immediate	Fülle die oberen 16 Bit mit Konstante

Tabelle 17: Arithmetische Befehle mit Konstanten

Befehl	Parameter	Beschreibung
MULT	rs, rt	Multipliziere zwei Register
MULTU	rs, rt	Multipliziere ohne Vorzeichenbeachtung
DIV	rs, rt	Dividiere zwei Register
DIVU	rs, rt	Dividiere ohne Vorzeichenbeachtung
MFHI	rd	Kopiere Spezialregister HI in anderes Reg.
MFLO	rd	Kopiere Spezialregister LO in anderes Reg.
MTHI	rd	Kopiere Register in Spezialregister HI
MTLO	rd	Kopiere Register in Spezialregister LO

Tabelle 18: Multiplikations- und Divisionsbefehle

Befehl	Parameter	Beschreibung
J	target	Springe direkt zum Ziel
JAL	target	Springe und vermerke vorher Adresse in $31
JR	rs	Springe an die im Register enthaltene Adresse
JALR	rs, rd	Springe indirekt, merke vorher Adresse in Reg.

Tabelle 19: Sprungbefehle

Befehl	Parameter	Beschreibung
BEQ	rs, rt, offset	Sprung, wenn beide Register gleich
BNE	rs, rt, offset	Sprung, wenn beide Register ungleich
BLEZ	rs, offset	Sprung, wenn Register kleiner gleich 0
BGTZ	rs, offset	Sprung, wenn Register größer 0
BLTZ	rs, offset	Sprung, wenn Register kleiner 0
BGEZ	rs, offset	Sprung, wenn Register größer gleich 0
BLTZAL	rs, offset	Sprung bei kleiner 0, merke Adresse in $31
BGEZAL	rs, offset	Sprung bei größer gleich 0, merke Adresse

Tabelle 20: Verzweigungsbefehle

Befehl	Parameter	Beschreibung
SYSCALL		Systemaufruf-Exception wird generiert
BREAK		Unterbrechungs-Exception wird generiert

Tabelle 21: Spezialbefehle

Befehl	Parameter	Beschreibung
LWC0	rt, offset(base)	Lese Wort für Koprozessorregister
SWC0	rt, offset(base)	Schreibe Wort aus Koprozessorregister
MTC0	rt, rd	Kopiere Register in Koprozessorregister
MFC0	rt, rd	Kopiere Koprozessorregister in Register
BC0T	offset	Springe, wenn BrCond-Eingang gesetzt ist
BC0F	offset	Springe, wenn BrCond-Eingang gelöscht ist
RFE		Status-Register wird restauriert

Tabelle 22: Koprozessorbefehle des IDT R3051

B Beschreibung wichtiger Register von Koprozessor 0

Zwei Register des Koprozessor 0, das Status- und das Cause-Register, enthalten viele den Systemzustand reflektierende Bits, die genauer beschrieben werden müssen. Die einzelnen im Status-Register enthaltenen Bits dienen vielen verschiedenen Zwecken. Deshalb werden sie nun einzeln mit kurzer Funktionsbeschreibung aufgezählt.

Bits 31-28/CUx Die Bits CU0 bis CU3 (Coprocessor Usability) geben an, ob der korrespondierende Koprozessor benutzbar ist. Beim IDT R3051 wird immer nur CU0 gesetzt sein, da nur der Koprozessor 0 vorhanden ist.

Bits 27-26: Diese Bits sind ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bit 25/RE: Mit dem RE-Bit (Reverse Endianess) kann die Byte-Order beeinflußt werden. Wird das RE-Bit gesetzt, so werden alle folgenden Speicherzugriffe im User-Modus mit inverser Byte-Order ausgeführt. Dieses Bit hat keinen Einfluß auf den Kernel-Mode.

Bits 24-23: Diese Bits sind ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bit 22/BEV: Das BEV-Bit (Bootstrap Exception Vector) sagt, welche Exception-Vektoren benutzt werden sollen (siehe Abschnitt 3.5.3).

Bit 21/TS: Wenn das TS-Bit (TLB Shutdown) gesetzt ist, dann ist der TLB nicht funktionsbereit (entweder nicht vorhanden oder aufgrund eines Fehlers angeschaltet).

Bit 20/PE: Dieses PE-Bit (Parity Error) sollte während der Initialisierung mit einer 1 beschrieben werden, alle darauf folgenden Lesevorgänge liefern dann eine 0 zurück.

Bit 19/CM: Das CM-Bit (Cache Miss) zeigt bei isolierten Caches einen Cache Miss an und kann zur Feststellung der Cachegröße verwendet werden.

Bit 18/PZ: Dieses PZ-Bit (Parity Zero) sollte immer mit einer 0 beschrieben werden.

Bit 17/SwC: Wenn für Initialisierungszwecke auf den I-Cache zugegriffen werden muß, bewirkt das Setzen des SwC-Bits (Swap Caches) die Vertauschung beider Caches.

Bit 16/IsC: Mit dem IsC-Bit (Isolate Caches) wird die Verbindung zwischen Caches und Speicherschnittstelle getrennt.

Bits 15-8/IM: Diese Bits enthalten die Interrupt-Maske. Die Bits 15-10 sind für Hardware-Interrupts zuständig, während die Bits 9-8 Software-Interrupts erlauben.

Bits 7-6: Diese Bits sind ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bit 5/KUo: Bei der vorletzten Exception hat dieses KUo-Bit (Kernel/User old) angezeigt, ob sich der Prozessor im Kernel- oder im User-Modus befindet.

Bit 4/IEo: Das IEo-Bit (Interrupt Enable old) sagt aus, ob bei der vorletzten Exception Interrupts erlaubt waren.

Bit 3/KUp: Das KUp-Bit (Kernel/User previous) sagt, ob der Prozessor bei der letzten Exception im User- oder im Kernel-Mode war.

Bit 2/IEp: Das IE-Bit der letzten Exception.

Bit 1/KUc: Das KUc-Bit (Kernel/User current) der aktuellen Exception.

Bit 0/IEc: Das IE-Bit der aktuellen Exception.

Auch das Cause-Register des Koprozessor 0 enthält interessante Bits, die insbesondere für die Exception-Verwaltung wichtig sind. Hier folgt die Aufzählung aller Bits mit dazugehöriger Beschreibung.

Bit 31/BD: Wenn der die Exception auslösende Befehl direkt hinter einem Sprung- oder Verzweigungsbefehl steht, dann befindet er sich in einem sogenannten Branch-Delay-Slot, was durch dieses Bit signalisiert wird.

Bit 30: Dieses Bit ist ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bit 29-28/CE: Wenn ein Koprozessor einen Fehler ausgelöst hat, dann ist in diesen beiden CE-Bits (Coprocessor Error) die Nummer des schuldigen Koprozessors zu finden.

Bits 28-16: Diese Bits sind ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bits 15-10/IP: Diese IP-Bits (Interrupt Pending) entsprechen den sechs Hardware-Interrupts und zeigen, ob Interrupts auf Bearbeitung warten.

Bits 9-8/Sw: Für die Software-Interrupts stehen die Sw-Bits zur Verfügung.

Bit 7: Dieses Bit ist ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Bit 6-2/ExcCode: Diese Bits enthalten den Exception Code, also die Nummer der Exception-Kategorie (siehe Tabelle 9).

Bits 1-0: Diese Bits sind ungenutzt. Bei Schreibzugriffen muß hier immer eine 0 angegeben werden.

Alle mit Bedeutung besetzten Bits des Cause-Registers sind bis auf die Sw-Bits nur lesbar.

C Verzeichnisstruktur und mitgelieferte Diskette

Innerhalb von HADES ist das Verzeichnis `hades/models/mips` angesiedelt, in dem alle zum MIPS-Simulationsmodell gehörigen Dateien abgelegt sind. Alle folgenden Pfadangaben beziehen sich auf das Basisverzeichnis `hades/models/mips`.

Zuerst wird eine Übersicht der Unterverzeichnisse, die die zum MIPS-Simulationsmodell gehörigen Quelltexte enthalten, gezeigt. Darauf folgen die weiteren Verzeichnisse, die die Beispiele, die HADES-Designs und die Dokumentation enthalten.

```
core              // die zum Kern gehoerigen Klassen
gui               // alle Klassen der Benutzerschnittstelle
instr             // die Befehlsklassen und deren Basisklassen
memory            // allgemeine Speicherklassen
mipsmemory        // spezielle Speicherklassen und das HadesInterface
tools             // Hilfsklassen

designs           // verschiedene HADES-Designs (u.a. die Beispiele)
examples          // C-Programme fuer das Simulationsmodell
examples/graphics // Programme, die das grafische LCD benutzen
examples/prim     // Primzahlenberechnungsbeispiel
examples/test     // weitere Tests
text              // Dokumentation zum MIPS-Simulationsmodell
text/grafik       // Bilder dieser Diplomarbeit
text/vortrag      // HTML-Dateien (also die Folien) des Vortrags
```

Die zu dieser Diplomarbeit gehörige Diskette enthält ein Archiv mit dem Namen `mips.zip`. Das Archiv wird auch über die HADES-Homepage [Hendrich 98] zu bekommen sein. Dieses Archiv enthält das oben beschriebene Verzeichnis `mips` mit allen Unterverzeichnissen und kann direkt auf der Verzeichnisebene `hades/models` entpackt werden. Danach ist das MIPS-Simulationsmodell sofort direkt von HADES aus benutzbar.

D Erstellung der Übersetzungswerkzeuge

Zur Benutzung der Übersetzungswerkzeuge müssen diese vorher selbst übersetzt und an ihre Umgebung angepaßt werden. Für die hier folgende Beschreibung wird als Basissystem ein PC mit SuSE Linux 6.2 (Kernel 2.2.10) als Betriebssystem eingesetzt. Der hierauf installierte Compiler ist ein egcs-2.91.66. Für andere Umgebungen kann nicht garantiert werden, daß die angegebenen Vorgehensweisen zu einem Erfolg führen. Alle in diesen Kapiteln angegebenen Aktionen werden, wenn nicht anders angegeben, im Verzeichnis `/opt/mips` ausgeführt. Dieser Pfad frei ist an andere Voraussetzungen anpaßbar.

Zuerst muß der Crossassembler erstellt werden. SuSE 6.2 enthält das Quelltextarchiv `binutils-2.9.1.0.25.tar.gz`. Nachdem dieses Archiv mit dem Installations- und Konfigurationsprogramm YAST installiert worden ist, muß es zuerst entpackt werden.

```
mkdir /opt/mips/sources
cd /opt/mips/sources
```

```
tar -xvzf /usr/src/packages/SOURCES/binutils-2.9.1.0.25.tar.gz
cd binutils-2.9.1.0.25
```

Jetzt muß das Paket konfiguriert werden. Dazu dient folgender Aufruf:

```
./configure --target=mips-idt-elf --prefix=/opt/mips
```

Hier wird als Zielmaschine ein MIPS-Prozessor von IDT mit ELF-Binärdateien eingestellt. Der Pfad, unter dem das Ergebnis nach einem `make install` installiert wird, ist hier auch anzugeben. Die letzten beiden Schritte

```
make
make install
```

übersetzen die Binutils und kopieren sie an die angegebene Stelle. Nun stehen also ein Assembler, ein Linker und einige andere nützliche Programme für den Umgang mit MIPS-IDT-ELF-Dateien zur Verfügung. Assembler und Linker sind zwingende Voraussetzung für die Benutzung des GCC.

Probleme bei der Übersetzung können auftreten, wenn lex oder flex fehlen. In diesem Fall meldet der configure-Aufruf einen Fehler und flex muß vor einem erneuten Versuch installiert werden.

Nachdem die Binutils mit Crossassembler und Linker vorliegen kann nun der Crosscompiler erstellt werden. In der SuSE 6.2 Distribution ist auch der GCC im Quelltext vorhanden. Das Paket nennt sich `gcc-2.7.2.3.tar.gz`. Eigentlich ist auch der modernere EGCS-Compiler Teil der Linux-Distribution, leider schlugen jedoch die Übersetzungsversuche für einen Crosscompiler fehl. Deshalb wird nun mit dem GCC fortgefahren, der für die Zwecke hier vollkommen ausreicht.

Die Schritte der Übersetzung sind ähnlich der Vorgehensweise für die Binutils. Nur muß vorher dafür gesorgt werden, daß für die Übersetzung alle notwendigen Header-Dateien vorhanden sind. Die folgende Liste zeigt alle Header-Dateien, die für das beschriebene System zur Übersetzung des Crosscompilers notwendig sind. Obwohl diese Dateien von einem Intel-Linux-System stammen, müssen an diesen Dateien keine Änderungen vorgenommen werden. Die enthaltenen Werte sind auch für den MIPS-Crosscompiler gültig.

```
mkdir /opt/mips/mips-idt-elf/include
cd /opt/mips/mips-idt-elf/include
mkdir bits
mkdir sys
mkdir gnu
cp /usr/include/stdio.h .
cp /usr/include/bits/types.h bits/
cp /usr/include/bits/stdio_lim.h bits/
cp /usr/include/libio.h .
cp /usr/include/features.h .
cp /usr/include/_G_config.h .
cp /usr/include/bits/stdio.h bits/
cp /usr/include/sys/cdefs.h sys/
cp /usr/include/gnu/stubs.h gnu/
```

Auf die Vorbereitung der Header-Files folgt nun die Installation der Quelltexte:

```
cd /opt/mips/sources
tar -xvzf /usr/src/packages/SOURCES/gcc-2.7.2.3.tar.gz
cd gcc-2.7.2.3
```

Nachdem diese Vorbereitungen beendet sind, folgen nun die Schritte für die Übersetzung:

```
./configure --target=mips-idt-elf --with-gnu-as --with-gnu-ld \
            --prefix=/opt/mips
make LANGUAGES=c
make LANGUAGES=c install
```

Falls der erste make-Aufruf Probleme machen sollte, hilft das Einfügen der Zeile `mkdir include` in Zeile 2115 im `Makefile`.

Als Ergebnis steht nun der C-Crosscompiler zur Verfügung, der unter `/opt/mips/bin/mips-idt-elf-gcc` zu finden ist.

E initcode.h und initcodeelf.s

Als minimaler Ersatz eines Laufzeitsystems steht `initcodeelf.s` zur Verfügung. Dieser Quelltext ist grundsätzlich der einzig notwendige Assembler-Quelltext, da alle weiteren Quelltexte auf diesem basieren und entsprechend auch in C verfaßt sein können.

Die Initialisierungssequenz ist abhängig vom Linkformat, da unterschiedliche Linkformate unterschiedliche vordefinierte Symbole für den Linker voraussetzen. Zur Benutzung des ELF-Dateiformats müssen die Symbole `_start` und `__main` definiert sein. `_start` ist der Einsprungpunkt in das Programm, von hier aus muß also die `main`-Prozedur aufgerufen werden. Diese wiederum ruft dann immer direkt `__main` auf, so daß hier im Normalfall bis auf einen Rücksprung nichts weiter getan werden muß.

Zusätzlich werden im `initcode` noch einige Hilfsmethoden definiert, die für Debug-Zwecke gedacht sind:

print: Die `print`-Prozedur kann einen Integer-Parameter auf der Standard-Ausgabe anzeigen.

breakpoint: Diese Prozedur verhält sich wie print, nur wird zusätzlich dafür gesorgt, daß der Programmablauf angehalten wird.

time: Mit dieser Prozedur kann die Anzahl der Millisekunden, die seit dem Start des MIPS-Simulationsmodells vergangen sind, ausgegeben werden.

exit: `exit` versucht, die Ausführung des Programms zu beenden, indem die wahrscheinliche Rücksprungadresse vom Stapelspeicher geladen wird.

Die Header-Datei `initcode.h` im einzelnen:

```
/* Interface to the syscall instruction */

/** Print the integer value to System.out */
void print(int value);

/** Print the integer value to System.out and stop execution */
void breakpoint(int value);

/** Jump to the end of main() and print value */
void exit(int value);

/** Print milliseconds since start */
void time();
```

Die dazugehörige Assembler-Implementation der Initialisierungssequenz für ELF-Programme `initcodeelf.s`:

```
        .file   1 "initcodeelf.c"

        .text
        .align  2
        .globl  _start
        .ent    _start
_start:                     // Diese Marke zeigt absoluten Anfang
        jal     main        // Sprung nach void main()
        jal     time        // Spring zu time(), s. u.
        j       _start      // Danach wieder von vorn
        .end    _start

        .globl  __main
        .ent    __main
__main:                     // Diese Marke wird von main() gerufen
        j       $31         // Gleich wieder zurueck
        .end    __main

        .globl  print
        .ent    print
print:                      // Ausgabe eines int nach Stdout
        addu    $5, $0, $4  // Wert ins zweite Parameterregister
        addiu   $4, $0, 0   // Indikator fuer print() ins erste
        syscall             // Und Aufruf
        j       $31         // Sprung zurueck
        .end    print

        .globl  breakpoint
        .ent    breakpoint
breakpoint:                 // Ausgabe eines int und Halt
        addu    $5, $0, $4  // Wert ins zweite Parameterregister
        addiu   $4, $0, 1   // Indikator fuer breakpoint() ins erste
        syscall             // Und Aufruf
        j       $31         // Sprung zurueck
        .end    breakpoint
```

```
        .globl  time
        .ent    time
time:                       // Ausgabe der Millisekunden seit Start
        addiu   $4, $0, 2   // Indikator fuer time() ins erste
        syscall             // Und Aufruf
        j       $31         // Sprung zurueck
        .end    time

        .globl  exit
        .ent    exit
exit:                       // Programmende
        lw      $31, -4($0) // Aufrufadresse vom Stack
        addiu   $30, $0, 0  // Link-Register loeschen
        addiu   $29, $0, 0  // Link-Register loeschen
        jr      $31         // Und Sprung zurueck
        .end    exit
```

Die Übersetzung und die Benutzung des `initcode` ist in Abschnitt 8.2.3 beschrieben.

F graphics.h und graphics.c

Im Abschnitt 8.3.3 wird die Ansteuerung eines grafischen LCD und dessen Benutzung für Textausgaben beschrieben. Hier werden nun die dafür notwendigen Quelltexte vorgestellt.

Als Header-Datei dient `graphics.h`.

```
/* Interface to the Samsung KS0108 LCD Controller */

/** Enable or disable display (enable=1, disable=0) */
void enableDisplay(int on);

/** Clear the display */
void clearDisplay();

/** Set position of text cursor */
void setTextPos(int x, int y);

/** Print text to the current cursor position */
void text(char *string);

/** Print big text to the current cursor position */
void bigtext(char *string);
```

Dazu gehört dann die Implementation `graphics.c`. Die in diesem Quelltext enthaltene Definition des Zeichensatzes ist wegen der besseren Übersichtlichkeit gekürzt.

```
#include "graphics.h"

#define ENAMASK 0x2000
#define CHIP1 0x0800
```

```
#define CHIP2 0x0400

#define DISPLAYON 0x103e
#define SETPAGEADR 0x10b0
#define SETBYTEADR 0x1040
#define SETSTARTADR 0x10c0
#define WRITE 0x1200

int xpos = 0;
int ypos = 0;
int chipselect = CHIP1;

unsigned char charset[128][7] = {
  { 0,0,0,0,0,0,0 },              // 0
  { 0,0,0,0,0,0,0 },              // 1
  { 0,0,0,0,0,0,0 },              // 2
  { 0,0,0,0,0,0,0 },              // 3
  // ...
  { 3, 0, 0, 0, 0, 0, 0 },        // Space
  { 3, 47, 0, 0, 0, 0, 0 },       // !
  { 4, 3, 0, 3, 0, 0, 0 },        // "
  { 6, 20, 62, 20, 62, 20, 0 },   // #
  // ...
  { 5, 30, 37, 35, 30, 0, 0 },    // 0
  { 5, 4, 34, 63, 32, 0, 0 },     // 1
  { 5, 34, 49, 41, 38, 0, 0 },    // 2
  { 5, 18, 33, 37, 26, 0, 0 },    // 3
  // ...
  { 6, 30, 33, 45, 41, 6, 0 },    // @
  { 5, 62, 5, 5, 62, 0, 0 },      // A
  { 5, 63, 37, 37, 26, 0, 0 },    // B
  { 5, 30, 33, 33, 18, 0, 0 },    // C
  // ...
  { 3, 1, 2, 0, 0, 0, 0 },        //
  { 5, 24, 36, 36, 56, 0, 0 },    // a
  { 5, 63, 36, 36, 24, 0, 0 },    // b
  { 4, 24, 36, 36, 0, 0, 0 },     // c
  // ...
  { 4, 0, 63, 0, 0, 0, 0 },       // |
  { 4, 33, 30, 4, 0, 0, 0 },      // }
  { 5, 2, 1, 2, 1, 0, 0 },        // ~
  { 6, 85, 170, 85, 170, 85, 170 } // 127
};


/** Give one command to the display controller */
inline void command(int value)
{
  int *adr;

  adr = 0xa0000000;
  *adr = value;
}
```

```
/** Enable or disable display (enable=1, disable=0) */
void enableDisplay(int on)
{
  command(DISPLAYON | on);
}


/** Clear the display */
void clearDisplay()
{
  int x=0;
  int y=0;
  int *adr;

  adr = 0xa0000000;

  *adr = SETBYTEADR;
  for (y=0; y<8; y++) {
    *adr = SETPAGEADR | y;
    for (x=0; x<64; x++) {
      *adr = WRITE;
    }
  }
}


/** Set position of text cursor */
void setTextPos(int x, int y)
{
  x = x*5;
  if (x >= 64) {
    chipselect = CHIP2;
    x = x & 63;
  } else {
    chipselect = CHIP1;
  }
  ypos = y;
  xpos = x;
}


/** Print text to the current cursor position */
void text(char *string)
{
  int j;
  int value;
  int len;
  char ch;
  int *adr;

  adr = 0xa0000000;
```

```
  command(SETBYTEADR | xpos);
  command(SETPAGEADR | ypos);

  while (*string) {
    ch = *string & 127;
    len = charset[ch][0];
    if ((ch == 10) || ((xpos+len >= 64) && (chipselect == CHIP2))) {
      if (ch != 10) {
        string--;
      }
      len = 64-xpos;
      for (j=0; j<len; j++) {
        *adr = WRITE | chipselect;
      }
      xpos = 0;
      *adr = SETBYTEADR;
      if (chipselect == CHIP1) {
        for (j=0; j<64; j++) {
          *adr = WRITE | CHIP2;
        }
      }
      ypos = (ypos + 1) & 7;
      *adr = SETPAGEADR | ypos;
      chipselect = CHIP1;
    } else {
      for (j=1; j<=len; j++) {
        *adr = WRITE | chipselect | charset[ch][j];
        xpos++;
        if (xpos > 63) {
          xpos = 0;
          *adr = SETBYTEADR;
          chipselect = CHIP2;
        }
      }
    }
    string++;
  }
}


/** Print big text to the current cursor position */
void bigtext(char *string)
{
  int j;
  int value;
  int len;
  char ch;
  int *adr;

  adr = 0xa0000000;

  command(SETBYTEADR | xpos);
```

```
  command(SETPAGEADR | ypos);

  while (*string) {
    ch = *string & 127;
    len = charset[ch][0];
    if ((ch==10) || ((xpos+(len*2)>=64) && (chipselect==CHIP2))) {
      if (ch != 10) {
        string--;
      }
      len = 64-xpos;
      for (j=0; j<len; j++) {
        *adr = WRITE | chipselect;
      }
      xpos = 0;
      *adr = SETBYTEADR;
      if (chipselect == CHIP1) {
        for (j=0; j<64; j++) {
          *adr = WRITE | CHIP2;
        }
      }
      ypos = (ypos + 1) & 7;
      *adr = SETPAGEADR | ypos;
      chipselect = CHIP1;
    } else {
      for (j=1; j<=len; j++) {
        *adr = WRITE | chipselect | charset[ch][j];
        xpos++;
        if (xpos > 63) {
          xpos = 0;
          *adr = SETBYTEADR;
          chipselect = CHIP2;
        }
        *adr = WRITE | chipselect | charset[ch][j];
        xpos++;
        if (xpos > 63) {
          xpos = 0;
          *adr = SETBYTEADR;
          chipselect = CHIP2;
        }
      }
    }
    string++;
  }
}
```

G Begriffserklärungen

Dieser Anhang bietet einen Überblick über einige eventuell nicht allgemein geläufige Begriffe und Abkürzungen mit deren in diesem Text verwendeter Bedeutung.

ALU: ALU steht für *Arithmetical Logical Unit*. Die ALU berechnet arithmetische, logische und Shift-Operationen und arbeitet mit ganzen Zahlen.

Cache: Der Begriff *Cache* steht für einen kleinen schnellen Zwischenspeicher, der zur Beschleunigung von Speicherzugriffen eingesetzt wird. Das Prinzip des Cache basiert auf der Annahme, daß Speicherzugriffe lokal gehäuft auftreten und kann überdies beschleunigend eingesetzt werden, wenn bereits aus dem Speicher gelesene Werte mehrmals wieder gelesen werden. Da für den Begriff des Cache kein adäquates Gegenstück in der deutschen Sprache existiert, wird in diesem Text der englische Begriff verwendet.

DMA: Greifen innerhalb einer Schaltung andere Komponenten als der zentral steuernde Mikroprozessor auf den Speicher zu, so nennt sich das direkter Speicherzugriff. Im Englischen wird dafür der Ausdruck *Direct Memory Access*, DMA, benutzt.

Exception: Sowohl in der Hardware-Welt als auch innerhalb der Software-Entwicklung wird von *Exceptions* gesprochen, jedoch können sich dahinter viele verschiedene Bedeutungen verbergen. Für die MIPS-Terminologie ist eine Exception eine Unterbrechung des regulären Programmablaufs, egal ob diese Unterbrechung ihre Ursache innerhalb oder außerhalb des Prozessors hat. In diesem Text wird (wenn nicht genauer erläutert) diese Bedeutung für den Begriff Exception verwendet. Wenn von Java-Software-Exceptions die Rede ist, wird das auch explizit erwähnt.

FPU: Die FPU ist die *Floating Point Unit*, also die Fließkommaeinheit, und bietet Befehle zur Berechnung verschiedener komplexer mathematischer Funktionen, deren Operanden und Ergebnisse durch Fließkommazahlen repräsentiert werden.

HADES-Design: Ein HADES-Design ist die Zusammenschaltung mehrerer HADES-Modelle zu einem simulierbaren Ganzen. Als Beispiel sei hier das MIPS-Simulationsmodell zusammen mit einem Taktgenerator, einem Adreßdekoder und einem Speicher genannt.

Koprozessor: In der MIPS-Terminologie bezeichnet ein Koprozessor nicht unbedingt einen vom Hauptprozessor physikalisch getrennten weiteren Prozessor. Vielmehr ist damit das logische Konzept einer weiteren Einheit zur Befehlsausführung, getrennt vom eigentlichen Prozessorkern, gemeint. Die Realisierung von mehreren Koprozessoren kann also durchaus zusammen mit dem Prozessorkern auf einem Chip stattfinden.

Pipeline: Wörtlich übersetzt bedeutet *Pipeline* Rohrleitung oder auch Verbindung. Als allgemein auch in deutscher Fachliteratur etablierter Begriff wird dieser nicht durch ein deutsches Gegenstück ersetzt. Der Begriff Pipeline beschreibt in dieser Diplomarbeit den internen Aufbau eines Prozessors, der mehrere Befehle in Teilaktionen auftrennt und diese zeitversetzt gleichzeitig ausführt.

TLB/MMU: Es gibt Ausführungen der IDT-R30xx-Familie mit MMU (*Memory Management Unit*, Speicherverwaltungseinheit). Innerhalb der MIPS-Terminologie wird statt MMU der Ausdruck TLB (*Translation Lookaside Buffer*) benutzt, da der Aufbau der MMU sehr einfach gehalten ist und nichts weiter als den TLB enthält, der bei MMU-Implementationen anderer Architekturen nur ein Teil der gesamten Einheit ist. In diesem Text werden deshalb die Begriffe MMU und TLB synonym verwendet.