SoCRocket - Eine flexible erweiterbare Virtuelle Plattform zum Entwurf robuster Eingebetteter Systeme

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Dipl.-Ing. Thomas Schuster
geboren am 28.03.1976
in Räckelwitz

Eingereicht am: 11.11.2014
Disputation am: 08.04.2015

1. Referent: Prof. Dr.-Ing. Mladen Berekovic
2. Referent: Prof. Dr.-Ing. Harald Michalik

(2015)

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Braunschweig, den 15.05.15 Thomas Schuster

Vorwort

Diese Doktorarbeit ist der Ergebnis einer langjährigen und fruchtbaren Zusammenarbeit mit Prof. Dr.-Ing. Mladen Berekovic, den ich 2004 als Praktikant und späterer Angestellter am IMEC in Belgien kennenlernte und 2007 als Wissenschaftlicher Mitarbeiter an die Technische Universität Braunschweig begleitete. Ich möchte ihm hier ganz besonders für seine Unterstützung danken.

Ganz besonderer Dank gebührt ebenfalls Prof. Dr.-Ing. Harald Michalik, der mich durch einen gemeinsamen Projektantrag (2009), in Kontakt zur Europäischen Raumfahrtagentur (ESA) brachte und damit nicht nur wesentlich zur Finanzierung, sondern auch zur fachlichen Ausrichtung der vorliegenden Arbeit beitrug.

Der Weg zur Entstehung dieser Arbeit war spannend, selten geradlinig und erwies sich als kniffliger Balanceakt zwischen Industrieprojekten, wissenschaftlicher Arbeit, Lehre und Projektorganisation; ein Weg den ich ohne die Unterstützung meiner Kollegen am Lehrstuhl für Technische Informatik nicht hätte bewältigen können. Ganz herzlichen Dank an Rolf Meyer, der mir mit seinen großartigen Ideen und technischem *Know-How* während der Arbeit an *SoCRocket* zur Seite stand. Vielen Dank auch an Rainer Buchty, für seine wertvollen Anregung und die Durchsicht des Manuskripts, an Jan Wagner für seine Ideen zur Datenanalyse, Sören und Sönke Michalik für die fachlichen Diskussionen rund um GRLIB und Syed Abbas Ali Shah für die Zusammenarbeit auf dem Gebiet der *Power*-Simulation.

Ich danke ebenfalls Luca Fossati von der ESA für das Teilen seiner Erfahrungen und die Zusammenarbeit in *SoCRocket* und Christian Sauer von *Cadence* der mein Verständnis moderner Entwurfsmethoden durch sein *Mentoring* wesentlich beeinflusst hat.

Nicht zuletzt danke ich meiner Frau Kerstin, meinen Eltern, meinen Schwiegereltern und Oma Helene, ohne deren unermüdliche Unterstützung, Verständnis und auch gelegentliche Aufbauarbeit all dies nicht möglich gewesen wäre.

Braunschweig, am 15. Mai 2015

Thomas Schuster

Kurzfassung

Die moderne Halbleitertechnologie ermöglicht eine kostengünstige Fertigung von integrierten Schaltungen bestehend aus mehreren Milliarden Transistoren. Logik dieser Komplexität kann jedoch mit konventionellen Entwurfsmethoden, zum Beispiel auf Register-Transfer-Ebene, nicht effizient beschrieben werden. Grund ist der zur Erzeugung von synthetisierbaren Schaltungen erforderliche hohe Detailgrad, der eine sinnvolle Architekturexploration unmöglich macht. Um die vorhandenen technischen Möglichkeiten besser ausschöpfen zu können, werden heute drei Ansätze verfolgt: 1) die Integration immer größerer Speichermengen auf dem Chip, 2) die systematische Wiederverwendung von Logikkomponenten (IP-Reuse) und 3) die Erhöhung des Abstraktionsniveaus im Entwurfsprozess. Der Schwerpunkt dieser Arbeit liegt in der Erhöhung des Abstraktionsniveaus, speziell dem Entwurf von Systemen auf Basis von Virtuellen Plattformen (VPs), *Transaction-Level*-Modellierung (TLM) und *SystemC*. Es wird eine ganzheitliche Methode vorgestellt, mit der komplexe eingebettete Systeme effizient modelliert werden können. Ergebnis ist eine der RTL-Synthese nahezu gleichgestellte Genauigkeit bei wesentlich höherer Abstraktion, Flexibilität und Simulationsgeschwindigkeit.

Die der vorliegenden Arbeit zu Grunde liegenden Konzepte sind seit geraumer Zeit Thema zahlreicher Forschungsaktivitäten. Die Akzeptanz und Umsetzung in die Praxis durch die Industrie ist bisher jedoch nur in Ausnahmefällen gelungen. Ganze Branchen, beispielsweise die Automobilindustrie oder der Luft- und Raumfahrtsektor, zögern in der Anwendung der neuen Methodik. Gründe dafür sind Standardisierungslücken, welche die Interoperabilität und Wiederverwendbarkeit von TL-Modellen einschränken, die Abschreckung durch hohe Investitionen in Entwicklungswerkzeuge und der Mangel an geschultem Personal zur effizienten Umsetzung abstrakter Entwurfsmethoden. Diesen Beschränkungen wird durch die Entwicklung einer vollständig offenen Virtuellen Plattform begegnet. Das *SoCRocket*-System orientiert sich dazu an existierenden Standards wie IEEE 1666 SystemC/TLM2.0 und stellt Methoden zu deren effizientem Einsatz zur Verbesserung von Simulationsgeschwindigkeit und Simulationsgenauigkeit vor. So wird unter anderem gezeigt, wie moderne Multi-Kanal-Protokolle mit Split-Transfers durch Ausgleich des Intertransaktions-Timings ohne die Einführung zusätzlicher Protokollphasen zeitlich genau modelliert werden können. Standardisierungslücken in den Bereichen Speichermodellierung und Systemkonfiguration werden durch standardoffene Lösungen geschlossen. Darüber hinaus wird neue Infrastruktur zur Modellierung von Signalkommunikation auf Transaktionsebene, der Verifikation von Komponenten und der Modellierung des Energieverbrauchs vorgestellt.

Die entwickelte Methodik beruht auf der Kapselung der Grundfunktionalitäten des Systems – wie Busschnittstellen, Konfigurationsinformationen, Speicherelemente, Zeitverhalten oder Energiemodelle – in Bibliotheksbasisklassen. Neue Komponenten können durch Vererbung von diesen Basisklassen auf einfache Weise generiert werden. Systeme werden mit Hilfe von *Templates* erzeugt, die zur Laufzeit dynamisch konfiguriert werden. Dadurch lassen sich alle Systemparameter, bis hin zur Abstraktion der Busschnittstelle, ohne erneute Übersetzung variieren. Zur Demonstration wurden die Kernkomponenten einer im europäischen Raumfahrtsektor maßgeblichen Hardwarebibliothek modelliert. Neben einem aus einer Integereinheit entwickelten LEON2/3-Prozessorsimulator mit Cachesystem und Memory-Management-Einheit entstanden ein Busmodell für AMBA 2.0, ein Speichercontroller und diverse Peripheriekomponenten wie *Timer* und Interruptcontroller. Alle Komponenten wurden zunächst in *Unit*-Tests verifiziert und anschließend in einem Systemprototypen integriert. Zur Verifikation der Funktion, sowie Bestimmung von Simulationsgeschwindigkeit und zeitlicher Genauigkeit, wurde dieser für unterschiedliche Abstraktionsstufen konfiguriert und mit einem in VHDL beschriebenen RISC-Referenzentwurf (LEON3MP) verglichen. Das System mit losem *Timing* (LT) und blockierender

Kommunikation ist im Durchschnitt 561-mal schneller als die RTL-Referenz und weist eine durchschnittliche *Timing*-Abweichung von 7,04% auf. Das System mit näherungsweise akkuratem *Timing* (AT) und nicht-blockierender Kommunikation ist 335-mal schneller. Die durchschnittliche *Timing*-Abweichung beträgt hier nur noch 3,03%, was einer Standardabweichung von 0.033 und damit einer sehr hohen statistischen Sicherheit entspricht. Die verschiedenen Abstraktionsniveaus können zur Realisierung mehrstufiger Architekturexplorationen eingesetzt werden. Dies wird am Beispiel einer hyperspektralen Bildkompression verdeutlicht, die auf ein Mehrprozessorsystem abgebildet wurde. Die dynamische Rekonfigurierbarkeit des Systems ermöglicht es, einen Entwurfsraum aus 1280 Prototypen in weniger als 24 Stunden vollständig zu durchsuchen.

Abstract

Modern semiconductor technology enables cost-efficient production of integrated circuits consisting of several billions of transistors. Logic of this complexity cannot be solely designed at register-transfer level anymore. Reason is the high degree of detail required for generating synthesizable circuits making reasonable architecture exploration impossible. Today, three approaches are being followed for better exploitation of the existing technological capacities: 1) integration of ever larger amounts of memory on the chip, 2) systematic reuse of logic building blocks (IP-Reuse), and 3) raising the abstraction level of the design process. The focus of this work is the raise of the abstraction level, especially for the design of systems based on Virtual Platforms (VPs), Transaction Level Modeling (TLM), and SystemC. A holistic method for efficient modeling of complex embedded systems is presented. Results are accuracies close to RTL synthesis but at much higher abstraction, flexibility, and simulation performance.

The underlying concepts of this work have been subject to numerous research activities. However, general acceptance by the industry is still low, except some special cases like mobile communications. Whole domains, such as the automotive industry or aerospace, hesitate adapting the new methodology. The reasons for this are standardization gaps which limit interoperability and reuse of models, required high investments in new development tools, absence of experienced personnel for efficient application of abstract design methods, and lack of reliable simulation models. These problems are addressed by developing a completely open and extensible Virtual Platform. The *SoCRocket* system integrates existing standards like IEEE 1666 SystemC/TLM2.0, and introduces new methods for improvement of simulation performance and accuracy. It is shown, amongst others, how modern multi-channel protocols with split transfers can be accurately modeled by compensating inter-transaction timing without introducing additional protocol phases. Standardization gaps in the area of memory modeling and system configuration are closed by standard-open solutions. Furthermore, new infrastructure for modeling signal communication on transaction level, verification of components, and estimating power consumption are presented. The developed methodology is based on encapsulation of system core functionality – such as bus interfaces, configuration mechanism, storage elements, and timing and power models – in library base-classes. From these base-classes, new components can be generated by inheritance in a very simple way. Systems are assembled using run-time reconfigurable design templates, therefore, system parameters up to the level of bus interface abstraction can be varied without compilation. As a proof of concept core components of a decisive aerospace hardware library have been modeled. In addition to a LEON2/3 processor simulator with caches and memory management unit, developed from a preexisting integer unit, a bus model for AMBA 2.0, a memory controller, and various peripherals such as timer and interrupt controller are available. All components have been verified in unit tests and were subsequently integrated in a system prototype. For functional verification, as well as measurement of simulation performance and accuracy, the prototype was configured for different abstractions and compared to a VHDL-based RISC reference design (LEON3MP). The loosely-timed platform prototype with blocking communication (LT) is in average 561 times faster than the RTL reference and shows an average timing deviation of 7,04%. The approximately-timed system (AT) with non-blocking communication is 335 times faster. Here, the timing deviation is only 3,03 %, corresponding to a standard deviation of 0.033, proving a very high statistic certainty. The system's various abstraction levels can be exploited by a multi-stage architecture exploration. This is demonstrated by the example of a hyperspectral image compression, which was mapped to a multi-processor system. Thanks to the VP's reconfiguration capabilities, a design space of 1280 prototypes can be exhaustively explored in less then 24 hours.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Ziele

Eingebettete Systeme sind in der Regel elektronische Systeme, die in größere Systeme integriert sind. Sie sind im heutigen Leben allgegenwärtig und werden in der Unterhaltungselektronik, dem Automobilbau, dem Maschinenbau, der Luft- und Raumfahrtechnik und vielen anderen Bereichen eingesetzt. Wie auch integrierte Schaltungen allgemein haben sie sich seit der Einführung der ersten Mikrochips enorm weiterentwickelt. Bereits im Jahre 1965 formulierte Gordon Moore ausgehend von Beobachtungen der vergangenen Jahre die Vermutung, dass sich die Komplexität von Mikrochips in den folgenden Jahren jährlich verdoppeln würde. Diese Vermutung ist in leicht abgewandelter Form[1] als Moore's Law bekannt und hat bis heute Gültigkeit [Moo06]. Der im Jahre 1971 vorgestellte erste Mikroprozessor, der *Intel 4004*, bestand aus nur 2300 Transistoren. Aktuelle Systeme wie der 2013 eingeführte *IBM POWER8* bringen es auf bis zu 5 Milliarden Transistoren. In der Tat entwickelte sich die Halbleiterfertigungstechnik derart schnell, dass speziell im Verlauf der letzten ca. 15 Jahre eine immer größere Lücke zwischen der theoretisch möglichen Chipkapazität und deren sinnvoller Nutzbarkeit entstand. Diese sogenannte Produktivitätslücke wurde erstmals 2004 in der *International Technology Roadmap for Semiconductors* (ITRS) beschrieben (Abbildung 1.1 - *ITRS Roadmap* 2007 [itr14]).

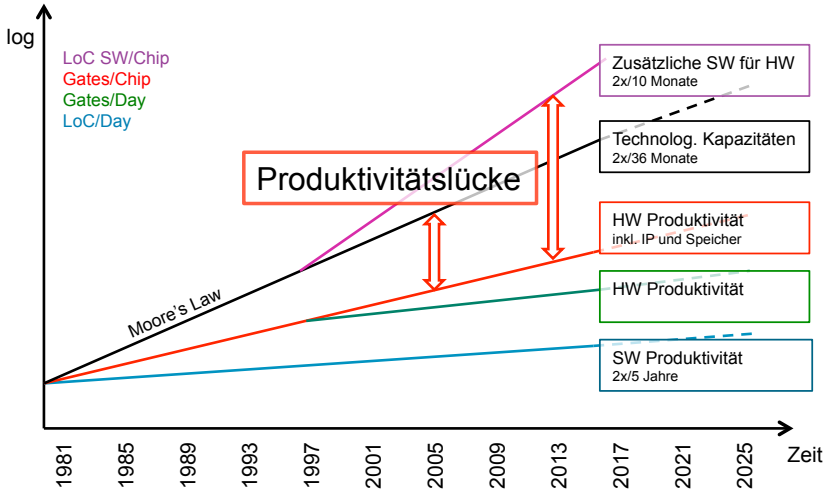


Abbildung 1.1: HW & SW Produktivitätslücke

Ansätze zur Lösung oder Linderung dieses Problems sind verstärkte Wiederverwendung von Komponenten (*IP-Reuse*), Integration immer größerer Speicher auf dem Chip und Weiterentwicklung der Entwurfsmethodik im Rahmen des *Electronic System Level Design* (ESLD)[Den06]. Während *IP-Reuse* und die Vergrößerung der *On-Chip*-Speicher in allen Anwendungsgebieten eingebetteter Systeme angenommen wurden, bereitet die Umsetzung von ESLD große Schwierigkeiten. Die Gründe dafür sind folgende:

1. Hohe Kosten für die Umstellung von *Workflows* und die Einführung neuer Werkzeuge,
2. Mangel an geeignetem Personal mit Expertise für abstrakte Entwurfsmethoden,

1 Mooresches Gesetz: Verdoppelung der Komplexität alle ca. 24 Monate

3. Mangel an verlässlichen Simulationsmodellen
4. und eingeschränkte Wiederverwendbarkeit abstrakter Modelle auf Grund fehlender Standards.

Getrieben durch kurze Produktzyklen ist ESLD bislang nur von der Mobilfunksparte adaptiert und wird ansonsten aufgrund der genannten Punkte zu zögerlich oder gar nicht umgesetzt. Ein Beispiel hierfür ist der europäische Raumfahrtsektor, in dem ESLD bis heute keine Rolle spielt. Zur Förderung moderner Entwurfsmethoden auf diesem Gebiet wurde in der *Abteilung Technische Informatik* der TU Braunschweig in Zusammenarbeit mit der Europäischen Raumfahrtagentur (ESA) die Virtuelle Plattform *SoCRocket* entwickelt. *SoCRocket* adressiert die oben genannten Probleme wie folgt:

1. Entwicklung offener Simulationsmodelle und Werkzeuge zur Konstruktion von *Data Processing Units* (DPU) auf unterschiedlichen Abstraktionsstufen,
2. Vereinfachung des Entwurfs von Modellen und Virtuellen Prototypen durch die Kapselung von Komplexität und die Bereitstellung einfacher Programmierschnittstellen,
3. umfassende Verifikation und Genauigkeitsuntersuchungen bezüglich der entworfenen Modelle und Werkzeuge
4. und ausschließliche Verwendung standardisierter und standardoffener Lösungen.

In dieser Arbeit werden am Beispiel von *SoCRocket* verschiedene Aspekte des Entwurfs von Simulationsmodellen und der dazu erforderlichen Infrastruktur erörtert und gezielt weiterentwickelt. Schwerpunkte sind der Aufbau und die Struktur von Modellen mit Hilfe von *SystemC*, effiziente Modellierung von *TLM*-Kommunikation, Handhabung von Metadaten, sowie Modellierung von Verhalten, Zeit (*Timing*), Leistungsaufnahme und Speicherelementen. Ziel ist die Bereitstellung geeigneter Modelle und Werkzeuge zur Steigerung der Produktivität beim Entwurf von Weltraum-DPUs. Die vorgestellten Lösungen und Konzepte sind dabei nicht auf den Raumfahrtsektor beschränkt und können auf beliebige andere Einsatzgebiete eingebetteter Systeme übertragen werden.

1.2 Virtuelle Plattform und ESLD – Definition

Der Begriff ESLD hat sich im Verlauf der letzten zehn Jahre als Ersatz und Erweiterung der Bezeichnung *System-Level-Design*, d.h. Entwurf auf Systemebene, herausgebildet. Die Abwandlung soll im Allgemeinen die Einbeziehung höherer Abstraktionsebenen in den Entwurfsprozess hervorheben. Die ITRS-*Roadmap* von 2004 definiert ESL vage als „eine Entwurfsebene über der Register-Transfer-Ebene“. Darüber hinaus gibt es verschiedene Versuche einer genaueren Begriffsklärung. Gartner/Dataquest Report beschreibt ESL als das „gleichzeitige Design von Hardware und Software“. Bailey und Martin umschreiben es als:

„... die Benutzung von Abstraktion zur Verbesserung des Verständnisses eines Systems, und Steigerung der Wahrscheinlichkeit zu dessen erfolgreichen Implementierung auf kosteneffiziente Art und Weise, unter Beachtung aller Randbedingungen[1]*.“*[Bai07]

Das Schlüsselwort in dieser Definition ist “Abstraktion“. Abgeleitet vom lateinischen Begriff *abstractus* (deutsch: abgezogen) bezeichnet es einen “induktiven Denkprozess des Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres“ [Pre08]. In Bezug

1 "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints

auf elektronische Systeme ist Abstraktion ein schwieriger Prozess, da es für die Wahl des richtigen Abstraktionsniveaus zur Modellierung und damit für die Entscheidung was zur Steigerung des Verständnisses weggelassen werden kann, nur wenige Hilfsmittel gibt. Die im Mittelpunkt dieser Arbeit stehenden Virtuellen Plattformen stellen eines der wichtigsten Werkzeuge dar, die in diesem Zusammenhang eingesetzt werden. Virtuelle Plattformen sind:

"... Softwareimplementierungen prozessorbasierter Systeme auf einem dem Anwendungszweck (Use Case) angemessenen Abstraktionsniveau."[Bai07]

Das primäre Ziel zum Einsatz Virtueller Plattformen ist es, im Vergleich zu konventionellen Systemmodellen (z.B. auf Register-Transfer-Ebene (RTL)) Simulationsgeschwindigkeit zu gewinnen. RT-Modelle simulieren, abhängig von der Komplexität des Modells, mit Geschwindigkeiten im ein- oder zweistelligen Kilohertz-Bereich. Dies ist gegebenenfalls gerade schnell genug für zyklengenaue Verifikation, aber viel zu langsam für Architekturexploration, die Analyse von Geschwindigkeit und Bandbreite oder Softwareentwicklung. Um annähernd in Echtzeit, also mit mehreren hundert Kilohertz oder Megahertz simulieren zu können, müssen RT-Modelle durch schnelle Simulationsmodelle mit reduzierter Komplexität und geringerer Anzahl paralleler Prozesse ersetzt und Kommunikationsschnittstellen durch vereinfachte abstrakte Protokolle modelliert werden. Den Durchbruch für diese Technologie kennzeichneten die Standardisierung der C++-Ergänzungsbibliothek *SystemC* im Jahre 2005 (IEEE 1666) und die Einführung der zweiten Version der *Transaction Level Modeling*-Bibliothek (TLM2.0) im Jahre 2008. Weitere Vorteile virtueller Plattformen sind die zeitige Verfügbarkeit von Simulatoren, wodurch die Entwicklung von Software beginnen kann, noch bevor echte Hardware zur Verfügung steht. Darüber hinaus sind VPs reine Software und können daher problemlos dupliziert werden, was es erlaubt mit mehreren Entwicklern in parallel zu arbeiten. Außerdem besteht im Unterschied zu ASIC- oder FPGA-Prototypen volle Sichtbarkeit auf alle Systemkomponenten, wodurch die Analyse von Simulationen und das Beheben von Fehlern erheblich erleichtert werden.

1.3 Eingebettete Systeme in der Europäischen Raumfahrt

Die vorliegende Arbeit wurde zur Steigerung der Effizienz des Entwurfsprozesses für eingebettete Systeme im Raumfahrtbereich durch die Europäische Raumfahrtagentur angeregt und orientiert sich an den besonderen Anforderungen dieses Industriezweigs. Integrierte Schaltungen zum Einsatz in Satelliten oder anderen Raumfahrzeugen müssen hochrobust sein, da sie starken Erschütterungen und hohen Strahlungsdosen ausgesetzt werden. Der Ausfall eines einzelnen Systems kann eine ganze Mission gefährden und gegebenenfalls großen Schaden anrichten. Daher sind eingebettete Systeme für Raumfahrtanwendungen im Vergleich zu anderen Einsatzgebieten, etwa der Unterhaltungselektronik, sehr konservativ ausgelegt. Dies betrifft sowohl die verwendeten Prozessoren, die – einmal qualifiziert – intensiv wiederverwendet werden, als auch Zieltechnologie und Entwurfsmethodik. Besonders die Entwicklung von ASICs ist extrem teuer, da die benötigten Stückzahlen in der Regel sehr klein sind. Hier kommen spezielle strahlungsfeste Logikbibliotheken und Speicher zum Einsatz. Aktuell gängige Strukturbreiten sind 0.18 - 0.13 µm. *ST Microelectronic* und die israelische Firma *Ramonchips* arbeiten an der Bereitstellung erster *Flows* basierend auf konventioneller 65nm-Technologie. Die verwendeten Standardzellen schützen die Implementierung mit Hilfe redundanter Schaltungen vor *Single Event Upsets* (SEU) [Red05]. Speicher werden durch *Error Detection and Correction*-Mechanismen (EDAC) oder aktive Speicherkorrektur (*Scrubbing*) vor Ein- oder Mehrbitfehlern geschützt. Auf Grund der hohen Kosten und der eingeschränkten Flexiblität werden für viele Anwendung FPGA-basierte Lösungen bevorzugt. Entsprechende strahlungsfeste Geräte werden durch die Firma *Atmel* angeboten [atm14]. Diese haben jedoch im Vergleich zu konventionellen FPGAs nur sehr geringe Logikkapazität (max. 280k Gates) und wenig Speicher, was ihr Einsatzgebiet einschränkt. Abbildung 1.2 [Eur11] vermittelt einen Überblick über die bisher relativ übersichtliche Landschaft der Basissysteme zum Einsatz

in Weltraum-DPUs (*Data Processing Units*).

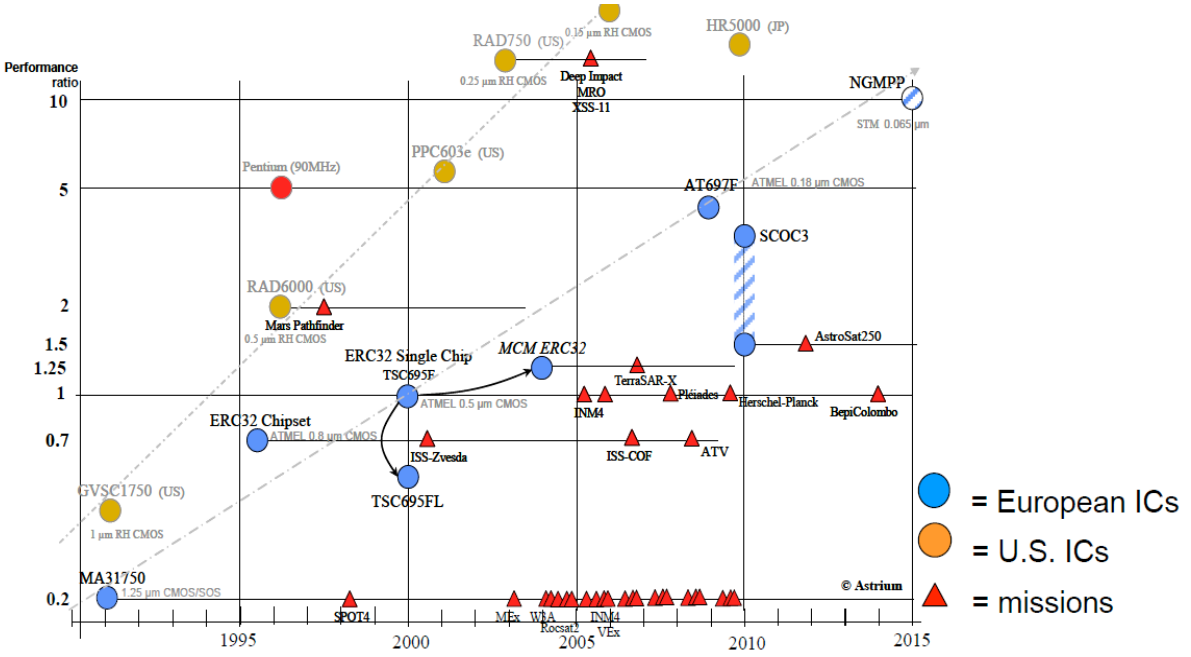


Abbildung 1.2: Basissysteme zum Einsatz in Weltraum-DPUs

Ein in der europäischen Raumfahrt intensiv genutzter Chip ist der *Atmel ATF697F*. Es handelt sich um einen 32-bittigen RISC-Prozessor mit *SPARCv8*-Architektur, der auch als *LEON2* bezeichnet wird. Der *LEON2* ist eine Weiterentwicklung des *ERC32*, der auf der *SPARCv7*-Architektur aufbaut. Die Grafik verdeutlicht, dass der *ATF697F* in etwa die Rechenleistung eines *Intel Pentium* mit 90 MHz liefert. Die momentan eingesetzten Basissysteme bleiben also in Bezug auf ihre Rechenleistung 10-15 Jahre hinter den in anderen Industriebereichen eingesetzten Systemen zurück. Der *LEON2*-Prozessor wird durch die amerikanisch-schwedische Firma *Aeroflex Gaisler* weiterentwickelt [gai13]. Das Nachfolgemodell *LEON3* ist im wesentlichen baugleich, liefert jedoch eine verbesserte Methodik zur Systemkonstruktion. Alle Komponenten wurden in ein *Plug & Play*-System integriert, wodurch sie am Bus automatisch identifiziert und durch Software erkannt werden können. Außerdem wurde mit dem *LEON3MP*-Architekturtemplate die Unterstützung für symmetrische Parallelverarbeitung (SMP - *Symmetric Multi-Processing*) eingeführt. Der Entwurfseinstieg findet jedoch weiterhin auf RT-Ebene statt. Eine weitere inkrementelle Weiterentwicklung stellt der *LEON4* dar. Er bietet im Vergleich zum *LEON3* eine geringfügig höhere Leistung (1.7 DMIPS[1]/MHz vs. 1.4 DMIPS/MHz) durch verschiedene Optimierungen der Mikroarchitektur, wie der Einführung von Sprungvorhersagen (*Branch Prediction*). Da sich mit den aktuell vorhandenen strahlungsfesten *FPGA*- und *ASIC*-Technologien nur Geschwindigkeiten von 50 bis maximal 200 MHz erreichen lassen, ist dies selbst im Multiprozessorbetrieb für moderne Signalverarbeitungsaufgaben, wie zum Beispiel die Kompression hochauflösender multispektraler Bilder, nicht ausreichend. Gegenwärtig verfolgt die europäische Raumfahrt drei Ansätze zur Überwindung dieses Problems:

1. Man versucht, kommerzielle Signalprozessoren (*COTS - Common of the shelf*) durch Abschirmung auf *Board*-Ebene für den Raumfahrtbereich einsetzbar zu machen. In [Tra11] werden dazu Versuche mit einer C6727 DSP von *Texas Instruments* und einem *SHARC IP-Core* von *Analog Devices (AD21469)* beschrieben. Dadurch können Geschwindigkeiten bis zu 1.6 GFLOPS erreicht werden. Die erzielbare Abschirmung bietet jedoch nur Schutz

1 Dhrystone MIPS - Millionen Instruktionen pro Sekunde im Dhrystone Test (hauptsächlich Ganzzahl- und Zeichenkettenoperationen)

gegen geringe Strahlungsdosen und ist somit nur für nicht-kritische Raumfahrtanwendungen einsetzbar.

2. Die existierende LEON-basierte Infrastruktur soll weiter optimiert werden. Dazu treibt die ESA die Entwicklung eines *Next Generation Multi-Processor* (NGMP) voran. Der NGMP wird vier LEON4-Prozessoren in ein SoC integrieren (Abb. 1.3 [And12]). Im NGMP werden bewährte und erprobte Komponenten lediglich neu arrangiert, wodurch Entwicklungskosten eingespart werden können. Allerdings ist die Leistungsfähigkeit beschränkt. Der Entwurf soll in 65nm-Technologie bei *ST-Microelectronic* gefertigt werden und bietet eine Verarbeitungsgeschwindigkeit von 1 GFLOPS bei einer Leistungsaufnahme von 10 Watt.

3. Es werden Forschungsprojekte zur Entwicklung von auf Weltraumanwendungen spezialisierten Signalprozessoren gefördert. Ein Beispiel dafür ist *MacSpace* [mac]. Der *MacSpace*-Prozessor besitzt eine massiv-parallele Architektur bestehend aus 64 lose gekoppelten Verarbeitungseinheiten. Ein erster Prototyp wird für 2016 erwartet. Die Verarbeitungsgeschwindigkeit soll bei 51.2 GOPS bzw. 12.8 GFLOPS liegen. Dies würde die Implementierung komplexer Signalverarbeitungsaufgaben in Software ermöglichen, die bisher auf FPGA oder in dedizierte Hardware ausgelagert werden mussten.

Alle drei vorgestellten Ansätze steigern die Komplexität von Weltraum-SoCs beträchtlich. Wie die Erfahrungen aus anderen Industriebereichen zeigen, ist der Übergang zu abstrakteren Entwurfsmethoden erforderlich, um die Entwicklungskosten einzudämmen und die Überdimensionierung von *Hardware* (*Overdesign*) zu verhindern.

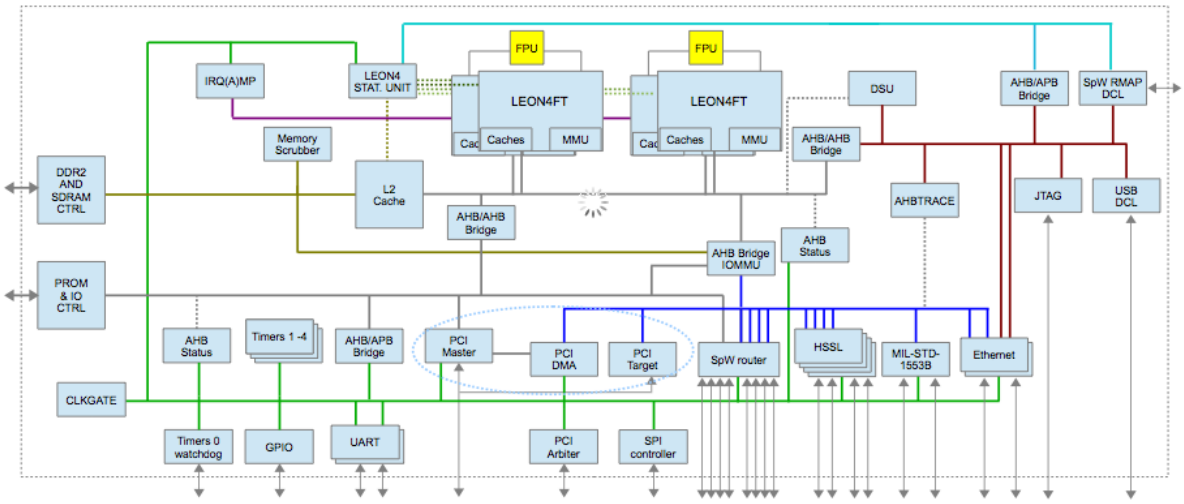


Abbildung 1.3: ESA - Next Generation Multi-Processer (NGMP)

GRLIB

Viele der oben genannten Hardwarekomponenten zum Entwurf von Weltraum-DPUs, wie die LEON-Prozessoren, sowie Verbindungsstrukturen und Peripherie des NGMP, sind in einer gemeinsamen Hardware-Bibliothek zusammengefasst. Die GRLIB wird durch die schwedisch-amerikanische Firma *Aeroflex Gaisler* [gai13] unterhalten und durch die ESA und andere Partner kontinuierlich erweitert. GRLIB enthält heute fast 100 Komponenten, die unter verschiedenen Lizenzmodellen vertrieben werden. Kernkomponenten, wie die Prozessoren und Busse, sind unter GPL- und kommerzieller Lizenz erhältlich. Die fehlertolerante Version des LEON3 ist eine rein kommerzielle IP, genau wie verschiedene Busbrücken und *Controller* (z.B. AHB2AHB, GRCAN). Einzelne IPs, wie der LEON2-Speichercontroller (MCTRL) stehen unter LGPL. Grund dafür sind der Ursprung der Komponenten und deren Entstehungsgeschichte. Eine Beschreibung aller GRLIB-IPs und der zugehörigen Lizenzmodelle kann [Gai10] entnommen

werden. Die wichtigsten zum Aufbau einer funktionsfähigen DPU erforderlichen Komponenten sind in Tabelle 1.1 zusammengefasst.

Name	Funktion
LEON3	SPARC V8 32-bit Prozessor
AHBCTRL	AMBA AHB Bus-Controller mit *Plug & Play*
APBCTRL	AMBA APB Busbrücke mit *Plug & Play*
GPTimer	Mehrzweck-*Timer*
IRQMP	Multi-Prozessor-*Interrupt-Controller*
MCTRL	8/16/32/64-bit PROM/SRAM/SDRAM-*Controller*
SRAM (GENMEM)	SRAM-Simulationsmodell mit SRECORD-Ladefunktion
AHBMEM	RAM mit AHB-Schnittstelle
APBUART	Programmierbarer UART mit APB-Schnittstelle

Tabelle 1.1: GRLIB-Kernkomponenten zur Konstruktion von Weltraum-DPUs

GRLIB ist ein bus-zentriertes System. Alle enthaltenen IP-Komponenten werden mit Hilfe eines *On-Chip*-Busses verbunden und verfügen über die entsprechenden Schnittstellen. Auf Grund seiner weiten Verbreitung werden dafür AMBA 2.0 [Lim11], insbesondere AHB und APB, verwendet. Das System verfügt über verschiedene hilfreiche Features, welche die Konstruktion von SoCs erleichtern. So sind alle Komponenten mit *Plug & Play-Records* ausgerüstet, mit deren Hilfe sie eindeutig am Bus identifiziert werden können. Das System nutzt diese Information außerdem zum automatischen Aufbau eines zentralisieren Adressdekodierers. Darüber hinaus wurden Interruptleitungen in die AHB-Schnittstellen integriert, so das jede Komponenten, ohne zusätzlichen Verdrahtungsaufwand, jeden Interrupt treiben kann. Alle IPs in GRLIB sind in VHDL auf RT-Ebene implementiert. Die Konsistenz und Geschlossenheit des Systems machen es zu einem optimalen Ansatzpunkt zur Entwicklung einer Virtuellen Plattform. Die weite Verbreitung im Europäischen Raumfahrtsektor lässt auf eine zügige Adaption der neuen Methodik durch die Industrie hoffen.

1.4 Forschungsbeitrag

Der primäre Forschungsbeitrag dieser Arbeit besteht in der Entwicklung einer flexiblen, modularen Methodik:

- zum schnellen und kostengünstigen Entwurf abstrakter *Hardware*-Simulationsmodelle,
- zur effizienten Konstruktion, Simulation und Analyse Virtueller Prototypen basierend auf herstellerunabhängiger, offener und standardkonformer Infrastruktur.

Darüber hinaus wurden folgende spezielle Beiträge geleistet:

1. Entwicklung einer Bibliothek von Basisklassen zur Kapselung der Grundfunktionalität von *Hardware*-Simulationsmodellen aufbauend auf neu entworfenen (z.B. *AMBA* 2.0 *TL*-Schnittstellen, Signalkommunikation) und existierenden *Open Source*-Lösungen (z.B. Konfiguration, Speichermodellierung)
2. Entwicklung von *SystemC/TL*-Simulationsmodellen für Kernkomponenten von *Payload*-Prozessoren im Raumfahrtbereich (z.B. LEON2/3 CPU[1], *AHB/APB-Controller*, Speicher-*Controller*).
3. Entwicklung einer flexiblen, erweiterbaren Virtuellen Plattform zur Simulation und Analyse von Multi-Prozessorsystemen basierend auf derselben Komponentenbibliothek (LEON2/3MP).

1 Erweiterung der *Trap*-LEON Intergereinheit um *Caches*, *MMU* und *Scratchpad*-RAMs)

a) Zur Beschleunigung der Architekturexploration kann das System zur Laufzeit vollständig dynamisch rekonfiguriert werden.

b) Die Unterstützung unterschiedlicher Abstraktionsniveaus ermöglicht einen mehrstufigen Explorationsansatz mit unterschiedlichen Abwägungen zwischen Simulationsgeschwindigkeit und -genauigkeit.

Das weiten Teilen dieser Arbeit zugrunde liegende Projekt wurde durch die Europäische Raumfahrtagentur (ESA) angeregt [Fos13] (ICT AO/1-6025/09/NL/JK), um die Effizienz des Systementwurfes für Raumfahrtanwendungen zu steigern. Dies schlägt sich speziell in der Auswahl der modellierten Basiskomponenten und Systeme nieder, schränkt die allgemeine Anwendbarkeit der vorgestellten Konzepte jedoch in keiner Weise ein. Darüber hinaus wurde ein konsequenter *Open Source*-Ansatz verfolgt. Gemeinsam mit der weitest möglichen Unterstützung für existierende Standards und standardoffene Lösungen sollen Akzeptanz und Adaptierbarkeit abstrakter Entwurfsmethoden im Raumfahrtbereich gefördert werden [Sch14]. Langfristiges Ziel ist es, die in *SoCRocket* bereitgestellten Schlüsselkomponenten kontinuierlich zu erweitern und zu ergänzen. Die dadurch entstehende Infrastruktur- und Komponentenbibliothek wird durch die ESA an ihre Lieferanten verteilt, die dadurch die Möglichkeit erhalten Systeme auf höherem Abstraktionsniveau zu entwerfen, Kosten zu senken und qualitativ höherwertige Arbeitsprodukte zu liefern. Auf Basis von *SoCRocket* werden zurzeit im Auftrag der ESA an unserem Lehrstuhl Hardware und hardwarenahe Software zur Implementierung des *CCSDS File Delivery Protocols* entwickelt (ICT AO/1-7150/12/NL/LvH). Des Weiteren entwickelt die Firma *Terma* mit Hilfe von *SoCRocket* ein Modell des ESA-*Next Generation Multi-Processor* (NGMP) [ter14] (ICT AO/1-7150/12/NL/LvH).

Die vorliegende Arbeit wurde durch Erfahrungen ermöglicht die ich in langjähriger erfolgreicher Zusammenarbeit mit der Firma *Coware*[1] sammeln konnte. Dabei entstanden ein Prozessorsimulator für eine *Very Long Instruction Word*-Architektur (VLIW) mit Fokus auf Basisband-Signalverarbeitung [Sch06b][Nov08], ein *Application Specific Instruction-set Processor* (ASIP) für Zeitsynchronisation in Mobilfunkanwendungen [Sch07] [Bou12] und ein Virtueller Prototyp für einen *Software-Defined-Radio*-SoC [Bou06][Ng07][Der10]. Diese Arbeiten entstanden am *Interuniversity Micro-Electronic Center* (IMEC) in Belgien in Kooperation mit *Samsung* und *Toshiba* und kennzeichnen den Beginn meiner Arbeit mit Prof. Dr.-Ing. Mladen Berekovic, den ich daraufhin an die TU-Braunschweig begleitete.

1.5 Organisation der Arbeit

Im Anschluss an diese Einleitung werden in Kapitel 2 allgemeine Grundlagen und Prinzipien des Entwurfes von *System-on-Chips* auf Systemebene erläutert. Darüber hinaus werden die historische Entwicklung, sowie aktuelle Trends und Forschungsarbeiten bezüglich ESLD und Virtuelle Plattformen untersucht. Kapitel 3 erläutert die zum Entwurf einer flexiblen erweiterbaren Virtuellen Plattform erarbeiteten Modellierungskonzepte. Dazu werden verschiedenen Aspekte des Entwurfs von Simulationsmodellen identifiziert, in Hinblick auf den *State-of-the-Art* untersucht und gezielt weiterentwickelt. Die auf dieser Grundlage erstellten Kernkomponenten zum Entwurf robuster eingebetteter Systeme im Raumfahrtbereich werden in Kapitel 4 beschrieben. Kapitel 5 erläutert, wie diese Komponenten mit Hilfe des *SoCRocket*-Entwurfsflusses zur Konstruktion Virtuellen Prototypen verwendet werden können. Darüber hinaus wird der Einsatz des Systems zur Architekturexploration auf unterschiedlichen Abstraktionsebenen und zur Entwicklung von hardwarenaher *Software* anhand praktischer Beispiele untersucht. In Kapitel 6 werden die gewonnenen Erkenntnisse zum Aufbau eines DPU-Prototypen (LEON3MP) auf Systemebene verwendet. Mit Hilfe des Prototypen werden Benchmarks zur Untersuchung von Simulationsgeschwindigkeit und -genauigkeit durchgeführt. Im Anschluss werden die Ergebnisse

1 jetzt zugehörig zu *Synopsys Inc.*

zusammengefasst und mögliche weiterführende Forschungstätigkeiten vorgestellt (Kapitel 7). Installationsanweisungen und eine Übersicht der wichtigsten Systemkommandos befinden sich in Anhang A. Ein Überblick über die im Rahmen der Arbeit entwickelten Simulationsmodelle und Infrastrukturkomponenten kann Anhang B entnommen werden. Anhang C enthält zusätzliche Simulationsergebnisse.

2 Grundlagen

In diesem Kapitel werden der historische Hintergrund und aktuelle Entwicklungen von ESLD und speziell Virtuellen Plattformen basierend auf SystemC/TLM untersucht und näher erläutert. Darüber hinaus werden aktuell praxisrelevante Werkzeuge klassifiziert und vorgestellt. Im Anschluss werden die ESL-Entwurfsmethodik für HW/SW-Co-Design, sowie die Grundprinzipien von *SystemC* und TLM erläutert.

2.1 Historischer Hintergrund

Es existiert eine Vielzahl von Forschungsprojekten, die sich mit der Umsetzung von *ESL*-Design unter verschiedenen Gesichtspunkten beschäftigt haben; einige davon reichen in die 1990er-Jahre zurück. In [Gup93] werden 1993 die Vorteile einer Abwägung von Entwurfsoptionen zur Implementierung von Systemen bestehend aus Prozessoren und anwendungsspezifischen Schaltungen beschrieben. Die Autoren stellen die Sprache *HardwareC* vor, mit deren Hilfe Komponenten auf hohem Abstraktionsniveau spezifiziert und synthetisiert werden können. *HardwareC* bildete ebenfalls die Grundlage für *Olympus* [DM90], eines der ersten Werkzeuge zur Erzeugung von Netzlisten aus einer Hochsprache. Andere frühe *High-Level*-Synthesewerkzeuge waren *Cathedral* von *IMEC* [DM86] oder *Architects Workbench* von der *Carneggie-Mellon University* [Wal89]. Ebenfalls 1993 beleuchtet G. De Micheli die Möglichkeit der Erweiterung von CAD-Werkzeugen zur Unterstützung von sogenanntem *HW/SW-Co-Design* [DM93]. Als Beispiel hierfür wird mit *Ptolemy* [Buc91] eine erste objektorientierte Entwurfsumgebung (C++) zur Entwicklung von Kommunikations- und Signalverarbeitungssystemen angeführt. Hindernisse für die weitere Entwicklung sieht der Autor im Fehlen abstrakter Systemmodelle, konsistenter Modellierungssprachen und Verifikationsmethoden; Probleme, die zum Teil bis heute nicht vollständig überwunden sind. In [Ern93] präsentiert R. Ernst einen software-zentrierten Syntheseansatz zur automatischen Optimierung der Implementierungskosten. Dabei werden ausgehend von einem Einprozessorsystem bei Verletzung von zeitlichen Randbedingungen zusätzliche Hardware-Elemente (z.B. Beschleuniger) erzeugt.

Die genannten frühen Ansätze zur Werkzeugunterstützung auf Systemebene basieren auf relativ einfachen Architekturen mit beschränkter Unterstützung für Parallelverarbeitung oder *Pipelining* von Komponenten. Dies änderte sich in den Folgejahren schnell. In [Bli98] wird ein Syntheseansatz für heterogene Multiprozessorsysteme vorgestellt. Dabei wird eine *Task*-Beschreibung mit Hilfe evolutionärer Algorithmen auf die Architektur abgebildet und der Prozess der Partitionierung als Optimierungsproblem dargestellt. Des weiteren entwickelten sich formale Methoden zur Erleichterung des Entwurfseinstiegs. Ein exzellenter Überblick über die dazu erforderlichen mathematischen Grundlagen findet sich in [Thi00]. In [Hen05] wird darüber hinaus ein Werkzeug zur *Performance*-Analyse auf Systemebene mittels formeller *Scheduling*-Techniken vorgestellt. Der Entwurf zunehmend größerer Systeme auf hohem Abstraktionsniveau und die automatische Synthese einzelner Komponenten erforderte die Entwicklung von Co-Simulationstechniken [Lie97]. Dabei werden Simulationsmodelle, wie z.B. Prozessorsimulatoren, gemeinsam mit *Register-Transfer-Level*-Komponenten simuliert. Die Technik erlaubt die schrittweise Transformation eines Entwurfs von der Systembeschreibung auf die physikalische Ebene, wodurch die Verifikation erheblich erleichtert wird. Zusätzlich können die Simulationszeiten durch die Verschiebung unkritischer Systemteile auf hohes Abstraktionsniveau verringert werden. Zur Optimierung dieses Effekts wird in [Ziv96] die Nutzung von *Compiled Simulation* für Co-Simulation vorgeschlagen. In [VR96] wird als weiteres Problem der Systemsynthese die Heterogenität der Entwicklungswerkzeuge angeführt. Effiziente Entwurfsmethodik wird in der Zukunft nicht nur die Wiederverwendung von

Komponenten, sondern auch die Wiederverwendung von Werkzeugen erfordern. Dazu müssen Wege gefunden werden, Modelle und Werkzeuge von Zulieferern einzubinden. Als Beispiel dafür werden die sich immer stärker durchsetzenden *ARM*-Prozessoren angeführt, die gemeinsam mit einem eigenen Simulator und *Software*-Entwicklungswerkzeugen als *IP*-Block ausgeliefert werden. Die größten Schwierigkeiten werden in der Synthese der Schnittstellen erkannt. Es existiert noch keine einheitliche Darstellung von Kommunikation auf Systemebene. Das von den Autoren vorgestellte *CoWare*-System basiert auf Containern mit abstrakten Schnittstellen (*Encapsulations*). Mit Hilfe der Container können Hardware- und Softwarekomponenten in unterschiedlicher Abstraktion und Darstellung (z.B. *C*, *DFL*, *VHDL*) gekapselt werden. Abbildung 2.1 zeigt einen gekapselten *C*-Prozess auf einem gekapselten Prozessor [VR96].

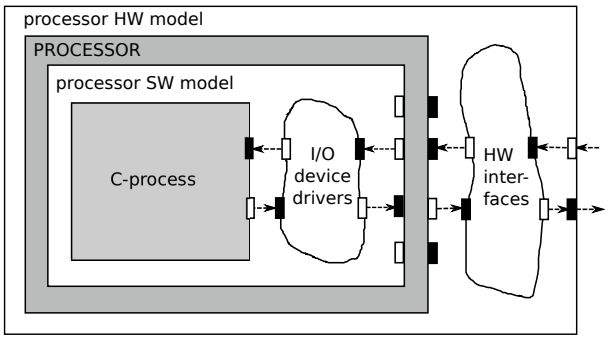


Abbildung 2.1: Gekapselter *C*-Prozess auf gekapseltem Prozessor im *Coware*-System

Die Container sind über sogenannte *Channels* verbunden. Kommunikation wird durch *Remote Procedure Calls (RPC)* realisiert. Für den Entwurf von Komponenten wird eine strikte Trennung von Funktionalität und Kommunikationsverhalten vorgeschlagen. Damit werden wichtige Konzepte der späteren *SystemC*- und TLM-Standards vorweggenommen (siehe Abschnitt 2.5).

Ab Anfang der 2000er-Jahre erlaubte es die Fertigungstechnik über kleinere Multiprozessorsysteme hinaus ganze Netzwerke von Prozessoren, sogenannte *Network on Chips* (NoC), zu fertigen. Dadurch erweiterte sich das vorhandene *Scheduling*-Problem um ein *Routing*-Problem: Je nach Netzwerk-Topologie und Einstellung (*Policy*) benötigt Kommunikation mehr oder weniger Zeit (Netzwerk-*Hops*). Besonders für Echtzeitsysteme ergeben sich daraus schwierige Probleme, die bis heute Gegenstand der Forschung sind [Mar09].

Außerdem erkannte man den Energieverbrauch von Systemen zunehmend als begrenzenden Faktor für die Größe und Leistungsfähigkeit von integrierten Schaltungen. Ein Ansatz zur Lösung dieses Problem war die Einführung von Heterogenität in MPSoCs. Durch die Entwicklung anwendungsspezifischer Prozessoren (ASIPs) versuchte man, für spezielle Signalverarbeitungsaufgaben immer nur genau so viel Flexibilität wie nötig bereitzustellen und dadurch den Energieverbrauch zu senken [Sch06b][Nov08]. Mit der Integration von ASIPs in MPSoCs beschäftigten sich unter anderem die Universität von Bologna und die RWTH Aachen: In [Ang06] wird ein offenes *Framework* zur Integration unterschiedlicher IPs auf Systemebene vorgestellt. In einem großen Anwendungsbeispiel werden MPARM [oB04] und verschiedene LISA-Modelle [Hof02] in einer VP integriert. Weitere ASIP-Werkzeuge werden in Abschnitt 2.3.3 (Prozessorgeneratoren) vorgestellt. Eines der ersten Werkzeuge mit dem primären Ziel der Abschätzung des Energieverbrauches von MPSoCs auf Systemebene war Avalanche [Hen02]. Das System stellt Modelle für Prozessoren, Caches und Speicher bereit, die mit analytischen Funktionen zur Berechnung der Leistung ausgestattet sind. Zur Parametrisierung der Modelle wird das System mit Hilfe von auf Gatterebene gewonnenen Eckdaten annotiert. Dies ist ein sehr aufwendiger Prozess, für den aber bis heute noch kein vollwertiger Ersatz gefunden werden konnte.

Darüber hinaus wurde in den frühen 2000er-Jahren die Grundlage für viele heute kommerziell oder als *Open Source*-Projekt verfügbare ESL-Werkzeuge gelegt. Beispiele dafür sind *Synopsys Platform Architect* (früher *Coware Platform Designer*) [Inc14d], *SoCLib* [soc13] oder die freie IP-Plattform *OpenCores* [ope14]. Auf diese und weitere heute praktisch relevante Werkzeuge

wird in Abschnitt 2.3 näher eingegangen.

2.2 Aktuelle Entwicklungen

Die jüngste Entwicklung ist dadurch gekennzeichnet, dass eine zunehmende Anzahl an eingebetteten Systemen Verbindung zum Internet herstellt (Internet of Things [Bar13]). Dadurch kommt es zu neuen *Workloads* und neuen Mischungen aus harten und weichen Echtzeitanforderungen. Systeme werden sich in der Zukunft zunehmend selbst vernetzen, selbst organisieren und ihrer Umgebung anpassen [Som14]. Weitere aktuelle Herausforderungen sind die aktive Erkennung und Korrektur von Fehlern, das Austauschen und Hinzufügen von Komponenten zur Laufzeit oder die Entwicklung von Garantien bezüglich Sicherheit und Zuverlässigkeit. Zu dieser Ansicht kommt auch David Fuller in seiner *Keynote* zur DATE-Konferenz 2014 [Ful14]. Er erläutert, dass die Anforderung an die Rechenleistung eingebetteter System weiter dramatisch steigen wird. Die Erwartungen der Konsumenten und die *Roadmaps* der Industrie zeichnen ein Bild einer elektronisch verbundenen Welt intelligenter Geräte. Diese Vision kann nicht mehr durch einen einzigen monolithischen Entwurfsprozess realisiert werden. Es gilt, neue Entwurfsmethoden zu entwickeln und umzusetzen. ESLD kann dabei eine entscheidende Rolle spielen. Ein großes Problem bezüglich des Zusammenwachsens von Systemen ist deren Heterogenität in Bezug auf Entwurf und Schnittstellen. Für analoge und digitale Komponenten, Mikro-Sensoren, Aktoren und MEMS wurden bislang unterschiedliche Ansätze verfolgt.

Um diese neuen Herausforderungen bewältigen zu können, müssen verschiedene Probleme gelöst werden. Dazu gehören Entwurfsmethoden, die Modelle unterschiedlicher Hersteller, Abstraktionen und Sprachen auf Metaebene integrieren können. Außerdem muss untersucht werden, wie das in heutigen VPs realisierte Konzept der Erkundung des Entwurfsraumes über die Grenzen von SoCs hinaus erweitert werden kann. Eines der größte aktuellen Hindernisse sind darüber hinaus die sequentielle Natur von *SystemC* und die Modellierung des Energieverbrauchs von Schaltungen auf Systemebene. Darüber hinaus muss es gelingen, ESLD auf neue Einsatzgebiete zu übertragen, um eine weitere Akzeptanz der neuen Methodik herzustellen. In den folgenden Abschnitten werden aktuelle Forschungsarbeiten auf diesen Gebieten zusammengefasst.

Entwurfsmethoden

Wissenschaftler um Franco Fummi von der Universität von Verone und *Agilent Technologies* arbeiten an der Integration heterogener Komponenten in ein gemeinsames *Framework* [Fum14]. Ziel ist es, sich von klassischen Co-Simulationsansätzen zu entfernen und eine auf C++/*SystemC* basierende homogene Umgebung zu schaffen. Ein ähnlicher Ansatz zur Vereinigung unterschiedlicher *Models of Computation* (MoC) wurde bereits mit *Ptolemy* an der Universität von Berkeley verfolgt [Buc91] (siehe 2.3).

Das beschriebene Heterogenitätsproblem besteht nicht nur in der Systemsimulation sondern auch in der Systemsynthese. Dies wurde auch durch die Industrie erkannt. Wolfgang Ecker von *Infineon Technologies* beschreibt in [Eck14] ein Werkzeug zur Beschreibung von Systemen auf Metaebene. Mit Hilfe von *metagen* können verschiedene Code-Generatoren, mit unterschiedlichen Eingabe- und Ausgabesprachen integriert werden. Dem Entwickler wird eine einheitliche Nutzerschnittstelle präsentiert, mit der heterogene Systemteile gemeinsam synthetisiert werden können. Dadurch lassen sich einzelne Entwurfsschritte um bis zu Faktor 20 beschleunigen. Für den kompletten Entwurf eines Chips (Spezifikation bis *Tapeout*) verkürzt sich die Entwicklungszeit dadurch auf ein Drittel.

In Zukunft werden sich die uns bekannten elektronischen Systeme mehr und mehr über Chipgrenzen hinaus ausdehnen und mit Hilfe von Sensoren und Aktoren direkt mit der Umwelt interagieren. Derartige Systeme werden auch als Cyber-Physische Systeme (CPS) bezeichnet. In [Mue12] beschreiben Wolfgang Müller von der Universität Paderborn und Anthony Di Pasquale von der *Northwestern University* in Boston die Erweiterung einer Virtuelle Plattform für diesen sich vergrößernden Blickwinkel. Als Beispiel dient ein zweirädriges elektrisches Gefährt, dass

durch einen ARM-Prozessor gesteuert wird. Beschleunigungssensor, Gyroskop, Servomotor und Abstandsmesser sind über einen CAN-Bus angebunden. Das Prozessorsubsystem wird mit Hilfe von QEMU und *SystemC* simuliert. Zur Simulation von Analogkomponenten wird *SystemC*-AMS eingesetzt. Physikalische Effekte werden mit *Open Dynamics Engine* (ODE) emuliert. Die Ergebnisse zeigen einen 40000-fachen Geschwindigkeitsgewinn im Vergleich zu einem Logik-Analysator. Trotzdem ist das System noch zu langsam für Echtzeituntersuchungen.

Eine Lücke in der aktuellen Entwurfsmethodik besteht hinsichtlich Systemsimulationen mit mittlerer Abstraktion. Der überwiegende Teil der Forschungsarbeiten beruht auf losem *Timing*, für Softwareentwicklung und schnelle Exploration, oder zyklengenauem *Timing* für Verifikation und Validierung. Für Architekturexploration und *Performance*-Modellierung schlägt der TLM2.0-Standard einen näherungsweise akkuraten Modellierungsstil vor. Dieser wurde bisher, vermutlich aufgrund seiner höheren Komplexität, kaum adaptiert. Die Notwendigkeit mittlere Abstraktionsstufen in Zukunft mehr einzubeziehen, wird unter anderem durch Sascha Roloff von der Universität Erlangen-Nürnberg motiviert [Rol12]. Dabei wird beschrieben, dass zukünftige Systeme hunderte an Prozessoren enthalten können. Diese müssen unter Einbeziehung ihres dynamischen Verhaltens simuliert werden. Nur so sind eine sinnvolle Dimensionierung der Hardware zur Erreichung der *Performance*-Ziele und die Verifikation der Software erreichbar. Abstrakte lose gekoppelte Systeme sind dafür zu ungenau. Zyklengenaue Modelle hingegen können die benötigte Simulationsgeschwindigkeit nicht liefern.

Ein großes Problem für die Akzeptanz der ESL-Methodik sind ungeklärte Fragen bezüglich der Verifikation. Damit befassen sich unter anderem Wissenschaftler des *TIMA Laboratories* in Frankreich. In [Pie13] gibt Laurence Pierre einen Überblick über moderne *Assertion-Based*-Verifikationstechniken. Darüber hinaus wird eine Methodik vorgestellt, mit der *Assertions* und Systemanforderungen aus der frühen Entwurfsphase, von *SystemC*/TLM, auf die RT-Ebene übersetzt werden können. Die Entwicklung einer derartigen Lösung würde die Konsistenz im Entwurfsfluss beim Übergang zu ESLD erhöhen. Bisher existiert leider keine Implementierung, welche die entsprechenden Transformationen automatisch umsetzen kann.

Forscher der Universität Bremen beschäftigen sich mit der Fehlerlokation in TLM-Designs. In [Le13] wird ein Mechanismus zur Analyse von Fehlerspektren in *SystemC* beschrieben. Ein Spektrum ist dabei ein Abbild von Ereignissen und Zuständen, die während der Ausführung eines Tests innerhalb des Entwurfs berührt werden. Dieses Abbild wird im Anschluss der Simulation mit einem Erwartungswert verglichen. Dadurch kann eine Aussage über die Wahrscheinlichkeit von Fehlern in bestimmten Systemkomponenten getroffen werden. Die Methodik wird an Beispieldesigns aus der TLM2.0-Bibliothek erprobt. Die betrachteten Komponenten sind vergleichsweise einfach und lassen daher keine finale Schlussfolgerung zu. Trotzdem erscheint der Ansatz sehr vielversprechend und könnte die Verifikation großer VPs in Zukunft sehr erleichtern.

Mit dem Einsatz von Virtuellen Plattformen zur Verifikation von Chipentwurf beschäftigen sich ebenfalls Wissenschaftler der *Portland State University*. In [Lei13] beschreiben Forscher um Li Lei, wie VPs zur Validierung von *Post-Silicon*-Designs eingesetzt werden können. Der Ansatz beruht auf dem Abgleich der Schnittstellenzustände zwischen Chip und VP. Jeder Zustandsübergang im abstrakten Modell muss einen äquivalenten Übergang im Hardwaredesign hervorrufen. Die Methodik wurde an verschiedenen Netzwerkadaptern erprobt. Dabei konnten verschiedene Arten von Fehlern schnell identifiziert werden.

Erkundung des Entwurfsraumes

Eines der primären Ziele von Virtuellen Plattformen ist die Erkundung des Entwurfsraumes (*Design-Space Exploration*, DSE) im frühen Entwurfsstadium. Da eine vollständige Durchsuchung aller möglichen Systeme zur Identifizierung der optimalen Lösung allein aufgrund der Vielzahl von Einstellungen und Parametern in der Regel nicht möglich ist, muss der Suchraum intelligent eingeschränkt werden. Sehr weit verbreitet ist der Einsatz von Heuristiken basierend auf *Simulated Annealing* [Tal06]. Taghavi, Pimentel und Thompson stellen in [Tag09] eine auf der Auswertung einer Baumstruktur basierende Lösung vor. Durch den Ausschluss von Optimierungsgruppen

(Ästen) wird der Suchraum effizient eingeschränkt. Ebenfalls denkbar ist die Verwendung von Techniken zum Maschinenlernen. Ein Beispiel zum Einsatz von neuronalen Netzen zur Steuerung von Entwurfsraumerkundungen kann [Ozi08] entnommen werden.

Mit TuneableVP wird in [Lin08] eine DSE-Analyseplattform für ARM-basierte Systeme vorgestellt. Das System stellt dazu *Wrapper* für unterschiedliche Modellklassen, wie *Programmers View*, *Timed Programmers View*, *zyklengenaue Abstraktion* und *Traces* bereit. Die Modelle erhalten dadurch eine gemeinsame Analyseschnittstelle, welche die Auswertung von Simulationen erheblich erleichtert. So können mit Hilfe einer graphischen Oberfläche *Timing* und Energieverbrauch, aber auch Komponentenaktivität und Busauslastung berechnet und angezeigt werden.

In [Uba14] wird durch Wissenschaftler der *Northeastern University Boston* mit *Multi2Sim* eine VP vorgestellt, mit der die Lastverteilung zwischen CPUs und GPUs untersucht werden kann. Das System stellt eigens dafür entwickelte Modelle bereit und ermöglicht die Einbindung externer IPs (z.B. OVP). Diese werden dazu in *SystemC-Wrapper* eingehüllt. Die Besonderheit ist ein Injektionsmechanismus, der das Einfügen von Fehlern in GPU-Modelle erlaubt. Dadurch kann, neben Zeitverhalten und Implementierungskosten (z.B. Fläche, Energie), auch die Zuverlässigkeit des Systems in die Erkundung einbezogen werden.

Ein gelungener Ansatz zur Übertragung plattformbasierten Designs auf neue Anwendungsbereiche wird in [Gra14] durch die *Universität Erlangen-Nürnberg* und die *Audi AG Ingolstadt* vorgestellt. Dabei wurden für SoCs und MPSoCs gängige Methoden der Entwurfsraumerkundung auf *Embedded Control Units* (ECU) im Automobilbereich übertragen. Die Autoren führen eine zusätzliche Variantenebene in die Exploration ein, welcher durch die Optimierung über verschiedene ECU-Typen hinweg eine sehr weit gefasste Untersuchung ermöglicht.

Trotz Abstraktion sind die Größe des Entwurfsraumes und damit die Anzahl der Freiheitsgrade beim Entwurf eines eingebetteten Systems durch die Geschwindigkeit der verfügbaren Simulationssysteme beschränkt. Ein Ansatz zur Generierung sehr großer MPSoCs mit bis zu 400 Verarbeitungselementen wird in [Cas14] durch Forscher der Universität *Sante Cruz do Sul* in Brasilien vorgestellt. Dabei werden Anwendungen als *Task*-Graphen modelliert und einem festen Architekturtemplate dynamisch zugewiesen. Allokation von Ressourcen und Interprozesskommunikation werden durch spezielle Managementeinheiten verwaltet. Der Ansatz erscheint sehr vielversprechend, muss aber den Nachweis seiner Leistungsfähigkeit noch erbringen.

Die Erkundung des Entwurfsraumes großer MPSoCs mit dem Schwerpunkt der Abschätzung des Energieverbrauches stand ebenfalls im Mittelpunkt des EU-Projektes (FP7) COMPLEX. Die Ergebnisse des Konsortiums unter der Leitung des *Institute for Information Technology Oldenburg* (OFFIS) werden in [Gru12] zusammengefasst. Das Ergebnis ist eine Entwurfsmethodik, die Techniken zur Optimierung des Energieverbrauches mit *Virtual Prototyping* verknüpft. Die zu untersuchenden VPs werden aus UML und einer funktionalen C/C++ Beschreibung generiert. Zur Modellierung von Prozessoren kommen unter anderem *ARMulator* und *SimpleScalar* zum Einsatz. Während der Plattformsimulation werden *Traces* zur Darstellung des Energieverbrauches über der Zeit generiert. Diese dienen als Eingabe für den Optimierungsmechanismus. Die DSE kann automatisch und halb-automatisch gesteuert werden. Leider werden die zugrundeliegenden Konzepte nicht näher erläutert. Aus den Ausführungen geht hervor, dass Erfahrungswerte des Entwicklers eine entscheidende Rolle spielen.

Parallelisierung und Beschleunigung

Weitere aktuelle Entwicklungen beschäftigen sich mit der Beschleunigung und Parallelisierung von Simulationen auf Systemebene. Konventionelle Simulatoren wie der OSCI-*SystemC*-Referenzsimulator verwenden nur einen *Thread* und nutzen moderne *Multi-Core-Workstations* damit nur ungenügend aus. Forscher der RWTH Aachen um Jan Hendrik Weinstock und Rainer Leupers stellen in [Wei14] mit *SCope* einen parallelen *SystemC*-Simulator vor, bei dem die Komponenten eines MPSoCs auf die verschiedenen Kerne eines *Multi-Core*-Systems verteilt werden. Zur Beschränkung des Synchronisationsaufwands wird ein *Lookahead*-Mechanismus eingesetzt, der den Kommunikationsbedarf der Kerne bündelt. Der Simulator wurde an einer im

Rahmen eines EU-Projektes entwickelten Virtuellen Plattform (FP7 EUROTILE) erprobt und erzielte einen Geschwindigkeitsgewinn von > 4.

Mit der Reduktion des Synchronisationsaufwandes in parallelen *SystemC*-Simulationen beschäftigt sich auch Dukyoung Yun von der *Seoul National University* in Korea [Yun12]. Im beschriebenen Ansatz werden die Komponenten eines MPSoCs auf die Kerne eines Mehrprozessorsystems verteilt. Jede Komponente wird mit einem Simulationscache ausgerüstet, der Zugriffe auf das *Interconnect* puffert. Dadurch wird ein großer Teil der Speicherzugriffe unterdrückt und im Vergleich zur ungepufferten Parallelsimulation ein Geschwindigkeitsgewinn von 3.3 erreicht. Es steht zu erwarten, dass sich diese Methodik negativ auf die Simulationsgenauigkeit auswirkt. Dazu konnten bisher jedoch noch keine Angaben gemacht werden.

In [Dom12] untersucht Rainer Dörner von der *University of California Irvine* verschiedene Ansätze zur ereignisbasierten Simulation von Modellierungssprachen auf Systemebene. Unter anderem wird das Konzept der diskreten parallelen Ereignissimulation (PDES) erläutert. *SystemC* garantiert dem Nutzer die ununterbrochene Ausführung von *Tasks* durch kooperatives *Multi-Threading*. Dies wird als Hauptproblem auf dem Weg zu einem tatsächlich parallel arbeitenden Simulator erkannt. Ein Ausweg ist die Ausnutzung der expliziten Parallelität in Virtuellen Plattformen. In der Arbeit wird die Auftrennung von Systemen an TLM-Schnittstellen untersucht. Bei Experimenten mit einem H.264 Dekodierer konnte ein Geschwindigkeitsgewinn von Faktor 2 erzielt werden. Das Konzept der *Out-of-order*-Verarbeitung von Ereignissen zur Beschleunigung von VP-Simulationen wurde durch die selbe Gruppe in [Che13] und [Che12] vorgestellt.

Ein anderer Ansatz zur Beschleunigung von *SystemC* wird am *Verimag*-Institut in Grenoble verfolgt. In [Moy13] erläutert Matthieu Moy ein Konzept für TL-Modelle mit losem *Timing* (LT). Dabei werden *SystemC-Threads* für eine vorbestimmte Zeit Prozessen des Betriebssystems fest zugewiesen. In diesem Zeitraum werden sie unabhängig und entkoppelt voneinander parallel ausgeführt. Die Autoren stellen mit *sc-duration* eine entsprechende Ergänzungsbibliothek für *SystemC* bereit. Der präsentierte Ansatz erscheint sehr vielversprechend, ist aber für Systeme mit höheren Genauigkeitsanforderungen (z.B. AT oder CT) ungeeignet.

Die von *Verimag* vorgeschlagene Lösung ähnelt sehr der am *LIP6*-Labor in Paris entwickelten Bibliothek TLM-DT [Mel10]. Im Gegensatz zu *sc-duration* verwendet TLM-DT jedoch nicht die globale *SystemC*-Zeit, sondern hält für jeden parallelen *Thread* eine unabhängige lokale Zeit vor. Entsprechend besten Wissens des Autors ist TLM-DT heute der einzige Ansatz zur Parallelisierung von *SystemC*, der in einer nicht nur akademisch relevanten VP erprobt wurde (siehe SoCLib, Kapitel 2.3).

Die Forscher um Rohit Sinha von der *University of Waterloo* in Kanada nutzen zur Beschleunigung von *SystemC*-Simulationen zusätzlich GPUs [Sin12]. Der vorgestellte Entwurfsfluss erfordert einen modifizierten *SystemC*-Kernel, der die parallele Ausführung von *Tasks* erlaubt. In einem Analyseschritt werden kontrollintensive *Tasks* CPUs und rechenintensive *Tasks* GPUs zugeordnet. GPU-*Tasks* werden von *SystemC* nach *CUDA* übersetzt und zur Kommunikation mit dem Restsystem in einen *SystemC-Wrapper* eingehüllt. Zur Erprobung wurde ein Virtueller Prototyp für eine *Set-Top*-Box erstellt. Bei der Verarbeitung von drei MPEG-Videoströmen mit 12 Prozessorkernen und einer GPU wurde im Vergleich zur sequentiellen *SystemC*-Simulation eine Beschleunigung von 12.6 gemessen.

Parallelisierung ist nicht die einzige Antwort auf den Geschwindigkeitsbedarf bei der Simulation von VPs. Besonders für hochabstrakte Modelle hat sich die Anwendung von Zeitlicher Entkopplung (*Temporal Decoupling*) bewährt [Ayn09]. Dabei wird einzelnen IPs erlaubt, der globalen Simulationszeit um ein festgelegtes Quantum vorauszueilen. Dadurch können jedoch Ressourcenkonflikte nicht in jedem Fall zeitlich korrekt dargestellt werden und es kommt zu zeitlichen Abweichungen. Mit diesem Problem beschäftigen sich unter anderem Wissenschaftler der Technischen Universität München. In [Lu13] wird ein Algorithmus zur Bestimmung der Ressourcennutzung von TLMs vorgestellt, mit dessen Hilfe der durch zeitliche Entkopplung eingeführte Fehler korrigiert werden kann. Außerdem wird die Bedeutung zeitlicher Entkopplung für die Simulationsgeschwindigkeit aufgezeigt. Diese kann, je nach Partitionierung des Systems, bis zu einem Faktor 19 betragen. Der Korrekturmechanismus benötigt keine zusätzlichen

Kontextwechsel und beeinträchtigt die Geschwindigkeit damit kaum.

Mit der Erweiterung des Konzeptes der zeitlichen Entkopplung beschäftigen sich ebenfalls Wissenschaftlicher von *CEA-Leti*, *Minatec*, *ST-Microelectronic* und *Verimag* in Frankreich. Die Gruppe stellt in [Hel13] eine Methodik vor, mit der *Temporary Decoupling* über Speicheradressierung hinaus auf FIFO-basierte Kommunikation übertragen werden kann. Es wird eine spezielle Implementierung für *SystemC*-FIFOs vorgeschlagen, bei welcher der Füllstand abhängig von einem lokalen Quantum berechnet wird. Dadurch ist es nicht mehr erforderlich, nach jedem Lese- oder Schreibzugriff die globale Systemzeit zu synchronisieren. In den in der Veröffentlichung beschriebenen Tests konnte damit ein durchschnittlicher Geschwindigkeitsgewinn von 42.3 % erzielt werden.

Jens Gladigau von der Universität Nürnberg-Erlangen schlägt den Einsatz von *Traces* zur Beschleunigung von VP-Simulationen vor [Gla12]. Dabei wird davon ausgegangen, dass die Ausführungsreihenfolge von SystemC-*Threads* in vielen Fällen a priori bekannt ist. Daher müssen diese nicht aufwendig ereignisbasiert simuliert werden. Das Verfahren wird in der Arbeit mit Hilfe eines Netzwerksimulators demonstriert. Die Ersetzung der *Task*-Umschaltungen durch sequentiell abgearbeitete *Traces* lieferte im Experiment eine Beschleunigung von 30 %.

Modellierung des Energieverbrauchs

Mit zunehmender Hochintegration und steigenden Taktraten gewinnt die frühzeitige Abschätzung des Energieverbrauchs von SoCs an Bedeutung. Drei aktuell verfolgte Ansätze werden durch Bernhard Fischer von *Siemens AG Österreich* in [Fis14] verglichen: ein konventioneller tabellenbasierter Ansatz (A), durch eine funktionale *SystemC*-Simulation stimuliere Energiemodelle (B) und ein durch Anwendungsszenarios stimuliertes reines Energiemodell (C). Varianten B und C sind deutlich genauer als die manuelle Addition von Energiezuständen (A). Außerdem ermöglichen sie eine Vorhersage des Energieverbrauches über der Zeitachse. Ansatz B benötigt für eine Simulation eine Größenordnung mehr Zeit als Ansatz C, liefert dafür allerdings höhere Flexibilität, da ohne die Ableitung von Anwendungsszenarien direkt funktional simuliert werden kann.

Es existieren diverse weitere akademische Ansätze zur Lösung dieses Problems, von denen sich bis heute aber noch keiner durchsetzen konnte. Eine der ersten ganzheitlichen Methoden für TL-Systeme, die über den Prozessor hinaus Verbindungsstrukturen und Peripherie einbezieht, wurde in Zusammenarbeit von *IBM* und der *Pennsylvania State University* entwickelt [Dha05]. Als Hauptproblem identifizierte man die Charakterisierung der Modelle, insbesondere die Zuordnung von Energiewerten zur funktionalen Beschreibung des Systems. Als Beispiel wurde eine *PowerPC/CoreConnect*-Plattform eingesetzt. Die in der Charakterisierung gewonnenen Energiewerte werden manuell in einer Baumstruktur organisiert und können aus den *SystemC*-Modellen heraus adressiert werden. Der präsentierte Ansatz kann für unterschiedliche Granularitätsstufen eingesetzt werden. In den Experimenten wurde eine sehr hohe Genauigkeit ermittelt (ca. 3% Abweichung). Leider sind die verwendeten Anwendungsszenarien zu klein (Einzeltransaktionen) und damit nicht aussagekräftig.

Tayeb Bouhadiba von *IMAG* in Grenoble [Bou13] beschäftigt sich mit der Validierung von Energiemodellen in der frühen Entwurfsphase. Es wird eine Methode zur Annotation des Energieverbrauchs und der Temperaturentwicklung für Virtuelle Plattformen vorgeschlagen. Dabei werden *SystemC*/TL-Modelle mit Automaten zur Modellierung diskreter Leistungszustände ausgerüstet. Durch Beobachtung, welche Komponente sich wie lang in welchem Zustand befindet, lassen sich Energieprofile für das Gesamtsystem ableiten. Darüber hinaus kann mit Hilfe eines *Thermal Solvers* die Entwicklung der Temperatur abgeschätzt werden. Zur Erprobung des Ansatzes wird eine VP bestehend aus einem *Microblaze*-Simulator und entsprechender Peripherie verwendet. Der Mehraufwand für eine feingranulare Schätzung von Energie und Temperatur ist mit einem Faktor 10 im Vergleich zur TLM-Simulation mit LT-Abstraktion sehr hoch. Durch die Wahl größerer Untersuchungsintervalle kann der *Overhead* auf 1.25 reduziert werden. Die Genauigkeit dieses Verfahrens wurde bisher jedoch nicht nachgewiesen.

H. Lebreton von *CEA-Leti* beschäftigt sich mit der Integration von *Low-Power*-Techniken wie *DVFS* in die Energieabschätzung auf Systemebene [Leb08]. Dazu werden während der Simulation für jede involvierte Komponenten *Power*-Profile generiert, die den entsprechenden Zustand zu jedem beliebigen Zeitpunkt beschreiben. Die Profile können dann verwendet werden, um die Abschätzungen für den Normalbetrieb spezifisch anzupassen. Der Ansatz wurde anhand eines Signalverarbeitungsalgorithmus auf einem NoC verifiziert. Der beobachtete Fehler liegt bei 7 %.

Zu den neuesten Arbeiten auf dem Gebiet gehört *PETS* [Ret14] von *Barcelona Supercomputing Center*. In *PETS* werden generische funktionale *Power*-Modelle zur Steuerung des *Software-Mappings* (*Load Balancing*) in Multi-Prozessorarchitekturen verwendet. Dabei wird mit Hilfe einer VP eine durch Energiewerte gesteuerte DSE durchgeführt. Die generischen *Power*-Modelle können für spezifische Implementierungstechnologien parametrisiert werden. Der durchschnittliche Schätzungsfehler wird mit 4% angegeben. Leider ist anhand der Veröffentlichung nicht nachvollziehbar, wie dieser Wert ermittelt wurde.

Anwendungen

Wie bereits im Vorfeld erwähnt ist die Anwendung und Akzeptanz von Virtuellen Prototypen noch stark begrenzt. ESLD hat sich jedoch im Mobilfunksektor durchgesetzt. Besonders hier wurden verschiedene praktische Umsetzungen ausführlich publiziert. In [Sha14] beschreibt Tang Shan, vom Labor für Mobilkommunikation in Peking, seine Erfahrungen bei der Entwicklung eines MPSoCs für 4G-Basisstationen. Zur Konstruktion des Virtuellen Prototypen wurden *SystemC*, *LISA* und *Synopsys Platform Architect* eingesetzt (siehe 2.3). Die Autoren geben an, dass sich durch den Einsatz von ESLD die zur Optimierung des Entwurfs erforderliche Zeit von Monaten auf Tage reduzierte. Als wichtigste Vorteile werden die quantitative Analyse im frühen Entwurfsstadium und der zeitige Einstieg in die Softwareentwicklung hervorgehoben.

Verschiedene Bemühungen, ESLD im Automobilsektor zu etablieren, waren – ähnlich wie im Raumfahrtbereich – bisher wenig erfolgreich. Die Gründe dafür werden unter anderem in [Tha14] durch Manfred Thanner von *Freescale Deutschland GmbH* beschrieben. Hauptprobleme sind fehlende Simulationsmodelle und Beschränkungen hinsichtlich deren Einsetzbarkeit. So werden oft Modelle für einen Anwendungsfall speziell entworfen, können dann aber nicht wiederverwendet werden. Auch die Abstimmung von Modellen verschiedener Anbieter, insbesondere die Formulierung und das gemeinsame Verständnis von Anforderungen, bereiten Probleme.

Der potentielle Einsatz von ESLD und speziell VPs im Automobilbereich wird ebenfalls [Oet14] untersucht. Ein großes Konsortium bestehend aus den Universitäten Paderborn, Tübingen, München, Bremen und den Firmen *Bosch*, *Siemens* und *Infineon* kommt hier zur Schlussfolgerung, dass VPs in Kombination mit *SystemC* und der *Unified Verification Methodology* (UVM) eine ideale Grundlage für Fehlereffektsimulationen und die Identifikation von Schwachstellen in der frühen Entwurfsphase darstellen.

Dem Trend, ESLD-Methoden auf neue Anwendungsfelder zu übertragen, folgen auch Lorenzo Zuolo von der Universität Ferrara und Marco Indaco von der Universität Turin. Gemeinsam wurde dort eine Virtuelle Plattform zur feingranularen Erkundung von *Solid State Drives* entwickelt [Zuo14]. *SSDExplorer* ist eine spezialisierte Simulationsumgebung, die um ein ARM7 RTL-Prozessormodell konstruiert wurde. NAND-Flash und Schnittstellen wurden zyklengenau mit *SystemC* modelliert. Zur Darstellung des Zeitverhaltens der internen Logik werden abstrakte parametrisierbare Verzögerungsmodelle verwendet.

Sven Goossens von der *University of Technology Eindhoven* verwendete Virtuelle Plattformen zum Entwurf rekonfigurierbarer Speichercontroller für Systeme mit gemischter Kritikalität [Goo13]. Der entworfene *Controller* verfügt über einen rekonfigurierbaren Kommandoscheduler, der mit Hilfe eines speziellen Protokolls Speicherkommandos in vorhersagbarer Weise zuordnet und weiterverarbeitet. Der Entwurf wurde zunächst mit *SystemC*/TLM vorgenommen. Dadurch wurde die Herausarbeitung der Systemstruktur erleichtert. Als weiterer Vorteil des Einsatzes von ESLD wird die hohe Sichtbarkeit auf das System hervorgehoben. Dadurch konnten Fehler schneller aufgefunden und beseitigt werden.

Die *Cheng Kung University Tainan* in Taiwan stellte kürzlich mit NetVP eine auf die Entwicklung von Netzwerkbeschleunigern spezialisierte Virtuelle Plattform vor [Wan12]. Dazu setzt das System mit Hilfe eines vLAN-*Layers* auf der Linux-Netzwerkschnittstelle auf. Somit können virtuelle Komponenten auf einfache und natürliche Weise im Zusammenspiel mit realer Hardware verifiziert werden.

Zhimiao Chen von der RWTH Aachen beschäftigt sich mit der Integration analoger Komponenten in Virtuelle Plattformen. Da RF-Schaltungen nicht mit herkömmlichen ereignisbasierten Simulatoren simuliert werden können, müssen dafür geeignete Abstraktionen gefunden werden. In [Che14] wird eine Methode vorgestellt mit welcher der Zustand einer *Schematic*-Datenbank in ein *SystemC*-Modell überführt werden kann. Das Modell dient dann in der VP-Simulation als Platzhalter für Analogkomponenten, was die funktionale Verifikation des Gesamtsystems erheblich erleichtert und beschleunigt.

2.3 ESL-Werkzeuge

Werkzeuge im ESL-Bereich sind Hilfsmittel zur Erzeugung und zur Förderung des Verständnisses von Systemen. Der Werkzeugbegriff schließt sowohl Synthese-, Explorations- und Analysewerkzeuge, als auch Simulationsmodelle ein. In diesem Abschnitt werden verschiedene Ansätze zur Klassifikation vorgestellt. Auf dieser Grundlage werden im Anschluss aktuell praxisrelevante Werkzeuge gruppiert und im Detail beschrieben.

2.3.1 Klassifikation

In den vergangenen Jahren entstand eine Vielzahl von Werkzeugen zur Unterstützung der ESL-Methodik mit unterschiedlichem Anwendungszweck und unterschiedlicher Position im Entwurfsprozess. Die optimale Nutzung dieser Werkzeuge erfordert eine geeignete Klassifikation.

Erste Ansätze hierzu wurden Mitte der 1990er-Jahre durch die *Terminology Working Group* des *Rapid prototyping of Application Specific Processors* Programmes erarbeitet [Gro00]. Das Ergebnis war eine Modell-Taxonomie zur Unterscheidung zwischen Verhaltensmodellen (*Behavioral*), funktionalen Modellen (*Functional*) und strukturellen Modellen (*Structural*). Verhaltensmodelle beschreiben die Funktion und das Verhalten unter Abstraktion einer spezifischen Implementierung. Funktionale Modelle sind auf die reine Funktion eines Systems oder einer Komponente ausgerichtet und ignorieren dazu deren Zeitverhalten. Strukturelle Modelle repräsentieren ein System im Sinne seiner Hierarchie und Verbindungsstrukturen. Der Grad der Abstraktion wurde mit Hilfe von Koordinatenachsen dargestellt. Diese Darstellungsweise hat sich in abgewandelter Form bis heute erhalten. [Bai07] beschreibt dazu sehr anschaulich ein System aus vier Achsen: zeitliche Genauigkeit (Temporal), Datengranularität (Data), funktionale Genauigkeit (Functional) und strukturelle Details (Structural). Abbildung 2.2 demonstriert eine derartige Klassifikation am Beispiel der im Rahmen dieser Arbeit entwickelten SoC-Komponenten (Abschnitt 4).

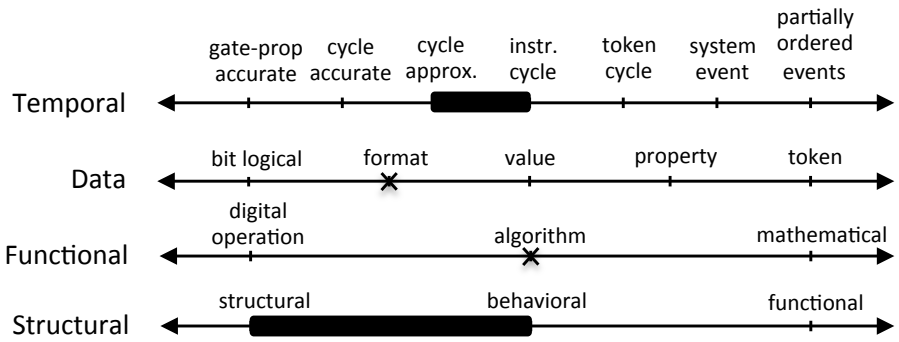


Abbildung 2.2: Klassifikationsbeispiel SoCRocket Modelle

Die betrachteten Modelle können für verschiedene zeitliche Auflösungen konfiguriert werden. Je nach Einstellung sind sie *instruction cycle*-akkurat oder *cycle approximate*. In jedem Fall wird der Simulation ein Taktsignal in Form einer zeitlichen Konstante zugrunde gelegt. Diese dient ausschließlich der Abschätzung des zeitlichen Verhaltens. Es wird keine taktgesteuerte Simulation durchgeführt. Die Auflösung der Daten ist verhältnismäßig hoch. Alle Daten werden aus Sicht des Programmierers in hardware-äquivalentem Format dargestellt. Dabei werden unter anderem Eigenschaften wie *Endianess*, Instruktionskodierung und die Belegung von Steuerregistern berücksichtigt. Bit-logische Genauigkeit wird jedoch nicht erreicht, da eine Abstraktion im Sinne der Abbildung auf physikalische Speicherelemente vorgenommen wurde. Auf der funktionalen Achse wird algorithmische Genauigkeit erreicht. Die implementierten Funktionen entsprechen der Spezifikation, werden aber nicht durch die in der Hardware vorhandenen Funktionseinheiten realisiert. Die modellierten Komponenten sind teilweise strukturell und bilden gleichzeitig Funktion und Zeitverhalten ab.

Neben der vorgestellten Modell-Taxonomie entwickelten sich weitere Klassifikationsverfahren mit den Schwerpunkten Verifikation oder hardwareabhängige Software. Ein hervorragender Überblick über die Thematik findet sich in [Bai05]. Darüber hinaus stellt [Den06] eine plattformbasierte Klassifizierung vor, welche die Einordnung von Werkzeugen in Bezug auf ihre Position im Entwurfsfluss (*Design Flow*) erlaubt. Das Konzept des plattformbasierten Entwurfs wurde von Sangiovanni-Vincentelli und Martin eingeführt [SV01]. Dabei wird der Entwurfsprozess als eine Sequenz von Schritten dargestellt, die sich beim Übergang von hoher zu niedrigerer Abstraktion wiederholen (Abb. 2.3).

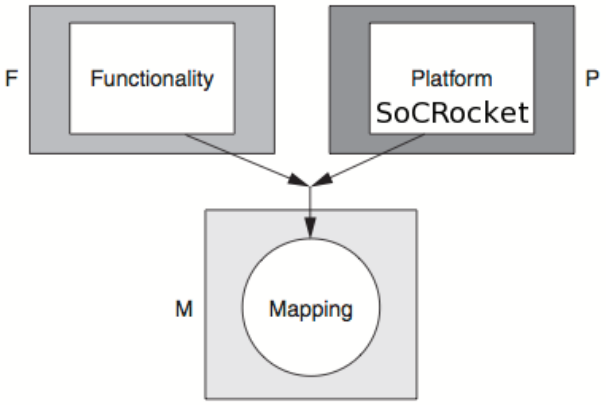


Abbildung 2.3: Plattform-basierte Klassifikation

Je Abstraktionsebene wird zwischen drei Werkzeugklassen unterschieden: Funktionalität, *Mapping* und Plattform (Klassen F/M/P). Klasse F bezieht sich auf die implementationsunabhängige Darstellung von Entwürfen und umfasst Werkzeuge zur Manipulation, Simulation und Analyse funktionaler Beschreibungen. Prominente Vertreter dieser Klasse sind u.a. *Matlab* und *SystemVision* (siehe Abschnitt 2.3.3). Klasse M umfasst Werkzeuge zur Umsetzung bzw. Synthese von funktionaler Beschreibung in eine Plattforminstanz auf einem niedrigeren Abstraktionsniveau. Die Werkzeuge der Klasse M unterstützen verschiedene Verarbeitungsmodelle (*Models of Computation* (*MoC*)) und sind in der Regel auf einen speziellen Anwendungsfall zugeschnitten (z.B. HL- oder *Behavioral*-Synthese). Klasse P enthält Bibliotheken und Module zur Implementierung von Funktionen. Dies sind Prozessoren, Co-Prozessoren, FPGAs, Speicher und Hardwarebeschleuniger, aber auch Softwarekomponenten wie Betriebssysteme oder *Middleware*. Zur Klasse P gehören ebenfalls Werkzeuge zur Analyse, Konfiguration und Manipulation dieser Komponenten. Beispiele für Werkzeuge der Klasse P sind die *OpenCores IP Bibliothek* oder *Synopsys Platform Architect* (siehe Abschnitt2.3.2).

In den folgenden Abschnitten werden wichtige Werkzeuge und aktuelle Forschungsprojekte der verschiedenen Werkzeugkategorien beschrieben.

2.3.2 Virtuelle Plattformen (Klasse P)

Entsprechend der vorgestellten Klassifizierung ist das in dieser Arbeit entwickelte *SoCRocket*-System ein Werkzeug der Klasse P. Es stellt *SystemC/TLM*-Komponenten und Werkzeuge zur Modellierung von Systemen auf hohem Abstraktionsniveau bereit. Derartige Systeme werden auch als Virtuelle Plattformen (VP) bezeichnet. Virtuelle Plattformen dienen der Erzeugung von Computer-Simulationsmodellen eines Systems, sogenannten Virtuellen Prototypen. VPs mit spezieller Ausrichtung auf die Luft- und Raumfahrtbereich existieren kaum. Mögliche Gründe sind der im Vergleich zu anderen Anwendungsfelder eingebetteter Systeme sehr kleine Markt und die in Abschnitt 1.3 aufgeführten technischen Beschränkungen, welche die abstrakte Modellierung von Systemen bisher weniger zwingend erscheinen ließen. Es ist jedoch durchaus denkbar, Werkzeuge aus anderen Bereichen, z.B. dem Mobilfunk oder der Unterhaltungselektronik, wiederzuverwenden. *SoCRocket* kann in diesem Fall durch die Bereitstellung von Simulationsmodellen mit standardkonformen Schnittstellen ein Verbindungsglied darstellen. Im folgenden werden die in diesem Kontext relevanten kommerziellen und akademischen Lösungen vorgestellt.

Cadence Virtual System Platform (VSP)

Die Firma *Cadence Inc.* gehört neben *Mentor Graphics Inc.* und *Synopsys Inc.* zu den Marktführern im Bereich Entwurfsautomatisierung. Jede der auch *die großen Drei* genannten Firmen bietet Virtuelle Plattformen als integrierten Teil der hauseigenen ESL-Lösung an. VSP basiert auf einem IP-Katalog der unter anderem *FastModels* von ARM und *Imperas*-Simulatoren enthält [Inc14b]. Die Einbindung verschiedenster Prozessormodelle wird durch eine generische Schnittstelle, den *Processor Abstraction Layer* (PAL), realisiert. *Cadence* entwickelt aber auch eigene Simulations-IP, zum Beispiel für die durch den Zukauf der Firma *Denali Inc.* erworbenen Speichercontroller. In der Regel lizenziert der Nutzer Simulationsinfrastruktur und Modelle getrennt voneinander. Zur Erzeugung eines Plattform-Prototypen stehen verschiedene Werkzeuge bereit. Die Modellbildung erfolgt in drei Schritten: Zuerst wird eine *Register VP* erzeugt. Diese enthält alle Register bzw. Bit-Felder des Entwurfs und repräsentiert die initiale Software-Schnittstelle. Alle dafür notwendigen Komponenten können aus einer IP-XACT- oder RDL-Beschreibung generiert werden. Der nächste Schritt ist die Entwicklung der *Functional VP*. Dies umfasst hauptsächlich die Modellierung des Verhaltens und der Kommunikationsschnittstellen. Modelliert wird in SystemC/TLM2.0. Alle Kommunikation ist *Loosely-Timed (LT)*. Der letzte Schritt ist das *Software Bring-up*. Dabei wird die VP mit Hilfe von Anwendungssoftware oder eines Betriebssystems optimiert. *Cadence* stellt VSP ein HLS-Werkzeug zur Seite, den sogenannten *C-to-Silicon Compiler*. Das Werkzeug erleichtert die Überführung abstrakter Simulationsmodelle

in synthetisierbaren Code. Durch die Einkopplung von RTL-Synthese wird dabei eine frühe Abschätzung des zeitlichen Verhaltens erreicht. Die HLS-Modelle können in VSP eingebettet und simuliert werden. Die eigentliche Simulation ist auf dem *Cadence Incisive Simulator* aufgebaut, der über eine eigene Implementierung des *SystemC*-Kernels und diverse Erweiterungen für *Debugging* und *Profiling* verfügt. Mit Hilfe von Analysewerkzeugen können Transaktionen nach Zeit, Typ oder Quelle gefiltert und dargestellt werden. Ähnlich wie das im folgenden vorgestellte System von *Synopsys* kann VSP als Ergänzung zu *SoCRocket* eingesetzt werden. *SoCRocket* fungiert in diesem Fall als standardisierte IP-Bibliothek und stellt die Basisfunktionalität für die Analyse und Optimierung des Systems bereit. Die Simulation mit VSP bietet sehr komfortable Möglichkeiten zur Aufbereitung und Auswertung von Simulationsergebnissen. Dies kann insbesondere für die Optimierung sehr großer Systeme hilfreich sein.

Synopsys Platform Architect (PA)

Platform Architect ist eine Weiterentwicklung des *Platform Creators* (PC), der durch den Zukauf der Firma *Coware* im Jahre 2010 erworben wurde [Inc14d]. PA/PC ist eine der ersten kommerziell erfolgreichen Virtuellen Plattformen und war bei seiner Einführung wegweisend. Dies umfasst sowohl die Kapselung von Hardware- und Softwarekomponenten (siehe Kapitel 2.1) zur Unterstützung des Partitionierungsprozesses, als auch die Standardisierung von Schnittstellen und Konfigurationsmechanismen. Basierend auf *Synopsys*-Simulationswerkzeugen (VCS) erlaubt PA heute die Integration und Co-Simulation von Modellen mit unterschiedlichem Abstraktionsgrad. Dem Benutzer wird eine komfortable GUI präsentiert, mit der sich Modelle mittels *Drag & Drop* im System platzieren und verdrahten lassen. Die Entwicklung neuer Komponenten wird durch Werkzeuge unterstützt, die TLM-Schnittstellen und Speicherelemente automatisch generieren. *Synopsys* stellt eine umfangreiche Bibliothek an Simulationsmodellen bereit. Neben proprietären Schnittstellen werden TLM2.0 und OCP unterstützt. Es ist theoretisch möglich, beliebige Modelle von Drittanbietern einzubinden. Darüber hinaus wird mit *Processor Designer* ein Werkzeug zur Erzeugung von anwendungsspezifischen Prozessoren (ASIP) bereitgestellt. Auf Grundlage einer Architekturbeschreibung (LISA2.0) können Simulatoren und synthesefähige Hardwaremodelle für RISC- und VLIW-Prozessoren erzeugt werden. Wie bereits erwähnt, lassen sich auch die ESL-Werkzeuge von *Synopsys* als Ergänzung zu *SoCRocket* einsetzen. Der Nutzer kann selbst entscheiden, ob er eine Investition in zusätzliche Hilfsmittel tätigen möchte. Die durch *SoCRocket* bereitgestellte Basisfunktionalität sowie die Transparenz und Portabilität der Komponenten werden hierdurch nicht beeinträchtigt.

Carbon SoC Designer / Model Studio

Carbon Design Systems ist eine auf *Virtual Prototyping* spezialisierte Firma aus der Region Boston (USA). Man erkannte früh, dass heute 80% der Komponenten in SoCs extensiv wiederverwendet werden und spezialisierte sich auf Werkzeuge zur Erzeugung abstrakter Simulationsmodelle für vorhandene konventionelle RTL IP (*Carbon Model Studio*). Auf der Firmenwebseite sind verschiedene technische *Whitepapers* verfügbar [car14]. Die generierten Modelle sind in der Regel weniger performant als manuell konstruierte und optimierte Modelle, sind aber wesentlich einfacher zu verifizieren. Dies ist ein entscheidender Vorteil und trug wesentlich zum Erfolg bei. *Carbon* kooperiert heute eng mit ARM Ltd. und bietet zyklengenaue Modelle für verschiedene ARM-Prozessoren an. Zur Komplettierung der Entwicklungsumgebung wurde darüber hinaus *SoC Designer Plus* entwickelt. Dabei handelt es sich um eine Benutzerschnittstelle ähnlich *Synopsys Platform Architect*, in der Simulationsmodelle auf unterschiedlichen Abstraktionsstufen mittels *Drag & Drop* zu Plattformprototypen verknüpft werden können.

Virtutech / SIMICS

SIMICS ist ein Systemsimulator der, anders als die meisten heute praktisch relevanten Systeme, nicht auf *SystemC* aufbaut. Ziel ist die möglichst schnelle funktionale Simulation von Virtuellen

Prototypen auf hohem Abstraktionsniveau zur Beschleunigung der Softwareentwicklung [Mag02]. SIMICS entstand ursprünglich am *Swedish Institute of Computer Science* in Stockholm, wurde dann in ein *Spin-off* (Virtutech) überführt, welches heute zu *Wind River* gehört. In SIMICS werden Prozessoren und Peripheriekomponenten in einer einfachen objektorientierten Sprache beschrieben. Die resultierenden Modelle werden dann mit Hilfe einer speziellen API, zum System hinzugefügt. Simulationen werden durch eine Kommandoschnittstelle gesteuert, die auch Pythonskripte interpretieren kann. Dadurch kann das System flexible zur Laufzeit analysiert und modifiziert werden. Die Komponentenbibliothek von SIMICS umfasst heute fast alle aktuellen Prozessoren von ARM, Intel, MIPS und Freescale (PowerPC), sowie eine Vielzahl an Peripherie [Inc14]. Ähnlich wie bei anderen konventionellen VPs ist es außerdem möglich, SIMICS zur Co-Verifikation von Komponenten auf niedrigerem Abstraktionsniveau, mit externen Simulatoren zu verbinden. Dies ist jedoch nicht der primäre Anwendungszweck. Der größte Vorteil des Systems besteht in seiner enorm hohen Simulationsgeschwindigkeit, die bereits auf einem *Host* mit nur 750 MHz Taktrate mehrere Millionen Instruktionen pro Sekunde beträgt. SIMICS bildet ebenfalls die Grundlage für den von AMD angebotenen x86-Simulator SIMNOW [sim14] und den darauf aufbauenden Systemsimulator *COTson* von HP-*Labs* [cot14].

ARM – Fast Models

ARM-Prozessoren sind heute dominant in vielen Einsatzgebieten eingebetteter Systeme. Als einer der wichtigsten *IP-Provider* erkannte man früh die Vorteile Virtueller Plattformen und die Notwendigkeit seinen Kunden schnelle Simulatoren, zur Steigerung der Effizient im Softwareentwurf bereitzustellen. *Fast Models* [fas14] sind hochabstrakte Modelle, welche die Sichtweise des Programmierers auf das System widerspiegeln. Viele Details der Mikroarchitektur, wie *Pipelining* werden abstrahiert, um die Simulationsgeschwindigkeit zu steigern. Der funktionale Kern der Modelle ist in C/C++ modelliert und wird innerhalb eines LISA+-*Wrappers* instantiiert, mit dessen Hilfe Register und Schnittstellen definiert werden. *Fast Models* sind in abgewandelter Form für alle kommerziellen VP-Systeme erhältlich. Mit *System Canvas* und *System Generator* bietet ARM darüber hinaus eigene Lösungen zur Erzeugung Virtueller Prototypen an. Über *Europractice* [eur14] können europäische Universitäten vergünstigt Zugriff erhalten. Die Raumfahrt ist einer der wenigen Bereiche, in denen ARM-Prozessoren nur geringe Bedeutung haben. Hier werden vorwiegend SPARC-CPUs (LEON) eingesetzt.

Imperas Open Virtual Platform (OVP)

Imperas vertreibt Simulatoren für verschiedene Prozessorarchitekturen. Verfügbar sind MIPS32 als Single-Core (z.B. 74Kc/Kf), Dual-Core (34Kc/Kf) und Quad-Core (1004Kc/Kf), ARM-Prozessorsimulatoren für die Architekturen v4 bis v7 sowie verschiedene CPUs von ARC, NEC und Freescale (PowerPC) [ovp13]. Alle Modelle sind *Open Source*, können aber nur mit dem kostenpflichtigen *Imperas*-Simulator (OVPsim) simuliert werden. OVPsim unterstützt TLM2.0 und kann damit auch zur Simulation von *SoCRocket*-Systemen eingesetzt werden. Darüber hinaus ist es möglich, OVP-Modelle mit einer Kompatibilitätsschicht auszurüsten. Dadurch können diese, auch umgekehrt, in *SoCRocket* eingebunden werden.

SoCLib

SoCLib ist eine der bekanntesten nicht-kommerziellen Lösungen für *Virtual Prototyping* von MPSoCs [soc13]. Schwerpunkte des Projektes sind die Simulationsbeschleunigung, die Konfiguration, das *Debugging* und die automatische Generierung von Simulationsmodellen, sowie die Entwicklung von Anwendungen für eingebettete Systeme. Ein wissenschaftlicher Beitrag besteht in der Entwicklung von TLM-DT [Mel10], einer Erweiterung zur verteilten Simulation von TL-Systemen. Die Plattform ist unter der GPL-Lizenz veröffentlicht und frei im Netz verfügbar. Sie enthält verschiedene Modelle zur Simulation von Prozessoren, Peripheriekomponenten oder Netzwerk-Routern. *SoCLib* wurde durch die *Agence Nationale de la Recherce* (ANR) gefördert.

An der Entwicklung des Systems waren sechs industrielle Partner, unter anderem *ST Micro-Electronic*, *Thales*, *Thompson* sowie 11 Forschungsinstitute beteiligt. Der größte Nachteil des Systems ist die fehlende Kompatibilität zu aktuellen Standards, insbesondere TLM2.0. Dadurch ist die praktische Anwendung stark eingeschränkt und eine kombinierte Nutzung von SoCLib und SoCRocket derzeit nicht möglich.

GreenSoCs

GreenSoCs ist eine Kooperationsplattform zur Entwicklung offener VP-Infrastruktur [gre13]. Es werden Lösung zur Modellkonstruktion, Modell-zu-Modell-Kommunikation, Modell-Werkzeug-Kommunikation, sowie verschiedene Simulationsmodelle und Entwicklungswerkzeuge angeboten. Hervorzuheben sind das *GreenReg-Framework* zur Modellierung von Registern, das aus einer ursprünglich von Intel entwickelten Lösung hervorging. Darüber hinaus werden spezialisierte *Sockets* mit spezifischen Erweiterungen zur zyklengenauen Modellierung von AMBA, PCIe und OCP angeboten. Speziell zur Integration mit Werkzeugen wurde *GreenControl* entwickelt. Dabei handelt es sich um eine Schnittstelle, mit der Modellparameter zur Laufzeit ausgelesen und rekonfiguriert werden können. Ein Großteil der *GreenSoCs*-Projekte, zum Beispiel *GreenControl*, *GreenAV* und die AMBA-*Sockets*, wurden in Kooperation mit der TU-Braunschweig entwickelt [Sch11][Gün11].

Unisim

Die UNISIM *United Simulation Platform* ist eine weitere im universitären Umfeld sehr verbreitete Virtuelle Plattform. Das Projekt wird durch das europäische *HiPEAC*-Netzwerk [hip14] unterstützt und hat seine Wurzeln bei *INRIA* in Frankreich. Der Schwerpunkt von *UNISIM* ist die Exploration von Software-Simulationstechniken. Das Projekt stellt den Versuch dar, eine einheitliche Simulationsinfrastruktur für die gemeinschaftliche, verteilte Entwicklung komplexer Systeme zu schaffen. Dazu wurden generische Simulatorschnittstellen entwickelt und eine Bibliothek mit Simulationsmodellen aufgebaut. Plattform und Simulationsmodelle stehen unter BSD-Lizenz und sind im Internet frei verfügbar [uni13]. Ähnlich wie bei *SoCLib* besteht keine Unterstützung für moderne Industriestandards wie TLM2.0, was die Weiterentwicklung des Systems und die Wiederverwendung der vorhandenen Komponenten erschwert.

ReSP / TrapGen

Die *Reflective Simulation Platform* (ReSP) [Bel09] ist ein Projekt der *Politecnico di Milano*. Der Quellcode steht unter GPL-Lizenz und ist auf *Google Code* frei verfügbar [res13]. Im Gegensatz zu den meisten anderen offenen Systemen (z.B. SoClib, Unisim) wird bereits TLM2.0 unterstützt. Die Innovation von ReSP liegt in seiner Verknüpfung von SystemC und Python. Jede SystemC-Klasse des Systems wird in Python gekapselt. Dazu müssen Struktur und Eigenschaften der jeweiligen Komponente in XML beschrieben werden. Die Systemsimulation wird dann in Python konstruiert. Dadurch ist es möglich, Spracheigenschaften wie Reflexion und Introspektion auszunutzen. Die Simulation ist damit in der Lage, ihre Struktur zu analysieren und, wenn nötig, dynamisch zu verändern. Darüber hinaus können Informationen zu Klassen oder Instanzen zur Laufzeit einfach abgefragt werden, was die Analyse erleichtert. Zur Umsetzung von Reflexion müssen Metainformationen im Binärcode des Programmes gespeichert werden. Daher liegt die Simulationsgeschwindigkeit unter der nativ kompilierter statischer Programme. Grund dafür sind die erforderlichen Stringvergleiche zur Identifikation von Klassen, Methoden und Attributen. Dies wird jedoch durch hohe Flexibilität ausgeglichen. So ist es möglich Komponenten zur Laufzeit neu zu strukturieren und ganze Simulationen dynamisch umzubauen. Leider wurde ReSP in den vergangenen Jahren kaum weiterentwickelt.

Eng verknüpft mit ReSP ist der *TRansactional Automatic Processor generator* (TRAP) [Fos10], der ebenfalls als offenes Projekt auf Google-Code verfügbar gemacht wurde [tra]. TRAP umfasst eine Architekturbeschreibungssprache und Werkzeuge zur automatischen Erzeugung

von Prozessorsimulatoren. Wie auch ReSP arbeitet TRAP auf Grundlage von Python. Die Spezifikation von Prozessoren erfolgt durch den Aufruf von Funktionen in der TRAP-API. Eine Architekturbeschreibung umfasst eine strukturelle Beschreibung (z.B. Register, Pipelinestufen), die Binärcodierung der Instruktionen und die Beschreibung des Verhaltens. Momentan stehen vier fertige Simulatoren bereit: Microblaze, ARM7, MIPS und LEON3. Die Modelle haben eine relativ geringe Komplexität (10k - 30k LoC). Es wurden nur die Integereinheiten modelliert. Caches und Speichermanagement (MMU) sind nicht vorhanden. Fließkommaberechnungen müssen durch Integeroperationen dargestellt werden. Eine besonders interessante Eigenschaft von TRAP ist die OS-Emulation. Die Simulatoren laden zu Beginn der Simulation ELF-Dateien. Dabei werden Systemfunktionen, wie zum Beispiel die Standardausgabe oder Dateizugriffe, auf den Host umgeleitet. Die entsprechenden Funktionen greifen mit Hilfe der TLM2.0-Debug-Transportschnittstelle auf den simulierten Speicher zu. Dadurch lässt sich die Simulation beschleunigen. Außerdem können im frühen Entwicklungsstadium auf einfache Weise Testausgaben und IO bewerkstelligt werden.

Weitere Werkzeuge

Über die beschriebenen Lösungen hinaus existieren diverse weitere Werkzeuge der Klasse P. *TLMCentral* [tlm14] ist eine Datenbank kommerziell und frei verfügbarer TL-Simulationsmodelle. Ausschließlich offene Lösungen werden über *OpenCores* angeboten [ope14]. *OpenCores* verfügt über eine große Auswahl an synthetisierbaren VHDL- und Verilog-Modellen, die Auswahl an Modellen zur abstrakten Modellierung auf Systemebene ist jedoch sehr gering und beschränkt sich auf einen NoC-Simulator und einige Filter. Darüber hinaus wird ein auf TLM2.0 aufbauendes OCP-*Modeling Kit* angeboten. Dieses enthält einfach zu nutzende *Sockets* mit integriertem *Payload*-Management, die im Vergleich zu TLM2.0 über zwei zusätzliche Abstraktionsstufen verfügen. Ein weiteres weit verbreitetes Werkzeug der Klasse P ist MPARM [Ben05] der Universität Bologna. Dem Namen entsprechend ermöglicht MPARM die schnelle Modellierung von ARM-basierten Multiprozessoren auf hohem Abstraktionsniveau. Die VP enthält eine Bibliothek mit entsprechenden Simulatoren und Peripheriekomponenten. Ebenfalls sehr beliebt zur Simulation von ARM-Prozessoren ist *ARMulator*. Ein praktisches Beispiel zum Design einer Virtuellen Platform basierend auf *ARMulator* kann [Lee05] entnommen werden. Das PTLSim-System der *New York State University* ist dahingegen auf zyklengenaue Simulation von x86-Architekturen zugeschnitten. In [You07] wird ein interessantes Experiment beschrieben, bei dem mit PTSim ein AMD-Athlon-Multiprozessor inklusive eines XEN-Hypervisors integriert wurde. Eine sehr flexible Virtuelle Plattform basierend auf einem generischen Prozessormodell wurde neulich durch Wissenschaftler des *Technological Educational Institute of Crete* in Griechenland vorgestellt [Gra13]. Das System baut auf einer NoC-Architektur auf und integriert eine spezielle *Monitoring*-Ebene zur Überwachung von Speicherzugriffen und der effizienten Aufteilung der verfügbaren Bandbreite. Eines der beliebtesten freien Werkzeuge zur Modellierung und Simulation von RISC-Prozessoren ist *SimpleScalar*. Unter anderem werden ARM, Power PC, x86 und Alpha unterstützt. Das System wird in [Aus02] durch Todd Austin, Eric Larson und Dan Ernst von der *University of Michigan* ausführlich beschrieben. Ähnlicher Popularität erfreut sich CACTI, ein Werkzeug zur Exploration von Verbindungsstrukturen und *Caches* auf Systemebene [Mur08]. CACTI hat keinen direkten Bezug zu Virtuellen Plattformen, kann aber unterstützend oder im Vorfeld zur Beschränkung des Entwurfsraumes eingesetzt werden.

2.3.3 Funktionalität und Mapping (Klassen F und M)

Der Schwerpunkt dieser Arbeit sind Virtuelle Plattformen (Klasse P). Trotzdem sollen hier die wichtigsten Werkzeuge und Forschungsaktivitäten bezüglich Funktionalität und *Mapping* (Klassen F und M) kurz zusammengefasst werden.

High-Level-Synthese

Typische Mitglieder der Klasse M sind *High-Level*-Synthesewerkzeuge. Als Ausgangspunkt moderner *HLS*-Werkzeuge dient die Beschreibung des zu implementierenden Algorithmus in einer Hochsprache wie C/C++ oder Matlab. Die Beschreibung umfasst die Funktionsweise, enthält jedoch keine Details zur Hardwarestruktur. Informationen zu Datentypen, paralleler und sequentieller Verarbeitung und der Abbildung von Variablen und Feldern auf Register und Speicher müssen ergänzt werden. Abhängig vom Werkzeug wird dies durch Pragmas im Quellcode oder durch Eingaben in einer graphischen Benutzeroberfläche bewerkstelligt. Die resultierende funktionale und strukturelle Beschreibung wird dann durch die Synthesesoftware in eine RTL-Beschreibung, also eine Darstellung des Systems auf dem nächst niedrigeren Abstraktionsniveau überführt. HLS-Werkzeuge setzen sich nur sehr langsam durch. Hauptgrund dafür ist die Komplexität der impliziten Strukturbeschreibung. Der generierte RTL-Code muss sich in seiner Qualität an handkodierten Entwürfen messen [Tob10]. Dies ist oft nur mit sehr viel Erfahrung zu erreichen. Die aktuell führende HLS-Werkzeuge sind *CatapultC* von *Calypto* (früher Mentor Graphics) [cal14], *C-to-Silicon Compiler* von *Cadence* [cto14] und *ImpulsC* von *Impulse Accelerated Technologies* [imp14]. *CatapultC* und *C-to-Silicon Compiler* zielen hauptsächlich auf die Erzeugung von Hardwarebeschleunigern für ASICs. *C-to-Silicon* verbindet sich dazu mit einem RTL-Synthesewerkzeug, dass Rückmeldung bezüglich des optimalen *Timings* von Pipelinestufen liefert. *ImpulsC* dagegen zielt vordringlich auf FPGA-Entwurf. Ansonsten sind die Unterschiede im Funktionsumfang der genannten Werkzeuge eher gering.

Prozessorgeneratoren

Eine Kombination aus Funktionsbeschreibung und Mapping (Klassen F und M) bilden die Prozessorgeneratoren. Diese werden ebenfalls oft in Virtuelle Plattformen oder Modellbibliotheken integriert und könnten damit unter Umständen auch der Klasse P zugeordnet werden. Da die Beschreibung der Struktur eines Prozessors sehr aufwendig ist, werden hier *Templates* verwendet, die durch den Nutzer mit Hilfe von Architekturbeschreibungssprachen (ADL) parametrisiert werden. Eine der bekanntesten ADLs ist die *Language for Instruction Set Architecture* (*LISA*). Sie wurde ursprünglich an der RWTH-Aachen entwickelt und wird heute zum Beispiel im *Processor Designer* von *Synopsys* (früher *Coware*) [pro14] eingesetzt. Mit Hilfe von *LISA* beschreibt man die Pipeline eines Prozessors, das Verhalten seiner Instruktionen sowie die Instruktionskodierung und deren Assembler-Syntax. Das Werkzeug generiert dann einen Simulator, der mit TLM-Schnittstellen ausgerüstet werden kann, Softwareentwicklungswerkzeuge (Assembler, Linker, Compiler) und eine RTL-Implementierung in VHDL oder Verilog. Der Entwurfsfluss erlaubt die schnelle Optimierung des Instruktionssatzes für spezielle Anwendungen (*Application Specific Instruction set Processors* – ASIP). Ein ähnliches Konzept wird durch *Tensilica* (jetzt *Cadence*) verfolgt. Der *Xtensa*-Basisprozessor kann mit Hilfe der TIE-Sprache [Tho13] um spezielle Funktionseinheiten und Spezialbefehle erweitert werden. Das *Trimaran*-System [tri14] ist auf VLIW-Architekturen spezialisiert. Erfolgreiche Generatoren für massiv parallele Prozessoren sind zum Beispiel *Silicon Hive* von *Intel* oder ADRES von IMEC. *Silicon Hive* generiert VLIW-Beschleuniger aus der ADL TIM [Pin06]. ADRES erzeugt ein *Coarse-Grained-Array* spezialisierter Funktionseinheiten mit Hilfe von *Simulated-Annealing* [Nov08].

MathWorks Matlab

Matlab und seine Erweiterung *Simulink* [Inc14c] sind wahrscheinlich die bekanntesten kommerziellen Werkzeuge der Klasse F. Besonders in den Bereichen Signalverarbeitung und Automatisierungstechnik hat sich *Matlab* als defacto Standard zur implementierungsunabhängigen funktionalen Beschreibung von Algorithmen durchgesetzt und dient beim Designeinstieg als ausführbare Spezifikation. Durch verschiedene Erweiterungen wird momentan versucht einen direkten Pfad zur Implementierung zu öffnen. So ist es möglich mit Hilfe von Konvertern C-Code zu generieren, der wiederum als funktionaler Kern für SystemC/TLM-Komponenten dienen

kann. Mit *Simulink* verfolgt man das Konzept des modellbasierten Designs. In einer graphischen Oberfläche können in Bibliotheken vordefinierte Komponenten arrangiert werden. Das Werkzeug generiert dann mittels Einfügen von Verbindungsstrukturen C-Code, VHDL oder Verilog.

Cascade Critical Blue

Cascade [cri05] ist ein Werkzeug zur automatischen Synthese von Coprozessoren und wird durch *CriticalBlue Ltd.* entwickelt und vertrieben. Die Besonderheit an *Cascade* sind seine Softwareanalysefunktionen. Das Werkzeug analysiert und optimiert die Coprozessorarchitektur und Software vor dem Syntheseschritt anhand benutzerdefinierter Implementierungsziele. Dadurch können auf einfache Weise unterschiedliche Optionen erprobt werden. Ergebnis sind ein oder mehrere Coprozessoren, die mit dem Hauptprozessor über den Systembus kommunizieren. Es wird eine RTL-Beschreibung des Systems generiert. Sowohl *HW*- als auch *SW*-Schnittstellen werden automatisch erzeugt.

SpaceStudio

SpaceStudio ist eine Virtuelle Plattform, die aus Forschungsaktivitäten der Universität von Montreal entstand [Fil07] und heute durch deren Ausgründung *SpaceCodesign* [spa14] weiterentwickelt wird. Es handelt sich um eine auf Eclipse basierende Entwicklungsumgebung, die den Entwickler auf dem Weg vom abstrakten Simulationsmodell bis hin zur FPGA- oder Chipimplementierung begleitet. Vorteil der Verwendung von Eclipse [ecl14] ist die einfache Integration externer Werkzeuge von Drittanbietern, wodurch Entwurfsprozesse individuell abgebildet werden können. Das System hat seine besonderen Stärken im Entwurfseinstieg. Mit Hilfe der Benutzeroberfläche können Verhaltensbeschreibungen in SystemC gekapselt und als Hardware oder Software gekennzeichnet werden. Eine Systemsynthese im eigentlichen Sinne findet nicht statt. Hardwarekomponenten zur Unterstützung der verschiedener Abstraktionsniveaus müssen extra beschafft und speziell verpackt werden. *SpaceStudio* unterstützt den Entwickler bei der Verbindung/Verdrahtung der Komponenten und stellt Analysewerkzeuge zur Auswertung des Zusammenspiels von Hardware und Software bereit.

Berkeley Ptolemy II

Das *Ptolemy-Framework* der *University of Berkley* war eines der ersten Werkzeuge zur Beschreibung von Systemen, dass verschiedene MoCs kombiniert [Buc91]. Entstanden in den frühen 1990er-Jahren hat es bis heute, besonders mit seinem Nachfolger *Ptolemy II*, große Bedeutung und ist in der akademischen Gemeinde weit verbreitet. Funktionen für *Ptolemy* können als Datenflussgraphen, *Discrete Event*- oder *Continuous Time*-Beschreibung, Kahn-Prozessmodell oder synchrones/reaktives Modell beschrieben werden [Bro10]. *Ptolemy* eignet sich damit potentiell zur Darstellung moderner cyber-physischer Systeme, in denen SystemC/TLM nur eines von vielen MoCs darstellt. Der Kern von *Ptolemy* besteht aus einer Anzahl von Java-Klassen, welche die hierarchische Strukturierung und Verbindung von Komponenten ermöglichen. Zur Integration in das *Framework* müssen alle Modelle unabhängig vom MoC in XML beschrieben werden (MoML). Ergänzend zu *Ptolemy* entstanden verschiedene Werkzeuge, die den Einsatz auf verschiedenen Anwendungsgebieten erleichtern. Dazu zählen zum Beispiel *VisualSense* zur Modellierung von Drahtlosen Netzwerken, *Viptos* für den Entwurf von Sensornetzwerken oder *Kepler* für wissenschaftliche Berechnungen.

Weitere Werkzeuge

Es existieren diverse weitere Werkzeuge der Klassen F und M. Typische Vertreter sind unter anderem NoC-Generatoren wie NoC-Wizard/xENOC [Jov08], welche die Synthese von NoCs aus abstrakten XML-Beschreibungen erlauben. Weitere bekannte Werkzeuge zur funktionalen Beschreibung von Systemen sind z.B. *LabView*, *Mathematica*, *Maple*, *Esterel* und *Rhapsody*.

2.4 ESL in HW/SW-Co-Design

Der Prozess zum Entwurf eingebetteter Systeme ist heute in aller Regel *top-down*-orientiert und startet mit einer ausführbaren Spezifikation auf Systemebene. Basierend auf dieser Spezifikation werden *Hardware* und *Software* gleichzeitig entwickelt. Diesen Vorgang bezeichnet man als *HW/SW-Co-Design*. Die Entscheidung, welcher Teil der Funktionalität Eingang in *Hardware* oder *Software* findet, wird als Partitionierung bezeichnet und durch Explorationsergebnisse und die Erfahrung des Entwicklers gesteuert. Den Prozess der Auswahl einer geeigneten Architektur aus einem Raum verschiedener Möglichkeiten nennt man Systemsynthese. Die bekannteste Beschreibung der beim Entwurf eingebetteter Systeme relevanten Abstraktionsebenen geht auf D. D. Gajsky zurück [Gaj83]. Die Arbeit unterscheidet zwischen Systemebene, Register-Transfer-Ebene, Gatter-Ebene und Transistor-Ebene. Den beschriebenen Ebenen werden mit Hilfe des bekannten Y-Diagramms unterschiedliche Repräsentationen eines Entwurfs zugeordnet (Abbildung 2.4).

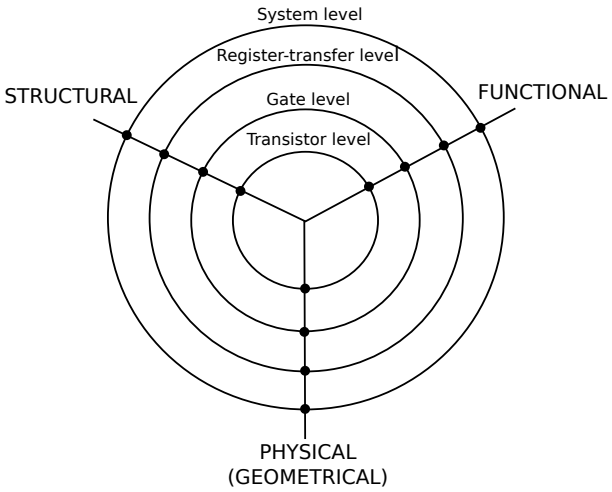


Abbildung 2.4: Gajsky's Y-Chart

Eine heute gängige Erweiterung des Y-Ansatzes mit dem Schwerpunkt *Hardware*- und *Software*-Entwicklung bildet das von J. Teich entwickelte *Double-Roof*-Modell (Abbildung 2.5 [Tei07]). Die linke Seite der Graphik beschreibt typische Abstraktionsebenen im *Software*-Entwicklungsprozess (Modul, Block). Die rechte Seite bezieht sich auf die Entwicklung von *Hardware* (Architektur, Logik). Beide Seiten teilen sich die Systemebene, auf der noch keine klare Unterscheidung zwischen *Hardware* und *Software* existiert. Jeder vertikale Pfeil stellt einen Syntheseschritt dar, bei dem eine Spezifikation auf eine strukturelle Implementierung der nächst niedrigeren Abstraktionsstufe abgebildet wird. Das Ergebnis der höheren Stufe dient als Eingabe/Spezifikation für den folgenden Syntheseschritt (horizontale Pfeile). Es werden zwei Dächer sichtbar, die verschiedenen Blickwinkeln auf das System entsprechen.

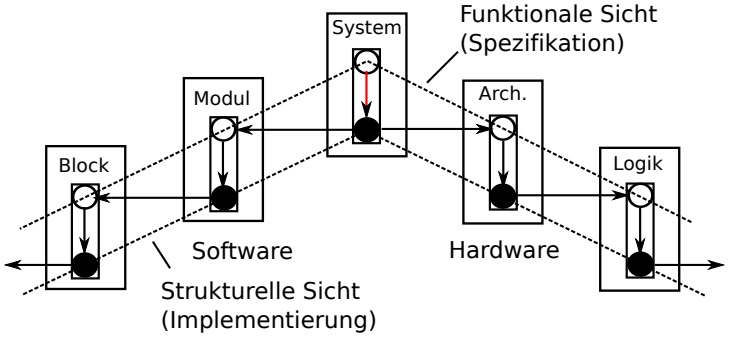


Abbildung 2.5: *Double-Roof-Model* des *HW/SW Co-Design*

Der Schwerpunkt der vorliegenden Arbeit liegt in der Systemebene und damit in der Überführung einer funktionalen Spezifikation in eine Hardware- und eine Softwarearchitektur (roter Pfeil). Die initiale Spezifikation zum Entwurf eines Systems wird typischerweise in Hochsprachen wie *C/C++* oder *Matlab* verfasst. Die Mächtigkeit dieser Sprachen erlaubt es, Algorithmen in ihrer Funktion ohne Rücksicht auf physikalische Einschränkungen, z.B. die Anzahl vorhandener Speicherports, zu erfassen. Zur Synthese effizienter Schaltungen ist diese Information nicht ausreichend. Zeitverhalten, Nebenläufigkeit und zur Verfügung stehende Ressourcen müssen annotiert oder zusätzlich beschrieben werden. Dies ist die Domäne typischer Hardwarebeschreibungssprachen wie VHDL oder Verilog. Zur Überbrückung der Lücke zwischen abstrakter Spezifikation und Hardwarebeschreibung wurden zwei Ansätze verfolgt: die Erweiterung der Hardwarebeschreibung in Richtung Hochsprache in Form von *SystemVerilog* auf der einen Seite und die Erweiterung von *C/C++* um Sprachelemente zur Beschreibung von Hardware-Ressourcen und Zeitverhalten in Form von *SpecC* und *SystemC* auf der anderen. Alle drei Sprachen haben heute praktische Bedeutung. In den Bereichen Systemmodellierung und ESL-Synthese hat sich jedoch *SystemC* als Standard durchgesetzt. *SystemC* ist eine Erweiterung von *C++* und im Gegensatz zu *SpecC*, einer Obermenge von ANSI-*C*, objektorientiert. *SystemVerilog* hat als eine der bedeutendsten Verifikationssprachen seine Bestimmung gefunden. Sowohl *SpecC* als auch *SystemC* verfügen über eine synthetisierbare Untermenge (*subset*). Die Reduktion reinen Hochsprachen-Codes auf diese Untermenge ist allerdings sehr aufwendig und birgt kaum Produktivitätsgewinn im Vergleich zur direkten Übersetzung in *VHDL* oder *Verilog*. In den vergangenen Jahren wurde verstärkt an der Vereinfachung dieses Prozesses gearbeitet. Heute sind die ersten kommerziellen *High-Level*-Synthesewerkzeuge verfügbar [syn13] [Inc]. In einer gemeinsam mit der Firma *Rhode & Schwarz* durchgeführten Studie konnten die Leistungsfähigkeit aber auch die Beschränkungen dieser Werkzeuge aufgezeigt werden [Tob10]. Ein hervorragender Überblick über Methoden und Problemstellungen der *High-Level*-Synthese findet sich in [Fin10][Gaj94].

Die Entwicklung neuer Komponenten und deren Synthese ist allerdings oft von untergeordneter Bedeutung. Der überwiegende Teil der Hardwarebausteine eines Systems wird in der Regel wiederverwendet oder in Form von IP-Blocks eingekauft. Hier ergibt sich fast zwangsläufig ein Kompatibilitätsproblem, da zur Durchführung einer Exploration oft Modelle verschiedener Hersteller gemeinsam integriert werden müssen. Daher wurde unter dem Dach der *Open SystemC Initiative*[1] [acc13] eine Interoperabilitätsschicht für Simulationsmodelle geschaffen. Die *Transaction Level Modeling* (TLM)-Bibliothek stellt wie *SystemC* eine Ergänzung zu C++ dar. Die TLM-Bibliothek in Version 2.0 enthält unter anderem verschiedene *Sockets*, ein auf Funktionsaufrufen basierendes Kommunikationsprotokoll und ein standardisiertes *Payload*-Objekt zur Modellierung von Datenübertragungen in speichergesteuerten Systemen. Damit sind theoretisch alle Voraussetzungen zum Einsatz von abstrakten Entwurfsmethoden auf Systemebene gegeben. Praktisch gestaltet sich die Umsetzung allerdings noch immer schwierig. Eines der entscheidenden Probleme ist die Notwendigkeit der Einbindung der gesamten Zulieferkette. Die Bereitstellung von Explorationsmodellen ist die Aufgabe des IP-Herstellers. Dieser hat aber nur geringen Einfluss und Einblick in die Arbeitsabläufe des Systemhauses. Die Entwicklung und Verifikation von Simulationsmodellen ist teils sehr kostspielig. Außerdem fehlt oft entsprechend qualifiziertes Personal, was die Abläufe zusätzlich verzögert.

2.5 Systementwurf mit SystemC und TLM

Wie im vorherigen Abschnitt erwähnt, spielt die Einführung von *SystemC* und deren Erweiterung um Methoden zur Modellierung von Kommunikation auf Transaktionsebene (TLM2.0) eine entscheidende Rolle für die Realisierung von *ESLD*. *SystemC* und *TLM2.0* bilden die Grundlage für alle im Rahmen dieser Arbeit entwickelten Komponenten. Daher sollen die zugrunde liegenden Modellierungskonzepte an dieser Stelle kurz näher erläutert werden. Detailliertere Informationen

1 jetzt *Accelera Systems Initiative*

zu allen Sprachkonstrukten und deren Einsatzzweck können zum Beispiel [Kes12] und [Bla10] oder den von *Accellera* [acc13] erhältlichen Referenzhandbüchern entnommen werden. Sehr empfehlenswert sind darüber hinaus die im Internet frei verfügbaren Tutorien von *Doulos* [dou14].

SystemC ist keine eigenständige Sprache sondern eine Erweiterungsbibliothek zu *C++*, welche die Modellierung von *Hardware*-Komponenten erleichtert. Die *SystemC*-Bibliothek besteht aus:

- einem Simulationskern basierend auf einem ereignisgesteuerten Simulationsalgorithmus, der Parallelverarbeitung sequentiell emuliert. *SystemC*-Simulationen sind daher immer deterministisch.
- einem Komponentenmodell, das die hierarchische Strukturierung eines Entwurfs ermöglicht. Dazu wird eine spezielle Modulklassen (*sc_module*) bereitgestellt, von der durch Vererbung abgeleitet werden kann. *SystemC*-Module entsprechen im erweiterten Sinne *Entities* in *VHDL* und können beliebig verschachtelt werden.
- Methoden zur einfachen Realisierung von Nebenläufigkeit. Mit Hilfe von Makros (*SC_THREAD*, *SC_METHOD*) können *C++*-Funktionen als parallele Prozesse gekennzeichnet und ähnlich wie zum Beispiel in *VHDL* mit einer *Sensitivity*-Liste versehen werden.
- sogenannten *Primitive Channels* zur Modellierung synchronisierter Kommunikation zwischen *SystemC*-Modulen und Prozessen. Wichtige spezialisierte *Channels* für zum Beispiel Signale (*sc_signal*) oder FIFOs (*sc_fifo*) sind vordefiniert und können beliebig erweitert werden.
- speziellen Hardware-Datentypen zur Auflösung unbestimmter und hochohmiger Zustände (*resolved logic*, z.B. *sc_logic*, *sc_logic_vector*).
- einem Konzept zur Modellierung von Zeit.

Durch die beschriebenen Eigenschaften kann *SystemC* zur Beschreibung von Hardware auf *RT*-Ebene eingesetzt werden. Der Schwerpunkt liegt jedoch in der Verwendung auf Systemebene, insbesondere in Kombination mit *TLM 2.0*. Wie in Abbildung 2.6 dargestellt, setzt *TLM* direkt auf *SystemC* auf und erweitert das System um Modellierungsstile und Mechanismen, welche die abstrakte Darstellung von Komponenten und Kommunikation erleichtern.

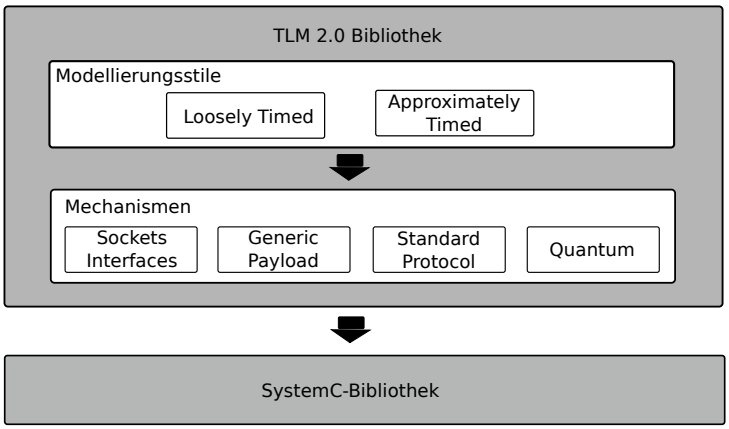


Abbildung 2.6: TLM 2.0 Bibliothek als Erweiterung zu SystemC

Einsatz und Kombination von Modellierungsstilen und Mechanismen sind vom Anwendungszweck (*Use Case*) des zu erstellenden Modells abhängig. Modelle mit losem Zeitverhalten (*Loosely-Timed* (*LT*)) sind für schnelle blockierende Kommunikation ausgelegt. Das Verhalten derartiger Modelle ist oft rein funktional; auf Nebenläufigkeit wird weitestgehend verzichtet. Darüber hinaus synchronisieren sich *LT*-Modelle oft nur in größeren Abständen (Übergabe

des Kontrollflusses an den Simulationskern), um zeitaufwendige Prozessumschaltungen zu minimieren. *TLM* realisiert dies mit Hilfe von *Quantum-Keeper*-Klassen, die es Modellen erlauben der Simulationszeit lokal vorauszueilen. Der *LT*-Modellierungsstil wird eingesetzt, wenn der betrachtete Anwendungsfall hohe Simulationsgeschwindigkeit erfordert und zeitlich Genauigkeit von geringerer Bedeutung ist. Ein typisches Beispiel hier ist die Entwicklung von Software. Modelle für diesen Anwendungsfall müssen die Perspektive des Programmierers auf das System einfangen und werden daher oft auch als *Programmer's View* (*PV*) bezeichnet. Bei der Bewegung großer Speicherinhalte oder der Abarbeitung rechenintensiver Schleifen erfolgt der Speicherzugriff oft direkt als sogenannter *Bypass* über einen vom Ziel (*Target*) bereitgestellten Zeiger (siehe auch *Direct Memory Interface – DMI*). Die Modelle sind funktional und bezüglich aller Speicherlokationen (Steuerregister, Bit-Felder, Speicher) korrekt, liefern aber nur gerade die Genauigkeit, die zum Start eines Betriebssystems erforderlich ist (z.B. *Interupts* in korrekter Abfolge). Der Modellierungsstil *Approximately Timed* (*AT*) wird eingesetzt, wenn zeitliche Genauigkeit eine höhere Bedeutung hat. Dies ist unter anderem für Architekturexploration oder die Analyse von Bandbreite und Kommunikationsverhalten erforderlich. Derartige Modelle werden oft auch als *Architect's View* (*AV*) bezeichnet, da sie die Perspektive des Systemarchitekten abbilden. Der *AT*-Modellierungsstil beinhaltet nicht-blockierende Kommunikation, ein Standardprotokoll zur Modellierung des Zeitverhaltens von Transaktionen mit bis zu vier Synchronisationspunkten pro Übertragung und die Modellierung von Nebenläufigkeit mit *SystemC*-Prozessen, FIFOs und *Event*-Queues.

Das Konzept der *TLM*-Kommunikation wird in Abbildung 2.7 vereinfacht dargestellt. Alle speichergesteuerten Systeme können in *Initiatoren* und *Targets* gegliedert werden. Die *TLM*-Bibliothek enthält *Sockets* und spezialisierte Kanäle (*Channels*) zur Unterstützung der Modellierung der Kommunikation. *Initiator*- und *Target-Sockets* sind durch einen Vorwärts- und einen Rückwärtstransportpfad miteinander verbunden. Die eigentliche Datenübertragung erfolgt durch die Übergabe eines standardisierten *Payload*-Objektes. Dieses enthält Datenfelder, welche unter anderem die Zieladresse, die *Payload*-Daten (Zeiger), die Länge der Übertragung in Byte und die Anzahl der gleichzeitig übertragbaren Bytes (*Streaming*-Weite) beschreiben. Die für den *LT*-Modellierungsstil empfohlene blockierende Kommunikation besteht aus nur einem Funktionsaufruf pro Transaktion, der entlang des Vorwärtspfades ausgeführt wird. Der *Inititator* ruft dabei eine Funktion in der Schnittstelle *tlm_fw_transport_if* (*b_transport*) des *Targets* auf und übergibt eine Referenz auf das Transaktionsobjekt. Ein *return* des *Targets* beendet die Transaktion. Im Falle nicht-blockierender Kommunikation (*AT*-Modellierungsstil), wird das Transaktionsobjekt dagegen mehrfach zwischen *Initiator* und *Target* vor und zurück gereicht. Jede Übergabe markiert einen Protokollschritt und kennzeichnet somit den Fortschritt der Transaktion. Dieser wird im Attribute *phase* des *Payload*-Objektes festgehalten, wodurch diesem eine besondere Bedeutung zukommt.

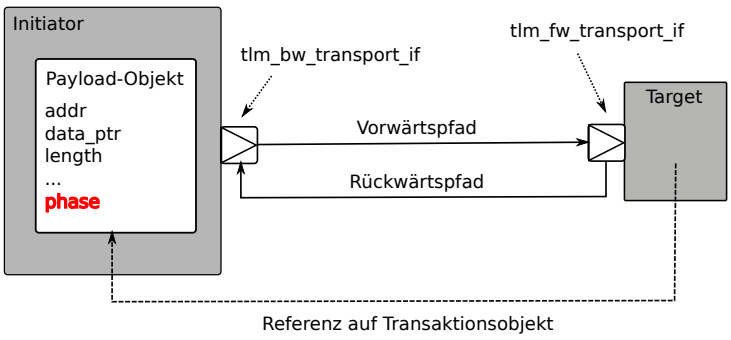


Abbildung 2.7: Konzept der TLM-Kommunikation

Da durch *TLM* nur Modellierungsstile und Mechanismen bereitgestellt werden, die je nach Anwendungszweck kreativ angepasst und kombiniert werden müssen, gestaltet sich die Umsetzung in der Praxis oft schwierig. Dies betrifft insbesondere die *AT*-Modellierung, die trotz unbestrittener

Vorteile noch keine weitreichende Akzeptanz gefunden hat. Besonders schwierig gestaltet sich die Modellierung moderner Mehrkanalprotokolle mit *Split*-Transfers. Auf wichtige Fragen wie etwa die Vereinbarkeit von Interoperabilität, Simulationsgeschwindigkeit und Simulationsgenauigkeit wird in Abschnitt 3.2 näher eingegangen.

3 Effizienter Entwurf von Simulationsmodellen

In diesem Kapitel werden unterschiedliche Aspekte zur Konstruktion von Simulationsmodellen und Virtuellen Plattformen identifiziert und in Hinblick auf den Stand der Technik untersucht. Dazu zählen Aufbau und Struktur von Modellen, TLM-Schnittstellen, Modellierung von Speicherelementen und Verhalten, Verwaltung und Handhabung von Metadaten sowie Modellierung des Energieverbrauchs, Debugging, Analyse und Verifikationsunterstützung.

Ziel ist die Entwicklung einer offenen erweiterbaren Infrastruktur, die sich möglichst nah an existierenden Standards orientiert und die Modellierung von Hardwarekomponenten aus dem Raumfahrtbereich ermöglicht. Abhängig von der Verfügbarkeit werden Lösungen zur Umsetzung in *SoCRocket* ausgewählt, mit neuen Ideen verknüpft und weiterentwickelt oder von Grund auf neu entworfen.

3.1 Aufbau und Struktur von Modellen

3.1.1 Stand der Technik

Es besteht Konsens darüber, dass Verhalten und Kommunikation von TL-Modellen getrennt voneinander implementiert werden sollten [Ini05]. Victor Reyes von *NXP Semiconductors* beschreibt dies in [Rey09] als Grundvoraussetzung für die Wiederverwendbarkeit von IP-Komponenten zur Abdeckung unterschiedlicher Anwendungsfälle und Abstraktionsniveaus. Dieses Konzept wird in *SystemC* und auch TLM2.0 durch die explizite Trennung der Schnittstellendeklaration von der Implementierung der zugehörigen Funktionen unterstützt [Ayn09]. Dadurch kann die Schnittstelle eines Modelles zur Außenwelt im Entwicklungsprozess gleich bleiben. In Abhängigkeit vom gewünschten Abstraktionsniveau ändert sich lediglich die Implementierung. Wie in nahezu allen heute verfügbaren Virtuellen Plattformen ersichtlich (Abschnitt 2.3.2), ist es oft zweckmäßig, Modelle noch tiefer zu strukturieren. Oft werden zusätzlich Verhalten und Speicher voneinander getrennt. *Synopsys* verwendet zur Modellierung von Speichern SCML [Inc13], *Cadence* ein auf einer offenen Registerimplementierung basierendes *Framework* (SC_REGISTER). Ähnlich der Trennung zwischen Verhalten und Kommunikation, erlaubt die Abspaltung des Speicherteils eine effizientere Wiederverwendung auf unterschiedlichen Abstraktionsniveaus. Außerdem kann die Implementierung zwischen unterschiedlichen Modellen geteilt werden. Infrastrukturkomponenten wie z.B. Registerfiles werden entweder durch Instantiierung oder durch Vererbung in das eigentliche TL-Modell eingebunden. Die Interaktion in Richtung der Infrastruktur wird durch Programmierschnittstellen realisiert. Andererseits kommuniziert die Infrastruktur mit dem Verhalten der Komponenten über Rückruffunktionen, sogenannte *Callbacks*, die individuell registriert werden können.

3.1.2 Strukturierung mit Bibliotheksbasisklassen

Die Kernkomponenten zur Modellierung von Raumfahrtanwendungen sind in einer gemeinsamen *Hardware*-Bibliothek integriert (*GRLIB* / siehe Abschnitt 1.3). Die Komponenten verfügen über einen gemeinsamen Konfigurationsmechanismus, teilen Typdeklarationen zur Schnittstellenbeschreibung und generische Speicher. Diese Gemeinsamkeiten können in *SystemC*-Basisklassen gekapselt und so in das TL-System übernommen werden. Die in Abbildung 3.1 dargestellte Struktur hat sich dazu als zweckmäßig erwiesen und bildet das Grundgerüst aller im Weiteren vorgestellten Komponenten.

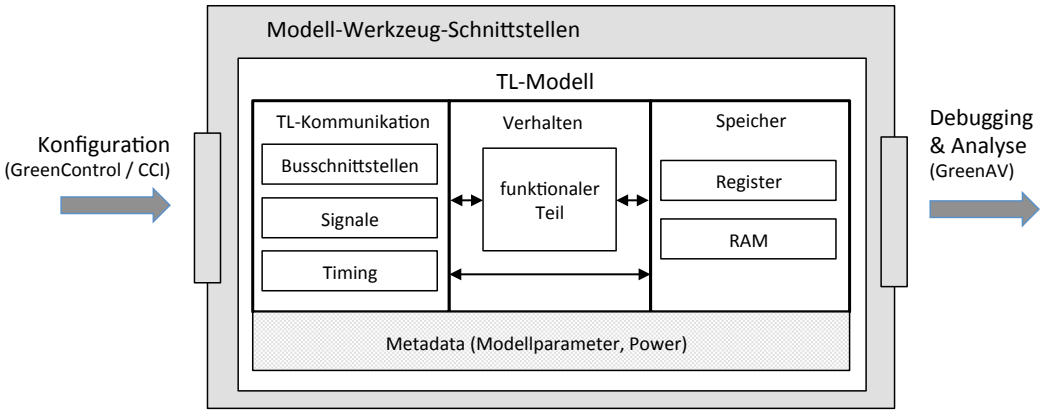


Abbildung 3.1: Aufbau von SoCRocket Komponenten

Die Gliederung umfasst vier Sektionen: Kommunikation, Verhalten, Speicher und Metadaten. Die Struktur entsteht implizit durch Vererbung von Bibliotheksbasisklassen, die generische Modellteile zur besseren Wiederverwendbarkeit kapseln. Alle Sektionen setzen auf einer gemeinsamen Modell-Werkzeugschnittstelle auf, die in Abschnitt 3.7 erläutert wird. Zur Realisierung der TL-Kommunikation wurden verschiedene Basisklassen entwickelt, von denen bei Bedarf mehrfach abgeleitet werden kann. So können durch einfache Vererbung Komponenten mit verschiedenen AMBA2.0-Schnittstellen und spezialisierten Signalports gebildet werden (Abschnitt 3.2). Ebenfalls durch Vererbung werden Registerfiles und Speicher eingebunden (Abschnitt 3.3). Der Entwickler wird dadurch entlastet und kann sich auf die Modellierung des Verhaltens konzentrieren (Abschnitt 3.4). Eine Übersicht der empfohlenen Basisklassen kann Tabelle 3.1 entnommen werden. Konfigurationsparameter und Informationen zum Energieverbrauch werden als Metadaten im Modell abgelegt und im Kontext des Gesamtsystems gespeichert. Dadurch sind alle Parameter global adressierbar und können bei Bedarf zur Laufzeit geändert werden (Abschnitt 3.5).

Typ	Basisklasse	Beschreibung
Busschnittstellen	AHBMaster	AHB-*Master*-Klasse blockierend (LT) und nicht blockierend (AT) AHB-*Master* API siehe 3.2.2 (Abb. 3.3)
	AHBSlave	AHB-*Slave*-Klasse blockierend (LT) und nicht blockierend (AT) AHB-*Slave* API siehe 3.2.2 (Abb. 3.4)
	APBDevice	APB-*Slave*-Klasse blockierend (LT); siehe 3.2.3
Signale	signalkit	Signalports für in/out/inout, vektorisierte Signale mux/demux siehe 3.2.5
Timing	clock_device	API zur Annotation der Taktzeit; dient der Ableitung der Verzögerung
Register	gr_device	Modell erhält ein *GreenReg* Registerfile, dass bei Bedarf direkt an einen APB-*Socket* gebunden werden kann; siehe 3.3.2
RAM	mem_device	Generischer Speicher als *Array* oder *Hashmap* siehe 3.3.3

Tabelle 3.1: Übersicht Bibliotheksbasisklassen

3.2 TL-Kommunikation

Ein nicht unerheblicher Teil der Komplexität bei der Modellierung von Simulationsmodellen besteht in der Abstraktion der Bus- und Signalkommunikation. Zur Schaffung einheitlicher Schnittstellen und Erleichterung der Erstellung von Komponenten wurde die notwendige Funktionalität in Basisklassen gekapselt. Dazu wird die bereits in Abschnitt 2.5 vorgestellte Interoperabilitätsschicht TLM2.0 genutzt. TLM2.0 ist keine *out-of-the-box*-Lösung zur Implementierung von Busschnittstellen, es ist vielmehr eine lose Sammlung von Werkzeugen und Techniken. Im Zentrum stehen ein standardisiertes *Payload*-Objekt und ein mehrphasiges Standardprotokoll zur Modellierung von Datenübertragungen (siehe Abschnitt 2.5). Zur Abbildung spezieller Protokolle können *Payload* und Standardprotokoll beliebig erweitert und modifiziert werden. Die Implementierung der zugehörigen Logik und Zustandsautomaten auf *Master*- und *Slave*-Seite obliegt dem Benutzer.

Für *SoCRocket* soll eine Lösung für TL-Kommunikation gefunden werden, welche die transparente Modellierung von Komponenten mit sowohl losem *Timing* (LT-Modus), als auch näherungsweise akkuraten *Timing* (AT-Modus) erlaubt.

3.2.1 Stand der Technik

Die meisten heute verfügbaren TL-Simulationsmodelle nutzen die durch die Erweiterungen von TLM2.0 gegebenen Möglichkeiten nur unzureichend aus. In der Regel werden, entsprechend dem LT-Modellierungsstil, nur blockierende Schnittstellen angeboten [Pul11]. Nichtblockierende Modelle mit näherungsweise akkuratem *Timing* (TLM2.0 AT) sind eher selten. Eine Suche auf *TLMCentral* [tlm14] liefert 272 LT- aber nur 20 AT-Modelle. Die Nutzung des AT-Modellierungsstils für *Performance Modeling* und *Design Space Exploration* auf ESL wird durch Martin Streubühr von der Universität Erlangen-Nürnberg in [Str09] erläutert. Es wird ein System beschrieben, das die zeitlichen Effekte konkurrierender Zugriffe auf geteilte Ressourcen modelliert. Als Anwendungsbeispiel wird ein *Motion*-JPEG-Dekodierer verwendet. Die Anwendung wird in aktororientiertem *SystemC* beschrieben und automatisch auf den MPSoC abgebildet. Leider werden keine Details der AT-Schnittstellenbeschreibung offenbart, es wird jedoch erwähnt, dass sich die Verwendung nichtblockierender Modelle nur dann lohnt, wenn Komponenten gleichzeitig auf Busse zugreifen. Gunar Schirner von der *University Irvine* beschreibt in [Sch06a] verschiedene Ansätze zur Modellierung von AMBA-Bussen auf Transaktionsebene. Neben einem zyklenbasierten Busmodell (BFM), werden ein abstraktes TLM-Modell und ein arbitriertes TLM-Modell beschrieben; letzteres verteilt Nutzertransaktionen auf mehrere Bustransaktionen. Eine ähnliche Vorgehensweise ist auch für die näherungsweise Modellierung mit TLM2.0 denkbar. Seit Februar 2011 bietet *Carbon Design Systems* eine offene Erweiterung für TLM2.0 an, mit der verschiedene AMBA-Protokolle [Sys14] auf unterschiedlichen Abstraktionsebenen modelliert werden können. Zur Konstruktion näherungsweise akkurater Schnittstellen definiert das System Erweiterungen für die TLM-*Payload* und das Standardprotokoll. Es werden jedoch lediglich *Sockets* bereitgestellt. Die Implementierung der Zustandsautomaten und die Entscheidung darüber welche Erweiterungen effektiv genutzt werden, obliegt dem Anwender. Die Vorarbeiten zur Entwicklung des *Carbon*-AHB-*Design-Kit* wurden unter dem Dach von *GreenSoCs* [gre13] am Institut für Theoretische Informatik der TU Braunschweig geleistet. Das System wurde unter anderem in [Poc11] durch Wissenschaftler der TU Berlin zur Validierung einer Methodik für *Model Checking* auf Transaktionsebene eingesetzt. Das dazu implementierte Beispielsystem besteht aus zwei AHB-*Master* und zwei *AHB-Slave* Komponenten, die über einen Bus miteinander verbunden sind. Leider werden keine Angaben bezüglich Genauigkeit, Geschwindigkeit oder Detailgrad der Implementierung gemacht.

3.2.2 AMBA High-Performance Bus (AHB)

AHB ist das Rückgrat der Kommunikation im GRLIB-System. Alle Komponenten die häufig und mit hoher Frequenz auf Speicher oder Peripherie zugreifen, verfügen über eine AHB-Schnittstelle. AHB ist außerdem das am weitesten verbreitete Protokoll der AMBA-Protokollfamilie [Lim11]. Es wurde 2003 als Teil von AMBA 2.0 eingeführt. AHB unterstützt mehrere *Master* an einem Bussegment, *Split*-Transfers und *Pipelining*. Übertragungen bestehen aus einer Adressphase und einer Datenphase und nehmen daher mindestens zwei Takte in Anspruch, sofern der Empfänger keine zusätzlichen Wartezyklen einfügt.

AHB-Modellierungskonzept

Zur Modellierung von Komponenten mit AHB-Schnittstelle wurden drei *SystemC*-Basisklassen entwickelt: *AHBDevice*, *AHBSlave* und *AHBMaster*. In GRLIB enthält jede AMBA2.0-Schnittstelle einen *Plug & Play*-Konfigurationsregistersatz [Gai10] bestehend aus acht 32-Bit-Registern. Diese teilen sich in vier Register mit herstellerspezifischen Angaben zur Identifikation der Komponente und vier Basis-Adressen-Register (BAR) zur Beschreibung des Adressraums. Die Konfigurationsregister werden durch den verbundenen Router (z.B. AHBCTRL), entsprechend der Gerätenummer (*hindex*) in einen fest definierten Speicherbereich abgebildet und können per Software ausgelesen werden. Der Mechanismus erlaubt die automatische Generierung von Adressierungstabellen und das konfigurationsabhängige Laden von Gerätetreibern beim Systemstart. *SoCRocket* kapselt GRLIB-Konfigurationsregister in der Klasse *AHBDevice*, die darüber hinaus Zugriffsfunktionen für alle relevanten Bit-Felder zur Verfügung stellt. In der Regel wird *AHBDevice* nicht direkt zur Erzeugung neuer Komponenten verwendet, sondern indirekt mit Hilfe von *AHBSlave* oder *AHBMaster* eingebunden (Abb. 3.2).

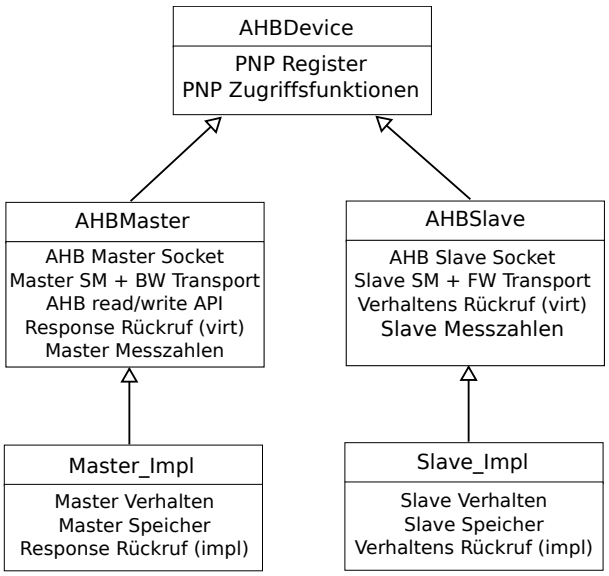


Abbildung 3.2: Konstruktion und Vererbung von AHB-Schnittstellen in *SoCRocket*

AHBMaster stellt dem Nutzer einen TLM/AHB-*Master-Socket* zur Verfügung. Die Klasse enthält den zur Modellierung des Protokolls notwendigen Zustandsautomaten, die Transportfunktion für den TLM-Rückwärtspfad, eine Programmierschnittstelle zur Einleitung von Buszugriffen, Rückruffunktionen zur Übermittlung der Ergebnisse (*Response*) und einen Messzahlcontainer mit Zugriffszählern. Dementsprechend stellt der *AHBSlave* einen TLM/AHB-*Slave-Socket* bereit. Dieser enthält ebenfalls einen Zustandsautomaten, Transportfunktionen für den TLM-Vorwärtspfad, eine Rückruffunktion zur Einkopplung des Modulverhaltens und einen Messzahlcontainer. Eine *SoCRocket*-Komponente, die eine AHB-Busschnittstelle benötigt, erbt diese von der *AHBMaster*- oder der *AHBSlave*-Basisklasse.

Abbildung 3.3 zeigt die für *Master*-Komponenten bereitgestellte Programmierschnittstelle. Die Schnittstelle enthält Lese- und Schreibfunktionen mit unterschiedlichen Signaturen. Neben der Zieladresse (*addr*) muss der Nutzer wenigstens einen Zeiger auf einen reservierten Speicherbereich mit *Payload*-Daten (**data*) und die Länge des Transfers (*length*) übergeben. Optional kann eine Initialverzögerung angegeben werden (*&delay*). Die Initialverzögerung wird zur Transferverzögerung der Transaktion addiert und am nächstmöglichen Synchronisationspunkt konsumiert. Darüber hinaus kann der Nutzer atomare Transaktionen erzwingen (*Bus Locking – is_lock*). Die Referenz *&cacheable* wird durch das adressierte TLM-*Target* auf eins gesetzt, wenn die Zieldaten in einem Cache gepuffert werden dürfen. Mit Hilfe der Referenz *&response* ist es außerdem möglich, den Transferstatus einer Transaktion direkt im Verhalten auszuwerten. Abhängig vom Abstraktionsniveau der Komponente löst *AHBMaster* eine blockierende (LT) oder nichtblockierende Transaktion (AT) aus. Im LT-Modus geben die *ahbread* und *ahbwrite* Funktionen erst nach Abschluss der kompletten Transaktion die Kontrolle an das Modul zurück. Im AT-Modus geschieht dies am Ende der AHB-Adressphase. Der *Master* darf dann unmittelbar eine neue Transaktion starten. Der Eingang von Lesedaten und damit der Beginn der AHB-Datenphase wird mit Hilfe einer Rückruffunktion (*response_callback*) signalisiert. Die Rückruffunktion ist virtuell und sollte im erbenden Modul (*Master_Impl*) überladen werden. Zur Nutzung des verzögerungsfreien TLM-Debugpfades stehen mit *ahbread_dbg* und *ahbwrite_dgb* spezielle Zugriffsfunktionen bereit. Diese werden in der Systemsimulation durch den Debugger und den OS-Emulationsmodus der CPU zum transparenten Zugriff auf alle Speicherelemente genutzt.

```
1  // AHB Lesezugriff (vereinfachte und volle Signatur)
2  void ahbread(uint32_t addr, unsigned char * data, uint32_t length);
3  void ahbread(uint32_t addr, unsigned char * data, uint32_t length,
4  sc_time &delay, bool &cacheable, bool is_lock,
5  tlm::tlm_response_status &response);
6
7  // AHB Schreibzugriff (vereinfachte und volle Signatur)
8  void ahbwrite(uint32_t addr, unsigned char * data, uint32_t length);
9  void ahbwrite(uint32_t addr, unsigned char * data, uint32_t length,
10 sc_time &delay, bool is_lock, tlm::tlm_response_status &response);
11
12 // Debug Lese-/Schreibzugriff
13 void ahbread_dbg(uint32_addr, unsigned char * data, uint32_t length);
14 void ahbwrite_dbg(uint32_addr, unsigned char * data, uint32_t length);
15
16 // TLM Response-Rueckruffunktion (AT-Modus)
17 virtual void response_callback(tlm::tlm_generic_payload * trans) {};
```

Abbildung 3.3: AHB Master - Programmierschnittstelle

Die Programmierschnittstelle der Basisklasse *AHBSlave* besteht aus nur einer rein abstrakten Rückruffunktion (*exec_func*), die von der implementierenden Klasse (*Slave_Impl*) überladen werden muss. Die Funktion wird beim Eintreffen einer neuen Transaktion ausgelöst. Im LT-Modus geschieht dies unmittelbar, im AT-Modus zum BEGIN_REQ Zeitpunkt. Die Funktion *exec_func* sollte auf nichtblockierende Weise implementiert und die übergebene Verzögerung (*&delay*) um den Gesamtwert der durch zusätzliche Wartezyklen benötigten Zeit erhöht werden. Die Basisverzögerung für den Transfer wird in *AHBSlave* anhand der Gesamtmenge der Daten und der Anzahl pro Takt übertragener Bytes abgeschätzt.

```
1  virtual uint32_t exec_func(tlm::tlm_generic_payload &gp,
2  sc_time &delay, bool debug=false) = 0;
```

Abbildung 3.4: AHB Slave - Programmierschnittstelle

Durch die beschriebene Kapselung der Busschnittstellen ist die Modellierung des Datentransfers für die Entwicklung von Komponenten weitestgehend transparent und beschränkt sich auf die Nutzung von Programmierschnittstellen und die Reaktion auf Rückruffunktionen. Die Dauer und Verzögerung eines Datentransfers werden durch die Zustandsautomaten der Basisklassen und den Weg der Transaktion durch das System bestimmt. Zusätzlich kann der *Slave* durch die Erhöhung der Transferzeit Wartezyklen simulieren.

AHB/TLM-Protokollmodellierung

Das AHB-Protokoll wird abhängig vom Abstraktionsniveau auf blockierende (LT) oder nicht-blockierende Weise (AT) modelliert. Dazu sind zwei grundsätzliche Schritte erforderlich: die Abbildung von RTL-Signalen auf die TLM-*Payload* und die Assoziation von Protokollpunkten mit TLM-Phasen.

Der erste Schritt ist für beide Modi äquivalent. Tabelle 3.2 zeigt die für die Modellierung von AHB getroffenen Zuordnungen. Auf Transaktionsebene erfolgt dabei keine Unterscheidung zwischen *Slave*-Schnittstellen von Bussen und *Slave*-Schnittstellen von Peripheriekomponenten. Einige der Signale, wie die Transferadresse (HADDR) oder die Datenbusse (HWDATA/HRDATA) können direkt auf das TLM-*Payload*-Objekt abgebildet werden, da dieses entsprechende Felder vorhält. Für protokollspezifische Zusatzinformationen, wie den in AHB vorgesehenen Speicherschutzmechanismus (HPROT) kann die *Payload* erweitert werden. Dazu wurde die Erweiterungsklasse *amba_ext* eingeführt. Alle Felder von *amba_ext* sind ignorierbar und verfügen über eine sichere Standardeinstellung, so dass die Kompatibilität zu anderen TLM2.0-Komponenten gegeben ist. Verfügt eine Transaktion über keine Erweiterung für HSIZE, so wird für den Transfer die volle Busbreite (32 Bit) angenommen. Wird die Erweitung *cacheable* nicht unterstützt, dann betrachtet das System die entsprechende Komponente als nicht pufferbaren Speicherbereich. Alle verbleibenden Signale, wie HSEL oder HREADY, werden nicht explizit modelliert. HSEL wird durch die Auswahl eines *Slaves* für die Weiterleitung einer Transaktion dargestellt. HREADY signalisiert die Bereitschaft des *Slaves* zum Empfang von Daten und wird durch das Einfügen von Wartezyklen realisiert. Der Vorgang der Busanforderung (HBUSREQ) und Buszuweisung (HGRANT) entspricht auf Transaktionsebene dem Weiterreichen der *Payload* vom *Master* zum Bus.

RTL Signal	TLM Abbildung	Beschreibung
HADDR	tlm_generic_payload.addr	Transfer-Adresse
HTRANS	nicht erforderlich	Transfer-Typ (NONSEQ, SEQ, IDLE, BUSY)
HWRITE	tlm_generic_payload.command	Lesen oder Schreiben
HSIZE	amba_ext::hsize	Bytes per Transfer (Takt)
HBURST	tlm_generic_payload.length	Burst-Typ (FIXED, INCR, WRAPPING)
HPROT	amba_ext::cacheable	Speicherschutzindikator
HLOCK	amba_ext::lock	Bus-Locking
HWDATA/HRDATA	tlm_generic_payload.data	Schreib-/Lesedaten
HBUSREQ, HGRANT	nicht erforderlich	Bus-Request-Handling
HSEL, HREADY	nicht erforderlich	Slave-Auswahl
HRESP	tlm_generic_payload.response	Response-Status

Tabelle 3.2: AHB Payload Abbildung

LT- und AT-Modus des *SoCRocket*-Systems unterscheiden sich grundlegend in der Art und Weise der Übertragung des *Payload*-Objektes zwischen *Master* und *Slave*. Im LT-Modus existieren nur zwei *Timing*-Punkte (siehe Abbildung 3.5). Das Diagramm zeigt einen einzelnen Lese- oder Schreibzugriff ohne Wartezyklen. Der erste Punkt markiert den Beginn der Transaktion und somit den Beginn der AHB-Adressphase. Dieser Punkt entspricht dem Aufruf der *ahb_read/write*

Schnittstellenfunktion, verschoben um eine etwaige Initialverzögerung. Der zweite Punkt markiert das Ende der AHB-Datenphase und die Rückgabe der Programmkontrolle an den Verhaltensteil der Komponente (*return* von *ahb_read/write*). Die Verwendung blockierender Kommunikation reduziert die Anzahl der Synchronisationspunkte pro Transaktion und ermöglicht eine hohe Simulationsgeschwindigkeit. In der Regel werden Speicherzugriffe durch nur einen direkten Funktionsaufruf realisiert. Die Verzögerungszeiten der Transferkomponenten im Kommunikationspfad werden akkumuliert und erst am Ende der Transaktion durch den *Master* konsumiert. Der hohen Geschwindigkeit steht ein Verlust an Simulationsgenauigkeit gegenüber. Da nur Anfangs- und Endzeitpunkt der Transaktion bekannt sind, können Pipelineeffekte nicht berücksichtigt werden. Für eine schnelle Systemexploration oder die Entwicklung von Software ist dies allerdings unerheblich.

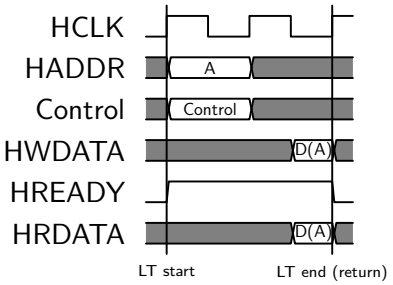


Abbildung 3.5: AHB - LT Timing Modellierung

Größere Simulationsgenauigkeit kann durch die Verwendung einer höheren Anzahl an Synchronisationspunkten, in Kombination mit nichtblockierender Kommunikation, erreicht werden. Der AT-Modus verwendet dazu die vier Phasen des TLM-Standardprotokolls: BEGIN_REQ, END_REQ, BEGIN_RESP und END_RESP. Die beiden ersteren markieren Beginn und Ende der AHB-Adressphase, die letzteren Beginn und Ende der AHB-Datenphase. Abbildung 3.6 illustriert diese Näherung am Beispiel eines einzelnen Lese- oder Schreibtransfers (NONSEQ). Der Transfer benötigt zwei Wartezyklen. Der BEGIN_REQ Zeitpunkt entspricht, wie auch im LT-Modus, dem Zeitpunkt des Aufrufes der *ahb_read/write* Schnittstellenfunktion einschließlich einer eventuellen Initialverzögerung. END_REQ markiert die Übernahme des Kommandos durch den *Slave*. Der Punkt BEGIN_RESP zeigt den Beginn der Datenübertragung an. Dabei wird nur die effektive Übertragung unter Ausschluss von Wartezyklen berücksichtigt. Die Grafik zeigt, dass die Schreibdaten des Transfer bereits zu einem früheren Zeitpunkt angelegt werden. Der Bus ist zu diesem Zeitpunkt jedoch blockiert, da der *Slave* nicht bereit ist ($HREADY$=Null). Dieser Punkt ist durch das AHB-Protokoll bestimmt und entspricht näherungsweise dem Ende der vorangegangenen Adressphase.

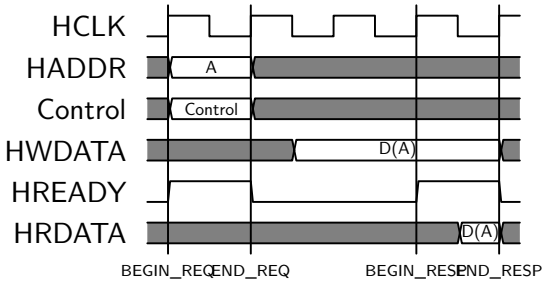


Abbildung 3.6: AHB - AT Timing Modellierung (Einzeltransfer)

AHB-Bursts bestehen aus mehreren aufeinanderfolgenden Adress- und Datenphasen (Abb. 3.7). Für die Simulation auf AT-Ebene ist es nicht sinnvoll, Anfang und Ende jeder dieser Phasen zu markieren. Das *SoCRocket*-System nutzt BEGIN_REQ zur Kennzeichnung des Beginns der ersten Adressphase eines Bursts. END_REQ markiert das Ende der letzten Adressphase. Dementsprechend kennzeichnet BEGIN_RESP den Beginn der ersten Datenphase und END_RESP das

Ende der letzten Datenphase des Transfers. Es ist dabei unerheblich, ob es sich um einen inkrementellen Burst oder einen Burst mit fester Länge handelt. Art und Dauer des Transfers werden durch die Anzahl pro Takt übertragener Bytes und die Länge der Daten bestimmt. Der Aufwand zur Simulation eines Bursts ist damit nicht höher als der Aufwand für einen Einzeltransfer. Im Vergleich zur vollständigen Markierung aller Takte werden $(N-1)*4$ (N: Länge des Bursts) Synchronisationspunkte eingespart, was einen erheblichen Geschwindigkeitsvorteil verspricht. Der Geschwindigkeitsgewinn wird durch die Vernachlässigung des *Intra-Burst-Timings* erkauft. Dadurch können *Split*-Transfers nicht ohne weiteres modelliert werden. Dies entspricht dem Konzept der TL-Modellierung, weniger relevante Eigenschaften zum Zwecke der Simulationsbeschleunigung zu abstrahieren. Sollte der Entwurf eine genauere Modellierung der *Split*-Funktionalität von AHB erfordern, empfiehlt sich die Einführung zweier zusätzlicher Phasen: SPLIT_START und SPLIT_END. Dies wird auch in [Sys14] vorgeschlagen. Der Router (AHBCTRL) kann diese Phasen zur Verlängerung der AHB-Datenphase verwenden oder, je nach gewünschter Genauigkeit, neu arbitrieren. Das Beispiel macht ebenfalls deutlich, dass der *Slave* im Falle eines Bursttransfers BEGIN_RESP vor END_REQ sendet. Dies entspricht nicht dem TLM-Standardprotokoll, ist jedoch der Verständlichkeit der Implementierung und dem Informationsgehalt der erhobenen Simulationsdaten zuträglich, da so die Eckdaten für die Übertragung der Adressen und Daten direkt zur Verfügung stehen.

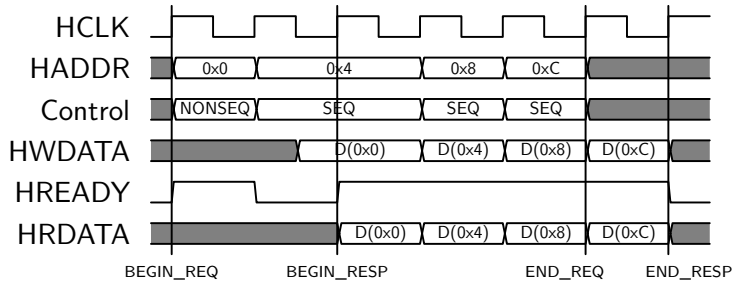


Abbildung 3.7: AHB – AT Timing Modellierung (Burst-Transfer)

Die getrennte Markierung von Address- und Datenphase des AHB-Protokolls ermöglicht die Berücksichtigung von Pipeline-Effekten in der Systemsimulation. Abbildung 3.8 verdeutlicht die Überlappung aufeinanderfolgender Transfers. Beide im Beispiel dargestellten Transaktionen sind Einzeltransfers mit jeweils einem Wartezyklus. Die erste Adresse (A1) wird an der ersten steigenden Taktflanke nach Anlegen des Kommandos vom *Slave* übernommen. Das TL-Modul sendet zu diesem Zeitpunkt END_REQ (ER1). Der *Master* ist nun zum Senden der nächsten Transaktion bereit. Dies geschieht ähnlich wie in zyklengenauer Simulation unmittelbar (BR2). Das zweite Kommando wird zwei Takte später, nach Ablauf der Wartezeit und Ende der Datenphase der ersten Transaktion (D1) akzeptiert. Die Gesamttransferzeit beider Transaktionen (BR1 - ERSP2) beträgt fünf Takte. Dies entspricht der durch das Protokoll vorgegebenen Verzögerung und damit dem in einer RTL-Simulation zu erwartenden Ergebnissen.

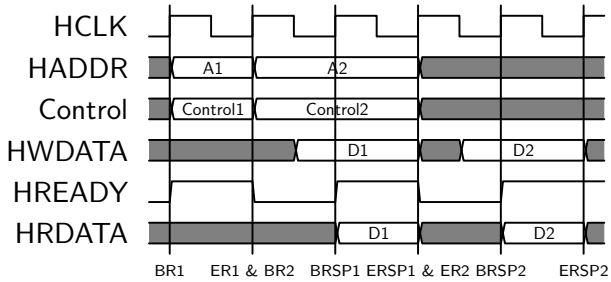


Abbildung 3.8: AHB – AT Timing Modellierung (Überlappende-Transfers)

In der Regel verzichten nicht-zyklengenaue AHB-Implementierungen auf die Modellierung des Pipelinings. Dadurch entsteht ein systematischer Fehler, den die vorgestellte Lösung beseitigt.

Ohne die Modellierung der Pipeline muss für jede Transaktion eine feste Transferzeit angenommen werden. Diese Transferzeit setzt sich aus der Länge des Bursts, der Anzahl der Wartezyklen und einem zusätzlichen Adressierungstakt zusammen. In vollem Betrieb wird der Adressierungstakt durch die Pipeline verdeckt. Dadurch ergibt sich eine pessimistische Abschätzung der Transferzeit. Bei direkter Abfolge von Einzeltransaktionen ohne Wartezyklen kann so ein Fehler von bis zu 100% entstehen.

3.2.3 AMBA Peripheral Bus (APB)

APB-Modellierungskonzept

APB-Busse verfügen über nur einen Master, im LEON-System ist dies die AHB/APB-Busbrücke (APBCTRL). Eine Basisklasse zur Entwicklung von *Master*-Komponenten ist daher nicht erforderlich. Zur Entwicklung von APB-*Slaves* stellt *SoCRocket* die Klasse *APBDevice* bereit. Diese enthält einen Konfigurationsregistersatz, bestehend aus zwei 32-Bit-Registern, wovon eines der Identifikation der Komponente und eines der Definition des Adressraumes dient [Gai10]. *APBDevice* übermittelt die Inhalte dieser Register beim Simulationsstart an die Busbrücke. Diese bildet die Konfigurationsdaten aller verbundenen Komponenten in ihren Konfigurationsbereich am oberen Ende des zugeordneten Speichers ab. Die Konfigurationsdaten dienen dem Aufbau des APB-Adressdekodierers und können durch die CPU per Software ausgelesen werden.

APB-Schnittstellen werden in der Praxis oft zur Ansteuerung von Kontrollregistern verwendet. Zur Vereinfachung der Modellierung derartiger Register wird in *SoCRocket GreenReg* verwendet (Abschnitt 3.3.2). APB-Komponenten mit Kontrollregistern erben daher zusätzlich von Klasse *gr_device* (Abb. 3.9). Dadurch werden der Komponente Container und Zugriffsfunktionen zur Modellierung von Speicherelementen bereitgestellt.

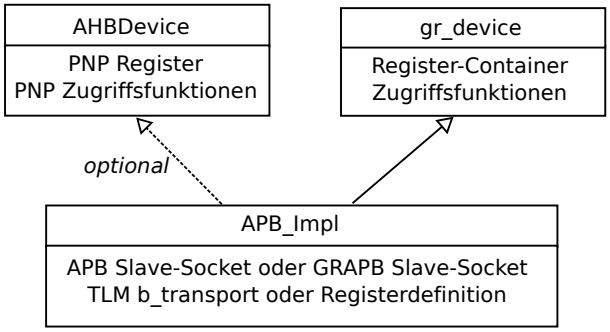


Abbildung 3.9: Konstruktion und Vererbung von SoCRocket APB Schnittstellen

Anders als bei der Modellierung von AHB-Schnittstellen werden APB-*Sockets* nicht in einer Basisklasse gekapselt. Die Instanziierung des *Sockets* obliegt dem Nutzer und wird in der implementierenden Klasse durchgeführt (*APB_Impl*). Dabei bestehen zwei Optionen. Soll die Schnittstelle mit Kontrollregistern gekoppelt werden, muss ein *GreenReg*-APB-*Socket* instanziiert werden. Der *Socket* wurde im Rahmen dieser Arbeit entwickelt [Mey10] und stellt eine Verbindung zwischen *GreenReg* und den TLM/AMBA-*Sockets* her. Der Quellcode des *Sockets* befindet sich im Verzeichnis *contrib*. Einfache Beispiele für die Instanziierung finden sich in den Komponenten *gp_timer* und *irqmp* (siehe Anlage B). Werden keine Kontrollregister benötigt, müssen ein TLM/APB-*Slave-Socket* instantiiert und die zugehörige blockierende Transportfunktion implementiert werden.

APB/TLM-Protokollmodellierung

Das APB-Protokoll ist für die Ansteuerung von Peripherie- und I/O-Komponenten mit geringer Bandbreite ausgelegt. Es verfügt über keinen Burstmodus und keine Spezialsignale für Speicherschutz und Pufferung. Diese Funktionen werden durch die AHB-Schnittstelle des APB-*Masters*

(die Busbrücke) realisiert. In aller Regel darf der komplette APB-Speicherbereich nicht in Caches gepuffert werden. Zur Abbildung der RTL-Signale des Busses auf die Transaktionsebene wird nur die TLM-*Payload* benötigt. Die AHB-Erweiterung *amba_ext* wird durch APB-Komponenten ignoriert. Die daraus resultierende Signalzuordnung ist in Tabelle 3.3 dargestellt. Die Bussignale zur Auswahl des *Slaves* werden nur implizit modelliert. PSEL markiert den Zeitpunkt der Übertragung der Transaktion an den APB-*Slave*. PENABLE steht in fester Relation zu PSEL und kann abstrahiert werden.

RTL Signal	TLM Abbildung	Beschreibung
PADDR	tlm_generic_payload.addr	Transfer-Adresse
PWRITE	tlm_generic_payload.command	Lesen oder Schreiben
PRDATA/PWDATA	tlm_generic_payload.data	Schreib-/Lesedaten
PSEL	nicht erforderlich	Decoder Auswahlsignal
PENABLE	nicht erforderlich	Decoder Enable-Signal

Tabelle 3.3: APB Payload Abbildung

Jeder APB-Transfer nimmt genau zwei Takte in Anspruch: *Enable* und *Select*. Es existieren keine überlappenden Übertragungen. Dadurch kann das Protokoll auf Transaktionsebene auf sehr einfache und genaue Weise dargestellt werden. Wie Abbildung 3.10 zeigt, genügen dafür zwei *Timing*-Punkte, die hinreichend durch blockierende Kommunikation darstellbar sind. Eine Unterscheidung zwischen LT- und AT-Abstraktion, in Hinsicht auf die Busschnittstelle, ist daher nicht erforderlich.

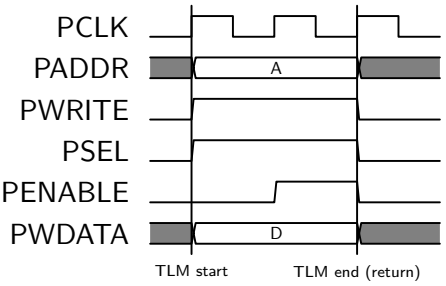


Abbildung 3.10: APB - Timing Modellierung

3.2.4 AMBA eXtensible Interface Bus (AXI)

In den vorangegangen Abschnitten wurde gezeigt, dass die AMBA2-Protokolle AHB und APB mit geringem Aufwand auf Transaktionsebene modelliert werden können. Der geringe Aufwand begründet sich in der Natur dieser Protokolle. Jedes verfügt über nur einen Kanal zur Übertragung von Daten zwischen *Master* und *Slave*. Dies gilt auch für das AHB-Protokoll, da AHB-Adressbus und -Datenbus in fester Relation zueinander stehen. AMBA2 ist gegenwärtig der *de-facto*-Standard für Datenbusse in Weltraum-DPUs. Aller Voraussicht nach, wird sich dies in naher Zukunft ändern, da die Ansprüche an die Flexibilität und Bandbreite von Verbindungsstrukturen auch in diesem Anwendungsfeld kontinuierlich steigen. Sehr wahrscheinlich werden zukünftige DPU-Komponenten über AMBA3-Schnittstellen verfügen. AMBA3 erweitert die AMBA-Busfamilie um den AMBA *eXtensible Interface Bus* (AXI). AXI ist ein modernes Mehrkanal-Protokoll mit getrennten Adressierungs- und Datenphasen, die im Gegensatz zu AHB in keinem festen zeitlichen Verhältnis zueinander stehen. Das Protokoll definiert fünf unabhängige Kanäle: *Read Address*, *Write Address*, *Read data*, *Write Data* und *Write Response*, was *full-duplex* Lese- und Schreiboperationen ermöglicht. Alle Kommunikation in AXI sind burst-basiert. Unabhängig von der Länge des Bursts existiert nur ein Adress-/Kommandotakt. Das Protokoll unterstützt sowohl mehrere ausstehende Transaktionen als auch die Umsortierung von Transaktionen zur Laufzeit.

AXI/TLM-Modellierungsvorschlag

Im Rahmen dieser Arbeit wurde ein Vorschlag zur Modellierung des AXI-Protokolls erarbeitet. Die beschriebene Vorgehensweise wurde in einem gemeinsamen Projekt mit der Firma *Cadence* zur Modellierung eines DDR3-Speichercontrollers erprobt [Sch13b]. Ziel ist die Bereitstellung einer Lösung unter alleiniger Nutzung des TLM2.0-Basisprotokolls. Die Schwierigkeit besteht dabei in der Erzielung einer hohen Modellierungsgenauigkeit bei gleichzeitiger Erhaltung der Kompatibilität zu beliebigen TLM2.0-Standardkomponenten. Darüber hinaus soll die Lösung, ähnlich wie die vorgeschlagenen Abbildungen für APB und AHB, einfach einsetzbar sein und keine unnötige Komplexität aufweisen. Trotz der beschriebenen getrennten Kanäle für Lese- und Schreiboperationen wird daher für *Master* und *Slave* nur jeweils ein TLM-*Socket* vorgesehen.

Tabelle 3.4 zeigt die Abbildung von RTL-Signalen auf die TLM-*Payload* und die *Payload*-Erweiterung *amba_ext*. Ähnlich wie für AHB und APB können Zieladressen, sowie Lese- und Schreibdaten durch die entsprechenden Felder der *Payload* modelliert werden. Das Erweiterungsobjekt enthält eine Identifikationsnummer zur eindeutigen Zuordnung der Transaktion. Dadurch wird die im AXI-Protokoll vorgesehene Priorisierung und Umgruppierung von Operationen ermöglicht. Länge und Weite des Bursts werden durch die Erweiterungen *length* und *size* beschrieben. Der *Write-Response*-Kanal wird abstrahiert und auf das *Response*-Statusfeld der TLM-*Payload* abgebildet.

RTL Signal	TLM Abbildung	Beschreibung
ARADDR/AWADDR	tlm_generic_payload.addr	Transfer-Adresse
RDATA/WDATA	tlm_generic_payload.data	
	tlm_generic_payload.length	
WRESP	tlm_generic_payload.response	Write Response
ARID/AWID	amba_ext::id	Transaktions-ID
ARLEN/AWLEN	amba_ext::length	Länge des Bursts
ARSIZE/AWSIZE	amba_ext::size	Bytes per Takt
ARBURST/AWBURST	amba_ext::burst_type	Typ des Bursts
ARLOCK/AWLOCK	amba_ext::lock	Bus-Locking
ARCACHE/AWCACHE	amba_ext::cache	Cachetyp (pufferbar)
ARPROT/AWPROT	amba_ext::prot	Speicherschutzindikator
ARVALID/AWVALID	nicht erforderlich	Bereit-Signale des Masters
ARREADY/AWREADY	nicht erforderlich	Bereit-Signale des Slaves
WSTRB, WRLAST	nicht erforderlich	Byte-Enables, Burstende

Tabelle 3.4: AXI Payload Abbildung

Abbildung 3.11 zeigt einen AXI-Lesetransfer für ein Datenwort. BEGIN_REQ markiert den Zeitpunkt des Anlegens der Leseadresse an den *Read-Address*-Kanal (*ARADDR*). Die Übernahme des Kommandos durch den *Slave* wird durch END_REQ markiert. Die dargestellte Transaktion hat zwei Wartezyklen, danach werden die Lesedaten bereitgestellt. Zu diesem Zeitpunkt sendet der *Slave* BEGIN_RESP. Der *Master* beendet den Transfer mit END_RESP, nach vollständiger Übertragung aller Daten. Da AXI, im Gegensatz zu AHB, Kommandos unabhängig von der Burstlänge in nur einem Takt überträgt, ist sichergestellt, dass BEGIN_RESP in jedem Fall nach END_REQ gesendet wird. Diese Abfolge entspricht dem TLM-Standardprotokoll.

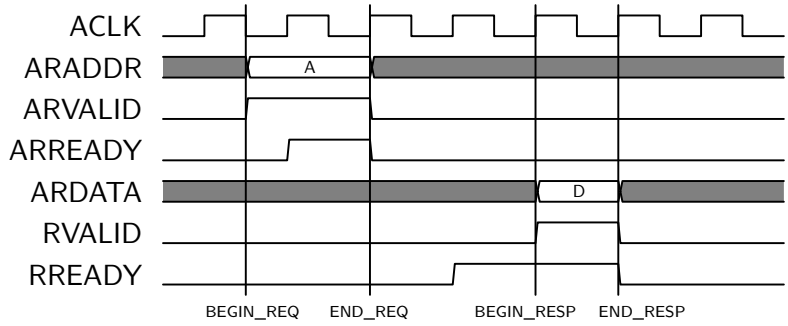


Abbildung 3.11: AXI - Timing Modellierung (Lese-Burst)

AXI unterstützt Burstlängen von bis zu 16 Takten (256 Takte in AXI4). Oft ist der *Slave* nicht in der Lage, die gesamten Lesedaten in ununterbrochener Reihenfolge zu liefern. In diesem Fall werden zusätzliche Wartezyklen eingefügt (RVALID=0), was ein Problem für die vorgeschlagene Markierung der Burstgrenzen darstellt. Wie bereits erwähnt sendet der *Slave* BEGIN_RESP zur Markierung des ersten Datentaktes. Ohne Verwendung zusätzlicher TLM-Phasen besteht keine Möglichkeit, den *Master* über nachträglich eingefügte Wartezyklen zu informieren. Der *Master* kann dadurch END_RESP nur auf Grundlage des BEGIN_RESP-Zeitpunktes, eigener Wartezyklen (RREADY=0) und der Länge des Bursts berechnen. END_RESP wird also zu früh gesendet. Der dadurch entstehende Fehler, kann aber in der Simulation ausgeglichen werden, weil der *Slave* die Anzahl der durch ihn eingefügten Wartenzyklen kennt und deshalb die nachfolgende Transaktion entsprechend verzögern kann. Die beschriebene Vorgehensweise wird in Abbildung 3.12 dargestellt. Die Darstellung zeigt zwei aufeinanderfolgende Lesezugriffe: A1 und A2. Die erste Operation ist ein Burst der Länge zwei. Für die Übertragung der zugehörigen Lesedaten (D1.1 und D1.2) werden drei Takte benötigt, da der *Slave* vor dem Senden des zweiten Datenpaketes (D1.2) einen Wartezyklus einfügt. Da der *Master* darüber hinaus nicht informiert werden kann, sendet dieser END_RESP einen Takt zu früh (!ERSP1!). Ohne den beschriebenen Korrekturmechanismus zum Ausgleich des Fehlers würde der *Slave* die Daten des zweiten Zugriffes (D2) ebenfalls um einen Takt zu früh senden. Eine weitere mögliche Fehlerquelle besteht in der Annahmeverzögerung der Lesedaten (RREADY=0 Zyklen). Die dadurch entstehende zusätzliche Verzögerung kann dem *Slave* durch eine Verzögerung der END_RESP-Phase übermittelt werden.

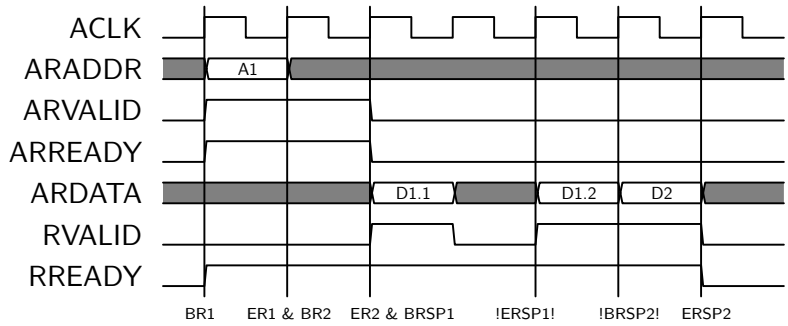


Abbildung 3.12: AXI - Timing Modellierung (Lese-Burst Korrektur)

Auch die Modellierung von Schreibzugriffen birgt einige Schwierigkeiten. Die BEGIN_RESP-Phase wird durch den *Slave* generiert, kann also nur zur Markierung des Transferendes verwendet werden, da der *Slave* nicht weiß, wann der *Master* die Schreibdaten an den *Write-Data*-Kanal anlegt. Unter der Prämisse der Verwendung des TLM-Standardprotokolls, kann man sich hier nur mit einer Schätzung behelfen. Ist der *Write-Data*-Kanal zum Zeitpunkt des Anlegens der Schreibadresse unbenutzt, so kann eine feste Verzögerung zwischen Adresse und Daten angenommen werden. In vielen Fällen ist diese Verzögerung eine statische Eigenschaft des *Masters*, die in einem TLM-Modell mit Hilfe eines Konfigurationsparameters eingestellt werden kann. Ist der Bus zum Kommandozeitpunkt nicht frei, so wird die Datenübertragung relativ zum Ende der

vorangegangenen Transaktion verzögert. In diesem Fall jedoch, sendet der *Slave* BEGIN_RESP zu früh. Dieses Problem tritt ebenfalls auf, wenn der *Master* nicht alle Schreibdaten in einem lückenlosen Burst liefern kann (WVALID=0 Takte). Der *Master* kann den so entstandenen Fehler durch die END_RESP-Phase ausgleichen. Die relative Verzögerung von END_RESP gegenüber BEGIN_RESP muss dabei der Summe aller Haltezyklen des *Masters* entsprechen. Der END_RESP-Zeitpunkt und die Summe der Wartezyklen des *Slaves* (WREADY=1) bestimmen das Ende des Transfers. Abbildung 3.13 verdeutlicht dies an einem Beispiel. Zum Zeitpunkt des Kommandos A1 ist der *Write-Data*-Kanal frei. Bei Annahme einer festen Verzögerung von einem Takt zwischen Adresse und Daten, einem Wartetakt und zwei Takten für den Transfer der Daten (D1.1 und D1.2) sendet der *Slave* BEGIN_RESP zum angegebenen Zeitpunkt (!BRSP1!). Da der *Master* in der Übertragung einen Takt pausieren musste (WVALID=0), entspricht dies nicht dem wahren Ende des Transfers. Zur Korrektur des Fehlers schickt der Master ein abschließendes END_RESP mit einer Verzögerung von einem Takt.

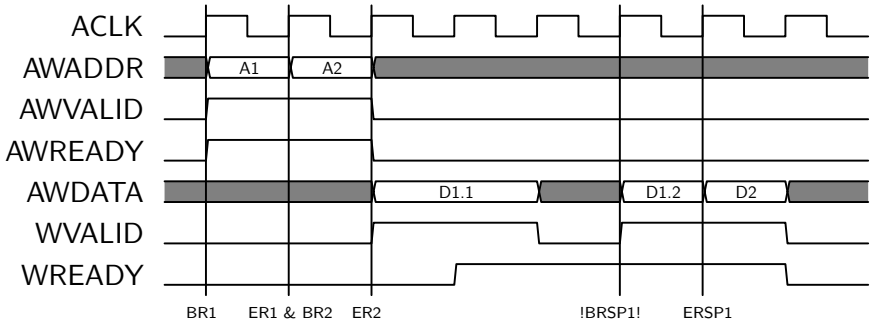


Abbildung 3.13: AXI - Timing Modellierung (Schreib-Burst)

Mit Hilfe des beschriebenen Ausgleichsmechanismus können durch Daten-*Splits* verursachte Abweichungen bei der Modellierung von Punkt-zu-Punkt-Transfers mit dem TLM2.0-Standardprotokoll ohne das Einfügen zusätzlicher Protokollphasen vollständig eliminiert werden.

3.2.5 Signale und Interrupts

In der Regel besitzen *SystemC*-Modelle neben den TLM-Schnittstellen zur Modellierung der Busanschlüsse verschiedene Eingabe- und Ausgabeports (*sc_in/out*). Diese werden zur Herstellung von *Interrupt*-Verbindungen, zur Modellierung von *Reset*-Leitungen oder Ansteuerung sonstiger Kontrollfunktionen verwendet. Der Standardweg zur Verbindung von *SystemC*-Ports verschiedener Module in Virtuellen Plattformen sind *SystemC*-Signale (*sc_signals*). Dies ist eine sehr hardwarenahe Beschreibung, wie sie ähnlich auch in VHDL oder *Verilog* verwendet wird. Ähnlich wie in den traditionellen Hardwarebeschreibungssprachen kann man *SystemC*-Signale auch innerhalb von Komponenten zur Synchronisation von *Threads* einsetzen. Daher sind sie besonders für Hardwareentwickler einfach verständlich und werden gern benutzt. Diese spezielle Hardwarenähe ist aber nicht in jedem Fall erwünscht. Besonders in den höheren Abstraktionsebenen ist Simulationsgeschwindigkeit sehr wichtig. TLM-Entwickler versuchen dazu, die Anzahl der Synchronisationspunkte der *Threads* in ihren Entwürfen und damit die Aufrufe des *SystemC-Scheduler* zu reduzieren. Da jede Änderung eines *SystemC*-Signals einen Kernelaufruf zur Folge hat, ist deren Verwendung für den gewünschten Zweck ungeeignet. Aus diesem Grunde ist eine abstraktere Modellierung von Signalkommunikation erforderlich. Gegenwärtig gibt es noch keinen allgemein anerkannten Lösungsansatz für dieses Problem. Ein vielversprechender Vorstoß wird in [Swa12] beschrieben. TLM_WIRE befindet sich jedoch noch in der Konzeptionierungsphase. Nach meinem besten Wissen ist die bisher einzige praktische verfügbare Lösung *GreenSignal*, ein Projekt der *Open Source*-Initiative *GreenSoCs* [gre13]. *GreenSignal* verwendet TLM-*Sockets* zur Modellierung von Eingabe- und Ausgabeports. Alle Signale eines Systems werden durch eine einzige TLM-*Payload*-Erweiterung dargestellt. Jede Änderung eines Signals (eines Feldes der *Payload*-Erweiterung), wird an alle Komponenten propagiert. In den Komponenten registriert

der Nutzer die Signale/Felder auf die er reagieren möchte. Alle übrigen werden ignoriert. Der *GreenSignal*-Ansatz ist interessant, da er die eigentlichen Verbindungen abstrahiert und jeder Komponente, im Stile eines *Full Crossbar*, jedes Signal zur Verfügung stellt. Für die Verständlichkeit des Systems und eine leichtere Überführbarkeit auf ein niedrigeres Abstraktionsniveau (z.B. RTL), ist es jedoch günstiger Signalverbindungen explizit darzustellen. Darüber hinaus erfordert *GreenSignal* die Erzeugung und Verwaltung von TLM-Objekten, was für den angestrebten Zweck einen unnötigen Mehraufwand bedeutet.

Aus diesen Gründen wurde für *SoCRocket* ein neues Werkzeug zur Modellierung von Signalverbindungen geschaffen. Ziel der Implementierung ist die Kombination der hardwareähnlichen Nutzbarkeit von *SystemC*-Signalen, mit der Effizienz direkter Funktionsaufrufe. Der entsprechende Quellcode befindet sich im Unterverzeichnis *common* (siehe Anlage B). Der Codeblock in Abbildung 3.14 verdeutlicht die einfache Handhabung von *SoCRocket*-Signalen am Beispiel einer einfachen Signalquelle. Das implementierende Module muss die *Header*-Datei *signalkit.h* inkludieren (Zeile 1) und das Makro SK_HAS_SIGNALS aufrufen (Zeile 5). Dadurch werden das Modul bei der Infrastruktur registriert und alle notwendigen Datentypen und Zugriffsfunktionen bereitgestellt. Das Modul definiert einen Ausgang vom Typ *Integer* mit dem Namen *out* (Zeile 8). Der Zugriff auf den Port kann genau wie bei SystemC-Ports durch eine Zuweisung (Zeile 19) oder einen Aufruf der Port-Funktion *write* erfolgen.

```
1  #include "signalkit.h"
2
3  class source : public sc_module {
4
5    SK_HAS_SIGNALS(source);
6    SC_HAS_PROCESS(source);
7
8    signal<int>::out out;
9
10   // Constructor
11   source(sc_module_name nm) :
12     sc_module(nm), out("out") {
13
14      SC_THREAD(run);
15   }
16
17   void run() {
18     // ...
19     out = i;
20     // ...
21   }
22 }
```

Abbildung 3.14: TLM Signal - Source Module

Ähnlich einfach gestaltet sich die Kodierung von *Target*-Komponenten (Abb. 3.15). Der Eingabeport des Moduls ist in Zeile 7 instantiiert (*in*). Der Konstruktor verdeutlicht die Registrierung einer Rückruffunktion. Im Beispiel wird die Funktion *onsignal* bei jeder Änderung von *in* aufgerufen.

```
#include "signalkit.h"

class dest : public sc_module {

  SK_HAS_SIGNALS(dest);

  signal<int>::in in;

  // Constructor
  dest(sc_module_name nm) :
    sc_module(nm), in(&dest::onsignal, "in") {
   }

  // Signal handler for input in
  void onsignal(const int &value, const sc_time &time) {
    // do something
  }
}
```

Abbildung 3.15: TLM Signal - Destination Module

Zur Herstellung der Verbindung zwischen *Initiator* und *Target* stellt *SoCRocket* die *connect* Methode bereit (siehe Abb. 3.16). Darüber hinaus enthält das System Methoden zum *Multiplexing* und *De-Multiplexing* von Signale. Des weiteren ist es möglich, mehrere Signale gleichen Typs durch Angabe einer Signalnummer zu bündeln. Eine genaue Beschreibung aller Einsatzmöglichkeiten kann dem *SoCRocket*-Handbuch entnommen werden [Sch12b].

```
#include "source.h"
#include "dest.h"

int sc_main(int argc, char *argv[]) {

  source src;
  dest dst;

  connect(src.out, dst.in);

   ...
  return 0;

}
```

Abbildung 3.16: TLM Signal - Connecting signals

Die in Kapitel 4 vorgestellten Kernkomponenten zum Entwurf robuster eingebetteter Systeme, verwenden den hier vorgestellten Mechanismus zur Modellierung von *Interrupt*-Verbindungen, zur *Reset*-Verteilung und zur Realisierung von Cache-Kohärenz (DBUS-*Snooping*).

3.3 Modellierung von Speicherelementen

3.3.1 Stand der Technik

Einer der ersten und wichtigsten Schritte zur Erstellung eines Plattformprototypen ist die Modellierung aller direkt und indirekt adressierbaren Speicherelemente. Trotz der Essenzialität dieses Problems existiert bis heute kein allgemein akzeptierter Standard. Die meisten Plattformen nutzen hierfür eigene, teilweise proprietäre Infrastruktur. *Synopsys* setzt auf die *SystemC*

Modeling Library (SCML), deren Quellcode durch *Coware* im Jahre 2006 öffentlich zugänglich gemacht wurde [Inc13]. *Cadence*-Plattformen erstellen Modelle basiert auf SC_REGISTER, einem Register- und Speichermodellierungswerkzeug, dass bisher noch nicht frei verfügbar ist. Die bisher einzige wirklich offene *Open Source*-Lösung ist *GreenReg* [gre13]. *GreenReg* wurde ebenfalls unter dem Dach der *GreenSoCs*-Kooperationsplattform entwickelt und stellt eine Weiterentwicklung des von *Intel* freigegebenen *Design- und Register-Framework* (DRF) dar. Dem Autor ist kein Projekt bekannt, in dem *GreenReg* bisher umfassend eingesetzt und erprobt wurde. *SoCRocket* nimmt somit eine Vorreiterrolle zur Etablierung eines offenen Standards ein.

3.3.2 Speichermodellierung mit GreenReg

Um *GreenReg* für die Modellierung von Registern in *SoCRocket* nutzen zu können, wurde das System erweitert und an die verwendeten AMBA/TLM2.0-*Sockets* angepasst. Alle dafür entwickelten *Patches*, sowie der Quellcode der *Sockets* befinden sich im Verzeichnis *contrib* der VP (siehe Anlage B). *GreenReg* stellt einen wichtigen Pfeiler der *SoCRocket*-Infrastruktur dar. Es wird in allen Bibliothekskomponenten zur Modellierung von Registern und Registerfeldern eingesetzt. Dadurch hat *GreenReg* eine besondere Bedeutung für das Verständnis des Systems und soll an dieser Stelle kurz erläutert werden.

Die *GreenReg*-Klasse *gr_device* wird in *SoCRocket* als eine Basisklasse zur Komponentenbildung integriert. Alle Module mit Steuerregistern erben von dieser Klasse und erhalten dadurch Zugriff auf Datentypen und Funktionen zum Aufbau von Registerbänken, Registern und Bit-Feldern. Der übergeordnete Registercontainer wird durch die Infrastruktur mit einem für unser System angepassten TLM-*Socket* (*AMBA-greenreg_socket*) verbunden. *Socket* und Registercontainer kommunizieren mittels blockierender TLM-Transaktionen. Das folgende Beispiel verdeutlicht den durch den Einsatz von *GreenReg* erzielbaren Produktivitätsgewinn bei der Modellierung von Komponenten am Beispiel des Mehrzweck-*Timers gp_timer*. Abbildung 3.5 zeigt ein Steuerregister des *Timers*, dass mit Hilfe von *GreenReg* modelliert und an einen *Socket* verbunden werden soll.

31 … 10	9	8	7 … 3	2 … 0
Reserved	DF	SI	IRQ	TIMERS

Tabelle 3.5: GPTimer Steuerregister (config)

Dazu sind folgende Schritte erforderlich:

1. Inkludieren des Socket-Headers:

   ```
   #include "greenreg_ambasockets.h"
   ```

2. Modul von der *GreenReg*-Basisklasse ableiten:

   ```
   class gp_timer : public gs::reg::gr_device
   ```

3. Dem System durch Aufruf eines Makros mitteilen, dass die neue Komponente *GreenReg-Callbacks* benötigt.

   ```
   GC_HAS_CALLBACKS();
   ```

4. Den TLM-Socket für das Registerfile erzeugen:

   ```
   gs::reg::greenreg_socket<gs::amba::amba_slave<32> > my_sock;
   ```

5. Im Konstruktor des Modules *GreenReg* initialisieren:

   ```
   // Erzeugen eines Registerfiles mit bank_size Bytes und Wort-Adressierung
   gr_device(name, gs::reg::ALIGNED_ADDRESS, bank_size, NULL)
   ```

6. Die Initialisierung von *GreenReg* generiert einen Zeiger auf einen Registercontainer (r). Dieser Zeiger wird zur Initialisierung des *Sockets* verwendet. Darüber hinaus müssen Start- und Endadresse, sowie Protokoll und Abstraktionsniveau angegeben werden.

```
// Initialisierung des Sockets und Registrierung der Registerbank
my_sock("sock", r, start_address, end_address, amba::amba_APB,
amba::amba_LT, false);
```

7. Innerhalb der Registerbank ein neues Register (config) erzeugen:

```
r.create_register("config", "GP_Timer Steuerregister (config)", offset,
type (z.B. gs::reg::STANDARD_REG), Initialwert, Schreibmaske,
Registerweite (32bit));
```

8. Bit-Felder innerhalb des neuen Registers registrieren (Tabelle 3.5):

```
r[config].br.create("TIMERS", 0, 3);
r[config].br.create("IRQ", 3, 5);
r[config].br.create("SI", 8, 1);
r[config].br.create("DF", 9, 1);
```

9. Rückruffunktionen für Bit-Felder definieren (Beispiel SI-Feld):

```
GR_FUNCtion(gp_timer, SI_written);
GR_SENSITIVE(r[config].br[SI].add_rule(gs::reg::POST_WRITE,
SI_written, gs::reg::NOTIFY);
```

10. Implementierung der Funktion SI_written. Die Funktion wird nach jeder Schreiboperation auf Feld SI aufgerufen.

3.3.3 Simulationsspeicher

Neben Registern und Registerbänken hat die einfache Modellierung von Speichern große Bedeutung für den praktischen Einsatz einer VP. *SoCRocket* stellt dafür zwei Klassen bereit, die flexibel erweitert und modifiziert werden können: *array_memory* und *map_memory*. Die entsprechenden Implementierungen befindet sich im Verzeichnis *models/memory* der Plattform (siehe Anlage B). Wie durch die Benennung angedeutet unterscheiden sich *array_memory* und *map_memory* bezüglich der Organisation der zu speichernden Daten. Der Speicher *array_memory* verwendet ein statisches Feld (C/C++ Array) und eignet sich besonders zur Modellierung kleiner und dicht besiedelter Speicher. Zur Modellierung großer und dünn besetzter Speicher sollte *map_memory* verwendet werden. Der Speicher *map_memory* basiert auf einer *hashmap* aus C++/STL. Zur Adressierung der Daten wird die Zugriffsadresse als Schlüssel verwendet, wodurch sich sehr leicht sehr große Speicherbereiche abbilden lassen.

Innerhalb der Komponentenbibliothek werden die beschriebenen Speicher zur Modellierung von I/O, PROM, SRAM und SDRAM verwendet. Die Speicher sind zur Anbindung über den *MCTRL*-Speichercontroller bestimmt (siehe Abschnitt 4.3.3). Alle derartigen Speicher müssen zur Unterstützung des *Plug & Play*-Mechanismus von der Bibliotheksbasisklasse *mem_device* abgeleitet werden. Die Klasse *mem_device* dient der Identifizierung des Speichers und seiner Eigenschaften im System und enthält keine Adresseinstellungen (Tabelle 3.6).

Attribut	Beschreibung
device_type m_type	enum { ROM, I/O, SRAM, SDRAM }
uint32_t m_banks	Anzahl der Bänke bei SRAM oder SDRAM
uint32_t m_bsize	Größe je Bank (alle Bänke gleich groß)
uint32_t m_bits	Zugriffsweite (Bits pro Wort)
uint32_t m_cols	Speicher-Spalten pro Zeile (nur SDRAM)

Tabelle 3.6: Attribute der Basisklasse *mem_device*

Zusätzlich zu den generischen Speichern zur Modellierung von Komponenten enthält das System einen SRAM-Speicher mit AHB-*Slave*-Schnittstelle (*abh_mem*). Dieser wird in Abschnitt 4.3.5 näher beschrieben.

3.4 Verhaltensmodellierung

3.4.1 Stand der Technik

Die größte Herausforderung bei der abstrakten Modellierung von SoC-Komponenten besteht in der Darstellung des Verhaltens (*Behavior*). Dies kann abhängig vom Anwendungszweck auf unterschiedlichste Weise geschehen. Verhalten ist dabei nicht nur Funktion, sondern umfasst auch Zeit und Verzögerung. Zur Modellierung des Verhaltens von Hardwarekomponenten in ESL können beliebige Hochsprachen und verschiedene MoCs eingesetzt werden. Gängige Varianten sind Datenflussgraphen, *Discrete Event*- oder *Continuous Time*-Beschreibungen, Kahn-Prozessmodelle oder synchrone/reaktive Modelle [Bro10]. Als Sprachen kommen zum Beispiel *Matlab*, *SpecC*, *Handel-C*, *Esterel*, *SystemVerilog* oder C/C++ zum Einsatz. Für den Entwurf von SoC-Komponenten in Virtuellen Plattformen hat sich in den vergangenen Jahren jedoch die funktionale Beschreibung mit *SystemC* durchgesetzt. *SystemC* ist eine Erweiterung zu C/C++, welche die Beschreibung typischer Hardwarebausteine wie Ports, Signale und Fifos erleichtert. Eine Einführung in *SystemC* und die zugehörige *Transaction Level Modeling*-Bibliothek TLM2.0 wurde in Abschnitt 2.5 gegeben. Die große Akzeptanz und Verbreitung der Sprache gab den Ausschlag für den Einsatz in *SoCRocket*.

3.4.2 Modellierung von SoCRocket-Komponenten mit SystemC

Die Grundvoraussetzung für die Akzeptanz von *SoCRocket* in der europäischen Raumfahrtindustrie ist die Verfügbarkeit geeigneter Simulationsmodelle. Zum Aufbau einer entsprechenden Modellbibliothek müssen daher wichtige Kernkomponenten, die bisher nur als VHDL-Code vorliegen abstrakt mit Hilfe von *SystemC*/TLM modelliert werden. Die in Abbildung 3.17 dargestellte Vorgehensweise wurde im Rahmen einer Masterarbeit entwickelt, die der Autor 2010 an der TU Braunschweig betreuen durfte [Mey10]. Im folgenden werden die dabei gewählten Schritte erläutert. Die Methodik ist auf den Entwurf neuer Komponenten ohne RTL-Vorlage übertragbar. In diesem Falle entfällt Schritt 2.

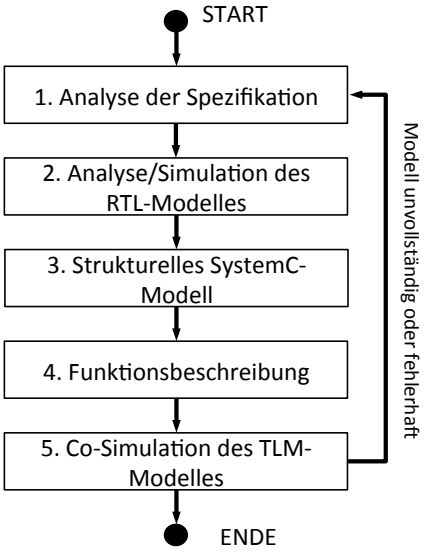
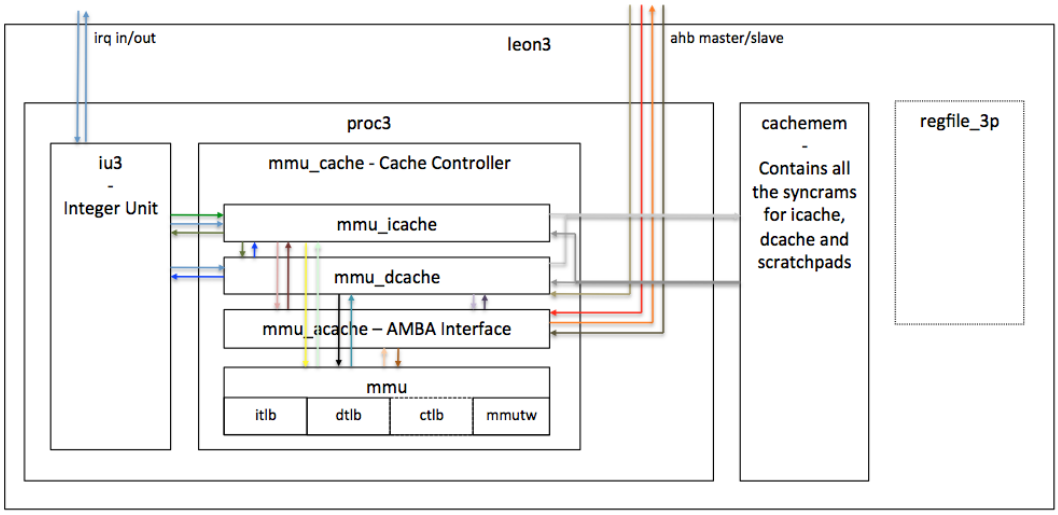


Abbildung 3.17: Überführung von RTL-Komponenten aus GRLIB nach SystemC/TLM2.0

Analyse der Spezifikation (1)

Vor der Modellierung einer Komponente auf Transaktionsebene muss ihre Funktionalität detailliert bekannt sein. Idealerweise ist eine vollständige und exakte Beschreibung der Funktionalität in der Spezifikation gegeben, die als erster und wichtigster Anhaltspunkt für die Analyse dient. Die Herangehensweise an die Aufarbeitung der enthaltenen Information ist abhängig von Art und Umfang des zu modellierenden Moduls. Komplexe und umfangreiche Komponenten, sind in ihrer Gesamtheit schwierig zu erfassen und sollten daher zunächst auf ihre Kernbestandteile reduziert werden. Für die Konstruktion von TLMs sind zuerst die Schnittstellen von besonderer Bedeutung, da diese direkt oder indirekt alle Operationen auslösen. Während bei komplexen Komponenten auch die Schnittstellen zunächst auf ihren Kern reduziert werden müssen, können einfache Module – ausgehend von ihren Schnittstellen – schnell und vollständig modelliert werden. In speicheraddressierten Systemen werden den einzelnen Komponenten Registeradressen zugeordnet. Relativ zu einer Basisadresse definiert die Spezifikation ein Registerfile, in das alle relevanten Funktionen abgebildet sind. Über diese Register ist die Konfiguration und Steuerung des Moduls zur Laufzeit möglich. Zusätzlich zu den Registern enthält die Schnittstelle einer Komponente Signale, deren taktgenaue Koordination der Kommunikation mit anderen Komponenten in der Regel deutlich abstrahiert werden kann (z.B. Interruptleitungen oder Reset). Auf Grundlage der Schnittstellen kann eine Liste der zu implementierenden Funktionseinheiten angefertigt werden. Diese sollte erste Aufschlüsse über die spätere Segmentierung geben und in eine Zeichnung überführt werden (Abb. 3.18). Zusätzliche Informationen können aus der VHDL-Beschreibung des Modells gewonnen werden. Dabei handelt es sich um spezifische Details, wie die Reaktion auf Ausnahmesituationen oder das Verhalten im uninitialisierten Zustand.



Bus	Description	From	To
ici	Instruction Cache In	iu3	icache
ico	Instruction Cache Out	icache	iu3, dcache
dci	Data Cache In	iu3	dcache
dco	Data Cache Out	dcache	iu3, dcache
ahbi	AMBA Master IN	leon3	acache
ahbo	AMBA Master OUT	mmu_acache	leon3
ahbsi	AMBA Slave IN	leon3	acache
ahbso	AMBA Slave OUT Vector	leon3	acache
crami	Cache RAM IN	icache, dcache	cachemem
cramo	Cache RAM OUT	cachemem	icache, dcache

Bus	Description	From	To
mcii	Memory Contr. Instr. IN	icache	acache
mcio	Memory Contr. Instr. OUT	acache	icache
mcdi	Memory Contr. Data IN	icache	acache
mcdo	Memory Contr. Data OUT	acache	dcache
mcmmi	Mem. Contr. Mem Manager IN	mmu	acache
mcmmo	Mem. Contr. Mem Manager OUT	acache	mmu
mmuici	Mem. Manager Icache IN	icache	mmu
mmuico	Mem. Manager Icache OUT	mmu	icache
mmudci	Mem. Manager Dcache IN	dcache	mmu
mmudco	Mem. Manager Dcache OUT	mmu	dcache

Abbildung 3.18: Beispiel für eine Strukturskizze nach Analyse der Spezifikation (LEON-Prozessor)

Analyse/Simulation der Hardwarebeschreibung (2)

Die Funktionalität ist theoretisch in der Spezifikation (z.B. Benutzerhandbuch) vollständig beschrieben, weist aber praktisch oft Lücken auf. Daher ist eine detaillierte Analyse des RT-Modelles unerlässlich. Der VHDL-Code der modellierten GRLIB-Komponenten ist in der GPL-Version

nahezu unkommentiert und enthält eine Vielzahl undokumentierter Implementierungstricks. Komplizierte Codepassagen müssen zum besseren Verständnis aufgearbeitet und sukzessive mit eigenen Kommentaren versehen werden. Dies ist ein sehr aufwendiger Prozess, hilft aber, eine solide Grundlage für die spätere funktionale und zeitliche Analyse zu schaffen.

Je besser das Verständnis des zu modellierenden Entwurfs, desto größer der Spielraum zur Abstraktion!

Ähnlich wie bei der Analyse der Spezifikation kann zur Untersuchung des VHDL-Modelles von den Schnittstellen ausgegangen werden. Die Kommunikation erfolgt über Ports, die unterschiedliche parallele Prozesse auslösen. Diese Prozesse modellieren die Funktionalität und können durch interne Signale untereinander synchronisiert werden. Die VHDL-Prozesse sollten, wenn möglich, nicht in das *SystemC*/TLM-Design übernommen werden. Umstrukturierung und Sequentialisierung der enthaltenen Logik führt meist zu einer Steigerung der Simulationsgeschwindigkeit, bei vernachlässigbaren Genauigkeitseinbußen. Zur Unterstützung der Analyse der Hardwarebeschreibung kann das VHDL-Modell simuliert werden. Im Falle der in dieser Arbeit betrachteten GRLIB-Modelle war die Spezifikation bezüglich des Zeitverhaltens oft unvollständig. Darüber hinaus mussten zusätzliche Informationen über Grenz- und Ausnahmefälle (z.B. Fehlerverhalten) gewonnen werden. Zur Simulation war eine *Testbench* erforderlich, welche die Eingänge der Modelle durch das Anlegen von Testvektoren stimuliert. Nach der Simulation wurden die internen Signale und Ausgänge der Entwürfe mit dem erwarteten Verhalten verglichen. Die *Testbench* wurde komplett in *SystemC* implementiert und mit Hilfe von *Transaktoren* co-simuliert. Der beschriebene Aufbau bildete die Grundlage, für die in Abschnitt 3.8.1 beschriebene Testumgebung (Testschnittstelle: *direct r/w*).

Strukturelles Modell (SystemC-Skelett) (3)

Nach der Analyse der Spezifikation und des Hardwaremodelles sind Funktionalität und Zeitverhalten der zu modellierenden Komponente bekannt. Nun kann das strukturelle Skelett des zukünftigen *SystemC*/TLM-Modelles konstruiert werden. Dies wird durch die in *SoCRocket* bereitgestellte Infrastruktur erheblich erleichtert. Wie in Abschnitt 3.1 beschrieben, können Modelle einfach durch Vererbung von Bibliotheksbasisklassen gebildet werden. Das Modell erhält dadurch Bus- und Signalschnittstellen, Register, Speicher und GRLIB-spezifische Einstellungen (z.B. *Plug & Play*). Die Basisklassen stellen der implementierende Klasse APIs zum Auslösen verschiedener Ereignisse bereit und können ihrerseits Funktionen im Verhalten aufrufen (Ereignisbenachrichtigungen / *Callback*-Funktionen).

Funktionsbeschreibung und -zuordnung (4)

Im nächsten Schritt muss die Funktionalität sinnvoll gegliedert und in *SystemC* implementiert werden. Ein charakteristisches Merkmal von VHDL-Komponenten sind in sich abgeschlossene Funktionsabläufe (z.B. Prozesse oder Verkettungen von Zuweisungen), die durch einen externen oder internen Schalter (Register, Signal oder Port) ausgelöst werden. Diese Funktionsabläufe lösen wiederum andere Schalter aus oder beschalten Ausgangssignale. Für jeden identifizierten Funktionsablauf kann eine Funktion in *SystemC* implementiert werden. Im Anschluss müssen diese Funktionen den durch die Infrastruktur bereitgestellten Funktionstypen zugeordnet werden. Dabei handelt es sich um *SystemC*-Prozesse, Softwareroutinen (C++) oder *Callbacks* von *SoCRocket*-Basisklassen wie *gr_device* (siehe 3.3.2), *ahb_master/slave* (siehe 3.2.2) oder *signalkit* (siehe 3.2.5). Zum Auslösen von Schaltern werden APIs von Basisklassen beschrieben. So kann zum Beispiel durch den Aufruf der Schnittstellenfunktion *ahb_write* der Basisklasse *ahb_master* eine Bustransaktion generiert werden. Das Abstraktionsniveau des TLM ist abhängig vom Anwendungsfall und den Genauigkeits-/Geschwindigkeitsanforderungen.

SoCRocket soll im Kontext der Europäischen Raumfahrt zunächst zur Entwicklung und Validierung hardwarenaher Software eingesetzt werden. Ein weiteres Einsatzziel ist die Erkundung des Entwurfsraumes (DSE) für Weltraum-DPUs.

Wie bereits im Abschnitt 3.2 beschrieben, ist es möglich diese Anwendungsfälle auf die TLM2.0-Programmierstile LT und AT abzubilden. Für Softwarevalidierung und schnelle Exploration modelliert *SoCRocket* Kommunikation mittels blockierender Funktionsaufrufe, wodurch die Anzahl der Synchronisationspunkte/Kernelaufrufe in der Simulation minimiert und die schnelle Simulation auch großer Systeme ermöglicht werden. Dazu nimmt man bewusst eine geringere Simulationsgenauigkeit in Kauf. Für detaillierte Systemexploration werden Bustransaktionen dagegen durch nichtblockierende Funktionsaufrufe modelliert. Die Unterteilung einer Transaktion in mehrere TLM-Phasen ermöglicht die Abbildung von Parallelitätseffekten. Dadurch steigt die Anzahl der Synchronisationspunkte und die Simulationsgenauigkeit. Die Simulationsgeschwindigkeit jedoch sinkt. *SoCRocket*-Busschnittstellen sind in Bibliotheksbasisklassen gekapselt und kommunizieren mit dem Verhalten über Rückruffunktionen. Bei jedem Aufruf einer Rückruffunktion wird dem Verhalten ein Transaktionsobjekt übergeben. Das Verhalten muss daraufhin den Zustand der Komponente mittels Auswahl der entsprechenden Transferfunktion ändern und die Anzahl der zur Bearbeitung benötigten Wartezyklen an die Busschnittstelle zurückliefern. Im einfachsten Fall ist die Anzahl der Wartezyklen konstant. Beispiele hierfür sind eingebettete Speicher (z.B. *ahb_mem*) oder Registerbänke. Die Verzögerung für derartige Komponenten kann mit Hilfe eines Konfigurationsparameters fest eingestellt werden. Für eine weitere Klasse von Komponenten lassen sich Zugriffsverzögerungen statisch berechnen. Diese Möglichkeit wird durch die AMBA2-Protokolle begünstigt, da weder APB noch AHB mehrere ausstehende Transaktionen für das gleiche Ziel unterstützen. Transaktionen werden daher im Verhalten meist sequentiell abgearbeitet. Abhängig von der Transferfunktion ergibt sich dann eine Verzögerung, die durch den Zustand der Komponente eindeutig bestimmt ist. Ein Beispiel dafür ist der GRLIB-Speichercontroller (*mctrl* – siehe Anlage A). Der Speichercontroller verfügt über eine TLM/AHB-*Slave*-Schnittstelle, von der aus Transaktionen wahlweise an einen ROM, I/O-Speicher, SRAM oder SDRAM weitergeleitet werden. Abhängig vom Typ der Transaktion (Lesen oder Schreiben), der Art des adressierten Speichers, dem internen Zustand (z.B. Speicherbank offen oder geschlossen) und der Einstellung des Controllers (z.B. *Read-Modify-Write*) ergeben sich unterschiedliche Transferfunktionen. Zur Berechnung der Zugriffsverzögerung ist keine Synchronisation mit dem *SystemC*-Kernel erforderlich. Speicherzugriffe werden im *Context* der Rückruffunktion der Busschnittstelle ausgeführt. Tabelle 3.7 zeigt einige Beispiele für die Berechnung der Verzögerung in *mctrl*. Alle abgebildeten Funktionen wurden empirisch ermittelt und unter Einbeziehung zusätzlicher Faktoren an das Verhalten der RTL-Komponente angepasst. Für PROM, I/O und SRAM können in der Registerbank des Speichercontrollers feste Wartezeiten eingestellt werden. Die SDRAM-Zugriffszeit orientiert sich am Zustand des Speichers. Abhängig von Adresse und Länge der Transaktion müssen verschiedene Zeilen (Rows) und Spalten (Cols) geöffnet oder geschlossen werden.

Speichertyp	Transferzeit	Funktion
PROM	$T_R =$	$(1 + (2 + N_{PROMRWS}) * BL) * T_{clk}$
	$T_W =$	$(3 + N_{PROMWWS}) * BL * T_{clk}$
I/O	$T_R =$	$(5 + N_{IORWS}) * BL * T_{clk}$
	$T_W =$	$(3 + N_{IOWWS}) * BL * T_{clk}$
SRAM*	$T_R =$	$(4 + N_{RAMRWS}) * BL * T_{clk}$
	$T_W =$	$(3 + N_{RAMWWS}) * BL * T_{clk}$
SDRAM**	$T_{RW} =$	$(2 + N_{RCD} + N_{CAS} * BL/BW) * T_{clk}$

$N_{PROMRWS/PROMWWS}$: PROM-Wartezyklen
$N_{IORWS/IOWWS}$: I/O-Wartezyklen
$N_{RAMRWS/RAMWWS}$: SRAM-Wartezyklen
N_{RCD}: RAS-to-CAS-Wartezyklen (Zeilen- zu Spaltenaktivierung)
N_{CAS}: CAS-to-CAS-Wartezyklen (SDRAM-Burstabstand)
BL: Burstlänge (Worte); BW: SDRAM-Burstgröße (Worte); T_{clk}: Taktzeit

* kein *Read-Modify-Write*, ** Bank geschlossen

Tabelle 3.7: Verzögerungszeiten für ausgewählte *MCTRL*-Transferfunktionen

Die beschriebene einfache statische Vorausberechnung der Transferzeit ist nicht möglich bei Komponenten die mehrere sich gegenseitig beeinflussende Transaktionen in parallel verarbeiten. Denkbar wäre ein DDR-Speichercontroller mit AXI-Schnittstelle der mehrere unvollständige Transaktionen annehmen kann und diese gemäß ihrer *Bank*- und *Row*-Adressen zur Optimierung des Zugriffs umsortiert. Weitere Beispiele sind Transferkomponenten, die zwischen mehreren *Master*-Komponenten arbitrieren müssen. Die Wartezeit einer Transaktion im Arbiter hängt vom Arbitrierungsschema und damit von anderen Transaktionen ab, die gegebenenfalls vorgezogen werden. Unter den in *SoCRocket* bereitgestellten Komponenten, nimmt der AHB-Controller (*ahbctrl*) daher eine Sonderrolle ein. Der *ahbctrl* ist ein AHB-*Slave* und ein AHB-*Master*, erbt seine Busschnittstellen aber von keiner der Bibliotheksbasisklassen. Im Gegensatz zu den im vorab beschriebenen einfachen Komponenten mit fester oder deterministischer Verzögerung macht es für komplexe Komponenten, wie den *ahbctrl*, Sinn, auch im Verhaltensteil eine Unterscheidung gemäß des Abstraktionsniveaus vorzunehmen. Für Softwareentwicklung oder schnelle Systemexploration sind zum Beispiel Arbitrierungseffekte weitgehendst irrelevant. Ein LT-Modell kann die dafür erforderlichen *SystemC*-Threads einsparen. Das Verhalten wird dabei soweit vereinfacht, dass immer nur eine Transaktion weitergeleitet werden kann. Alle anderen Transaktionen werden blockiert. Das dafür erforderliche *Lock* kann durch eine globale Variable realisiert werden. Die Auswahl des *Masters* wird dem Zufall (dem SystemC-Kernel) überlassen. Wie in Abbildung 3.19 dargestellt, kann die Kernfunktionalität des Routers in die blockierende Transportfunktion eingebettet werden. Ist der Bus besetzt (*busy*), versetzt Zeile 5 alle zwischenzeitlich zugreifenden *Master* in den Wartezustand. Die Transaktion wird in Zeile 11 dekodiert und in Zeile 14 an den ausgewählten *Slave* weitergeleitet. Die Freigabe des Busses erfolgt in den Zeilen 17 und 18. Die dargestellte Lösung erfordert keine zusätzlichen *Threads*, ist damit sehr schnell und erfüllt die Anforderungen eines Softwareentwickler.

```
1  void AHBCtrl::b_transport(uint32_t id, payload_t &trans, sc_time &delay) {
2
3          ...
4
5          // Warte hier bis frei
6          while(busy) wait(unblock_event);
7
8          // Markiere als beschaeftigt
9          busy = true;
10
11         // Slave auswaehlen
12         index = decode(trans);
13
14         // Transaktion an Slave weiterleiten
15         ahb_out[index]->b_transport(trans, delay);
16
17         // Bus frei
18         busy = false;
19         unblock_event.notify();
20
21         ...
22 }
```

Abbildung 3.19: *AHBCTRL* - Abstrahiertes LT-Verhalten (vereinfacht)

Für detaillierte Architekturexploration ist eine höhere Genauigkeit erforderlich. Es ist unter anderem notwendig, den Arbitrierungsmechanismus des Busses zu modellieren. Dazu sind mehrere Prozesse erforderlich. Eine genaue Beschreibung der AT-Implementierung des AHB-Controllers befindet sich in Abschnitt 4.2.1 (siehe u.a. Abbildung 4.16).

Simulation des SystemC/TLM-Modells (5)

Im abschließenden Schritt muss das TLM-Modell simuliert und mit der RTL-Referenz oder im Falle eines Neuentwurfes mit der Spezifikation verglichen werden. Dazu kann wiederum die in Abschnitt 3.8 beschriebene Testumgebung verwendet werden. In der Entwurfsphase hat sich die Verwendung gerichteter (*directed*) Tests als zweckmäßig erwiesen. Es empfiehlt sich, unmittelbar nach der Implementierung einer Funktion entsprechende Testvektoren hinzuzufügen. Die Testumgebung unterstützt die Erzeugung von *Traces* und *Waveforms*. Darüber hinaus können Erwartungswerte für Funktionen und Zeitverhalten annotiert werden (*Assertions*).

3.5 Verwaltung und Handhabung von Metadaten (Konfiguration)

3.5.1 Stand der Technik

Ein essentieller und bisher von der Standardisierung vernachlässigter Bereich des Entwurfs von IP-Komponenten ist die Handhabung von Metadaten. Unter Metadaten versteht man alle Informationen die der Einstellung, Beschreibung und Konfiguration eines Modelles dienen oder zu dessen Analyse annotiert werden. Neben Konfigurationsdaten können dies zum Beispiel Daten bezüglich des Energieverbrauchs oder der Verlässlichkeit von Komponenten sein (Abschnitt 3.6). Für die effiziente Wiederverwendbarkeit von Simulationsmodellen ist es wichtig, dass diese Daten durch einheitliche Werkzeugschnittstellen zugänglich gemacht werden.

Einer der ersten erfolgreichen Verstöße auf diesem Gebiet wurde durch das SPIRIT-Consortium entwickelt (seit 2009 zu *Accellera* gehörig [acc13]). Der Erfolg von SPIRIT beruht auf einem sprachunabhängigen Metadatenformat zur Beschreibung von Konfigurationsparametern und integrationsrelevanten Einstellungen. Dazu werden Simulationsmodelle in eine XML-Bibliothek

integriert. Dies ermöglicht einen SoC-Entwurfsprozess, in dem der Entwickler IP-Modelle charakterisiert und zum Aufbau eines Systems aus einer Modellbibliothek instantiiert. Aus SPIRIT ging die IP-XACT-Spezifikation [Com10] hervor. IP-XACT ist ebenfalls auf IP-Wiederverwendbarkeit spezialisiert und definiert ein XML-basiertes Austauschformat zur Schnittstellenbeschreibung, dass 2010 durch den IEEE standardisiert wurde (IEEE 1685). Darüber hinaus legt IP-XACT Schnittstellen zur konfigurationsabhängigen Generierung von Modellen fest. Der Standard erleichtert die Beschreibung von IP-Schnittstellen und ermöglicht somit die Integration von Modellen unterschiedlicher Hersteller in verschiedene Entwicklungsumgebungen, trifft aber keine Aussagen über die interne Handhabung von Metadaten, deren Implementierung und Zugriffsfunktionen.

Einen praktischen Ansatz zur Behebung dieses Problems stellt die *SystemC Modeling Library* (SCML) [Inc13] bereit. SCML bietet neben der einfachen Beschreibung von Speicherelementen (Abschnitt 3.3) die Möglichkeit, Modelle mit konfigurierbaren Parametern auszustatten. Diese sogenannten SCML-*Properties* werden während der Konstruktion des Objektes durch einen *Property Server* initialisiert, der die zu setzenden Initialwerte aus einer XML-Datei einliest. Der *Property Server* ist integrierter Teil der ESL-Umgebung *Synopsys Platform Architect* und wie auch das XML-Format nicht quelloffen. Einen ähnlichen Ansatz verfolgen *ARM* und *Carbon Design Systems* mit der *RealView-ESL-API* und dem darin integrierten *Cycle Accurate Simulation Interface* (CASI). CASI dient vordringlich der generischen Programmierung von *Ports* und *Channels* zur zyklengenauen Simulation von Modellen. Darüber hinaus enthält es eine Schnittstelle für Konfigurationsparameter, die ähnlich wie SCML-*Properties* initialisiert und zur Laufzeit überschrieben werden können. Die Initialisierung erfolgt jedoch nicht über ein in die Werkzeuge integrierten Server, sondern über eine Konfigurations-API in der Elaborationsphase der Simulation (vor dem Verbinden der Komponenten). Einen weiteren verwandten Ansatz zur Standardisierung von Metadaten und Modellspezifikationen verfolgt *Synopsys* in seiner *Innovator*-Software[1] [inn14]. *Innovator* verwendet CCSS-Parameter zur Darstellung von SystemC-Konfigurationsparametern. CCSS-Parameter sind Objekte die sich aus Zugriffsfunktionen, einem *String* zur Bezeichnung des Datentyps und einem Wert des entsprechenden Datentyps zusammensetzen. Es existiert kein globales Parameterverzeichnis. Die *Innovator*-Software durchsucht den Quellcode nach Parametern und zeigt diese in einer IDE grafisch an. CCSS-Parameter sind bedeutsam, da sie durch eine große Anzahl von Simulationsmodellen unterstützt werden (*Designware-System-Level-Bibliothek*). Die Firma *Cadence* fördert die Konfiguration von Systemkomponenten im Rahmen der *Open Verfication Methodology* (OVM). OVM erlaubt es, Modellparameter über einheitliche Programmierschnittstellen zu setzen und auszulesen. Die Parameter müssen vor der Instantiierung der Komponente initialisiert werden. Dabei wird ein hierarchischer Ansatz verfolgt. Komponenten höher in der Hierarchie können Parameter tiefer liegender Komponenten überschreiben. Für jeden Lesezugriff wird eine hierarchische Suche durchgeführt. Als Ergebnis wird der Wert des ersten gefundenen Eintrags zurückgeliefert.

Es ist deutlich erkennbar, dass sich alle vorgestellten Ansätze inherent ähneln. Alle Lösungen arbeiten entweder mit einer Konfigurationsschnittstelle (z.B. CASI, OVM) oder stellen spezielle Parameterklassen bereit (z.B. SCML, CCSS). Eine Funktionsschnittstelle besteht in der Regel aus virtuellen Funktionen die im Modell implementiert werden müssen und für den Benutzer (oder die Werkzeugumgebung) extern sichtbar sind. Mit Hilfe dieser Funktionen können Modellparameter gesetzt, ausgelesen und manipuliert werden. Ein Beispiel für eine einfache generische Konfigurationsschnittstelle ist in Abbildung 3.20 dargestellt.

1 Zum Entstehungszeitpunkt dieser Arbeit integriert Synopsys *Innovator* in sein neues *Virtualizer Development Kid* (VDK).

```
1 // Modellparameter setzen
2 void set_parameter(string name, int value);
3
4 // Modellparameter auslesen
5 int  get_parameter(string value);
```

Abbildung 3.20: Beispiel für eine einfache Konfigurationsschnittstelle

Eine Alternative zur Konfigurationsschnittstelle stellt die Verwendung einer Kapselungsklasse dar. Diese kann Modellparameter transparent ersetzen. Die Bereitstellung von Kapselungsklassen ist aus der Sicht der Infrastruktur aufwendiger als die Definition einer Schnittstelle, da die Implementierung der Zugriffsfunktionen nicht dem Modell überlassen bleibt. Dadurch ergibt sich jedoch eine höhere Verlässlichkeit, die den geringen Mehraufwand in der Bereitstellung mehr als ausgleicht. Ein Beispiel für eine einfach Parameterklasse ist in Abbildung 3.21 gegeben. Zur Unterstützung unterschiedlicher Datentypen empfiehlt sich der Einsatz von C++-Templates.

```
1 // Definition der Parameterklasse (vereinfacht)
2 template <typename T>
3 class parameter {
4  private:
5   T param;
6  public:
7   void set_parameter(T value) {...}
8   T get_parameter() {...};
9 };
```

Abbildung 3.21: Beispiel für eine einfache Parameterklasse

3.5.2 Standardoffene Konfigurations-Middleware

Wie im vorherigen Absatz beschrieben sind die in aktuellen Werkzeugen eingesetzten Lösungen zur Handhabung von Metadaten vom Ansatz ähnlich, jedoch keinesfalls kompatibel. Simulationsmodelle können, sofern Sie über eine IP-XACT-Schnittstellenbeschreibung verfügen, in unterschiedliche ESL-Umgebungen integriert werden. Darüber hinaus erlaubt die Unterstützung des TLM2.0-Standards die gemeinsame Simulation von Komponenten unterschiedlicher Hersteller. Schwierigkeiten bereitet die einheitliche Modellkonfiguration, insbesondere der Austausch von Metadaten zwischen Modellen, sowie Modellen und Werkzeugen.

Ein vielversprechender Ansatz zur Lösung dieses Problems wird in [Sch11] beschrieben. Die Arbeit wurde ebenfalls an der TU Braunschweig verfasst und entwickelt eine generische *Middleware*, mit deren Hilfe Modelle mit unterschiedlichen Ansätzen zur Metadatenverwaltung durch beliebige Werkzeuge konfiguriert werden können. Darüber hinaus stellt die Arbeit einen ersten Schritt zur Entwicklung einer generischen Werkzeugschnittstelle dar und dient als Grundlage für die Arbeit der *Accellera*-Arbeitsgruppe für *Configuration, Control & Inspection* (CCI) [Ace14], die mit dem Entwurf eines Standards für *Meta*-Interoperabilität begonnen hat. Abbildung 3.22 wurde in abgewandelter Form aus [Sch11] übernommen und verdeutlicht das Konzept der standardoffenen Konfigurations-*Middleware*.

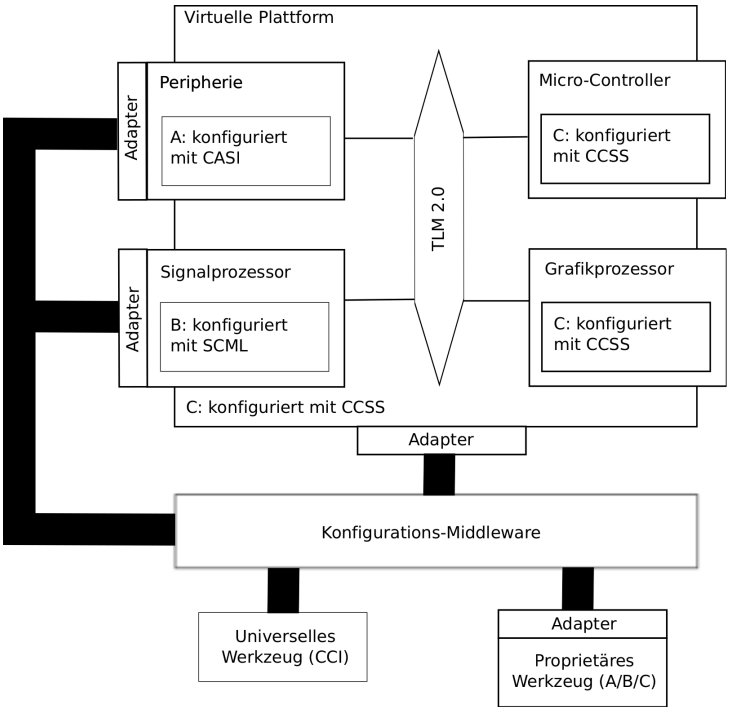


Abbildung 3.22: Konzept der Konfigurations-Middleware (GreenControl/Config)

Die Abbildung zeigt eine Virtuelle Plattform, die mehrere Simulationsmodelle mit Hilfe des TLM2.0-Standards verbindet. Die Plattform und die Mehrzahl der Komponenten verwalten ihre Metadaten in CCSS-Parametern (C). Es könnte sich somit um ein *Synopsys*-System mit *System-Level-Designware*-Komponenten handeln, welches zur Bearbeitung in *Innovator* vorgesehen ist. Die Plattform enthält jedoch auch einen Signalprozessor (B), der mit SCML konfiguriert werden muss und eine *Peripherie*-Komponente mit CASI-Schnittstelle (A). Diese Komponenten können demnach nicht durch *Innovator* initialisiert und analysiert werden. Der standardoffene *Middleware*-Ansatz beschreibt eine generische Konfigurationsschnittstelle, die durch die unterschiedlichen proprietären Werkzeuge angesteuert werden kann. Der Aufwand zur Entwicklung der dafür erforderlichen Adapter ist gering. Komponenten und Werkzeuge die den zukünftigen CCI-Standard unterstützen, können direkt verbunden werden.

Zur Verbindung mit der beschriebenen Konfiguration-*Middleware* werden die Parameter aller *SoCRocket*-Komponenten in *GreenSoCs*-Parameter (GS_PARAM) umgewandelt. GS_PARAMs sind Kapselungsklassen (siehe Abb. 3.21) denen mit Hilfe von *Template*-Parametern ein beliebiger Datentyp zugewiesen werden kann. Zu Beginn der Simulation werden alle vom Nutzer instantiierten GS_PARAMs automatisch bei einer zentralen Datenbank der *Middleware* registriert. Die Datenbank kann durch den Nutzer oder angekoppelte Werkzeuge nach Parametern und deren Eigenschaften durchsucht werden. Das Lesen und Beschreiben von Parametern erfolgt über die *GreenConfig*-API, die ebenfalls in [Sch11] ausführlich beschrieben wird. Zur Realisierung dieser Arbeit wurden GS_PARAMs auf Systemebene implementiert. Der in Kapitel 5 beschriebenen Methodik folgend, werden dazu alle Konfigurationsparameter eines Explorationsprototypen erfasst. Abbildung 3.23 verdeutlicht dies am Beispiel der Instantiierung des AHB-Busses (*AHBCTRL*) im Explorationsprototypen LEON3MP (Abschnitt 6.1).

```
1  // Definition der Konfigurationsparameter
2  gs::gs_param_array p_ahbctrl("ahbctrl", p_conf);
3  gs::gs_param<unsigned int> p_ahbctrl_ioaddr("ioaddr", 0xFFF, p_ahbctrl);
4  gs::gs_param<unsigned int> p_ahbctrl_iomask("iomask", 0xFFF, p_ahbctrl);
5  ...
6  gs::gs_param<bool> p_ahbctrl_rrobin("rrobin", false, p_ahbctrl);
7  ...
8
9  // Instantiierung des AHB-Busses
10 AHBCtrl ahbctrl("ahbctrl",
11         p_ahbctrl_ioaddr,
12         p_ahbctrl_iomask,
13         ...
14         p_ahbctrl_rrobin,
15         ...
16 );
```

Abbildung 3.23: Konfiguration des AHBCTRL im LEON3MP-Explorationsprototyp

Zeile 2 definiert eine Parametergruppe *p_ahbctrl* zur Aufnahme aller Konfigurationsparameter des AHBCTRL und ordnet diese dem übergeordneten Namensraum *p_conf* zu. Im Anschluss (Zeilen 2-6) werden drei Konfigurationsparameter instantiiert, welche die I/O-Adresse und -Maske sowie das Arbitrierungsschema des Busses bestimmen. Die Parameter kapseln Daten der Typen *unsigned int* und *bool*, welche dem AHBCTRL im Konstruktor übergeben werden (Zeilen 11 - 14). Alle Konfigurationsparamater des AHBCTRL werden durch die Übergabe eines Zeigers der Parametergruppe *p_ahbctrl* zugeordnet. Die Erzeugung von Untergruppen, die ihrerseits wiederum Parameter oder weitere Untergruppen besitzen, ist in ähnlicher Weise möglich. Durch den beschriebenen Mechanismus spannt sich ein Namensraum auf, der es erlaubt Parameter eindeutig zu identifizieren. So liefert die Konfigurationsdatenbank bei einer Anfrage mit dem Schlüssel *"conf.ahbctrl.rrobin"* den Arbitrierungsparameter *p_ahbctrl_rrobin* zurück.

SoCRocket verwendet den globalen Namensraum zur automatischen Initialisierung von Explorationsprototypen. Die Standardwerte aller Konfigurationsparameter des Systems können mit Hilfe eines Konfigurationsfiles überladen werden. Konfigurationsdateien werden von Hand erstellt oder mit dem *SoCRocket-Configuration Wizard* generiert (siehe Abschnitt 5). Um sowohl einfache Lesbarkeit sowie auch einfache Verarbeitung in Werkzeugen zu ermöglichen, werden Konfigurationsdateien in *JavaScript Object Notation* (JSON) verfasst [Cro14]. Wie in Abbildung 3.24 verdeutlicht, eignet sich JSON ideal zur Datenübergabe basierend auf Schlüssel/Wert-Paaren. Als Alternative wäre die Verwendung von XML denkbar. Der damit verbundene *Overhead* würde jedoch die Lesbarkeit beeinträchtigen.

```
1  "conf":{
2    "ahbctrl": {
3      "ioaddr": 0xE00,
4      "iomask": 0xFF0,
5      ...
6      "rrobin": true,
7      ...
8    }
```

Abbildung 3.24: Parameterinitialisierung mit JSON

Ein weiteres Entscheidungskriterium für die Verwendung von JSON zur Beschreibung von Systemkonfigurationen ist die Verfügbarkeit von *Parsern* für fast alle gängigen Skriptsprachen. *SoCRocket* verwendet den von Craig Mason-Jones entwickelten JSON4LUA-Parser [MJ14]. Die Konfigurationsdatenbank gleicht die eingelesenen Schlüssel mit den registrierten Parametern ab

und übernimmt im Falle eines Treffers den angegebenen Wert zur Initialisierung der Simulation.

Meines Wissens nach ist *SoCRocket* die erste Virtuelle Plattform, die *GreenControl*-Infrastruktur praktisch einsetzt. Durch die zum Einlesen von Konfigurationsdateien geschaffene Schnittstelle, können Systeme unkompliziert, ohne erneutes Kompilieren, rekonfiguriert werden. Die standardoffene-*Middleware* verspricht Zukunftssicherheit, nicht zuletzt in Hinblick auf den angekündigten CCI-Standard, und Kompatibilität zu verschiedensten proprietären ESL-Lösungen.

3.6 Modellierung des Energieverbrauches

3.6.1 Stand der Technik

Die durch ein Hardwaremodul konsumierte elektrische Leistung (P_{gesamt}) besteht aus einer statischen und einer dynamischen Komponente.

$$P_{gesamt} = P_{dynamic} + P_{static}$$
$$P_{dynamic} = P_{internal} + P_{switching}$$

Die dynamische Komponente ($P_{dynamic}$) setzt sich aus der Schaltleistung $P_{switching}$ und der zellinternen Leistung $P_{internal}$ zusammen. $P_{switching}$ ist abhängig von der Anzahl der Schaltvorgänge innerhalb eines Intervalls und der zu treibenden Last. $P_{internal}$ wird als abhängig von der Taktrate aber unabhängig von Schaltvorgängen angenommen. Die zellinterne Leistung umfasst Querströme, sowie die für D-Flops zur Erhaltung des Zustands erforderliche Leistung. Die statische Leistung P_{static} repräsentiert den Leckstrom (*Leakage*) einer Schaltung. Für Abschätzungen des Energieverbrauchs kann P_{static} als schaltungs- und taktunabhängig betrachtet werden.

Da heute Energieverbrauch neben Verarbeitungsgeschwindigkeit und Siliziumfläche das wichtigste Optimierungskriterium für digitale Schaltungen ist, hat die möglichst frühe Abschätzung der Leistungsaufnahme an Bedeutung gewonnen. Gewöhnlich erfolgt die erste Schätzung durch Simulation der Netzliste auf Gatterebene. Dies ist extrem langsam, da selbst in Systemen mittlerer Größe nur wenige hundert Takte pro Sekunde verarbeitet werden können. Außerdem kommen die Ergebnisse für eine sinnvolle Optimierung zu spät. Moderne kommerzielle Simulatoren wie *Questa/Modelsim* [que14] oder *Incisive* [inc14a] erlauben abstraktere Abschätzungen auf RT-Ebene. Dabei werden während der RTL-Simulation Schaltoperationen gezählt und in einer *Switching Activity*-Datei abgelegt oder als *Waveform* (z.B. VCD) über einer Zeitachse gespeichert. Die gewonnenen Daten können dann zusammen mit der synthetisierten Netzliste ausgewertet werden. Durch die Gewinnung der Schaltinformation auf RT-Ebene kann die Simulationsgeschwindigkeit um Faktor 10 bis 100 gesteigert werden. Die Verwendung von *Waveforms* eignet sich allerdings nur für sehr kurze Simulationen, da schnell riesige Datenmengen anfallen, für moderne *MPSoCs* ist diese Methode daher ungeeignet. Um sinnvolle Rückschlüsse für den Entwurfsprozess gewinnen zu können, muss die erste Abschätzung des Energieverbrauchs auf Systemebene vorgenommen werden. Entwickler verwenden hierzu heute oft manuelle tabellenbasierte Ansätze (*Spreadsheet Approach*). Dies funktioniert relativ gut für statische Leistungskomponenten, lässt aber Aktivität und Anwendung unberücksichtigt. Aktuelle Forschungsarbeiten zum Thema wurden bereits in Abschnitt 2.2 beschrieben.

3.6.2 *Power*-Modellierung in SoCRocket

In *SoCRocket* wird ein simulativer Ansatz zur Schätzung des Energieverbrauchs auf Systemebene verfolgt. Die Energieberechnung erfolgt auf Grundlage normalisierter Energie- und Leistungswerte, die als Metadaten in der *GreenControl-Middleware* abgelegt werden [AS14]. Art und Anzahl der Eingabeparameter sind vom Modell abhängig. In den meisten Fällen sind pro Komponente oder Unterkomponente drei Parameter erforderlich, welche die statische, die zell-interne und die dynamische Leistungsaufnahme beschreiben. Eine vollständige Übersicht befindet sich in [Sch12c].

Die statische Leistung ist überwiegend unabhängig von der auf dem Prozessor ausgeführten Anwendung und kann als unabhängig von der Taktrate angesehen werden. P_{static} steigt linear mit der durch eine Komponente eingenommenen Chipfläche und ist stark technologieabhängig. Zur Normalisierung wird daher ein Komplexitätswert mit direktem Bezug zur Fläche benötigt. Für Speicherelemente wie Cache, RAM oder ROM wird die normalisierte statische Leistung in pW/bit angegeben, für Router kann die Anzahl der *Master*- und *Slave*-Ports herangezogen werden und für Prozessoren die Breite des Datenpfades. Die zellinterne Leistung ist ebenfalls unabhängig von der ausgeführten Anwendung, weist aber eine lineare Abhängigkeiten zur Fläche und zur Taktrate auf. Die Normalisierung erfolgt mit Hilfe eines Komplexitätswertes und der Frequenz. Für Speicher wird die normalisierte zellinterne Leistung in $\mu W/bit/Hz$ angegeben. Router und Prozessoren verwenden neben der Frequenz wiederum die Anzahl der *Master*- und *Slave*-Ports oder die Breite des Datenpfades. Der anwendungsabhängige Teil der dynamischen Leistungsaufnahme ist die Schaltleistung $P_{switching}$. Schaltleistung wird an Bussen, Pins oder Speicherelementen verbraucht, die ihren Zustand wechseln. Leistung stellt einen durchschnittlichen Verbrauch von Energie über eine bestimmte Zeit dar und ist keine günstige Einheit für ereignisbasierte diskrete Messungen. $P_{switching}$ wird in *SoCRocket* daher indirekt mit Hilfe von Energie/Zugriff-Kennzahlen ermittelt. Zugriffe können dabei abhängig von der Natur der modellierten Komponente Speicherzugriffe, Bustransaktionen oder auf einem Prozessor verarbeitete Instruktionen sein. Während der Simulation wird die Anzahl der Zugriffe gezählt und anschließend mit einem $\mu J/Zugriff$-Wert multipliziert. Die Größe der Zählintervalle bestimmt die Genauigkeit der Schätzung.

Power-Modelle

Die Modelle zur Schätzung des Energieverbrauchs sind in die Simulationsmodelle der *SoCRocket*-Bibliothek direkt integriert und können mit Hilfe des Konfigurationsparameters *pow_mon* ein und ausgeschaltet werden. Ist *Power-Monitoring* aktiviert, so wird aus der *SystemC*-Systemfunktion *start_of_simulation* zum Simulationsbeginn die Funktion *power_model* aufgerufen. Dadurch werden die normalisierten Eingabewerte mit Hilfe der aktuellen Konfiguration denormalisiert. Im folgenden wird dieser Vorgang am Beispiel des Datencaches erläutert:

Die statische Leistung des Datencaches setzt sich aus einem annähernd konstanten Wert für den Cachecontroller $p_{staticctrl}$ und variablen Werten abhängig von Größe und Anzahl (Bänke) der *Tag*-Speicher und Datenspeicher zusammen.

$$P_{static} = P_{staticctrl} + n_{sets} * (P_{staticnormdtag} * N_{bitsdtag} + P_{staticnormddata} * N_{bitsddata})$$

Die zellinterne Leistung wird aus einem frequenznormalisierten Anteil für den Controller und sowohl größen- als auch frequenznormalisierten Anteilen für die Bänke gewonnen.

$$P_{internal} = f * (P_{intctrlnorm} + n_{sets} * (P_{intnormdtag} * N_{bitsdtag} + P_{intnormddata} * N_{bitsddata}))$$

Zur Bestimmung der Schaltleistung ($P_{switching}$) wird die Energie aller Zugriffe addiert. Dazu muss die Zugriffsenergie für Lese- und Schreiboperation auf *Tags* und Speicherbänke ebenfalls in Funktion *power_model* denormalisiert werden. Das Beispiel zeigt dies anhand der Energie für Lesezugriffe auf den *Tag*-RAM. N_{bits} enthält die Anzahl der Bits pro *Tag* und $width_{bits}$ die Weite des Speichers.

$$E_{dtagread} = E_{dtagreadnorm} * width_{bits} * N_{bits}$$

Die vollständigen Normalisierungsfunktionen aller im Rahmen dieser Arbeit entwickelten Komponenten können wiederum [Sch12c] entnommen werden. Statische und zellinterne Leistung ändern sich während der Simulation in der Regel nicht. Eine Ausnahme bilden Komponenten wie der GRLIB-Speichercontroller (siehe Abschnitt 4.3.3), die verschiedene Energiesparmodi implementieren. Die aus der Denormalisierung gewonnenen Werte P_{static} und $P_{internal}$ werden

daher meist direkt auf Modellparameter abgebildet, die mittels der *GreenControl-Middleware* ausgelesen werden können (siehe Tabelle 3.8).

Parameter	Beschreibung
*.power.sta_power	Statische Leistung des Modells
*.power.int_power	Zellinterne Leistung
*.power.swi_power	Schaltleistung im Messintervall

Tabelle 3.8: Parameterschnittstelle zum Auslesen des Energieverbrauchs

Die aktuelle Schaltleistung wird bei jedem Zugriff auf den Parameter **.power.swi_power* neu berechnet. Dazu wird eine *pre-read Callback*-Funktion verwendet.

$$P_{switching} = \frac{(E_{dtagread} * n_{dtagr}) + (E_{dtagw} * n_{dtagw}) + (E_{ddatar} * n_{ddatar}) + (E_{ddataw} * n_{ddataw}))}{T_{now} - T_{start}}$$

Wie in der Gleichung dargestellt, wird zur Berechnung von $P_{switching}$ die Energie per Zugriff mit der Anzahl der Zugriffe multipliziert. Das Produkt wird dann durch die Dauer des Messintervalls geteilt. T_{now} ist dabei die aktuelle Simulationszeit zum Zeitpunkt des Zugriffs auf die Analyseschnittstelle. T_{start} wird zum Simulationsbeginn mit Null initialisiert und enthält im Anschluss den Endzeitpunkt des vorherigen Messintervalles.

Power-Monitor

Die im vorherigen Abschnitt beschriebene Modellschnittstelle kann zur Integration mit Werkzeugen zur Leistungsanalyse verwendet werden. Als *Proof-of-Concept* wurde für *SoCRocket* ein einfacher *Power*-Monitor (PM) entwickelt. Das Werkzeug befindet sich im Verzeichnis *common* der VP (siehe Anhang B) und setzt auf der in Abschnitt 3.5.2 beschriebenen Infrastruktur zur Verwaltung von Metadaten auf. In der Grundeinstellung wird als Messintervall die Simulationsdauer angenommen. Das heisst, PM berechnet die durchschnittliche Leistung für die gesamte Simulation. Im Detail wird dazu wie folgt verfahren:

1. Zugriff auf die *GreenControl*-API am Ende der Simulation (*end_of_simulation*):
   ```
   gs::cnf::cnf_api *mApi = gs::cnf::Gcnf_Api::get_ApiInstance(NULL)
   ```
2. Alle registrierten *Power*-Parameter (siehe Tab. 3.8) in einer Datenstruktur sammeln (Abschnitt 3.7 / Abb. 3.28).
3. Liste nach Modellen sortieren (je Modell P_{static}, $P_{internal}$, $P_{Switching}$)
4. Bericht zum Energieverbrauch der Komponente ausgeben
5. Verbrauch der Komponente zum Gesamtverbrauch des Systems addieren
6. Nächste Komponente (weiter mit 3.), bis die in 2. aufgebaute Datenstruktur leer ist
7. Globalen Bericht zum Energieverbrauch des Systems generieren:
   ```
   @2130 us: Info:  ****************************************
   @2130 us: Info:  * Power Summary:
   @2130 us: Info:  * ------------------------------------
   @2130 us: Info:  * Static power (leakage): 1217.16 uW
   @2130 us: Info:  * Internal power (dynamic): 1920.78 uW
   @2130 us: Info:  * Switching power (dynamic): 898.567 uW
   @2130 us: Info:  * ------------------------------------
   @2130 us: Info:  * Total power: 2112.73 uW
   @2130 us: Info:  ****************************************
   ```

Die vorgestellte Lösung ist sehr flexibel, da die Berechnung des Energieverbrauches, je nach Bedarf, in größeren oder kleineren Intervallen durchgeführt werden kann. Die Bestimmung der durchschnittlichen Leistung einer Simulation verursacht fast keinen zusätzlichen Aufwand. Der Aufwand steigt, wenn Leistungsprofile berechnet werden sollen. Wie in Abbildung 3.25 ersichtlich, muss der PM dazu in Intervallen aufgerufen werden. Bei sehr hoher zeitlicher Auflösung können die gewonnenen Momentanwerte nicht mehr im Arbeitsspeicher gehalten werden und müssen auf die Festplatte ausgelagert werden.

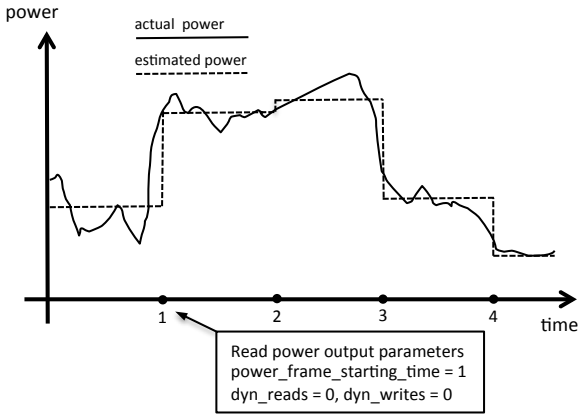


Abbildung 3.25: Beispiel: *Power*-Profil (keine Messwerte)

Der beschriebene PM wird im späteren Verlauf dieser Arbeit für Erkundung des Entwurfsraumes von Weltraum-DPUs eingesetzt (siehe Abschnitt 5.2). Zusätzliche Informationen, insbesondere zur Berechnung von *Power*-Profilen für verschiedene *SoCRocket*-Komponenten, können der Diplomarbeit von Etienne Kleine entnommen werden [Kle11].

3.7 Debugging und Analyse (Inspection)

3.7.1 Stand der Technik

Weitere für den praktischen Einsatz von TL-Modellen wichtige Aspekte sind deren *Debugging*- und Analysefähigkeiten. Unter *Debugging* versteht man dabei den Prozess der Fehlerbeseitigung zur Abstimmung von Hardware- und Softwarekomponenten in einer Virtuellen Plattform. Die Analyse liefert darüber hinaus Daten zur Optimierung und Anpassung des Systems für einen bestimmten Anwendungszweck. Zur effizienten Realisierung dieser Entwurfsaufgaben benötigt der Entwickler zur Simulationslaufzeit Zugriff auf alle Speicherelemente. Die dafür erforderliche Grundfunktionalität ist im TLM2-Standard als *Debug Transport Interface* (DTI) definiert. Mit Hilfe des DTI können alle im Adressbereich sichtbaren Speicherelemente ohne Verzögerung und Kontextwechsel direkt gelesen und beschrieben werden. Der dazu erforderliche *Debug*-Pfad wird bei der Bindung der TLM-*Sockets* implizit aufgebaut (Abb. 3.26).

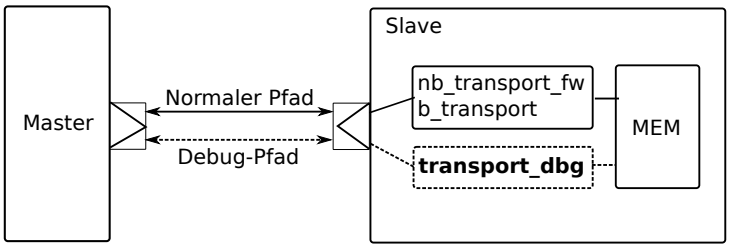


Abbildung 3.26: Debug-Transportpfad in TLM2.0

Zur Unterstützung von DTI müssen alle im Pfad enthaltenen Komponenten eine *Debug*-Transportfunktion (*transport_dbg*) definieren. Der Funktion werden im *Master* ein reguläres TLM-

Payload-Objekt, aber keine TLM-Phase und kein Verzögerungszeiger, übergeben. Seiteneffekte die den Zustand eines Modelles ändern werden auf dem *Debug*-Pfad ignoriert. Der DTI-Mechanismus ist einfach, schnell und wird in fast allen aktuellen TL-Simulationsmodellen unterstützt.

Über den *Debug*-Zugriff auf Speicherelemente hinaus ist es oft hilfreich, zeitliche Veränderungen von Registern und die Abfolge von Transaktionen zur späteren Auswertung aufzuzeichnen (*Tracing*). *SystemC* stellt dafür einen einfachen Mechanismus zur Speicherung von Signalen in *Waveform*-Dateien bereit (Abb. 3.27).

```
1 sc_trace_file *fp;
2 fp=sc_create_vcd_trace_file("wave");
3
4 sc_trace(fp, clk, "clk");
5 sc_trace(fp, mysignal, "mysignal");
6
7 sc_start(100, SC_NS);
8
9 sc_close_vcd_trace_file(fp);
```

Abbildung 3.27: *Tracing* in SystemC

Dafür muss im Quellcode eine VCD-Datei (*Value Changed Data*) geöffnet werden (Zeilen 1-2). Mit Hilfe des Kommandos *sc_trace* lassen sich dann beliebige Signal zur Aufzeichnung hinzufügen (Zeilen 4-5). Im Beispiel wird der Signalverlauf über 100 ns gespeichert (Zeile 7). Die Aufnahme endet mit dem Schließen der VCD-Datei (Zeile 9). VCD ist ein offenes Format, das von fast allen *Waveform*-Betrachtungswerkzeugen unterstützt wird. Eines der beliebtesten Werkzeuge für diesen Zweck ist *GTKWave* [gtk14]. Vorteile dieses Ansatzes sind seiner Einfachheit und Transparenz. Nachteile bestehen in der Beschränkung auf Signale und Ports. Besonders TL-Modelle auf hohem Abstraktionsniveau setzen häufig native Datentypen ein. Außerdem ist der Aufbau von *Traces* besonders bei tiefen Objekthierarchien umständlich. Eine geeignete Ergänzung bildet der auf *GreenControl* aufgebaute *Analysis und Visibility* Service *GreenAV* [gre13]. Der Service erlaubt den globalen Zugriff auf alle Systemparameter (GS_PARAMS) und *GreenReg*-Register über die in Abschnitt 3.5.2 beschriebene Konfigurations-*Middleware*. Alternativen zum integrierten *Tracing*-Mechanismus von *SystemC* werden in fast allen kommerziellen Simulatoren angeboten. *High-End* Simulatoren wie *Mentor Modelsim (Questa)* [que14] oder *Cadence Incisive* [inc14a] verwenden zur Simulation von *SystemC*/TLM-Entwürfen die gleichen aus VHDL- oder Verilog bekannten Schnittstellen. Die Benutzerschnittstellen dieser Werkzeuge erlauben es, Signale zur Laufzeit beim Simulationskernel zur Aufzeichnung anzumelden und deren zeitlichen Verlauf auf unterschiedliche Weise, zum Beispiel als *Waveform* oder Liste, darzustellen. Dadurch können Eingriffe in den Quellcode vermieden werden, was den Entwicklungsprozess vereinfacht.

Das *Tracing* von Transaktionen gestaltet sich schwieriger, da es hier im Gegensatz zu Signalen und Registern keine direkte Entsprechung zwischen Transaktionsebene und Register-Transfer-Ebene gibt. Es ist theoretisch möglich komplexe Objekte wie die TLM-*Payload* mit *sc_trace* aufzuzeichnen, sofern *Tracing*-Funktionen für alle *Payload*-Elemente existieren. Dies erscheint aber wenig zweckmäßig, da die so erfassten Daten sehr unübersichtlich sind und nicht effizient ausgewertet werden können. Nach meiner Erkenntnis gibt es zurzeit keine standardoffene Lösung für dieses Problem. Proprietäre Werkzeuge wie *Cadence VSP* behelfen sich mit für ihre Simulatoren speziell angepassten TLM- und *SystemC*-Bibliotheken. In diesen Bibliotheken werden TLM-Komponenten wie *FIFOs* oder *Sockets* transparent um Inspektionsschnittstellen ergänzt. Diese erlauben es dem Simulator, den Weg einer Transaktion durch das System nachzuvollziehen und für Analysezwecke aufzuzeichnen.

Essentiell für *Debugging* und Analyse von Simulationsmodellen ist die Möglichkeit zur Generierung strukturierter Terminalausgaben. Die dafür erforderliche Grundfunktionalität ist bereits in *SystemC* enthalten. So existieren überladene *Stream*-Operatoren für alle *Built-in*-Datentypen, sowie Funktionen zur Abfrage und Ausgabe von Modulnamen und der aktuellen Simulationszeit.

Außerdem können mit Hilfe der Funktion *sc_report* und den zu deren vereinfachten Benutzung definierten Makros SC_REPORT_INFO, SC_REPORT_WARNING, SC_REPORT_ERROR und SC_REPORT_FATAL Ausgaben generiert werden, die automatisch Informationen über die Position im Quellcode, den verursachenden Prozess und die Simulationszeit beinhalten. Für Meldungen der Stufen WARNING oder höher werden zu dem Ausnahmen (*Exceptions*) generiert, die an geeigneter Stelle abgefangen werden können.

Da Virtuelle Plattformen in aller Regel prozessorzentrische Systeme sind, kommt dem Zusammenspiel von Hardware und Software besondere Bedeutung zu. Der Entwickler muss in der Lage sein, die auf einem oder mehreren Prozessoren laufende Software zu analysieren. Dazu ist es erforderlich, Haltepunkte zu setzen (*Breakpoints*) und Speicherstellen zu überwachen (*Watchpoints*). Das verbreitetste Werkzeug für diesen Zweck ist der GNU-*Debugger* GDB [gdb14]. Der GDB-*Debugger* ist hochflexibel, rekonfigurierbar und unterstützt fast alle aktuellen Prozessorarchitekturen. GDB besteht aus einer Nutzerschnittstelle, einer Symbolseite und einer Architekturseite (*Target Side*). Die Symbolseite dient der Analyse von Objektdateien und der Zuordnung der darin enthaltenen *Debug*-Informationen zum Quellcode. Die Architekturseite ist für die Ausführung des Programmes und die Kommunikation mit der physikalischen oder virtuellen Hardware verantwortlich. Zur Anbindung von Prozessorsimulatoren stehen zwei unterschiedliche Mechanismen zur Verfügung. Im ersteren laufen Simulator und GDB auf dem selben System. Die Schnittstelle zwischen Simulator und GDB ist in diesem Fall prozedural. Der zweite Mechanismus unterstützt die Fernsteuerung von Simulatoren im Netzwerk über eine *Remote*-Schnittstelle. Dies ist vorteilhaft, wenn das Zielsystem kein leistungsfähiges Betriebssystem besitzt oder ein zusätzlicher *Debugger-Prozess* das System zu sehr belasten und damit die Simulationsgeschwindigkeit senken würde. Aus diesen Gründen verwenden Virtuelle Plattformen in der Regel die *Remote*-Schnittstelle. Der *Host*-Rechner ist dabei mit einem vollständigen GDB ausgestattet. Das Zielsystem verfügt lediglich über eine GDB-*Stub* genannte Rumpfimplementierung, die zum Beispiel über eine serielle Schnittstelle oder TCP/IP mit dem *Host* kommuniziert.

Im folgenden werden die für *SoCRocket* gewählten oder neu entwickelten Lösungen für den *Debug*-Zugriff auf Modelle, die Simulationsanalyse, Transaktionsaufzeichnung und Ausgabeformatierung erläutert.

3.7.2 Debug-Zugriff

Wie bereits erwähnt existiert mit dem TLM2.0 *Debug Transport Interface* (DTI) ein Standard zur Realisierung des *Debug-Zugriffes* auf Speicherelemente. Da sich dieser Mechanismus als sehr zweckmäßig erwiesen hat und in vielen Werkzeugen verwendet wird (z.B. GDB), wurde DTI in die *SoCRocket*-Infrastruktur integriert. Ein Modell erhält ein DTI durch Erbung einer Busschnittstelle von einer Bibliotheksbasisklasse. Die *Debug*-Transportfunktionen nutzen die gleichen Verhaltens-*Callbacks* wie der blockierende Transportpfad. Bei Eintreffen eines *Debug Transport Requests* ruft die Busschnittstelle unmittelbar die Funktion *exec_func* auf, die vom Verhaltensteil des Modelles überschrieben werden muss. Die Funktion erhält das TLM-*Payload*-Objekt, ein Verzögerungsargument und einen *Debug*-Indikator als Aufrufparameter und liefert die Anzahl gelesener oder geschriebener Bytes zurück:

```
virtual uint32_t exec_func(payload_t &gp, sc_time &delay, bool is_debug);
```

Im Falle blockierender Kommunikation (Aufruf aus *b_transport*) wird der Rückgabewert ignoriert. Der *Debug*-Transportpfad ignoriert den Parameter *delay* und läuft komplett verzögerungsfrei ab. Dazu müssen mit Hilfe des *is_debug* Schalters alle Synchronisationsaufrufe (*wait*) im Verhalten abgeschaltet werden. Darüber hinaus dürfen *Debug*-Zugriffe keine Nebeneffekte verursachen. Funktionen, die den Zustand eines Modells ändern, müssen daher ebenfalls mit Hilfe von *is_debug* ausgeklammert werden. Ein Sonderfall tritt ein, wenn ein *Debug*-Transport die Konsistenz des Systems gefährdet. Dies ist unter anderem der Fall, wenn eine Schreiboperation Speicherinhalte ändert, die im *Cache* einer CPU gepuffert wurden. In diesem Fall muss für die Invalidierung der entsprechenden *Cache*-Zeilen gesorgt werden.

3.7.3 Analyse-API

Zum *Tracing* von *SystemC*-Signalen und -Ports kann die im *SystemC*-Standard spezifizierte Funktion *sc_trace* verwendet werden. Wie bereits erwähnt ist diese in vielen Fällen nicht hinreichend und erfordert einen Eingriff in den Quellcode der Modelle. Für *SoCRocket* wurde daher eine auf der Konfigurations-*Middleware GreenControl* und dem *Analysis und Visibility* Service *GreenAV* aufbauende Analyse-API entwickelt. Wie in Abschnitt 3.5.2 beschrieben, verwendet *SoCRocket GreenControl* zur Laufzeitkonfiguration seiner Komponenten. Konfigurationsparameter werden dazu in einer Kapselungsklasse verpackt, was deren zentrale Verwaltung in einer Parameterdatenbank mit Hilfe eines globalen Namensraumes ermöglicht. Da die Kapselungsklasse GS_PARAM im System Rückruffunktionen für Lese- und Schreibzugriffe registriert, eignet sich dieser Mechanismus ebenfalls zur Aufzeichnung zeitlich bedingter Änderungen in *Waveforms* und *Trace*-Dateien. Diese Funktionen werden durch *GreenAV* bereitgestellt. Zur Organisation des Zugriffs werden alle Analyseparameter die keine Register oder Konfigurationsparameter sind, einem Parameterfeld *performance_counters* zugeordnet. Dieses Feld wird durch alle *SoCRocket*-Komponenten implementiert. Art und Anzahl der Analyseparameter sind abhängig von der Natur des Modelles. Zum Beispiel stellen die *Caches* Zähler für *Cache-Hits*, *Cache-Misses* und *Bypass*-Operationen bereit. Darüber hinaus können je nach Bedarf zusätzliche Parameter eingeführt werden. Dazu genügt es, eine beliebige globale Variable in einem GS_PARAM zu kapseln und dem Feld *performance_counters* zuzuweisen. Der Mechanismus erlaubt es, Analyseparameter an jeder beliebigen Stelle des Quellcodes mit Hilfe eines hierarchischen Pfades zu adressieren:

```
<module_path>.performance_counters.<param_name>
```

Eine vollständige Liste aller Analyseparameter im System kann [Sch12a] entnommen werden. Der Code in Abbildung 3.28 verdeutlicht die grundlegenden Zugriffstrategien.

```
1 // Zeiger auf Parameter-API
2 gs::conf::cnf_api * m_api = gs::cnf::GCnf_Api::getApiInstance(NULL);
3
4 // Erzeuge Vector alle Parameter im System (alle inst. Modelle)
5 std::vector<std::string> plist = m_api->getParamList();
6
7 // Parameter zurueckliefern fuer Lese- oder Schreibzugriff
8 gs::gs_param_base * hits =
9   m_api->getParam("top.mmu_cache.icache.performance_counters.read_hits");
```

Abbildung 3.28: Zugriff auf Analyseparameter mit *GreenControl*

In Zeile 2 wird ein Zeiger auf die Parameter-API generiert. Mit Hilfe der Funktion *getParamList* (Zeile 5) kann dann ein *Vector* aller Parameter zur Weiterverarbeitung oder Ausgabe erzeugt werden. Zum Auffinden einzelner Parameter dient die Funktion *getParam* (Zeile 9), welcher der hierarchische Name des aufzufindenden Parameters übergeben werden muss. Im Folgenden soll kurz beschrieben werden, wie zeitliche Änderungen von Analyseparametern in *SoCRocket* mit Hilfe von *GreenAV* aufgezeichnet werden können (Abbildung 3.29).

```
1  // GreenAV Instanz erzeugen
2  gs::av::GAV_Plugin analysisPlugin("AnalysisPlugin");
3
4  // Zeiger auf Parameter-API
5  gs::cnf::cnf_api * m_api = gs::cnf::GCnf_Api::getApiInstance(NULL);
6
7  // Zeiger auf GreenAV-API
8  boost::shared_ptr<gs::av::GAV_Api> mainGAVApi =
9   gs::av::GAV_Api::getApiInstance(NULL);
10
11 // Tracing mit VCD-Datei
12 gs::av::OutputPlugin_if * cache_vcd =
13  mainGAVApi->create_OutputPlugin(gs::av::VCD_FILE_OUT, "cache.vcd");
14
15 // Tracing mit Text-Datei
16 gs::av::OutputPlugin_if * cache_vcd =
17  mainGAVApi->create_OutputPlugin(gs::av::TXT_FILE_OUT, "cache.log");
18
19 // Parameter zum Trace hinzufuegen (Beispiel VCD)
20 mainGAVApi->add_to_output(cache_vcd,
21  mainApi->getPar("top.mmu_cache.icache.performance_counter.read_hits");
```

Abbildung 3.29: *Tracing* von Analyseparametern mit *GreenAV*

Zeilen 1-8 zeigen die Instantiierungen der erforderlichen Schnittstellen zu *GreenControl* und *GreenAV*. In Zeile 12 wird eine *Waveform*-Datei im VCD-Format angelegt. Alternativ können Daten als Liste in einer Textdatei ausgegeben werden (Zeile 16). In beiden Fällen wird im Anschluss die Funktion *add_to_output* der *GreenAV*-API genutzt, um dem *Trace* Parameter hinzuzufügen.

Wie auch *GreenControl* wurde *GreenAV*, unter dem Dach von *GreenSoCs*, schwerpunktmäßig an der TU Braunschweig entwickelt. *SoCRocket* setzt diese Infrastruktur erstmalig produktiv in einem *Open Source*-Projekt ein und tritt damit den Beweis für dessen praktische Einsetzbarkeit an. Eine detaillierte Beschreibung aller Funktionen von *GreenAV* kann [Sch] entnommen werden.

3.7.4 Transaktionsaufzeichnung (Tracing)

Eine besonders zweckmäßige Form der Erfassung von Transaktionsverläufen auf Systemebene sind *Message Sequence Charts* (MSC). Mit Hilfe von MSCs können Nachrichtenfolgen zwischen kommunizierenden Objekten auf einheitliche Weise dargestellt werden. Da auf Transaktionsebene Kommunikation zwischen Simulationsmodellen abhängig vom Abstraktionsniveau durch einen oder mehrere zusammengehörige Funktionsaufrufe modelliert wird, ist die Wahl dieser Präsentationsform naheliegend. Zur Generierung von MSCs für Transaktionsverläufe in *SoCRocket*-Simulationen wurde die Helferklasse *msclogger* entwickelt. Die Klasse *msclogger* ist ein *Singleton* und stellt dem System eine statische API zur Erfassung von TLM-Transfers bereit (Abbildung 3.9).

API-Funktion	Beschreibung
void msc_start(filename, nodes)	Aufzeichnung starten
void msc_end()	Aufzeichnung beenden
void forward(...)	*Forward*-Transport
void backward(...)	*Backward*-Transport
void return_forward(...)	*Forward*-Return
void return_backward(...)	*Backward*-Return

Tabelle 3.9: MSC-Logger API (vereinfacht)

Die API enthält für jeden möglichen TLM-Transportpfad eine Funktion. Diese Funktionen werden in den TLM-*Sockets* (z.B. Basisklassen *AHBMaster* oder *AHBSlave*) direkt vor dem Aufruf einer Transportfunktion ausgeführt. Die jeweilige Komponente übergibt dabei Zeiger auf sich selbst, den betroffenen TLM-*Socket*, das TLM-*Payload*-Objekt und die aktuelle Pfadverzögerung. Im Falle nichtblockierender Kommunikation wird zusätzlich die TLM-Phase übergeben. Der MSC-Logger generiert aus den so übermittelten Information eine Kommandodatei für den frei verfügbaren MSC-Renderer *Mscgen* [McT14]. Abbildung 3.30 verdeutlicht den Aufbau eines solchen Kommandofiles anhand eines Beispiels.

```
1  msc {
2
3     hscale="2";
4
5     Master, ahbctrl, ahbmem;
6
7     Master=>ahbctrl [label = "BEGIN_REQ(0x824a0b0/610ns/0s)",
8     linecolour = "#824a0b0", textcolour = "#824a0b0"];
9
10    ahbctrl>>Master [label = "TLM_ACCEPTED(0x824a0b0/610ns/0s)",
11    linecolour = "#824a0b0", textcolour = "#824a0b0"];
12
13    ahbctrl=>ahbmem [label = "BEGIN_REQ(0x824a0b0/610ns/0s)",
14    linecolour = "#824a0b0", textcolour = "#824a0b0"];
15
16    ...
17 }
```

Abbildung 3.30: MSC-Kommandofile generiert mit *mscgen*

Das Kommandofile registriert zunächst die *SystemC*-Namen der zur Transaktionsverfolgung vorgesehenen Komponenten. Im gegebenen Beispiel sind das eine *Testbench* (*Master*), der AHB-Bus (*ahbctrl*) und der AHB-Speicher (*ahbmem*) (Zeile 4). Es handelt sich um nichtblockierende Kommunikation. Zeilen 6-7 repräsentieren den *Forward*-Transport von *Master* zu *AHBCTRL* mit der TLM-Phase *BEGIN_REQ*. Der Zeiger auf das *Payload*-Objekt dient zur eindeutigen Identifizierung und wird auch zur Erzeugung eines Farbcodes verwendet. Die Annahme von *BEGIN_REQ* wird durch *ahbctrl* durch das Senden von *TLM_ACCEPTED* auf dem *Forward*-Return-Pfad bestätigt (Zeilen 10-11). Im Anschluss wird die Transaktion an *ahbmem* weitergeleitet. Die Ausgabe des MSC-Renderers für zwei verschachtelte nichtblockierende Transaktionen ist in Abbildung 3.31 dargestellt.

Abbildung 3.31: MSC für nichtblockierende Pipeline-Transaktionen (AHB)

Mit Hilfe der erzeugten Ausgabe lässt sich die Verschachtelung einzelner Transaktionen, insbesondere die Überlappung von Transaktionsphasen am AHB-Bus gut nachvollziehen. Darüber hinaus lässt sich das gewählte *Logging*-Format mit geringem Aufwand zur Gewinnung zusätzlicher Informationen, wie Busauslastung und Transaktionsverteilung verwenden. Der Mechanismus ist für externe Werkzeuge transparent und kann vollständig deaktiviert werden.

3.7.5 Ausgabeformatierung und Filterung

SoCRocket-Komponenten sind *SystemC*-Module und können daher alle durch *SystemC* unterstützen Methoden zur Ausgabeformatierung und Filterung verwenden. Dazu zählen die Standardausgabe mit *Streaming*-Operatoren von C++, Ausgaben mit *printf* im C-Stil und die integrierte Funktion *sc_report* mit der Meldungen gemäß ihrem Schweregrad (*Severity*) in abgestufter Form behandelt werden. Für *SoCRocket* wurde eine Methode entwickelt, die die natürliche Handhabbarkeit von C++-*Output Streams* mit den Fähigkeiten der Ausgabefilterung des integrierten *SystemC*-Mechanismus kombiniert. Der *SoCRocket-Verbosity*-Mechanismus stellt fünf vordefinierte Ausgabestufen bereit: *debug*, *info*, *report*, *warning* und *error* (Abb. 3.32).

```
1 // Regulaerer C++ Output Stream
2 std::cout << value << std::endl;
3
4 // SoCRocket Output Streams
5 v::error << value << v::endl;
6 v::warn << value << v::endl;
7 v::report << value << v::endl;
8 v::info << value << v::endl;
9 v::debug << value << v::endl;
```

Abbildung 3.32: SoCRocket - Ausgabeformatierung

Das System kann durch Angabe einer Ausgabestufe (*Verbosity Level*) derart konfiguriert werden, dass es Ausgaben die über der angegebenen Stufe liegen unterdrückt. Dies geschieht zur *Compile*-Zeit und nicht zur Laufzeit, wodurch gefilterte Ausgabekommandos aus dem Code entfernt werden und dadurch in der Simulation keinen *Overhead* verursachen können. Dies wird ermöglicht, da das System die Ausgabestufe nach der Konfiguration als Konstante betrachtet. Ausgabekommandos oberhalb der gewählten Stufe stellen somit nicht erreichbaren Code dar. Der GCC-Compiler führt die entsprechende Optimierung ab der zweiten Optimierungsstufe durch (-O2). Weitere Details zur Benutzung und Funktion des dargestellten Ausgabemechanismus können dem *SoCRocket*-Nutzerhandbuch [Sch12b] und dem Quellcode entnommen werden. Die Implementierung befindet sich im Verzeichnis *common* (siehe Anlage B) und umfasst die Klassen *color*, *number* und *msgstream*.

3.8 Verifikation

Zur Unterstützung der Verifikation von Komponenten wurde für *SoCRocket* ein zweistufiger Ansatz aus *Unit*-Tests und softwaregesteuerten Systemtests entwickelt. Sofern ein RTL-Referenzentwurf vorhanden ist, es sich also nicht um eine Komponentenneuentwicklung auf Grundlage einer Spezifikation handelt, können Verhalten und *Timing* mit Hilfe von *SystemC*/VHDL Co-Simulationen abgeglichen werden.

3.8.1 SoCRocket Testumgebung

Unit-Tests werden zur initialen Verifikation von Modulfunktionen und Zeitverhalten eingesetzt. Dafür sind verschiedene Grundfunktionen erforderlich, die in der Regel zwischen mehreren Tests geteilt und wiederverwendet werden können. Es empfiehlt sich daher der Einsatz einer Basisklasse, die diese Grundfunktionen in sich vereint und von den verschiedenen Testinstanzen ererbt werden kann. Abbildung 3.33 zeigt ein Beispiel für eine derartig strukturierte Testumgebung.

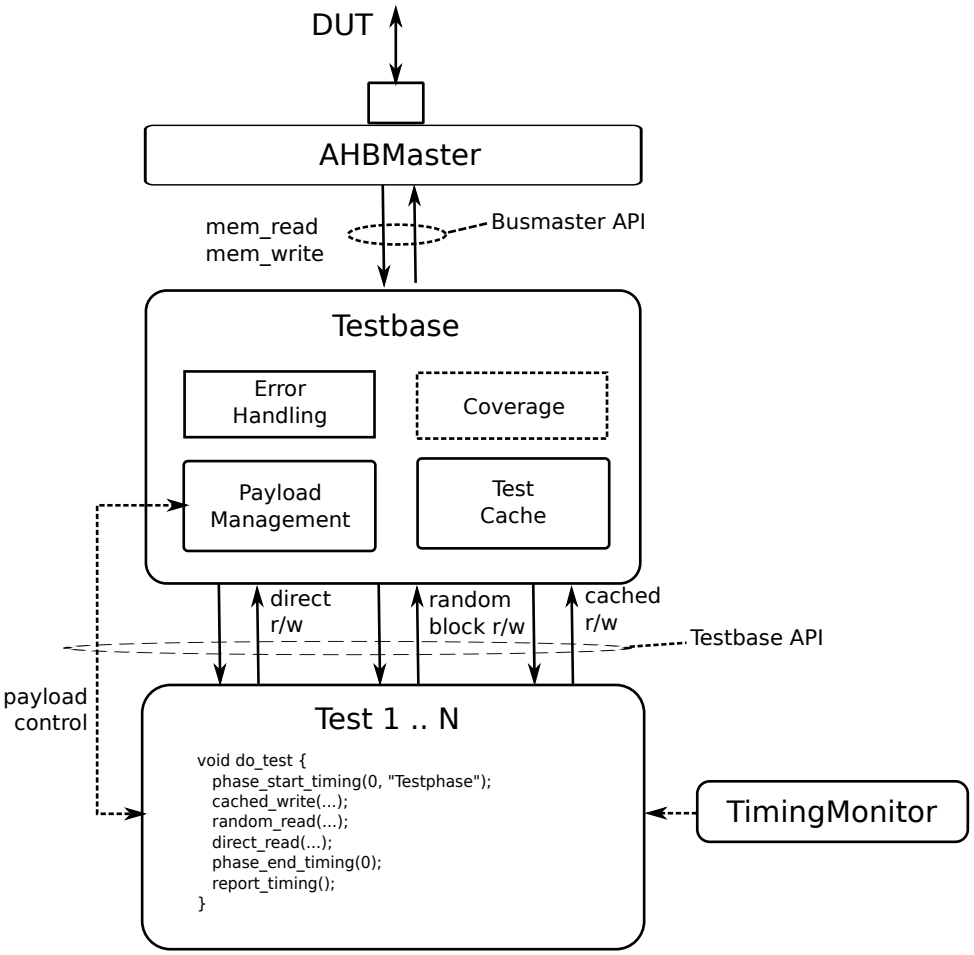


Abbildung 3.33: Umgebung für Unit-Tests

Die Testklasse *Testbase* erbt die Bibliotheksbasisklasse *AHBMaster* und erhält so eine AHB-*Master*-Schnittstelle über die verschiedenste Verbindungs- und Peripheriekomponenten angesteuert werden können. Der AHB-*Master* kann wahlweise für blockierende und nichtblockierende Kommunikation konfiguriert werden und stellt *Testbase* entsprechende Zugriffsfunktionen bereit. *Testbase* kann diese *Busmaster*-Schnittstelle direkt an die Testimplementierungen (Test 1 .. N) weiterreichen (Schnittstelle *direct r/w*) oder zum Aufbau komplexerer Testfunktionen verwenden. Zwei in der *SoCRocket*-Infrastruktur bereitgestellte Methoden sind die Generierung von zufallsgesteuerten/blockgesteuerten Operationen für einen definierten Speicherbereich (Schnittstelle *random/block r/w*) und der gepufferte Zugriff mit Hilfe eines Test-*Caches* (*cached r/w*). Zum Abgleich der Ergebnisse von Lese- und Schreiboperationen bietet sich die Nutzung des *Debug Transport Interfaces* (DTI) an (siehe Abschnitt 3.7). Zum Test von Leseoperationen können die entsprechenden Speicherstellen zunächst über den *Debug*-Transportpfad beschrieben werden. Im Anschluss werden die Daten dann über den blockierenden oder den nichtblockierenden Transportpfad zurückgelesen und mit den zuvor geschriebenen Werten verglichen. Im gleichen Sinne können Schreiboperationen abgeglichen werden, indem Daten zuerst über den blockierenden oder nichtblockierenden Transportpfad geschrieben und danach über den *Debug*-Transportpfad zurückgelesen werden. Bei direktem Zugriff auf die Busschnittstelle kann der Abgleich der Daten in der Testimplementierung vorgenommen werden.

Die Nutzung der direkten Lese-/Schreibschnittstelle eignet sich besonders für das gezielte Testen einzelner Speicherstellen wie Steuerregister. Als Alternative zur Nutzung des DTI können dabei erwartete Referenzergebnisse vorgegeben werden. Zum Vergleich von Transaktionsergebnissen mit erwarteten Referenzergebnissen stellt die Testumgebung die Funktion *check* bereit. Die Funktion lagert den Ergebnisvergleich in einen separaten *Thread* aus. Dies ist erforderlich, da die Testimplementierung im Falle nichtblockierender Kommunikation den Buszugriff nicht

durch Kontrollfunktionen blockieren darf. Im Falle von AHB-Transfers muss es möglich sein, mindestens eine weitere Transaktion zu starten, bevor die *Response* der ersten Transaktion abgeschlossen ist. Ansonsten geht der Vorteil der *Pipeline*-Modellierung im AT-Modus verloren (siehe Abschnitt 3.2.2). Dementsprechend müssen AXI-Tests eine höhere Anzahl von *in-flight* Transaktionen unterstützen, da das Protokoll deren Umsortierung während der Verarbeitung erlaubt (Abschnitt 3.2.4). Abbildung 3.34 verdeutlicht die Nutzung der direkten Lese-/Schreibschnittstelle und der Funktion *check*. Das Beispiel entstammt einem Test des CPU-*Cache*-Subsystems (Test 3 von *mmu_cache*, siehe Anhang B).

```
1  ...
2  // Cache-Zustand pruefen (I-Cache Configuration Register)
3  // ------------------------------------------------------
4  // args: Adresse, Daten, Bytes, ASI, flush, flushl, lock, Debug-Pointer
5  direct_dread(0x8, get_datap_clean(), 4, 2, 0, 0, 0, get_debugp_clean());
6
7  // Abgleich I-Cache Configuration Register gegen erwarteten Wert
8  // -------------------------------------------------------------
9  // [29:28] repl=0b11, [26:24] sets=0b011, [23:20] ssize=0b0000 (1kB)
10 // [19] lram=0, [18:16] lsize=0b010 (2^2 = 4 words per line)
11 // [15:12] lramsize=0, [11:4] lramstart=0x8e, [3] mmu=0
12 // args: Check ID, Daten, Referenzdaten, Bytes
13 check(2, get_datap(), get_refp_word(0x330208f0), 4);
14 ...
```

Abbildung 3.34: Testschnittstelle *direct r/w*

Zum Test des CPU-*Cache*-Subsystems wird die in Abbildung 3.33 gezeigte *AHBMaster*-Klasse durch einen CPU-Datensockel und einen CPU-Instruktionssockel ersetzt. Der Test liest das Konfigurationsregister des Instruktionscaches mit Hilfe der Funktion *direct_read* (Zeile 3). Das Register befindet sich auf Adresse 0x8 im Adressraum 0x2 (ASI 0x2 - Systemregister). Die Argumente der Funktion werden durch den entsprechenden Bus-*Master* auf die TLM-*Payload* oder deren Erweiterungen abgebildet. Um aufwendige Kopieroperationen in der Kommunikation zwischen Testimplementierung, Testbasis und Bus-*Master* zu vermeiden, werden die *Payload*-Daten mit Hilfe eines Zeiger direkt weitergereicht. Zur effizienten Implementierung dieses Mechanismus verfügt *Testbase* über eine *Payload*-Management-Einheit. Das *Payload*-Management unterhält einen statisch allokierten Speicherpool, welcher der Testimplementierung Zeiger auf Speicher zur Aufnahme von *Payload*-Daten, Referenzergebnissen und *Debug*-Informationen bereitstellt. Der Speicherpool ist als Ringpuffer implementiert, wodurch der Speicher während der Simulation mehrfach wiederverwendet werden kann. Außerdem wird durch den Einsatz des *Payload*-Managements die dynamische Allokation von Speicher zur Laufzeit verhindert (kein *malloc*). In Abbildung 3.34 wird die Funktion *get_datap_clean* eingesetzt (Zeile 5), um einen Zeiger auf einen 32-Bit-Speicherbereich zu generieren. Die Funktionen *get_debugp_clean* und *get_refp_word* liefern *Debug*- und Referenzdatenzeiger (Zeilen 5 und 13). Zusätzliche Informationen zum Aufbau der *Payload-Control*-Schnittstelle können dem Quellcode und der Quellcodedokumentation entnommen werden.

Zufallsgesteuerte Zugriffe oder Blockzugriffe werden in *Testbase* initialisiert und gegebenenfalls in mehrere Einzelzugriffe zerlegt. Die Testimplementierung initiiert lediglich eine Reihe von Transaktionen, erhält aber keine Informationen über die letztendlich gelesenen oder geschriebenen Daten. Daher muss die Ergebniskontrolle ebenfalls in die Basisklasse verschoben werden. Die Testschnittstelle *random/block r/w* stellt die in Tabelle 3.10 dargestellten Funktionen bereit.

Funktion	Beschreibung
random_read(size)	Zufälliges Lesen mit Zugriffsweite size im Speicherbereich der Testbench
random_write(size)	Zufälliges Schreiben
readCheck(start, end, size, fail)	Blockweises Lesen von Adresse start zu Adresse end mit Zugriffsweite size
writeCheck(start, end, size, fail)	Blockweises Schreiben

Tabelle 3.10: Verifikationsschnittstelle *random/block rw*

In der gegenwärtigen Implementierung generieren die Funktionen *random_read* und *random_write* nur eine einzelne Transaktion. Als einziger Aufrufparameter kann die Zugriffsweite bestimmt werden. Adresse und Schreibdaten werden zufällig bestimmt. Die Grenzen des Speicherbereiches werden für AMBA-Komponenten aus der Busadresse der *Master*-Schnittstelle abgeleitet (*haddr/hmask*). Die Funktionen *readCheck* und *writeCheck* erzeugen dagegen mehrere Transaktionen. Zur Markierung des Speicherbereichs können eine Start- und eine Endadresse angegeben werden. Der Adressbereich wird in aufsteigender Adressreihenfolge mit Transaktionen einer gegebenen Zugriffsweite gelesen oder beschrieben. Im Falle der Funktion *readCheck* wird der Speicherbereich dazu zunächst mit Hilfe des DTIs mit Zufallsdaten gefüllt. Die geschriebenen Daten werden in einem Puffer lokal zwischengespeichert und danach mit den Ergebnissen der Leseoperationen verglichen. Die Funktion *writeCheck* setzt ebenfalls das DTI zum Abgleich der Ergebnisse ein. Dazu werden zunächst Zufallsdaten über den blockierenden oder den nichtblockierenden Transportpfad geschrieben. Danach wird der gegebene Adressbereich mit Hilfe des DTIs ausgelesen. Zur Unterstützung von Negativtests kann den Blockzugriffsfunktionen der Parameter *fail* übergeben werden. Wenn *fail* gesetzt ist, generieren die Funktionen einen Fehler, sofern die entsprechende Operation, anders als beabsichtigt, erfolgreich war. Auf diese Weise lassen sich Zugriffe auf abgeschaltete oder nicht vorhandene Speicherelemente prüfen. Die Verifikationsschnittstelle *random/block rw* kann in Anlehnung an die vorgestellten Funktionen erweitert werden, um die Implementierung abgeleiteter Tests weiter zu vereinfachen.

Die dritte durch *Testbase* bereitgestellte Verifikationsschnittstelle (*cached r/w*) enthält gepufferte Zugriffsfunktionen. Dabei erfolgt der Datenabgleich automatisch mit Hilfe eines in *Testbase* integrierten *Caches*. Der *Cache* spiegelt alle durch die Testimplementierung geschriebenen Daten und verfügt über ein *Snooping Interface*. Dadurch werden kombinierte Tests unter Benutzung mehrerer unabhängiger *Master* möglich. Die Schnittstelle *cached rw* ähnelt der direkten Schnittstelle *direct rw*. Es wird jeweils eine Funktion zum gerichteten Lesen beziehungsweise Schreiben bereitgestellt. Beiden Funktionen (*check_read* und *check_write*) werden eine Zieladresse, ein Datenzeiger und eine Datenlänge übergeben. Zum Abgleich der Ergebnisse können innerhalb der Testimplementierung das DTI oder die bereits beschriebene Funktion *check* eingesetzt werden. Bei Verwendung des DTI in Kombination mit nichtblockierenden Transaktionen müssen eventuell Wartezyklen eingefügt werden, da die Simulationsergebnisse nicht unmittelbar nach dem Funktionsaufruf zur Verfügung stehen. In jedem Fall speichert die Funktion *check_write* eine Kopie aller geschriebenen Daten in einem *Cache*. Der *Cache* wurde als *Hashmap* implementiert und nutzt die Zieladresse als Schlüssel. Dadurch können auch große Speicherbereiche mit moderatem Aufwand abgebildet werden. Die Funktion *check_read* vergleicht gelesene Daten mit dem *Cache*-Inhalt und meldet bei Abweichungen einen Fehler. Das *Snooping Interface* registriert Schreibzugriffe anderer Test-*Master*. Im Falle eines Treffers werden die Daten im lokalen Test-*Cache* überschrieben.

Wie vorstehend beschrieben erfolgen bei Benutzung der Verifikationsschnittstellen *random/block rw* und *cached rw* sowohl Transaktionsgenerierung als auch Ergebnisabgleich in transparenter Weise innerhalb der Basisklasse *Testbase*. Detektierte Fehler werden direkt an das ebenfalls in *Testbase* implementierte *Error Handling* gemeldet. Der *Error Handler* besteht in seiner einfachsten Form aus einem Zähler, der durch jeden aufgetretenen Fehler inkrementiert wird. Dieser kann am Ende der Simulation als Rückgabewert der Hauptfunktion (z.B. *return* von *sc_main*) verwendet werden. Die Überprüfung des Rückgabewertes in einem übergeordneten

Skript ermöglicht die einfache Implementierung von Regressionen. Der *Error Handler* kann beliebig zur Aufnahme von Zusatzinformationen, wie der ID des fehlgeschlagenen Testvektors, der Quellcodeposition oder der betreffenden Simulationszeit erweitert werden. Die automatische Registrierung von Simulationsfehlern wird ebenfalls durch die mit der direkten Verifikationsschnittstelle verwendeten Funktion *check* unterstützt. Nur bei manueller Ergebnisskontrolle innerhalb der Testimplementierung (z.B. mit DTI) müssen Fehler direkt beim *Error Handler* gemeldet werden.

Mit den beschriebenen Methoden lassen sich Ergebnisse von Lese- und Schreiboperationen auf einfache Weise abgleichen. In vielen Fällen ist jedoch nicht nur das Transaktionsergebnis, sondern auch der Transportpfad zu dessen Zustandekommen von Bedeutung. Diese Information kann durch das *Tracing* von Transaktionen erlangt werden (siehe Abschnitt 3.7). Zum Einsatz in *Unit*-Tests ist dieser Ansatz jedoch zu kompliziert und langsam. Der Abgleich des Transportpfades kann zur Laufzeit erfolgen. Das Aufzeichnen von Transaktionen ist daher nicht erforderlich. Eine einfache Alternative stellt die Verwendung eines Statusfeldes dar, dass als ignorierbare Erweiterung an die TLM-*Payload* angehängt werden kann. Jedes Modul welches die Transaktion auf ihrem Pfad durchquert, kann dem Statusfeld Informationen hinzufügen. Dadurch kann der Test den erwarteten Pfad durch den einfachen Vergleich mit einem Bitmuster abgleichen. Dieser Ansatz wurde zur Verifikation des *Cache*-Subsystems erprobt. Zur Implementierung des Statusfeldes wurde ein 32-bit Integer verwendet. Tabelle 3.11 verdeutlicht den Aufbau der Erweiterung.

Bit	Feld	Beschreibung
21	MMU-Status	0 - TLB Hit, 1 - TLB Miss
20-16	TLB	Nummer der TLB
15	Reserviert	
14	Frozen Miss	1 - Gelesene Daten werden nicht im Cache gespeichert, da dieser gefroren (*frozen*) ist.
13	Cache Bypass	1 - Cache nicht verwendet (abgeschaltet oder erzwungen (ASI))
12	ScratchPad	1 - Zugriff ans Scratchpad weitergeleitet
11-5	Reserviert	
4	Flush	1 - Transaktion hat Cache-Flush ausgelöst
3-2	Cache Oper.	00 - Read-Hit, 01 - Read-Miss 10 - Write-Hit, 11 - Write-Miss
1-0	Cache Set	Read/Write Hit: Cache-Set mit dem Hit Read Miss: Cache-Set mit den neuen Daten Write Miss: 0b00 (kein Update bei Write Miss)

Tabelle 3.11: *Payload*-Erweiterung (Statusfeld) zur Verifikation des Transportpfades

Die Felder der Statuserweiterung können im einfachsten Fall mit Hilfe von Makros gesetzt und ausgelesen werden. Wie in Abbildung 3.35 gezeigt kann der erwartete Zugriffspfad der Funktion *check* optional als zusätzlicher Parameter übergeben werden. Im Falle einer Abweichung meldet *check* den Fehler dem *Error-Handler* und gibt eine entsprechende Nachricht aus. Das *Code*-Beispiel wurde Test 8 des CPU-*Cache*-Subsystems entnommen (siehe Anlage A).

```
1 // Schreiboperation ueber Schnittstelle direct rw
2 direct_dwrite(0x01041000, get_datap_word(0x00000000), \
3         4, 8, 0, 0, 0, get_debugp_clean());
4
5 // Abgleich von Transportpfadinformationen mit Status-Erweiterung
6 check(8, get_datap(), get_datap(), 4, get_debugp(), TLBHIT);
7 check(8, get_datap(), get_datap(), 4, get_debugp(), CACHEWRITEMISS);
```

Abbildung 3.35: Überprüfung des Transportpfades mit Statuserweiterung und Funktion *check*

Die Funktion *get_debugp_clean* (Zeile 2) gehört zur Schnittstelle des *Payload*-Managers und liefert einen Zeiger auf 32-bit-Speicher aus dem Speicherpool. Dieser wird in *Testbase* als Statusfeld an die TLM-*Payload* angehängt. Die Funktion *get_debugp* gibt den Zeiger auf das zuletzt angeforderte Statusfeld zurück. Dieser wird in den Zeilen 3 und 4, gemeinsam mit den erwarteten Pfadeigenschaften (TLBHIT, CACHEWRITEMISS), an die Funktion *check* übergeben. Genau wie zur Überprüfung der übertragenen Daten, erfolgt die Verifikation im blockierenden Fall unmittelbar und im nichtblockierenden Fall in verzögerter Form, mit Hilfe eines separaten *Threads*.

Ein wichtiges Maß zur Bestimmung der Effektivität der durchgeführten Tests ist die Testabdeckung (Code-*Coverage*). Diese wurde in Abbildung 3.33 nur zur Vervollständigung der Darstellung als unabhängige Komponente von *Testbase* eingetragen. Zur Bestimmung der funktionalen Codeabdeckung von *SystemC*-Komponenten können existierende *Open Source*-Lösungen wie *gcov* verwendet werden [gco14]. Das *SoCRocket-Build*-System unterstützt die Berechnung der Testabdeckung in Regressionstests mit *gcov* und *lcov* (Anlage A).

Zur Verifikation des Zeitverhaltens wird ein vereinfachter Ansatz gewählt. Vordringliche Anwendungsfälle der *SoCRocket*-Plattform sind die Architekturexploration auf unterschiedlichen Abstraktionsebenen, sowie die Entwicklung hardwarenaher Software. Für diese Anwendungsfälle ist ein hochabstrakter Simulationsmodus mit vereinfachtem Modellverhalten und ein näherungsweise akkurater Simulationsmodus erforderlich. Zur Realisierung dieser Simulationsmodi werden die im TLM2.0 beschriebenen LT- und AT-Modellierungsstile eingesetzt. Die Ansprüche an das Zeitverhalten sind dementsprechend verschieden. Für den LT-Fall ist die zeitliche Genauigkeit unerheblich. Diese Abstraktion soll nur gerade genug Genauigkeit zum Start eines Betriebssystem, zur korrekten Ansteuerung von Hardwarekomponenten und zur Handhabung von *Interrupts* liefern. Der AT-Modus erfordert zur Durchführung detaillierter Explorationen eine weit höhere Genauigkeit. Diese sollte abhängig von der Bedeutung der jeweiligen Komponente im Gesamtsystem im Durchschnitt bei 90% liegen und 80% nicht unterschreiten. Beide Abstraktionsebenen sind nicht zyklengenau und dahingehend ausgelegt, durch kalkulierten Verzicht auf zeitliche Genauigkeit Simulationsgeschwindigkeit zu gewinnen. Zur *Timing*-Verifikation in *Unit*-Tests ist es daher nicht zwingend erforderlich, Einzeltransaktionen separat auf RTL-Referenzoperationen abzubilden. Es interessiert vielmehr nur die Gesamtlaufzeit von Tests oder Testphasen. Aufbauend auf dieser Erkenntnis wurde für *SoCRocket* ein *Timing Monitor* entwickelt, der es erlaubt Tests in eine beliebige Anzahl an Phasen zu unterteilen und deren Simulationszeit abzugleichen. Je kürzer die Testzeit oder je höher die Anzahl der Testphasen, desto größer die potentiellen Abweichungen. Dies wird auch in Abbildung 3.36 deutlich.

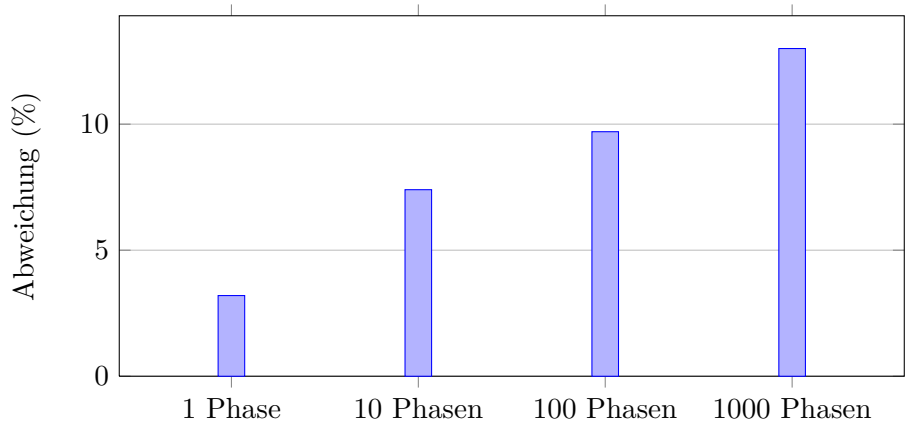


Abbildung 3.36: Abweichung in Abhängigkeit von der Testgranularität

Die Abbildung zeigt die gemessene durchschnittliche zeitliche Abweichung in einem Test der über den AHB-Bus mit zufallsgesteuerten Operationen auf SDRAM-Speicher zugreift. Der Test generiert 10000 Operationen. Wird der Test als eine einzige zusammenhängende Testphase betrachtet, so beträgt der Laufzeitunterschied im Vergleich zur *SystemC*/RTL Co-Simulation nur 3.2%. Wird der Test in eine höhere Anzahl gleich langer Phasen unterteilt, so steigt die durchschnittliche Abweichung pro Phase auf bis zu 13% (1000 Phasen mit je 10 Operationen). Grund dafür ist der höhere Einfluss gleichartiger Transaktionen, der zur Akkumulation systematischer Abweichungen führen kann, die in längeren Simulationen ausgeglichen werden. Außerdem kann es zu großen Abweichungen bei Einzeltransaktionen kommen, wenn das Modell Operationen in geänderter Reihenfolge abarbeitet. Derartige Effekte können an allen *Routern* und bei *out-of-order* Verarbeitung (z.B. AXI-Protokoll) auftreten. Abbildung 3.37 verdeutlicht die Funktion des *Timing Monitors*. Das Beispiel wurde einem Test des Speicher-*Controllers* entnommen (Test 3 von *MCTRL*, siehe Anlage B) und verwendet die Verifikationsschnittstelle *random/block rw*.

```
1 // Zeitnahme fuer Phase 3 starten
2 TimingMonitor::phase_start_timing(3, "64bit write to PROM");
3 // Blockweises 64bit-Schreiben im ROM-Speicherbereich
4 result |= writeCheck(rom_start, rom_end, 8, false);
5 // Zeitnahme fuer Phase 3 beenden
6 TimingMonitor::phase_end_timing(3);
7 ...
8 TimingMonitor::report_timing();
```

Abbildung 3.37: Überprüfung des Transportpfades mit Statuserweiterung und Funktion *check*

Der Beginn der Testphase wird durch das Kommando *phase_start_timing* eingeleitet. Dem Kommando kann eine Identifikationsnummer und eine textuelle Beschreibung übergeben werden. Bei Aufruf von *phase_start_timing* speichert der *Timing Monitor* die gegenwärtige Simulationszeit und die gegenwärtige Echtzeit (Systemzeit) gemeinsam mit der Phasenbeschreibung in einer *Hashmap* ab. Die Identifikationsnummer wird dabei als Zugriffsschlüssel verwendet. Nach Abschluss der Testphase werden die so gespeicherten Daten, mit Hilfe des Kommandos *phase_end_timing* (Zeile 6), um die abschließende Simulationszeit und Echtzeit ergänzt. Dieser einfache Mechanismus erlaubt beliebig viele und auch verschachtelte Zeitnahmen. Die Programmierschnittstelle des *Timing Monitors* erlaubt es, Simulationszeiten einzeln auszulesen. Außerdem kann ein *Timing*-Report zur Zusammenfassung aller erhobenen Daten generiert werden (Zeile 8).

Die vorgestellte Infrastruktur erlaubt es, *Unit*-Tests auf unterschiedlichen Abstraktionsniveaus wiederzuverwenden. AT-Modelle und RTL-Modelle werden in der Testimplementierung nicht blockiert, obwohl sie durch die gleichen Schnittstellen wie LT-Modelle gesteuert werden. Dies wird durch Fallunterscheidungen in der Testbasisklasse *Testbase* ermöglicht wird, die im Falle

nichtblockierender Kommunikation zusätzliche *Threads* erzeugt.

Nach Abschluss der *Unit*-Tests können neu entwickelte Komponenten in die VP integriert werden. Je nach Komplexität des Entwurfs empfiehlt sich die Entwicklung zusätzlicher Softwaretests. Dabei handelt es sich um Programme, die auf einem oder mehreren Prozessoren im System ablaufen und das zu testende Modell gezielt stimulieren. Für die im folgenden Kapitel beschriebenen Kernkomponenten zur Modellierung von DPUs konnten dazu die mit Komponententests des LEON-Prozessors aus der GRLIB verwendet werden. Die Vorgehensweise zur Implementierung und Ausführung von Software auf *SoCRocket*-VPs wird in Kapitel 6.1 anhand eines Beispiels erläutert. Eine Übersicht über die im Rahmen der Arbeit entwickelten *SystemC*/VHDL Co-Simulationsadapter befindet sich im Verzeichnis *adapters* der VP (siehe Anlage B).

4 Kernkomponenten zum Entwurf robuster Eingebetteter Systeme

Grundlage des *SoCRocket*-Systems ist eine *SystemC*/TLM2.0-Modellbibliothek, bestehend aus Kernkomponenten zum Aufbau robuster Echtzeitsysteme (Tab. 4.1). Wichtigster Bestandteil ist ein LEON2/3-Prozessorsimulator. Der Simulator besteht aus einer Integereinheit, die im Auftrag der ESA an der *Politecnico di Milano* entstand [Fos10] und im Rahmen dieser Arbeit grundlegend erweitert wurde. Die Erweiterungen bestehen in der Entwicklung einer flexiblen TLM-Schnittstelle, der Unterstützung von SparcV8-*Address Space Identifiers* (ASIs) zur Steuerung des Speicher-Subsystems, der Entwicklung von Caches, einer *Memory Management Unit* (MMU) und *Scratchpad*-RAM. Informationen zum Aufbau und Entwurf des Simulators befinden sich in Abschnitt 4.1. Des Weiteren gliedert sich die Bibliothek in Verbindungs- (4.2) und Peripheriekomponenten (4.3). Alle Komponenten unterstützen die im TLM2.0-Standard vorgeschlagenen Abstraktionsstufen *Loosely-Timed* und *Approximately-Timed* und setzen die in Abschnitt 3 entwickelte Infrastruktur praktisch um. Synthetisierbare VHDL-Hardwaremodelle der betrachteten Komponenten sind in der ebenfalls frei verfügbaren GRLIB-Bibliothek von *Aeroflex/Gaisler AB* zusammengefasst [gai13].

TLM	**Beschreibung**
leon2/3	LEON2/3-Prozessorsimulator mit Harvard Caches, MMU und Local-RAM
ahbctrl	AHB-Busmodell mit Plug & Play und Snooping
apbctrl	AHB/APB-Busbrücke
gptimer	*General Purpose Timer*
irqmp	*Multi-Processor Interrupt Controller*
mctrl	PROM, I/O, SRAM, SDRAM Speichercontroller
gen_mem	Generischer Speicher
ahbmem	Speichermodul mit AHB-Schnittstelle
apbuart	UART serial interface
socwire	SoCWire NoC-Schnittstelle
spacewire	SpaceWire Netzwerkschnittstelle

Tabelle 4.1: SoCRocket Kernkomponenten

4.1 LEON2/3 Prozessor Simulator

Übersicht

Die LEON3-CPU [Gai02] ist ein 32-Bit-Prozessor mit SparcV8-Architektur, der durch die schwedische Firma *Aeroflex Gaisler AB* [gai13] vertrieben wird. Der Prozessor liegt als synthetisierbares VHDL-Modell vor und kann unter den Bedingungen der GPL-Lizenz unbeschränkt für Forschung und Lehre eingesetzt werden. Darüber hinaus gibt es eine kommerzielle Lizenz, die den Einsatz für beliebige Anwendungsfälle erlaubt. Der LEON3-Prozessor ist eine komplexe RISC-Architektur mit 7-stufiger Pipeline, *Harvard Caches*, einer *Memory Management Unit*, *Scratchpad*-RAM und AMBA2.0-Busschnittstelle. Für sicherheitskritische Anwendungen wurden diverse Fehlerschutzmechanismen integriert, wie zum Beispiel *Single Event Upset* (SEU) Fehlerkorrektur in der Registerbank oder Speicherschutz für bis zu vier Fehler per *Cache Tag* oder 32-Bit-Speicherwort.

Die Korrektur von Fehlern wird unabhängig und für die Anwendungssoftware transparent durchgeführt [Gai10] (siehe auch Abschnitt 1.3).

Für den Prozessor ist ein GCC-Compiler-Backend für *Bare-Metal*-Implementierungen verfügbar. Darüber hinaus existieren Portierungen für diverse Echtzeitbetriebssysteme wie *RTEMS*, *eCos* oder *Embedded Linux* (*Snapgear*). Ebenfalls über *Aeroflex Gaisler* sind Simulatoren für Einzelprozessor- (TSIM) und Mehrfachprozessorsysteme (GRSIM) erhältlich. Die Simulatoren sind sehr schnell (> 1M Instruktionen/Sekunde), verfügen aber nur über geringe zeitliche Genauigkeit. Hauptanwendungsfall ist die Entwicklung von Software. Eine umfassende Systemexploration ist schon auf Grund der starren unterliegenden Architekturvorlagen nicht möglich. Aus diesem Grund initiierte die ESA die Entwicklung eines auf *SystemC* basierenden ISS. Der Simulator wurde mit Hilfe des *TrapGen*-Generators erzeugt [Fos10] und ist ebenfalls unter GPL-Lizenz frei verfügbar [tra]. Der *TrapGen*-LEON-ISS modelliert allerdings nur die Integereinheit (IU) des Prozessors, was seine Einsetzbarkeit stark einschränkt. Im Rahmen dieser Arbeit wurde der *TrapGen*-LEON-ISS um ein der Hardware entsprechendes konfigurierbares Cache-Subsystem und TLM-Schnittstellen erweitert. Dadurch werden der Einsatz auf Systemebene zu Explorationszwecken und die Entwicklung von hardwarenaher Software ermöglicht. Abbildung 4.1 verdeutlicht die Einbettung der *TrapGen*-IU in den *SoCRocket*-CPU-Simulator.

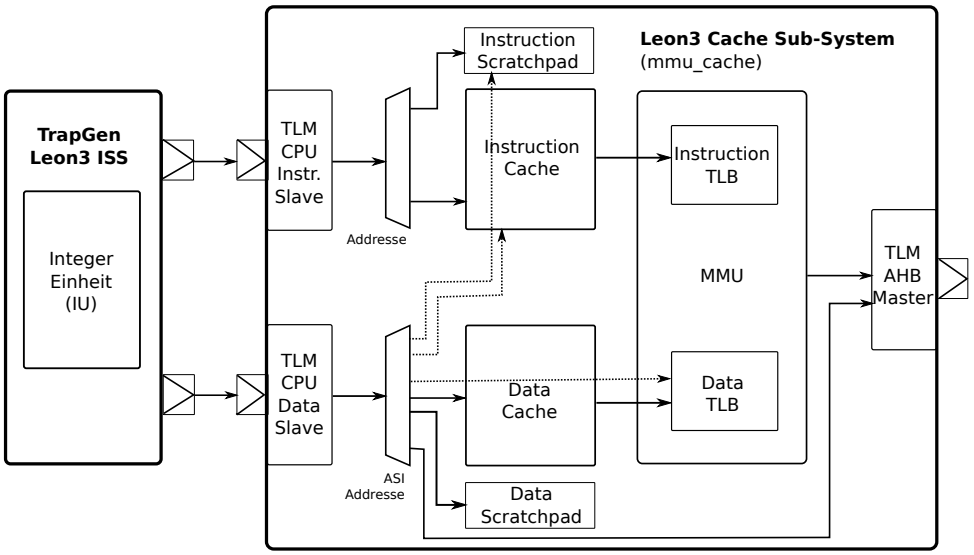


Abbildung 4.1: TL-Modell des LEON2/3

Die IU vernachlässigt Pipelineeffekte zur Steigerung der Simulationsgeschwindigkeit. Daher sind alle Komponenten des Cachesystems rein funktional und enthalten keine parallelen *Threads*. Die Zugriffsverzögerung wird abhängig vom Weg der Transaktion durchs System statisch annotiert. Durch den Einsatz einheitlicher Speicherschnittstellen ist das Cachesystem sehr flexibel. Alle Komponenten können mit Hilfe von Konstruktionsparametern zu oder abgeschaltet und beliebig verschachtelt werden. Die volle Konfiguration beinhaltet *Scratchpads*, *Caches* und *MMU*. Ein Speicherzugriff der IU würde somit abhängig vom Adressbereich zuerst an eines der *Scratchpads* oder die *Caches* gesendet. Der Cache enthält virtuelle Adressen, bedient den Zugriff oder leitet ihn an die MMU weiter. Je nach Konfiguration enthält die MMU getrennte oder geteilte *Translation Lookaside Buffers* (TLBs) für Instruktionen und Daten. Die ausgewählte TLB leitet die Transaktion an den AHB-TLM-*Master Socket* weiter. Ist die MMU nicht konfiguriert, so senden die *Caches* alle Zugriffe direkt an die Busschnittstelle. Es ist ebenfalls möglich, *Caches* zu entfernen und die Transaktion von der IU über die MMU, an den AHB-*Port* zu leiten, oder alle Komponenten des Cachesystems abzuschalten, wobei die IU dann direkt mit dem AHB-*Master* kommuniziert.

TLM-Schnittstellen

Die IU ist mit dem Cachesystem über zwei TLM2.0-*Sockets* verbunden (Abbildung 4.1). Dadurch können konzeptionell Instruktions- und Datenzugriffe parallel ausgeführt werden. Momentan wird diese Funktion nicht genutzt, da die IU zur Steigerung der Simulationsgeschwindigkeit die gesamte RISC-Pipeline in nur einem *Thread* simuliert, wodurch alle Speicherzugriffe serialisiert werden. Die Kommunikation zwischen IU und Cachesystem kann als blockierend oder nichtblockierend konfiguriert werden. Instruktions- und Datenzugriffe werden mittels der *TLM Generic Payload* realisiert. Zusätzlich werden an beiden Sockets optionale (ignorierbare) Erweiterungen zur Implementierung von Spezialfunktionenen übertragen (Tabelle 4.2).

Instruktions-Socket (icio)	
Erweiterung	Funktion
flush	Initiiert das Zurückschreiben des kompletten Instruktions-Caches in den Hauptspeicher
flushl/fline	Zurückschreiben der durch *fline* bestimmten Cache-Zeile in den Hauptspeicher
debug	Debug-Feld für direktes Testen des Instruktionspfades (siehe 4.1)
fail	Markierung für beabsichtigtes Fehlverhalten
Daten-Socket (dcio)	
Erweiterung	Funktion
asi	*Address Space Identifier* (siehe 4.1)
flush	Initiiert das Zurückschreiben des kompletten Daten-Caches in den Hauptspeicher
flushl/fline	Zurückschreiben der durch *fline* bestimmten Cache-Zeile in den Hauptspeicher
lock	Atomarer Zugriff auf den Datenbus (keine Unterbrechung durch andere Master)
debug	Debug-Feld für direktes Testen des Datenpfades (siehe 4.1)
fail	Markierung für beabsichtigtes Fehlverhalten

Tabelle 4.2: CPU Cache-Sockets - *Payload Erweiterungen*

Der LEON2/3 Prozessorsimulator erbt eine AHB-Initiatorschnittstelle von der Basisklasse *AHBMaster*, die gleichermaßen für Instruktionen und Daten genutzt wird. Darüber hinaus verfügt das Modell über *SoCRocket*-Signaleingänge (Abschnitt 3.2.5) zur Modellierung von Interrupts und DBus-*Snooping*.

ASIs & Steuerregister

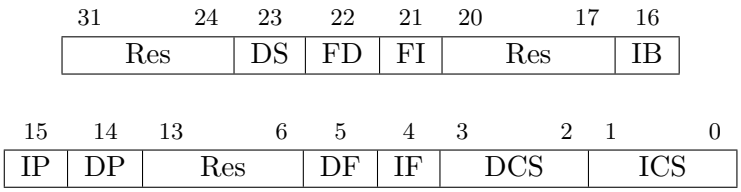
SPARC-Prozessoren nutzen einen 8-Bit großen *Address Space Identifier* (ASI) zur Aufteilung des Adressraumes in mehrere unabhängige 32-Bit-Bereiche. Der Großteil der ASIs wird dabei zur Steuerung des Cachesystems eingesetzt. Einige ASIs sind für Spezialfunktionen, wie das Erzwingen eines Cache-*Miss* oder eines Cache-*Flush* reserviert, andere ermöglichen den Zugriff auf Steuerregister. Eine Übersicht der im TL-Modell unterstützten ASIs, Steuerregister und deren Adressierung kann Tabelle 4.3 entnommen werden.

ASI	Adresse	Funktion/Register
0x00, 0x01, 0x03	alle	Cache-*Miss* erzwingen
0x02	0x00	Cache Control Register
	0x08	I-Cache Configuration Register
	0x0c	D-Cache Configuration Register
	0xff	Debug Status ausgeben
0x08-0x0b	alle	Normaler Zugriff
0x0c		Zugriff auf I-Cache Tags
0x0d		Zugriff auf I-Cache Data
0x0e		Zugriff auf D-Cache Tags
0x0f		Zugriff auf D-Cache Data
0x15	alle	I-Cache Flush
0x16	alle	D-Cache Flush
0x19	0x000	MMU Control Register
	0x100	MMU Context Pointer Register
	0x200	MMU Context Register
	0x300	MMU Fault Status Register
	0x400	MMU Fault Address Register

Tabelle 4.3: *LEON CPU* - Übersicht ASIs und Steuerregister

In *SoCRocket* werden ASIs als *Payload*-Erweiterung modelliert. Die Erweiterungen für Datenzugriffe sind in Klasse *dcio_payload_extension* implementiert. Das ASI-Feld der Erweiterung wird durch die CPU im Standardfall auf den Wert Acht (Normaler Zugriff) gesetzt. Für Zugriffe auf andere Adressbereiche verwendet der Prozessor die Befehle *load alternate* (*lda*), *store alternate* (*sta*) und *swap alternate* (*swapa*)[1]. Die aufgeführten Steuerregister wurden nicht mit *GreenReg* implementiert. Damit stellt der Prozessorsimulator in *SoCRocket* eine Ausnahme dar. Ein Grund dafür ist die Sonderstellung durch die Adressierung über ASIs. Des Weiteren können die Register nur direkt von Prozessorkern beschrieben werden und sind somit nicht über den AHB-*Socket* erreichbar. Die Steuerregister des CPU-Simulators entsprechen in ihrer Konfiguration weitgehendst dem Vorbild der VHDL-Implementierung. Abstraktionsstufen sind nur in Hinblick auf die Busschnittstellen von belang. Alle implementierten Bitfelder/Funktionen werden sowohl im LT- als auch im AT-Modus unterstützt. Tabelle 4.4 zeigt den Aufbau des *Cache Control Registers* (*CCR*). Das *CCR* dient der Steuerung der wichtigsten Funktionen des Cachesystems und wurde als private globale Variable implementiert. Zugriffe werden durch einen Adressdekodierer gesteuert, der direkt an den CPU-Daten-*Socket* angebunden ist.

1 Einschließlich verwandte Befehle für unterschiedliche Datenweiten, z.B. *load byte alternate* (*ldba*), *store half alternate* (*stha*)

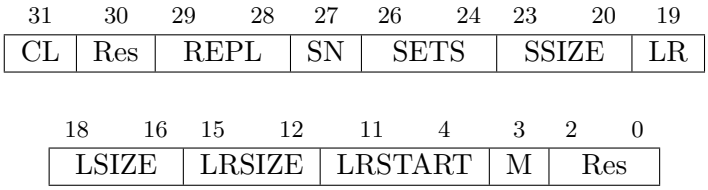


Bit-Feld	Beschreibung	Unterstützt
DS	Snooping im Datencache einschalten	LT, AT
FD	Flush Datencache	LT, AT
FI	Flush Instruktionscache	LT, AT
IB	Burstmodus für den Instruktionscache	LT, AT
IP	Instruktionscache Flush anhängig (pending)	nein
DP	Datencache einfrieren bei Interrupt	nein
IF	Instruktionscache einfrieren bei Interrupt	nein
DCS/ICS	Zustand des Daten-/Instruktionscaches:	LT, AT
	X0: abgeschaltet, 01: eingefroren, 11: eingeschaltet	

Tabelle 4.4: *LEON CPU* – Cache Control Register

Die Signal-Bits IP und DP zur Anzeige eines anhängigen Cache-*Flushes* werden im TL-Modell nicht unterstützt. Das Zurückschreiben des Cacheinhaltes in den Hauptspeicher (Cache-Flush) ist in aller Regel ein Ausnahmevorgang, der keinen oder nur sehr geringen Einfluss auf die Ausführungsgeschwindigkeit einer Anwendung haben kann. Zur Vereinfachung des Modells wird daher an dieser Stelle auf die Modellierung von Parallelität verzichtet. Das heißt, anders als auf RT-Ebene blockieren Cache-*Flushes* nachfolgende Instruktions- und Datenzugriffe der CPU. Eine weitere auf TL-Ebene abstrahierte Modelleigenschaft ist das Einfrieren des Instruktionscaches im Falle eines Interrupts (IF). Diese Funktion kann die Verdrängung von Programmdaten aus dem Cache verhindern und dadurch, unter Umständen, die Verarbeitungsgeschwindigkeit steigern. Alle bisher betrachteten Anwendungsfälle erforderten keine derartige Optimierung.

Neben dem CCR verfügt das Cachesystem, je nach Konfiguration, über ein Instruktions- und ein Datenkonfigurationsregister. Der Aufbau dieser Register ist für Instruktionen und Daten gleich (Tabelle 4.5). Die Register beinhalten Einstellungen, die zum Beispiel Größe (SSIZE), Assoziativität (SETS) und Aufbau (LSIZE) des jeweiligen Caches beschreiben. Das Register wird bei der Initialisierung des Cachesystems (*mmu_cache*) aus Konstruktorparametern initialisiert und kann zur Laufzeit nicht überschrieben werden. Informationen zu Aufbau und Funktionsweise aller weiteren Steuerregister des LEON-Modells können dem Benutzerhandbuch von *SoCRocket* entnommen werden [Sch12b].



Bit-Feld	Beschreibung	Unterstützt
CL	Gesetzt, wenn Cache-Locking möglich ist	LT, AT
REPL	Cache-Replacement Strategie 00: keine (direct mapped), 01: least recently used (LRU) 10: least recently replaced (LRR), 11: pseudo-random	LT, AT
SN	Bit ist gesetzt, wenn Daten-Cache Snooping möglichist	LT, AT
SETS	Cache-Organisation (Mapping) 000: direct mapped, 001: 2-Wege-Cache 010: 3-Wege-Cache, 011: 4-Wege-Cache	LT, AT
SSIZE	Größe pro Cache-Set in kB (2^{SSIZE})	LT, AT
LR	I/D Scratchpad RAM vorhanden	LT, AT
LSIZE	Worte je Cache-Zeile (2^{LSIZE})	LT, AT
LRSIZE	Größe des I/D Scratchpad RAMs in kB (2^{LRSIZE})	LT, AT
LRSTART	8 MSB Bits der I/D-Scratchpad-Startadresse	LT, AT
M	MMU vorhanden	LT, AT

Tabelle 4.5: *LEON CPU* – I/D Cache Configuration Register

Cache-Implementierung

Die *Caches* des LEON2/3-Simulators besitzen eine Mehrwege-Harvardarchitektur mit *Write Through*. Für *Write-Misses* werden Cacheeinträge invalidiert aber nicht neu allokiert. Der modellierte Cache hat damit eine Standardarchitektur, die unter anderem in [Han98] ausführlich beschrieben wird. Instruktions- und Datencaches werden getrennt von einander konfiguriert, sind aber sehr ähnlich aufgebaut, so dass sie in ihrer Kernfunktionalität als gemeinsame Klasse implementiert werden können. Abgewandelte Eigenschaften bezüglich des Instruktions- und Datenzugriffes werden durch abgeleitete Spezialisierungen realisiert. Abbildung 4.2 zeigt die Klassenhierarchie der Cacheimplementierung in vereinfachter Form. Zur Verbesserung der Übersicht wurde auf die Darstellung von Konstruktoren und die Mehrzahl an Attributen verzichtet. Ebenfalls nicht dargestellt sind Funktionen zur Simulationskontrolle und Modellierung des Energieverbrauchs.

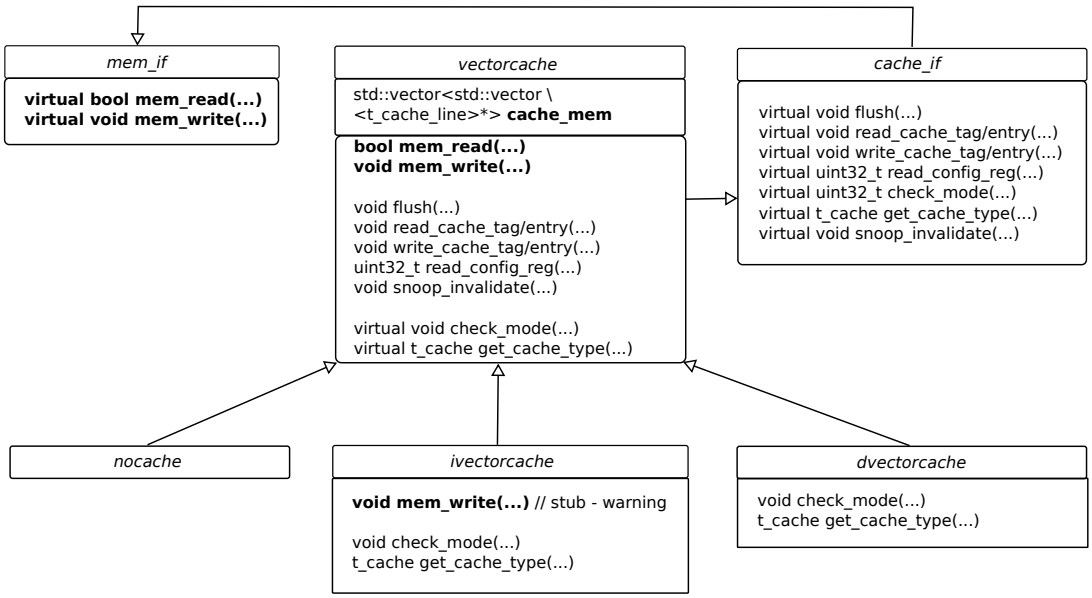


Abbildung 4.2: Klassenhierarchie Cache-Modell (UML)

Der Datencache wird durch die Klasse *dvectorcache* und der Instruktionscache durch die Klasse *ivectorcache* repräsentiert. Beide Klassen erben den größten Teil ihrer Funktionalität von *vectorcache*. Geringfügige Unterschiede bestehen in der Signatur der Konstruktoren. So kann der Instruktionscache im Falle eines *Misses* eine ganze Cachezeile in Form eines Bursts nachladen. Diese Eigenschaft wird der übergeordneten Klasse *vectorcache* fest übergeben und steht dem Nutzer extern nicht zur Verfügung. Des Weiteren implementieren *dvectorcache* und *ivectorcache* spezialisierte Versionen der Funktionen *check_mode* und *get_cache_type*. Dies wurde erforderlich, da sich beide Caches ein *Cache Control Register* (CCR, siehe Tab. 4.4) teilen, welches die jeweiligen Operationsmodi (ein/aus/gefroren) in unterschiedlichen Bit-Feldern speichert. Die Klasse *ivectorcache* überlädt darüber hinaus die Funktion *mem_write* der generischen Speicherschnittstelle (*mem_if*). An den Instruktionscache gerichtete Schreiboperationen verursachen eine Fehlermeldung und den Abbruch der Simulation.

Die Begrifflichkeit *vector* in den Klassenbezeichnungen geht auf die Art der Implementierung des Speichers im Cache zurück. Hierfür wurde der *Vector*-Container der C++ *Standard Template Library* (STL) gewählt. Ein *std::vector* ähnelt einem *Array* und kann in ähnlicher Weise, durch Indizierung mit einem *Offset*, adressiert werden. Im Vergleich mit anderen *Sequence*-Containern, wie *std::deque* und *std::list*, ist *vector* sehr effizient und schnell (ähnlich einem Array) [vec]. Darüber hinaus ist *vector* durch verschiedene vordefinierter Zugriffsfunktionen sehr komfortabel zu bedienen und mit einer dynamischer Speicherverwaltung ausgestattet, die eine einfache Verkleinerung und Vergrößerung des Zugriffsbereichs erlaubt. Die LEON2/3-Caches sind mehrdimensionale Strukturen, die aus bis zu vier assoziativen Bänken (*Sets*) bestehen. Jede Cachebank besteht aus einer Anzahl von Zeilen (*Lines*), die wiederum bis zu acht Einträge (*Entries*) und den zugehörigen *Tag* enthalten (Abbildung 4.3).

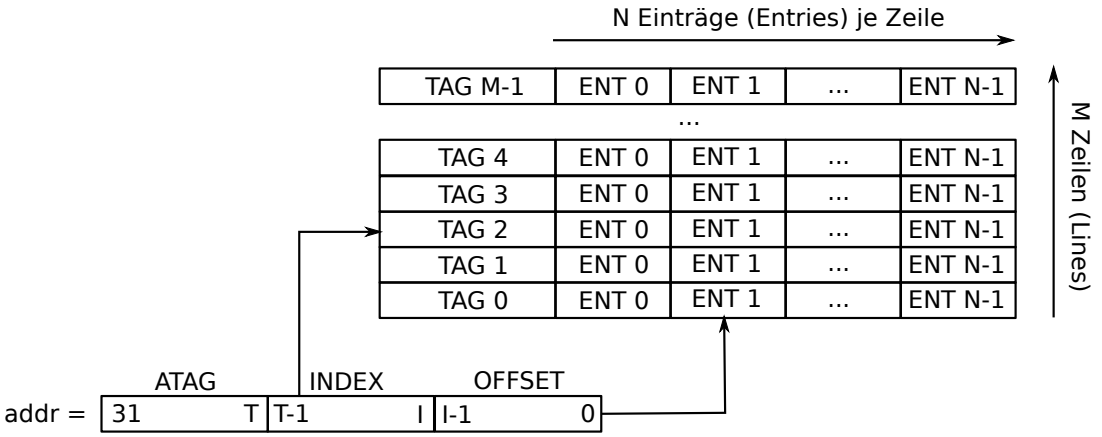


Abbildung 4.3: Aufbau einer Cache-Bank (*Set*)

Das TL-Modell implementiert diese Struktur als *vector of vectors* vom Typ *t_cache_line*:

```
std::vector<std::vector<t_cache_line>*> cache_mem;
```

Der äußere Vektor indiziert die Cachebänke, der innere Vektor die Zeile der jeweiligen Bank. Die Elemente *t_cache_line* bilden komplexe Datentypen, die den einfachen Zugriff auf *Tags* und Einträge (*Entries*) erlauben (Abb. 4.4).

```
1  typedef struct {
2    t_cache_tag tag;
3    t_cache_data entry[MAX_ENTRIES];
4  } t_cache_line;
5
6  typedef struct {
7    uint32_t atag;    // Address-Tag
8    uint32_t lrr;     // LRR-Ersetzungsindex
9    int32_t  lru;     // LRU-Ersetzungsindex
10   uint32_t lock;    // Zeilen-Lock
11   uint32_t valid;   // Gueltigkeit der Eintraege
12 } t_cache_tag;
13
14 typedef union {
15   uint32_t i;
16   unsigned char c[4];
17 } t_cache_data;
```

Abbildung 4.4: Aufbau von Cache-Zeilen im SystemC-Modell

Zur Vereinfachung des Zugriffs stellt die Klasse *vectorcache* die Funktion *lookup* bereit. Die Funktion *lookup* erwartet die Nummer der Cachebank und einen Adressindex als Eingabe und liefert die entsprechende Cachezeile (*t_cache_line*) zurück. Die Entscheidung über *Hit* oder *Miss* erfolgt dann durch den Vergleich des *Tag*-Feldes der Adresse mit dem ATAG-Feld des *Cache Tags*. Im Falle eines Treffers muss die Gültigkeit der Daten durch einen Abgleich des Zugriffsbereichs kontrolliert werden (Feld *valid* von *t_cache_tag*). Das Feld *valid* enthält ein Gültigkeitsbit für jedes Datum (32-Bit-Wort) der Cachezeile. Im Falle eines Doppelwortzugriffes müssen zwei Bits kontrolliert werden. Die Klasse *vectorcache* erleichtert dies durch die Funktion *offset2valid*, mit deren Hilfe der Zeilenoffset der Adresse (Abb. 4.3) unter Angabe der Zugriffsweite in eine Bitmaske umgerechnet werden kann. Dieser Vorgang ist für Lese- und Schreibzugriffe identisch. Sind die Daten gültig, so werden diese im Falle einer Leseoperation mit Hilfe einer *memcpy*-Anweisung in das Datenfeld der auslösenden Transaktion kopiert. Umgekehrt werden bei einem Schreibzugriff *Payload*-Daten in den Cache kopiert. Die Schreibdaten werden anschließend zur Wahrung der Konsistenz in den Hauptspeicher geschrieben (*Write Through*).

Der Kontrollfluss für Lesezugriffe auf den Datencache ist in Abbildung 4.5 dargestellt. Ein READ-*Miss* liegt vor, wenn der *Address Tag* in keiner Cachebank enthalten oder die Daten am entsprechenden *Offset* als ungültig markiert sind. Darüber hinaus kann ein READ-*Miss* durch Markierung mit $ASI = 0x0$ erzwungen werden. In beiden Fällen wird daraufhin die Länge des Zugriffes auf den Hauptspeicher berechnet. Diese kann ein oder zwei 32-Bit-Worte, im Instruktionscache aber auch die Differenz zum Ende der Cachezeile (*Instruction Burst Fetch*-Modus), also maximal acht 32-Bit Worte, betragen. Der eigentliche Speicherzugriff wird durch einen Aufruf der Speicherschnittstelle der Klasse *tlb_adapter* initiiert. Diese fungiert als *Proxy* und leitet den Aufruf abhängig von der Systemkonfiguration an die MMU oder den AHB-*Master* weiter. Die zurückgelieferten Daten werden dann in den Cache eingetragen. Dies geschieht unter Berücksichtigung der Ersetzungsstrategie (LRR/LRU/Random). Im Falle eines gefrorenen *Caches* (*Freeze*) werden die Daten nur ersetzt, wenn der *Address Tag* bereits in einer der Cachebänke gespeichert ist.

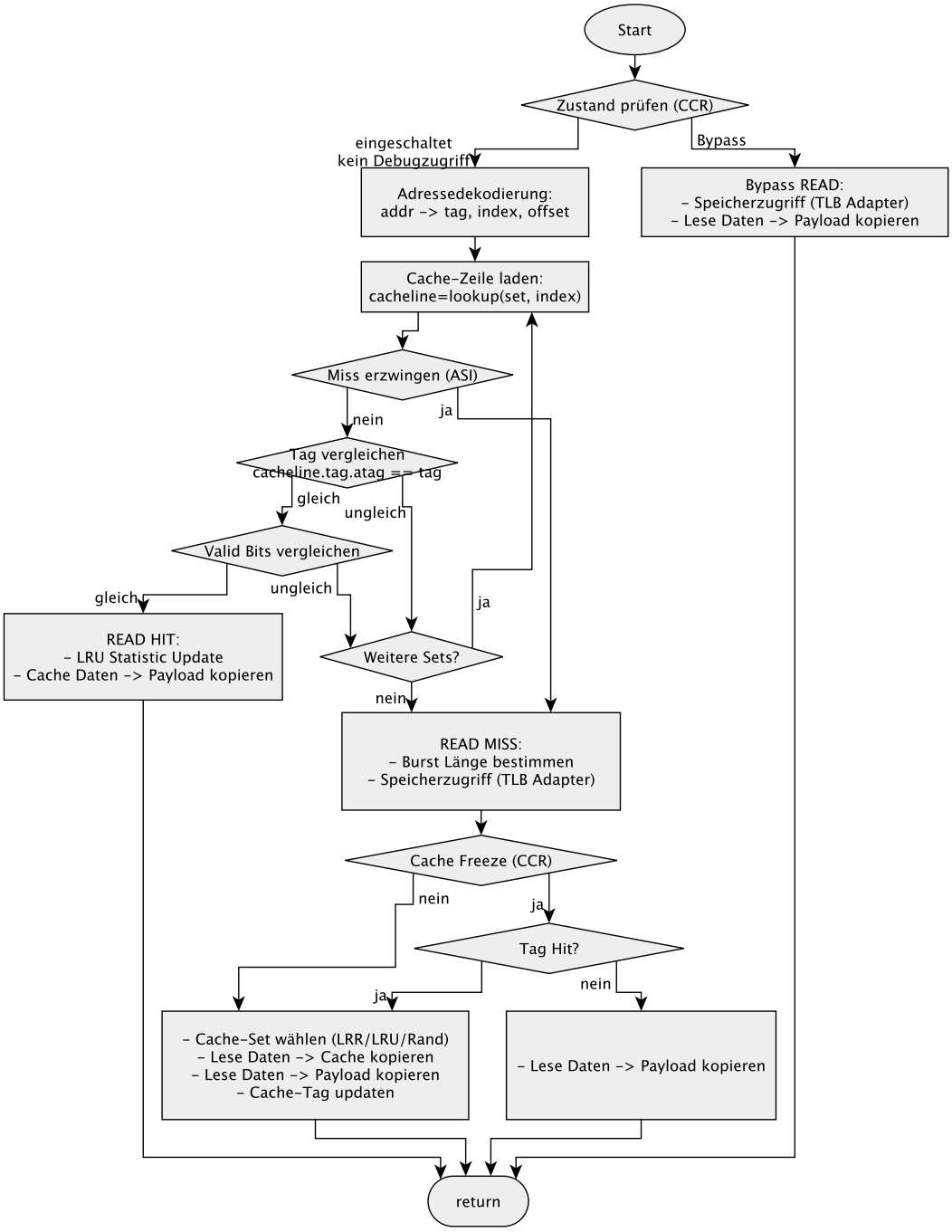


Abbildung 4.5: Daten-Cache Kontrollfluss für Leseoperationen

Die Unterscheidung zwischen *Hit* und *Miss* ist für Lese- und Schreiboperationen äquivalent (Abb. 4.6). Ein WRITE-*Hit* liegt vor, wenn der *Tag* der Transaktion in einer der Cachebänke gespeichert und die zugehörigen Daten gültig sind (*Valid*-Bits). Bei einem WRITE-*Hit* werden die *Payload*-Daten direkt in den Cache kopiert und im Anschluss zur Synchronisation mit dem Hauptspeicher an den TLB-Adapter weitergeleitet. Ein WRITE-*Miss* hat keinen Einfluss auf den Inhalt des *Caches* und wird ähnlich einem *Bypass*-Zugriff direkt an die Speicherschnittstelle der MMU oder des AHB-*Masters* gesendet (via TLB-Adapter).

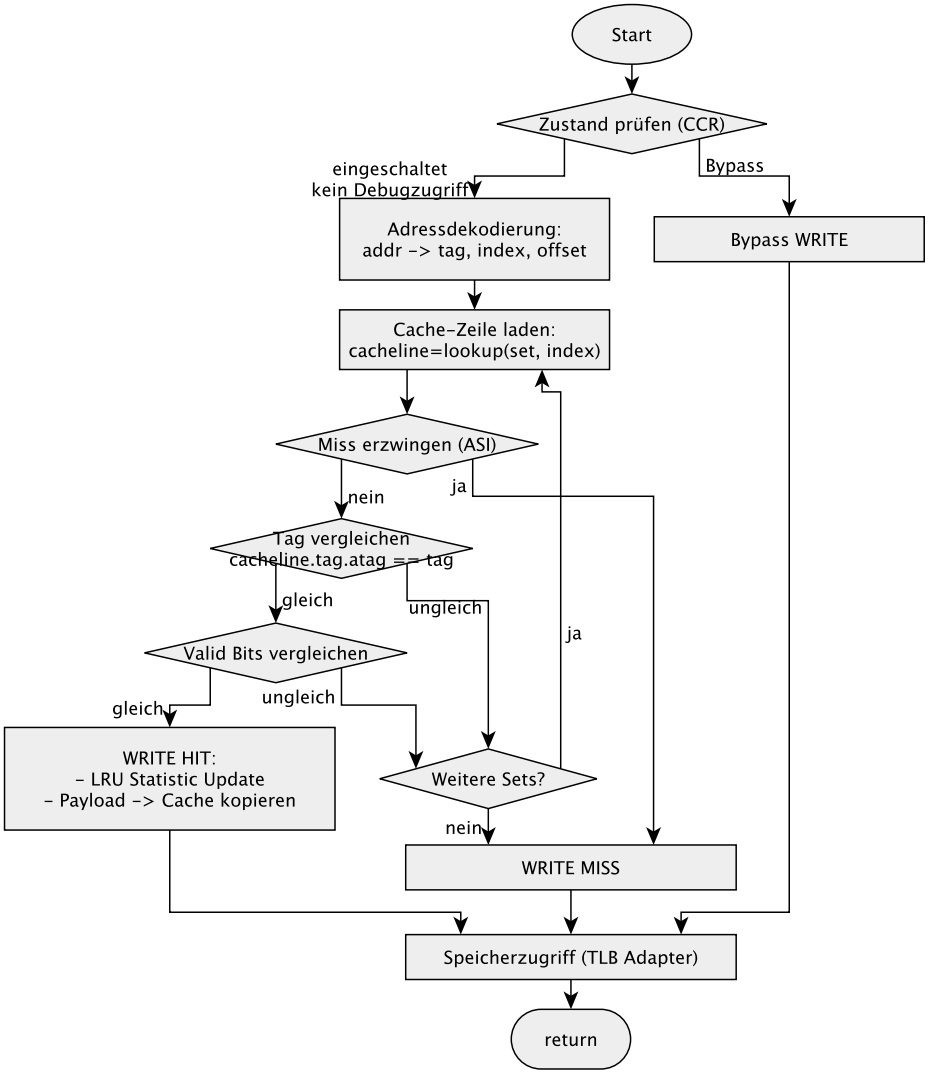


Abbildung 4.6: Daten-Cache Kontrollfluss für eine Schreiboperationen

Wie alle Komponenten des Cachesystems implementieren Instruktions- und Datencaches die Zugriffsschnittstelle *mem_if*. Dies geschieht indirekt durch die Erweiterung von *mem_if* zur Cache-Schnittstelle *cache_if*. Das System gewinnt dadurch an Modularität und kann abhängig von der Konfiguration dynamisch konstruiert werden. Die oberste Schicht des Systems definiert dazu lediglich Zeiger vom Typ *cache_if* die als Platzhalter dienen. Abbildung 4.7 verdeutlicht dieses Verfahren anhand des Cache-Datenpfades.

```
1  // Instanzierung des Cache-Datenpfades
2  dcache = (dcen == 1) ? (cache_if*)new dvectorcache("dvectorcache",
3                         (mmu_cache_if *)this,
4                         (mmu_en) ? (mem_if *)m_mmu->get_dtlb_if()
5                                  : (mem_if *)this,
6                          mmu_en, dsets, dsetsize, dsetlock,
7                          dlinesize, drepl, dlram, dlramstart,
8                          dlramsize, m_pow_mon)
9                        : (cache_if*)new nocache("no_dcache",
10                        (mmu_en) ? (mem_if *)m_mmu->get_dtlb_if()
11                                 : (mem_if *)this);
12
13 ...
14
15 // Beispiel: Lese-/Schreibaufruf des CPU Datensockels (mem_if)
16 dcache->mem_read(addr,asi,data_ptr,len,&delay,debug,is_dbg);
17 dcache->mem_write(addr,asi,data_ptr,len,&delay,debug,is_dbg,lock);
```

Abbildung 4.7: Cache Subsystem - Dynamische Instanzierung des Datenpfades

Der Konstruktorparameter *dcen* entscheidet, ob das System einen Datencache besitzt oder nicht. Entsprechend wird dem Zeiger *dcache* eine Instanz von *dvectorcache* (Zeile 2) oder eine Instanz des Platzhalters *nocache* (Zeile 9) zugewiesen. Die Klasse *dvectorcache* erhält einen Zeiger auf die Speicherschnittstelle der Klasse *mmu_cache*, welcher der Ansteuerung des AHB-*Masters* dient (Zeile 3) und abhängig vom Konstruktorparameter *mmu_en* einen Zeiger auf die Speicherschnittstelle der Daten-TLB der MMU (Zeile 4). Zugriffe im *Bypass*-Modus (ASIs 0, 1, 3) und bei softwareseitig abgeschaltetem Datencache (siehe *CCR* Tab. 4.4) werden durch *dvectorcache* an den AHB-*Master* weitergeleitet. Cache-*Misses* bei aktiviertem Cache werden an eine Ausgabeschnittstelle gesendet, die entweder auf die Daten-TLB oder direkt zum AHB-*Master* zeigt. Für den Cache selbst ist die Weiterverarbeitung der Transaktion transparent. Ähnliches gilt für den Platzhalter *nocache*. Die Klasse repräsentiert einen nicht vorhandenen Cache und wurde vordringlich aus Sicherheitsgründen eingefügt. Sie enthält *Stub*-Implementierungen aller Cache-Zugriffsfunktionen, die Warnungen für nicht erlaubte Operationen ausgeben (z.B. Beschreiben der Tags eines nicht vorhandenen Caches). Außerdem sind Konfigurationen denkbar, die keinen Datencache aber trotzdem eine MMU besitzen. In diesem Fall muss *nocache* Zugriffe abhängig von den Einstellungen der MMU-Register (siehe [Sch12b]) an den AHB-*Master* oder die Daten-TLB weiterleiten. Vorteil des erläuterten Ansatzes ist die einheitlich Signatur der Cache-Zugriffsfunktionen (Zeilen 16, 17). Aus Sicht des CPU-Datensockels sind diese konfigurationsunabhängig und immer gleich. Nur die interne Verschachtelung der Speicherschnittstellen und der Inhalt der Cache- und MMU-Konfigurationsregister entscheiden über den Weg der Transaktion durch das System.

Memory Management Unit (MMU)

Neben der Integereinheit (IU) und dem Cache stellt die *Memory Management Unit* (MMU) eine der zentralen und komplexesten Komponenten des LEON-CPU-Modelles dar. Aufgabe sind die Übersetzung virtueller in physikalische Adressen und der Schutz von Speicherbereichen in Softwareumgebungen mit mehreren unabhängigen Prozessen. Die Architektur der LEON-MMU ist an die im SparcV8-Architekturhandbuch [Inc92] beschriebene Referenzimplementierung angelehnt. Kernbestandteile sind *Translation Lookaside Buffers* (TLB), die im RTL-Modell als *Context Adressable Memory* (CAM) implementiert werden und eine Bank mit Steuerregistern, die über ASI 0x19 gelesen und beschrieben werden kann (Tabelle 4.3). Das TL-Modell implementiert TLBs mit Hilfe des Containers *std::map* der C++ *Standard Template Library* (STL). Eine *std::map* ist vollassoziativ und speichert Elemente als Schlüssel/Wert-Paare. Zur Identifikation wird ein binärer Suchbaum aufgespannt. Im Vergleich mit der RTL-Implementierung hat dies eine

erhebliche Komplexitätsreduktion zur Folge. Die Anzahl der vorhandenen TLBs, der TLB-Typ, die Ersetzungsstrategie und die Größe der abzubildenden Speicherseiten sind frei konfigurierbar. Abhängig vom TLB-Typ werden getrennte oder geteilte TLBs für Instruktionen und Daten instantiiert. Um dies zu realisieren, definiert die Klasse *mmu* zwei Zeiger vom Typ *std::map*:

```
std::map<t_VAT, t_PTE_context> * itlb, dtlb;
```

Abhängig vom TLB-Typ werden für diese Zeiger im Konstruktor der Klasse *mmu* eine einzelne oder zwei unterschiedliche Instanzen dynamisch erzeugt. Schlüssel zur Adressierung einer TLB ist ein *Virtual Address Tag* (*VAT*), der je nach Konfigurationsgröße der Speicherseiten 20 bis 17 MSB-Bits (4kB - 32kB Seiten) der virtuellen Adresse einnimmt. TLB-Einträge werden mit Hilfe des komplexen Datentyps *t_PTE_context* modelliert:

```
1  // Virtueller Adresstag
2  typedef unsigned int t_VAT;
3
4  // Page Descriptor Cache (TLB) - Eintrag
5  typedef struct {
6    unsigned int tlb_no;     // Nummer der modellierten TLB
7    unsigned int context;    // Prozesskontext
8    unsigned int pte;        // TLB-Eintrag
9    unsigned int lru;        // Ersetzungszaehler
10 } t_PTE_context;
```

Abbildung 4.8: Aufbau von TLB-Einträgen im SystemC-Modell

Wichtigstes Element von *t_PTE_context* ist der TLB-Eintrag *pte*. Im Falle eines Treffers (TLB-*Hit*) enthält dieser die physikalische Adresse der angesprochenen Speicherseite. Kann für eine virtuelle Adresse kein TLB-Eintrag gefunden werden (TLB-*Miss*), muss zur Übersetzung auf den Hauptspeicher zugegriffen werden. Der Hauptspeicher hält für jeden Prozess eine dreistufige Seitentabelle vor (Abbildung 4.9). Die Größe der einzelnen Tabellen und damit verbunden die Weite der Indexfelder im VAT sind konfigurierbar. Der Zeiger zur Wurzel der ersten Tabelle befindet sich in einer Kontexttabelle, die mit Hilfe des *Context Table Pointer Registers* adressiert und mit dem *Context Register* indiziert werden kann (siehe Tab. 4.3). Zur Adressierung der Seitentabellen wird der VAT in drei Indizes zerlegt. Die ersten zwei Tabellen enthalten Zeiger auf die Wurzel der Seitentabelle der jeweils nächsten Stufen. Die dritte Tabelle enthält die Adresse zur physikalischen Speicherseite, die dann entsprechend der Ersetzungsstrategie in einer der TLBs gepuffert werden kann.

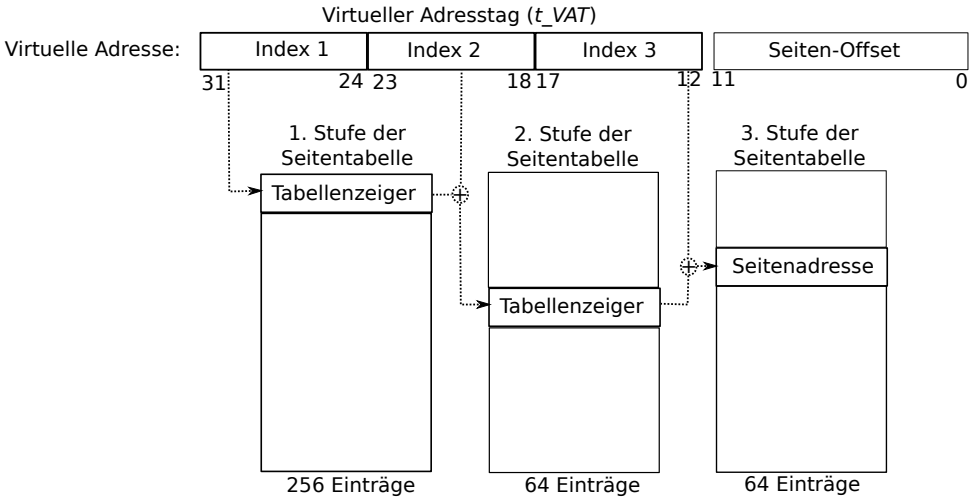


Abbildung 4.9: MMU - Aufbau virtuelle Adresse und Seitentabelle

Das TL-Modell der LEON-MMU implementiert die dafür nötige Funktionalität in der Funktion *tlb_lookup*. Die Funktion zerlegt zunächst die vom Cache übergebene virtuelle Adresse in einen *Address Tag* und einen Seiten-*Offset*. Danach wird die TLB (*std_map*) mit dem VAT indiziert. Im Falle eines Treffers baut *tlb_lookup* die physikalische Adresse auf und liefert diese zur Weiterleitung an den AHB-*Master* zurück. Wenn kein TLB-Eintrag gefunden werden kann, durchläuft die Funktion die beschriebene Seitentabelle (*Page Table Walk*). Dazu sind mehrere Speicherzugriffe erforderlich. Der Kontrollfluss für die Funktion *tlb_lookup* ist in Abbildung 4.10 dargestellt.

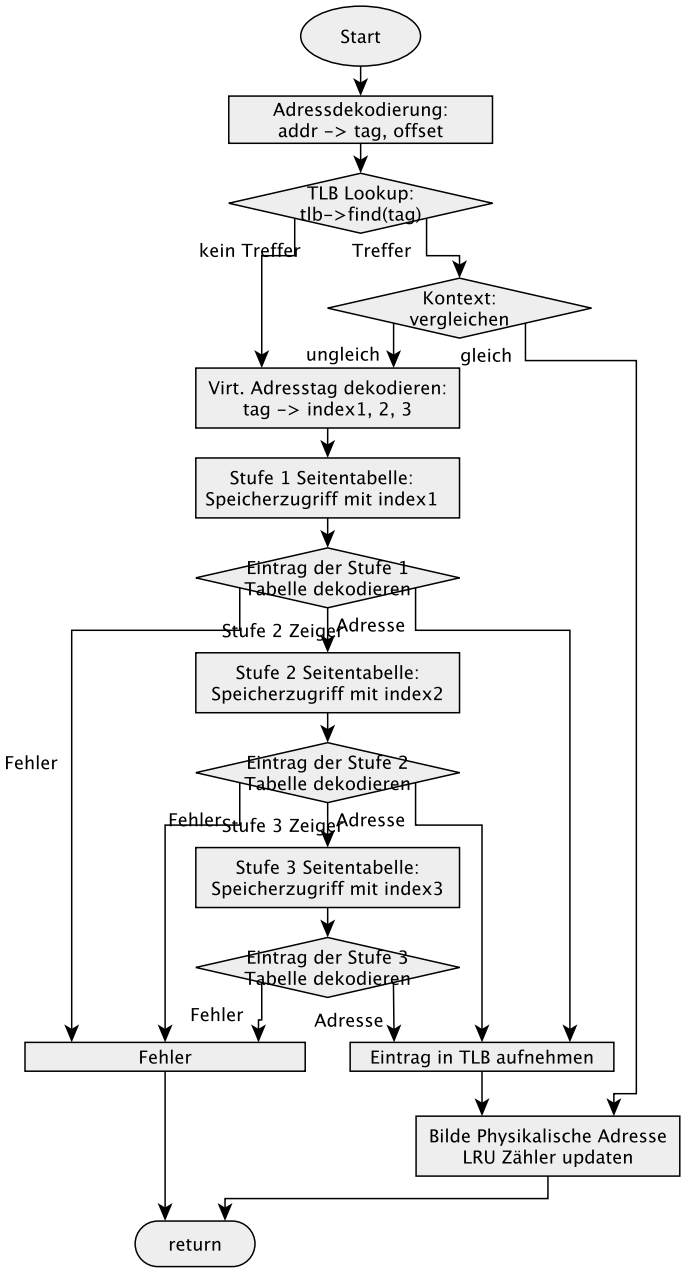


Abbildung 4.10: MMU - Kontrollfluss für TLB-*lookup* und *Page-Table-Walk*

Der Definition in [Inc92] folgend, sind flachere Seitentabellen mit nur einer oder zwei Stufen möglich. Daher wird die Struktur des Eintrags nach jedem Speicherzugriff geprüft. Sobald eine Adresse gefunden wird, erfolgt ein vorzeitiger Abbruch. In diesem Fall wird der entsprechende Eintrag unter Berücksichtigung der gewählten Ersetzungsstrategie in die TLB aufgenommen. Die physikalische Adresse des Zugriffs bildet den Rückgabewert der Funktion.

ScratchPad RAMs

Entsprechend dem RT-Modell stehen dem LEON-ISS eng gekoppelte Speicher, sogenannte *Scratchpad*-RAMs, zum schnellen Zugriff auf Instruktionen und Daten zur Verfügung. Die Funktionsweise von Daten- und Instruktions-*Scratchpad* ist sehr ähnlich. Beide Speicher können gelesen und geschrieben werden, wobei auf das Instruktions-*Scratchpad* im Regelfall nur 32-Bit-Operationen (*Instruction Fetch*) angewendet werden. Das Beschreiben der *Scratchpads* erfolgt über den Datensockel der CPU (Abb. 4.1). Daher können beide *Scratchpads* als gemeinsame Klasse implementiert werden. Das TL-Modell verfügt über Tests die verhindern, dass sich die für Instruktions- und Daten-*Scratchpad* reservierten Speicherbereiche überlappen. Außerdem ist die Instantiierung von *Scratchpads* nur möglich, wenn sich keine MMU im System befindet.

Die *Scratchpad*-Klasse (*localram*) implementiert die bereits vorgestellte generische Speicherschnittstelle *mem_if* (Abb. 4.2). Die Funktionalität ist direkt in die Zugriffsfunktionen *mem_read* und *mem_write* eingebettet. Der Simulationsspeicher wurde in Form eines C-Feldes vom Typ *t_cache_data* organisiert. Die Verwendung eines Feldes erscheint hier die optimale Lösung, da der abzudeckende Speicherbereich eher klein ist (maximal einige MB) und im Regelfall eine hohe Nutzungsdichte aufweist. Der Datentyp *t_cache_data* ist eine *Union*-Struktur, die zum Aufbau von Cachezeilen wiederverwendet wird (Abb. 4.4).

Die Verzögerungsberechnung der *Scratchpad*-RAMs ist statisch. Das Modell greift zur Bestimmung der Taktrate auf die Basisklasse *CLKDevice* zu. Die *Streaming*-Weite beträgt 32 Bit. Für jeden Transfer wird eine Verzögerung von einem Takt annotiert.

Diagnosezugriff und Debugging

Für Debugging und Diagnose stellt das Hardwaremodell Spezialfunktionen zum Zugriff auf Cache-*Tags* und Cache-Daten bereit. Alle Einträge in den *Caches* können mit Hilfe der ASIs *0xc* bis *0xf* gelesen und geschrieben werden (Tabelle 4.3). Da sich dies sehr gut für die Erzeugung von Testfällen verwenden lässt, wurde diese Funktionalität ins TL-Modell übernommen. Eine vollständige Beschreibung der dafür erforderlichen Kodierung der Adressen- und Datenfelder befindet sich in [Gai10]. Darüber hinaus wurde das TL-Modell mit einem Mechanismus zur zugriffsbasierten Zustandsüberprüfung ausgestattet. Dieser kommt in fast allen gerichteten Tests zum Einsatz und ermöglicht die Verfolgung des durch eine bestimmte Transaktion verwendeten Zugriffspfades. Dadurch kann die Testumgebung, neben der reinen Korrektheit der Ergebnisse, feststellen auf welche Weise diese gewonnen wurden (z.B. Cache-*Hit*, TLB-*Miss*). Die Testumgebung übergibt dazu einen Zeiger auf ein Integerdatum (*debug* – siehe Tabelle 4.2), dass als optionale Erweiterung an die entsprechende Transaktion angekoppelt wird. Auf dem Weg durch das Cachesystem erhält der Integer eine Bitmaske, welche die Art des Zugriffs eindeutig beschreibt (Tabelle 4.6). Das System stellt Makros zum Auslesen und Testen aller Bitfelder bereit.

31 22	21	20 16	15	14	13	12	11 5	4	3 2	1 0
Res	MMU	TLB	Res	FRM	BP	SP	Res	FL	CS	SET

Bit-Feld	Beschreibung	Funktion
MMU	MMU Status	0: TLB Hit, 1: TLB Miss
TLB	TLB Nummer	Nummer der TLB die den Hit oder Miss erzeugt hat
FRM	Frozen Read Miss	Bit wird gesetzt, wenn das Ergebniss eines *Read Miss* auf Grund eines *Cache Freeze* nicht in den Cache aufgenommen wird
BP	Bypass	Zugriff im Bypass-Modus (Cache abgeschaltet oder nicht konfiguriert)
SP	Scratchpad	Zugriff umgeleitet auf Scratchpad-RAM
FL	Flush	Zugriff hat einen Cache-Flush ausgelöst
CS	Cache State	00: Read Hit, 01: Read Miss, 10: Write Hit, 11: Write Miss
SET	Cache Set	Nummer betreffenden Cache-Sets (Hit/Miss-Verursacher)

Tabelle 4.6: *LEON CPU* - Debug-Erweiterung für Cache Analyse

Eine weitere Sonderfunktion des TL-Modelles besteht in der *Debug*-Ausgabe kompletter Cachezeilen. Eine solche Ausgabe kann per Software durch einen Zugriff mit ASI 0x2 und Adresse 0xff generiert werden. Die Nummer der auszugebenden Cachezeile wird im Datenfeld als 32-Bit-Wort übergeben. Die Ausgabe ist konfigurationsabhängig, umfasst alle Cachebänke und eine variable Anzahl an Einträgen.

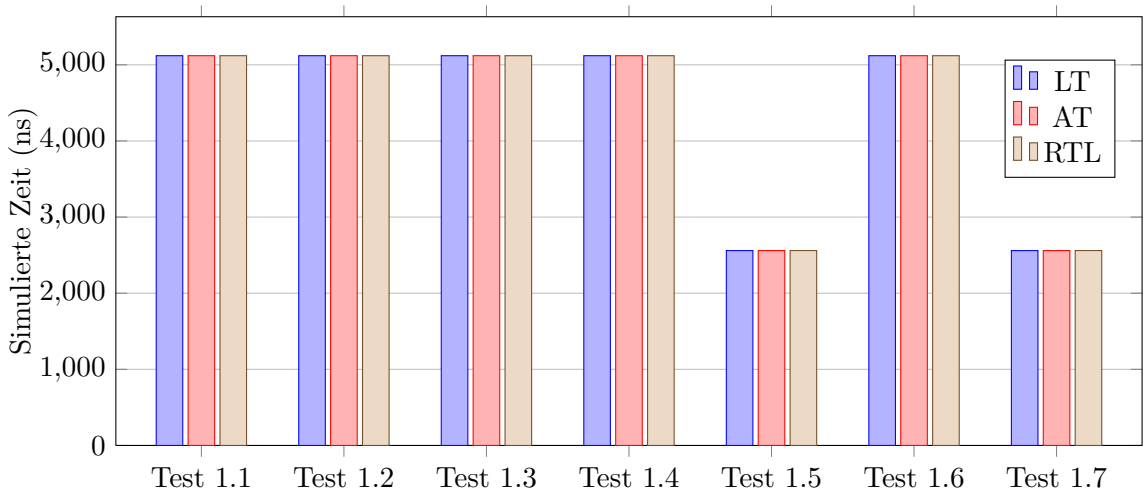
Verifikation und Leistungsfähigkeit

Zur Untersuchung von Simulationsgenauigkeit und Leistungsfähigkeit wurde das neu entwickelte Cachesystem des LEON2/3-Prozessorsimulators unabhängig von der bereits vorhandenen Integereinheit (IU) verifiziert. Verschiedene Tests des Gesamtsystems können Abschnitt 6 entnommen werden. Die Tests des Cachesystems werden durch einen IU-Platzhalter gesteuert. Alle Tests werden von einer gemeinsamen Basisklasse abgeleitet (*mmu_cache_test*). Die Klasse stellt *Sockets* zum Zugriff auf die Instruktions- und Datenschnittstellen des *Caches* und verschiedene Komfortfunktionen bereit. So unterhält *mmu_cache_test* einen Pool von *Payload*- und Referenzdatenzeigern, die durch den jeweiligen Test mit Hilfe von API-Funktionen abgefragt werden können. Die Testklasse übernimmt außerdem den automatischen Abgleich von Simulations- und Referenzdaten, und generiert Statistiken über Fehler und Zeitverlauf (Abschnitt 3.8). Die so generierten Tests können zur Simulation auf unterschiedlichen Abstraktionsniveaus wiederverwendet werden. Zur Co-Simulation der VHDL-Referenz müssen hierzu lediglich auf Seiten der Testklasse und des AHB-Busses Transaktoren eingefügt werden, welche die TL-Kommunikation der Testumgebung in zyklengenaue Signale umsetzen.

Die Abbildungen 4.11 und 4.12 zeigen die Ergebnisse der Co-Simulation. Die erzielte Simulationsgenauigkeit ist sehr hoch und erreicht in fast allen Fällen 100%. Da das Speicheruntersystem ein rein funktionales Modell ist, das sich im Sinne der Abstraktion fast ausschließlich an der AHB-Busschnittstelle unterscheidet, sind auch die Ergebnisse von LT- und AT-Modell gleich. Höhere Abweichungen zeigen sich teilweise bei der Verifikation von Spezialfunktionen, wie der Zeilenersetzungstrategie (Tabelle 4.7 – Test 5). Diese sind durch die unterschiedliche Behandlung von *Burst Transfers* bedingt.

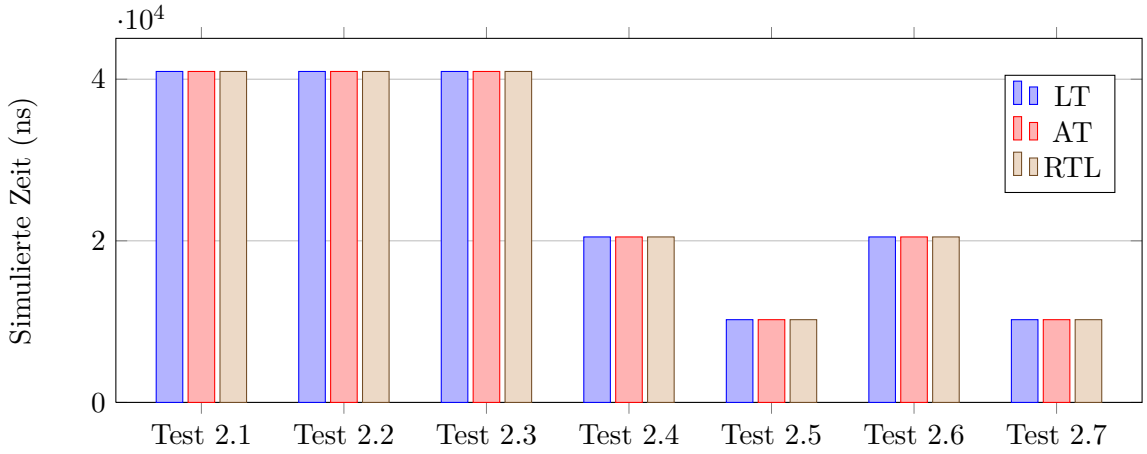
Das hier vorgestellte Verifikations-*Setup* dient vordringlich der Überprüfung der Simulationsgenauigkeit. Eine genauere Beschreibung der einzelnen Testsequenzen kann [Sch12d] entnommen werden. In Hinsicht auf die Simulationsgeschwindigkeit der Modelle lassen sich auf Grund des durch die Testumgebung bedingten *Overheads* keine verlässlichen Aussagen treffen. Die hier betrachteten Tests simulierten auf LT-Ebene im Durchschnitt 40 mal schneller als das

RTL-Referenzmodell.



Test 1	Beschreibung
1.1	Daten schreiben im Bypass-Modus (512 Worte)
1.2	Instruktionen lesen im Bypass-Modus (512 Worte)
1.3	Daten lesen im Bypass-Modus (512 Worte)
1.4	Instruktionen lesen, I-Cache-Miss (512 Worte)
1.5	Instruktionen lesen, I-Cache-Hit (512 Worte)
1.6	Daten lesen, D-Cache-Miss (512 Worte)
1.7	Daten lesen, D-Cache-Hit (512 Worte)

Abbildung 4.11: Cache-Subsystem / Test mit Minimalkonfiguration



Test 2	Beschreibung
2.1	Daten schreiben im Bypass-Modus für Tags 0x0 - 0x7 (2048 Worte)
2.2	Instruktionen lesen im Bypass-Modus von Tags 0x0 - 0x7 (2048 Worte)
2.3	Daten lesen im Bypass-Modus von Tags 0x0 -0x7 (2048 Worte)
2.4	Instruktionen lesen, I-Cache-Miss von Tags 0x0 - 0x7 (2048 Worte)
2.5	Instruktionen lesen, I-Cache-Hit von Tags 0x0 - 0x7 (2048 Worte)
2.6	Daten lesen, D-Cache-Miss von Tags 0x0 - 0x7 (2048 Worte)
2.7	Daten lesen, D-Cache-Hit von Tags 0x0 - 0x7 (2048 Worte)

Abbildung 4.12: Cache-Subsystem / Test mit Maximalkonfiguration

Test	Beschreibung	LT (ns)	AT (ns)
3	Test der Cache Grundfunktionalität: - Steuerregisterzugriff - Gezielte Hit/Miss Kombinationen über alle Sets von I/D-Cache - Diagnostischer Zugriff (ASIs 0xC - 0xF) - I/D Cache-Flush (ASIx 0x10, 0x11) und via CCR - Zugriffsweiten Byte, Half, Word und DWord - Cache Freeze	2540	2250
4	Lesen und Schreiben von Instruktions- und Daten-Scratchpads	29820	25930
5[1], 10	Test der Cache-Zeilenersetzungsstrategie (LRR, LRU) und des Cache-Lockings	16840[1]	9470[1]
6[2], 8, 9	Test der MMU mit unterschiedlich großen Speicherseiten (4[2], 8, 16kB)	410460[2]	492830[2]
7	Doppelwortzugriff und I-Cache Line-Fetch	98080	76950

Tabelle 4.7: Cache-Subsystem / Übersicht der TLM-Tests

4.2 Verbindungskomponenten

Verbindungskomponenten sind nicht-terminale Komponenten zur gesteuerten oder ungesteuerten Weiterleitung von Transaktionen, in der Regel also Busse und *Router*. Die wichtigsten Verbindungskomponenten der *SoCRocket*-Modellbibliothek dienen der Modellierung des AMBA2-Bussystems. Dazu werden ein AHB-Bus (AHBCTRL) und eine AHB/APB-Busbrücke (APBCTRL) bereitgestellt. Die Implementierung dieser Komponenten weist einige Besonderheiten auf. Zum einen besitzen beide Komponenten Multi-*Sockets*, welche die Verbindung beliebig vieler *Master*- und *Slave*-Komponenten erlauben. Der Hardware entsprechend, ist die Zahl der Bindungen auf 16 *Master* und 16 *Slaves* beschränkt. Zum anderen verfügt der AHBCTRL über getrennte Verhaltensteile für die LT- und die AT-Abstraktion. Dies steht im Gegensatz zu den meisten anderen Komponenten des Systems, die sich im Sinne der Abstraktion nur in der Busschnittstelle unterscheiden. Grund für die Trennung des Verhaltens ist die zentrale Rolle des AHBCTRL. Zur Erreichung einer hohen Genauigkeit im AT-Modus müssen Arbitrierungsmechanismus, Prioritäten und Pipelineeffekte berücksichtigt werden, Faktoren, die für eine schnelle Systemexploration im LT-Modus ignoriert werden können. Sowie AHBCTRL als auch APBCTRL nutzen die *Plug & Play*-Register der angebundenen Komponenten zum automatischen Aufbau von *Routing*-Tabellen. Die Zuordnung von Speicherbereichen erfolgt mit Hilfe von Adresse/Maske-Paaren. Im Gegensatz zur Hardwareimplementierung der GRLIB sind Interrupts nicht Teil der Busverbindung. Daher besitzen weder AHBCTRL noch APBCTRL Interrupteingänge oder -ausgänge. Interrupts werden durch TL-Signalverbindungen realisiert (siehe Abschnitt 3.2.5), wodurch sich auf Plattformebene eine höhere Flexibilität ergibt. Die Modellierung von Interrupts als *Payload*-Erweiterung würde die Zustandsautomaten in den AHB-*Sockets* unnötige verkomplizieren. Darüber hinaus stellen TL-Signalverbindungen direkte Funktionsaufrufe dar und können darum potentiell schneller simuliert werden. Das Bussystem verfügt über einen *Snooping*-Mechanismus. Dieser wurde ebenfalls mit Hilfe von TL-Signalverbindungen realisiert. Aufgabe des *Snoopings* ist die Herstellung von Cachekohärenz. Wenn ein beliebiger *Master* Daten in einen als *cacheable* gekennzeichneten Speicherbereich schreibt, müssen alle anderen *Master* des Systems über diesen Vorgang informiert werden und gegebenenfalls ungültig gewordene Einträge aus ihren Caches entfernen. Ist *Snooping* aktiviert, propagiert der Bus die *Master ID*, die Adresse und die Länge jedes Schreibvorganges. Die Kontrolle und Invalidierung der Caches erfolgt im Speichersystem des LEON-Simulators (siehe 4.1).

4.2.1 AHB-Controller (AHBCTRL)

Übersicht

Der Quellcode des *AHBCTRL*-TLM befindet sich im Unterverzeichnis */models/ahbctrl* der Plattform (Anhang B). Die Funktion des Modelles besteht in der Arbitrierung und Weiterleitung von Transaktionen zwischen den angeschlossenen Bus-*Master*- und *Slave*-Komponenten. Dazu wird eine interne Adressierungstabelle verwendet, die das TLM zu Beginn der Simulation aus Konfigurationsregistern (*Plug & Play*) aufbaut. Das Modell unterstützt *Round-Robin* und prioritätsbasierte Arbitrierung, Datenbus-*Snooping* zur Realisierung der Cachekohärenz in Multiprozessorsystemen und Bus-*Locking* für atomarer Speicheroperationen. Die Struktur des Modells ist in Abbildung 4.13 vereinfacht dargestellt.

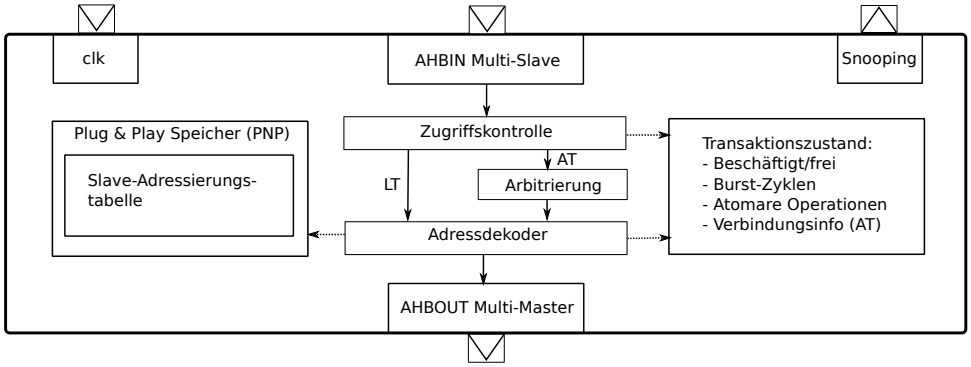


Abbildung 4.13: TL-Modell des AHB-Controllers (AHBCTRL)

TLM-Schnittstellen

Als zentrale Verbindungskomponente nimmt das AHBCTRL-TLM eine Sonderrolle ein. Die Komponente implementiert einen *Master-Socket* an den bis zu 16 *Master*-Komponenten und einen *Slave-Socket* an den bis zu 16 *Slave*-Komponenten gebunden werden können (TLM-Multi-*Sockets*). Die *Sockets* werden durch AHBCTRL direkt implementiert und nicht über Bibliotheksbasisklassen eingebunden. Der Grund dafür liegt in der Natur des Modells. Der AHB-Bus basiert auf Multiplexern und hat einen fast ausschließlich kombinatorischen Datenpfad. Wenn sich ein *Master* im Besitz des Busses befindet, so werden Transaktionen ohne Zeitverzögerung zum *Slave* weitergeleitet. Die für die Datenübertragung benötigte Zeit wird dabei einzig durch das Verhalten von *Master* und *Slave* bestimmt, welche Daten durch das Einfügen von Wartezyklen mit variabler Frequenz bereitstellen und weiterverarbeiten. Zur Darstellung dieses Verhaltens auf TL-Ebene ist eine enge Verzahnung von *Master*- und *Slave*-Schnittstelle erforderlich. Dies kann durch die in Abschnitt 3.2.2 eingeführten Programmierschnittstellen (Abbildungen 3.3 u. 3.4) nicht bewerkstelligt werden. Eine Erweiterung um eine spezifische Arbitrierungsfunktion und eine Rückruffunktion zur Signalisierung des Endes der AHB-Adressphase erscheinen auf Grund der Alleinstellung der Komponente nicht zweckmäßig. Zur Realisierung der *Timing*-Annotation erbt AHBCTRL die *SoCRocket*-Basisklasse *CLKDevice*. Die dadurch übergebene Taktrate wird zur Berechnung der Verzögerungszeit von Zugriffen auf den integrierten PNP-Speicher und zur Berechnung der Arbitrierungsverzögerung verwendet. Darüber hinaus besitzt AHBCTRL einen TL-Signalausgang zur Modellierung des Datenbus-*Snoopings*. In Multiprozessorsystemen können sich alle Bus-*Master* auf diesen Port verbinden und werden so über Schreibvorgänge in gemeinsam genutzten Speicher informiert (siehe 4.2.1).

Zugriffskontrolle

Alle Buszugriffe im LT-Modus sind blockierend. Die Zugriffskontrolle ist direkt in der an AHBIN gebundenen Transportfunktion *b_transport* realisiert. Transaktionen werden weitergeleitet, wenn

der Bus nicht durch eine andere Transaktion verwendet wird oder durch einen anderen *Master* gesperrt wurde (Abbildung 4.14). Die erforderlichen Sperren wurden mit Hilfe globaler Variablen definiert (Zeile 2). Da der *SystemC-Scheduler* kooperatives *Multi-Threading* verwendet, werden keine Semaphoren benötigt. Der Bus bleibt gesperrt, bis die vorherige Transaktion *busy=false* signalisiert und das Ereignis *free_event* auslöst. Dies geschieht nach der Rückkehr der Transaktion vom *Slave* und nach Konsum der annotierten Verzögerungszeit. Der Mechanismus sieht momentan keine zeitliche Entkopplung vor (*Temporary Decoupling*), kann dafür aber durch Verschieben des Synchronisationspunktes in den *Master* problemlos erweitert werden.

```
1 // Warte wenn Bus besetzt oder gesperrt durch anderen Master
2 while(busy || (is_lock && (id != lock_master)) wait(free_event);
3
4 // Bus jetzt gesperrt durch aktuelle Transaktion
5 busy = true;
6 ..
7 // Sperre Bus - wenn gefordert (lock - Payload Erweiterung)
8 is_lock     = ahbIN.get_extension<amba::amba_lock>(lock, trans);
9 lock_master = id;
```

Abbildung 4.14: AHBCTRL - Zugriffskontrolle im LT-Modus

Exklusiver Zugriff für atomare Operationen wird durch die AMBA-*Payload*-Erweiterung *lock* realisiert (siehe Abschnitt 3.2.2, Tabelle 3.2). Die Funktion *b_transport* liest die Erweiterung aus und speichert diese gemeinsam mit der ID des *Masters* (Zeilen 8, 9). Atomare Instruktionen wie zum Beispiel SWAPA bestehen aus zwei Speicheroperation (lesen und schreiben), die nicht unterbrochen werden dürfen. Der Prozessor setzt die *lock*-Erweiterung für die erste Speicheroperation und blockiert damit den Bus für alle anderen *Master*. Die zweite Speicheroperation wird ohne *lock*-Erweiterung gesendet, wodurch sich die Blockierung aufhebt.

Transaktionsarbitrierung

Die Transaktionsarbitrierung ist eine Funktion der AT-Abstraktion des AHBCTRL. In der LT-Konfiguration des Modelles wird auf die Modellierung von Arbitrierungseffekten zur Steigerung der Simulationsgeschwindigkeit verzichtet. Eintreffende LT-Transaktionen werden an der Arbitrierungsstufe des Modelles vorbei, direkt zum *Address Decoder* weitergeleitet. Im AT-Modus werden eingehende Transaktionen in einer Tabelle (*Request Map*) erfasst. Die Tabelle hat maximal 16 Einträge und kann damit genau eine Verbindungsanfrage für jeden angeschlossenen *Master* aufnehmen. Verbindungsanfragen werden durch eine Datenstruktur vom Typ *connection_t* dargestellt (Abb. 4.15). Die Pufferung des Transaktionszeigers ist bei nichtblockierender Kommunikation nicht hinreichend. Es müssen pro Transaktion zusätzlich die IDs des *Master*- und des *Slave*-Sockels gespeichert werden. Dadurch wird der einfache Aufbau des TLM-Rückkanals für die END_REQ und BEGIN_RESP Phasen ermöglicht. Darüber hinaus bleibt der Vorwärtskanal für END_RESP erhalten, wodurch die Adresse später nicht erneut dekodiert werden muss.

```
1  // Connection state
2  enum TransStateType { TRANS_INIT, TRANS_PENDING, TRANS_SCHEDULED };
3
4  // AHBCTRL connection descriptor
5  typedef struct {
6    uint32_t master_id;     // ID des Masters
7    uint32_t slave_id;      // ID des Slaves
8    sc_time start_time;     // Request Zeitpunkt (Wartezeitstatistik)
9    TransStateType state;   // Transaktionszustand
10   payload_t * trans;      // Transaktionszeiger
11 } connection_t;
```

Abbildung 4.15: Aufbau von AHBCTRL Verbindungsdeskriptoren

Zu jeder Zeit kann nur ein *Master* im Besitz des Busses sein, wobei das AHB-Protokoll die Überlappung zweier aufeinanderfolgender Transaktionen erlaubt. Es ist also durchaus möglich, dass ein *Master* die Adressphase einleitet, während ein anderer gleichzeitig seine Datenphase abschließt. Das AT-Modell des AHBCTRL stellt daher den Zustand des Busses mit Hilfe zweier globaler Variablen dar: *addr_bus_master* indiziert den gegenwärtigen Besitzer des Adressbusses und *data_bus_master* den Besitzer des Datenbusses. Abbildung 4.16 stellt die Handhabung von Transaktionen im AT-Modus vereinfacht dar.

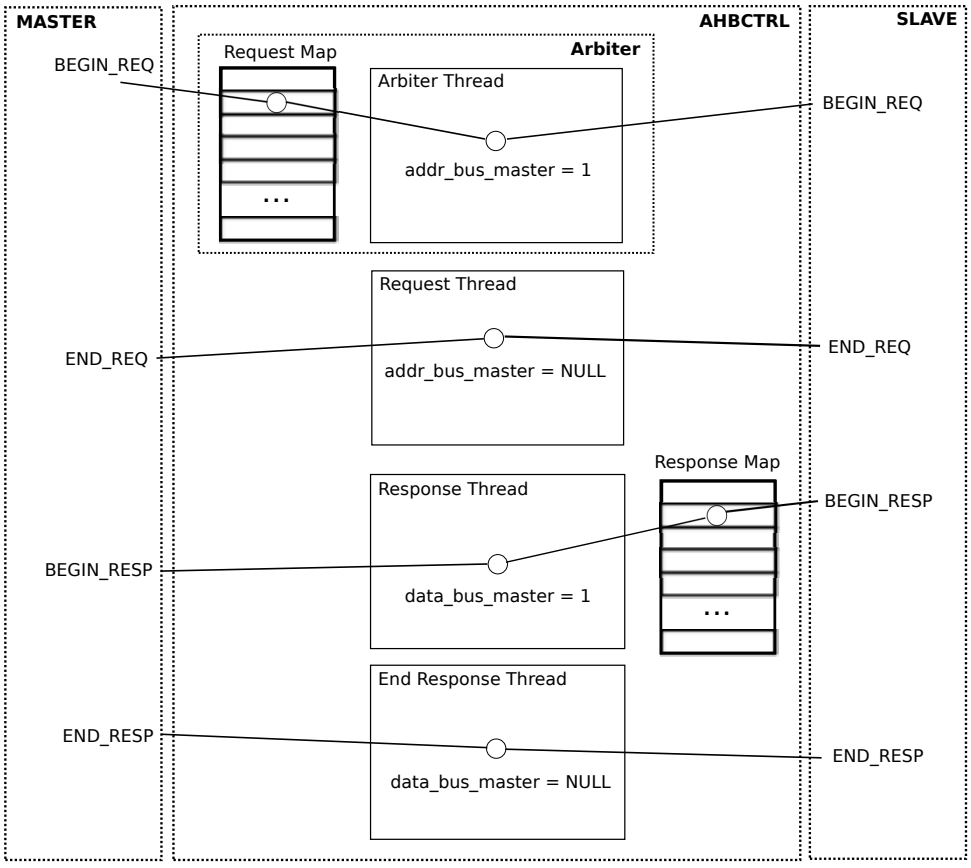


Abbildung 4.16: AHBCTRL - Transaktionsarbitrierung (AT)

Wenn der Adressbus frei ist (*addr_bus_master=NULL*), wählt der *Arbiter Thread* eine Transaktion zur Verarbeitung aus der *Request Map* aus. Die Auswahl wird abhängig von der Konfiguration durch die Priorität des *Masters* (*Master ID*) oder einen *Round Robin*-Zeiger gesteuert. Konnte eine Transaktion gefunden werden, so wird der Adressbus als besetzt markiert. Der *Arbiter* startet nun den *Address Decoder* zur Ermittlung der Zielkomponente für den adressierten Bereich. Der Rückgabewert des *Address Decoders* ist die Identifikationsnummer (ID) des *Slaves*,

die nun zur späteren Wiederverwendung in den Verbindungsdeskriptor eingetragen werden kann. Nach erfolgreicher Arbitrierung leitet der *Arbiter Thread* die Transaktion an den *Slave* weiter. Wie in Kapitel 3.2.2 beschrieben, sendet der *Slave* END_REQ zum geschätzten Zeitpunkt der Übernahme der Transferadresse. Der *Request Thread* des AHBCTRL leitet die Transaktion an den zuständigen Master weiter und gibt den Adressbus frei. Um in der *Request Map* eine neue Transaktion des selben *Masters* aufnehmen zu können, wird der entsprechende Verbindungsdeskriptor in die *Response*-Tabelle (*Response Map*) kopiert. Zum Zeitpunkt BEGIN_REQ markiert AHBCTRL den Datenbus als besetzt (*data_bus_master=1*). Nach *END_RESP* wird die Verbindung aus der *Response Map* gelöscht und der Datenbus freigegeben.

Address Decoder

Eine der Kernfunktionen des AHBCTRL ist die Adressdekodierung. Der *Address Decoder* wird zum Simulationsbeginn dynamisch initialisiert. Dies erfolgt mit Hilfe der Systemfunktion *start_of_simulation*. In einer Schleife werden alle auf *Master*- und *Slave*-Seite gebundenen Komponenten durchlaufen. Alle gültigen von *SoCRocket*-Basisklassen abgeleiteten Module besitzen *Plug & Play*-Konfigurationsregister und können bis zu vier einzeln zu dekodierende Unterkomponenten besitzen. Abbildung 4.17 veranschaulicht den Initialisierungvorgang.

```
1  // Zeiger zum Socket mit Binding-ID i<<2
2  socket_t * other_socket = ahbOUT.get_other_side(i>>2, 0);
3
4  // SystemC Object das den Socket enthaelt
5  sc_core::sc_object *obj = other_socket->get_parent();
6
7  // SystemC Objekt -> AHBDevice
8  AHBDevice *slave = dynamic_cast<AHBDevice *>(obj);
9
10 // AHB Adressinformationen abfragen (Unterkomponente j)
11 ahbaddr = slave->get_bar_base(j);
12 ahbmask = slave->get_bar_mask(j);
13 busid   = slave->get_busid();
14
15 // Daten in AHBCTRL Adresstabelle eingtragen
16 setAddressMap(i+j, busid, ahbaddr, ahbmask);
```

Abbildung 4.17: Initialisierung des AHB-Adressdekodierers

Mit Hilfe der durch den Sockel bereitgestellten Funktionen *get_other_side* und *get_parent* erhält die Initialisierungsroutine Zugriff auf alle angeschlossenen Komponenten (Zeilen 2 u. 5). Das zurückgelieferte *SystemC*-Objekt kann dann auf die Basisklasse AHBDevice abgebildet werden (Zeile 8). Die API der Basisklasse erlaubt den Zugriff auf die Adressinformationen der Komponente (Zeilen 11-13), die dann in die Adressierungstabelle des AHBCTRL übernommen werden (Zeile 16). Der AHBCTRL verwendet die Adressierungstabelle zur Laufzeit, um für jede Transaktion am Bus das zugehörige Ziel auszuwählen. Der dafür erforderliche *Decoder* ist in die Funktion *get_index* eingebettet. Für jeden Zugriff wird die Adressierungstabelle durchlaufen. Zur Auswahl der Zielkomponente wird die in der AMBA-Spezifikation vorgeschlagene Logikfunktion verwendet:

```
(addr ^ haddr) & hmask
```

Dabei definieren *haddr/hmask* den zu testenden Speicherbereich und *addr* die Adresse des Zugriffs. Ist der Ausdruck gleich Null, so befindet sich die Adresse im Bereich der Komponente. In diesem Falle kann *get_index* den Bindungsindex der Komponente zurückliefern. Wie bereits im vorherigen Abschnitt erwähnt, wird die ID des Zieles in den Verbindungsdeskriptor (Abb. 4.15) eingetragen. Der Bindungsindex dient zur Indizierung des *Multi-Sockets AHBOUT* (Abb. 4.18).

```
1 // Blockierende Kommunikation (LT)
2 ahbOUT[bindungsindex]->b_transport(trans, delay);
3
4 // Nicht-blockierende Kommunikation (AT)
5 status = ahbOUT[bindungsindex]->nb_transport_fw(*trans, phase, delay);
```

Abbildung 4.18: Transaktionsweiterleitung mit Bindungsindex am AHBOUT Multi-Socket

Snooping

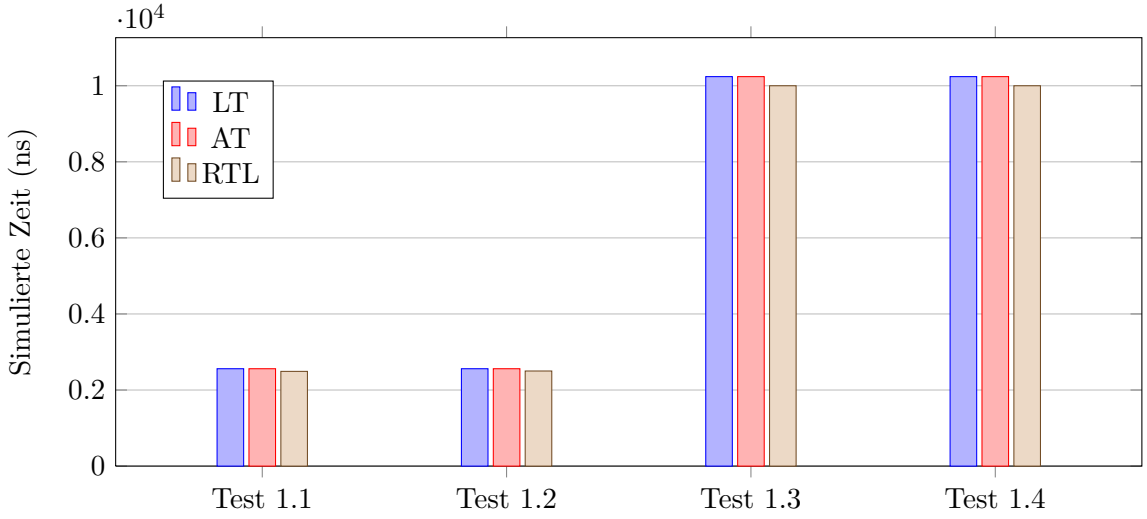
Zur Erhaltung der Cachekohärenz in Multiprozessorsystemen stellt AHBCTRL einen Mechanismus für Datenbus-*Snooping* bereit. Das Modell enthält dazu einen TL-Signalausgang vom Typ *t_snoop*, mit dem *Master*-ID, Adresse und Länge jedes Schreibzugriffes im System propagiert werden können. Jeder Bus-*Master* hat nun die Möglichkeit, einen *Handler* für diesen Ausgang zu registrieren, der nur aktiviert wird, wenn ein entsprechender Schreibvorgang vorliegt (siehe Abschnitt 3.2.5). Dieser Mechanismus ist sehr einfach und effizient, da die Daten im Stile eines *Broadcasts* automatisch verteilt werden. Außerdem ist im Vergleich zu anderen Lösungen nur eine geringe Anzahl an Aktivierungen erforderlich. Die angeschlossenen Komponenten müssen nicht aktiv am Bus lauschen und alle übertragenen Transaktionen auswerten, was eine wesentliche Komplexitätsreduktion zur Folge hat.

Verifikation und Leistungsfähigkeit

Alle Tests des AHBCTRL werden von einer zentralen Basisklasse (*ahb_ctrl_test*) abgeleitet, die auf die in Abschnitt 3.8 beschriebene Infrastruktur zurück geht. Die Klasse dient als Platzhalter für den LEON2/3-Prozessorsimulator und stellt einen AHB-*Master Socket* und einen TL-Signaleingang für Datenbus-*Snooping* bereit. Darüber hinaus enthält die Klasse Speicherpools zur effizienten Wiederverwendung von *Payload*- und Referenzdaten, sowie Zugriffsfunktionen zur Generierung gerichteter und zufallsgesteuerter Datentransfers. Zur Erzeugung eines Tests muss der Nutzer lediglich von *ahb_ctrl_test* ableiten und den zentralen Test-*Thread* mit Steuerinformationen füllen. Die Abstraktionsebene des Busmodelles ist dabei transparent.

Der *AHBCTRL* kann mit bis zu 16 *Master*- und 16 *Slave*-Komponenten verbunden werden. Zur Verifikation der Komponente wurden verschiedene Konfigurationen ausgewählt. Abbildung 4.19 zeigt die Ergebnisse eines Tests mit einem *Master*, der auf einen *Slave* gezielt (1.1, 1.2) und zufällig (1.3, 1.4) zugreift. Die Simulationsgenauigkeit im Vergleich zum VHDL-Referenzmodell liegt über 95%. Auf Grund der relativ einfachen Konfiguration sind kaum Unterschiede zwischen LT- und AT-Modus erkennbar. Dies ändert sich, sobald sich die Anzahl der Komponenten im System erhöht. Abbildung 4.20 zeigt die Simulationsgenauigkeit für einen Bus in Maximalkonfiguration mit Prioritätsarbitrierung. Die angeschlossenen *Master* versuchen gleichzeitig jeweils 10.000 zufallsgesteuerte Zugriffe. Der *Master* mit der niedrigsten ID erhält die höchste Priorität. Dies wird im RT- und im AT-Modell berücksichtigt. Das LT-Modell abstrahiert den Arbitrierungsmechanismus und leitet Transaktionen entsprechend ihrer Ankunftsreihenfolge weiter. Während das AT-Modell eine sehr hohe Genauigkeit aufweist (nahe 100%), ist das LT-Modell hier also sehr ungenau. Trotzdem ist die Gesamtausführungszeit des Tests auf allen Abstraktionsebenen nahezu gleich (ca. 1.6 µs). Unterschiede bestehen lediglich in der Abarbeitungsreihenfolge. Diese Beobachtungen lassen sich auf die in Tabelle 4.8 dargestellten weiteren Tests übertragen, insbesondere die in Test 6 betrachtete *Round Robin*-Arbitrierung.

Die Komponententests des AHBCTRL dienen hauptsächlich der Bestimmung der Simulationsgenauigkeit. Auf Grund des durch die Testumgebung eingeführten *Overheads* können keine genauen Aussagen zur Simulationsgeschwindigkeit getroffen werden. In der Maximalkonfiguration (Test 5) simulieren das LT-Modell 80x schneller und das AT-Modell 50x schneller als die VHDL-Referenz.



Test 1	Beschreibung
1.1	Gezieltes Schreiben (256 Worte)
1.2	Gezieltes Lesen (256 Worte)
1.3	Zufallsgesteuertes Schreiben (1000 Zugriffe)
1.4	Zufallsgesteuertes Lesen (1000 Zugriffe)

Abbildung 4.19: AHB-Bus / Test mit Minimalkonfiguration (1x Master, 1x Slave)

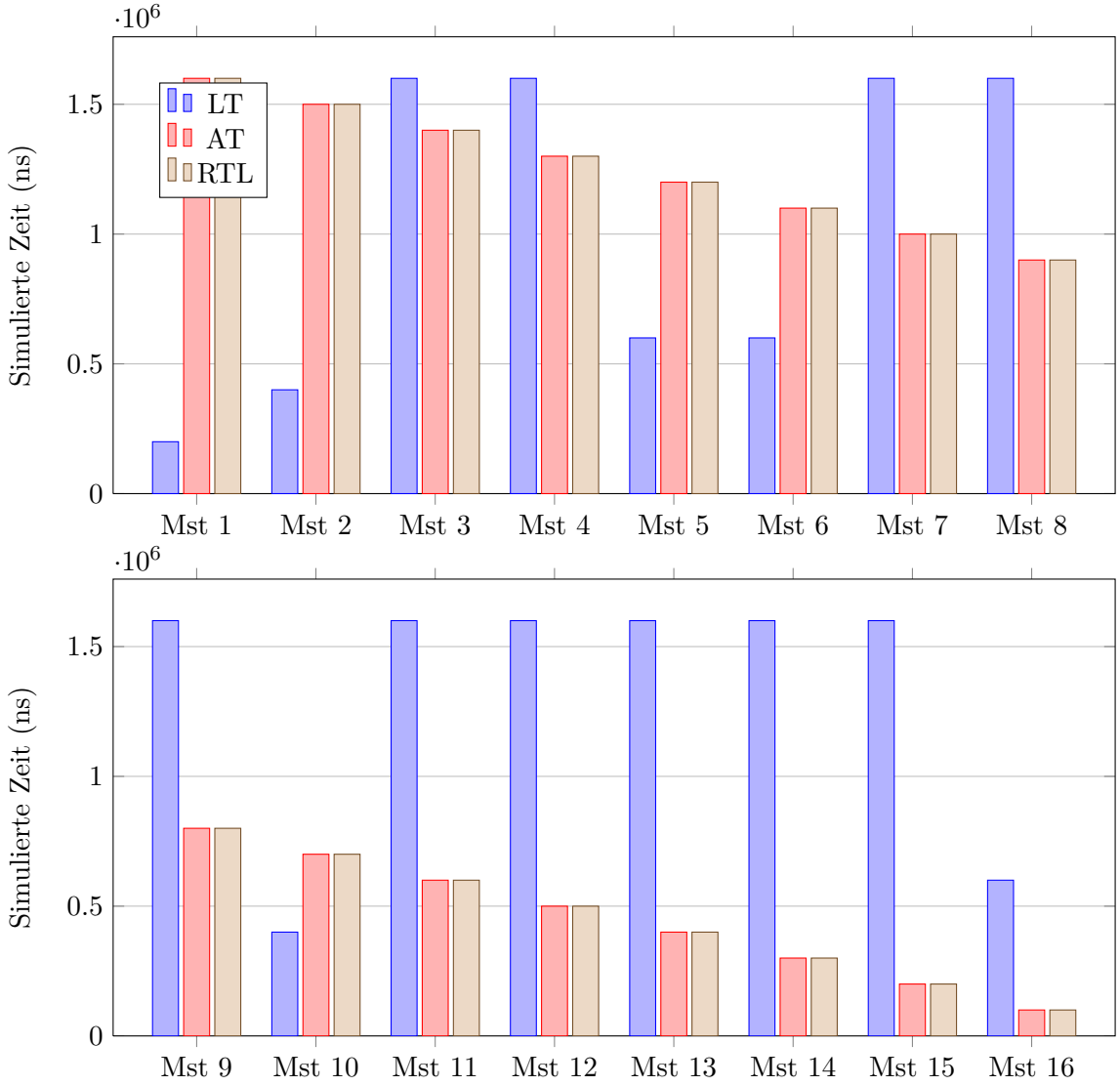


Abbildung 4.20: AHB-Bus / Test mit Maximalkonfiguration (16x Master, 16x Slave)

Test	Beschreibung	LT (ns)	AT (ns)	RTL (ns)
2^1, 7^2	Konfiguration mit zwei *Master* und zwei *Slave* Prioritätsarbitrierung[1] oder *Round-Robin*-Arbitrierung[2]:			
	- Einzeloperationen mit gezielter Adressierung	4000[1]	4000[1]	3990[1]
	- Gleichzeitiges Lesen und Schreiben	7980[1]	6000[1]	6000[1]
	- Test des *Snoop*-Mechanismus	2000[1]	2000[1]	1990[1]
3	16 *Master* und ein *Slaves*			
	- Gleichzeitiges zufallsgesteuertes Schreiben (je 1000x)	19990 - 160000	(16-ID) *10000	(16-ID) *10000
	- Gleichzeitiges zufallsgesteuertes Lesen (je 1000x)	19990 - 160000	(16-ID) *10000	(16-ID) *10000
4	Ein *Master* und 16 *Slaves*:			
	- Zufallsgesteuertes Schreiben (100000x)	1000000	1000000	999990
	- Zufallsgesteuertes Lesen (100000x)	1000000	1000000	1000000
6	16 *Master* und 16 *Slaves* mit *Round-Robin*-Arbitrierung:			
	- Gleichzeitiges zufallsgesteuertes Schreiben (je 10000x)	19990 - 160000	160000	159950
	- Gleichzeitiges zufallsgesteuertes Lesen (je 10000x)	19990 - 159990	159940	159950

Tabelle 4.8: AHB-Bus / Übersicht weitere TLM-Tests

4.2.2 APB-Controller (APBCTRL)

Übersicht

Die Implementierung des APBCTRL-Simulationsmodells befindet sich im Unterverzeichnis */models/apbctrl* der Plattform (Anhang B). Der APBCTRL stellte eine Brücke zwischen einem AHB-Bus auf der *Master*- und einem APB-Bus auf der *Slave*-Seite dar (Abb. 4.21). Der *Master-Socket* (AHBIN) dient zur Anbindung des AHBCTRL. An den APB-*Socket* können beliebige Komponenten mit APB-Schnittstelle verbunden werden. *SoCRocket*-Komponenten mit APB-Schnittstelle sind zum Beispiel MCTRL, IRQMP oder GPTimer. Zur Auswahl der *Slaves* verwendet APBCTRL eine interne Adressierungstabelle. Ähnlich wie im AHBCTRL wird diese zum Simulationsstart automatisch aus den *Plug & Play*-Registern der angeschlossenen Komponenten aufgebaut.

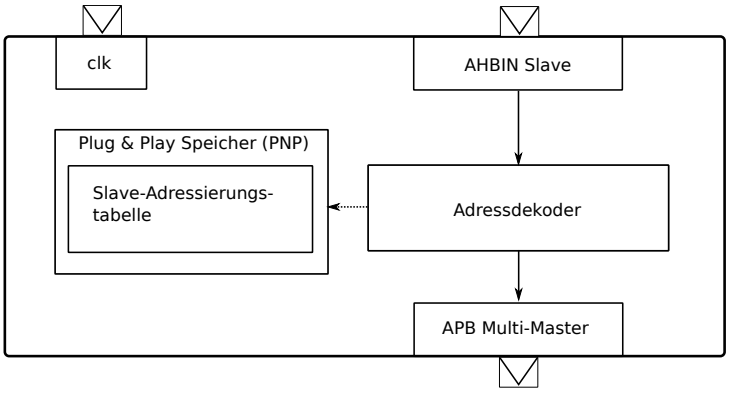


Abbildung 4.21: TL-Modell des APB-Controllers (APBCTRL)

TLM-Schnittstellen

Der APBCTRL präsentiert sich auf der AHB-Seite als gewöhnliche AHB-*Slave*-Komponente mit festem Adressbereich. Das Modell erbt von der Bibliotheksbasisklasse *AHBSlave* und erhält dadurch ein *Plug & Play*-Registerfile und einen AHB-*Slave-Socket* der wahlweise für LT- und AT-Kommunikation konfiguriert werden kann. Da APBCTRL der einzig sinnvolle Anwendungsfall für eine APB-*Master*-Komponente ist, wurde von der Kapselung in einer Basisklasse abgesehen. Der Aufwand zur Implementierung des *Sockets* ist gering, da alle Kommunikation auf der APB-Seite als blockierend betrachtet wird (siehe Abschnitt 3.2.3). Darüber hinaus werden alle Zugriffe auf im APB-Bereich verbundene Komponenten als nicht-*cacheable* betrachtet. Diese Information wird mit Hilfe der *Payload*-Erweiterung *amba_ext* transportiert. Zur Annotation des Timings erbt APBCTRL die Basisklasse *CLKDevice*.

Address Decoder

APB-Komponenten identifizieren sich gegenüber APBCTRL durch eine Adressen/Masken-Kombination (*paddr/pmask*). Aus globaler Sicht befinden sich alle APB-Komponenten in dem durch die Busbrücke am AHB-Bus reservierten Speicherbereich. Die APB-Basisadresse *paddr* ist 12 Bit breit und entspricht Bits 8-20 der globalen AHB-Adresse. Die ebenfalls 12 Bit breite Zugriffsmaske *pmask* definiert die Größe des durch die Komponente belegten Adressbereiches. Ähnlich wie im AHBCTRL ist der *Address Decoder* des APBCTRL in einer einzelnen Funktion implementiert (*get_index*). Die Funktion erhält die Zieladresse der Transaktion als Eingabe und vergleicht diese in einer Schleife mit allen Einträgen der Adressierungstabelle. Dazu wird folgende Logikfunktion verwendet:

```
(addr ^ paddr) & pmask
```

Wird der Ausdruck für einen der Einträge Null, so konnte ein gültiger *Slave* für den Zugriff gefunden werden. Die Funktion liefert dann den entsprechenden Bindungsindex des APB-Sockels zurück. Überschneidungen in der Adressierungstabelle werden durch die Funktion *checkMemMap* verhindert.

Transaktionsweiterleitung AHB/APB

Wie bereits erwähnt, wurde der APBCTRL von der Bibliotheksbasisklasse *AHBSlave* abgeleitet. Der AHB-Sockel kommuniziert mit dem Verhaltensteil der Komponente über die *Callback*-Funktion *exec_func*. Diese wird im LT-Modus unmittelbar durch die blockierende Transportfunktion (*b_transport*) und im AT-Modus nach dem Empfang von BEGIN_REQ aufgerufen. Die Verhaltensfunktion überprüft zunächst, ob die Transaktion an den APB-Bus weitergeleitet werden muss oder an den Konfigurationsbereich der Busbrücke (*Plug & Play*) gerichtet ist. Konfigurationszugriffe werden unmittelbar mit einer Verzögerung von einem Takt pro Wort abgearbeitet und beenden die Transaktion. Für reguläre APB-Zugriffe ruft *exec_func* zur Ermittlung des Bindungsindizes des Transaktionszieles den in die Funktion *get_index* eingebetteten *Address Decoder* auf. Danach wird das *Payload*-Objekt der AHB-Transaktion ausgelesen. Alle für den APB-Transfer relevanten Informationen werden in ein statisches APB-*Payload*-Objekt kopiert. Dieses wird dann unter Angabe des Bindungsindex über den APB-Sockel weitergeleitet. Das Kopieren der *Payload* stellt einen geringen *Overhead* dar, sorgt aber für eine saubere Trennung zwischen den AHB- und APB-Bereichen der Busbrücke. Pro Transaktion werden 20 Byte kopiert. Die APB-*Payload* benötigt keinen Speicherpool, da aufgrund der Natur des APB-Protokolls immer nur eine Transaktion existieren kann (Abschnitt 3.2.3). Die Verzögerung von APB-Transfers wird durch die angeschlossenen *Slaves* bestimmt. Alle Kernkomponenten von *SoCRocket* implementieren eine Verzögerung von einem Takt pro Zugriff. APBCTRL addiert dazu einen weiter Takt zur Modellierung der APB-*Setup*-Phase.

Verifikation und Leistungsfähigkeit

Zur Verifikation des APBCTRL wurde die zum Test des AHBCTRL verwendete Infrastruktur wiederverwendet (siehe Abschnitt 4.2.1). Der beschriebene Stimulusgenerator wird als alleiniger *Master* an den AHB-Bus angeschlossen. Dieser leitet die Testdaten an die APB-Brücke weiter, welche verschiedene APB-*Slave*-Komponenten adressiert. Auf Grund des trivialen *Timings* der APB-Busbrücke wurde dabei auf RTL-Co-Simulation verzichtet. Die Tests lesen und schreiben den APB-Konfigurationsbereich sowie alle angeschlossenen Register. Jede APB-Operation im Test beansprucht zwei Taktzyklen. Das Abstraktionsniveau der AHB-*Master*-Schnittstelle (LT-/AT-Modus) hat keinen Einfluss auf das Zeitverhalten. Die beobachtete Simulationsgenauigkeit entspricht somit 100%. Die genaue Testabfolge kann dem Quellcode und [Sch12d] entnommen werden. Zusätzliche Tests, auch in RTL-Co-Simulation, wurden in Zusammenhang mit der Verifikation des Speichercontrollers durchgeführt (Abschnitt 4.3.3).

4.3 Peripherie-Komponenten

Peripheriekomponenten sind terminale Komponenten, die zur Bereitstellung einer bestimmten Funktion für den Prozessor an den Systembus angeschlossen werden. Dieser Abschnitt stellt mit dem *General Purpose Timer* (GPTimer), dem *Multi-Processor Interrupt Controller* (IRQMP), dem Speichercontroller (MCTRL) und dem APB-UART die wichtigsten derartigen Komponenten der *SoCRocket*-Modellbibliothek vor.

4.3.1 General Purpose Timer (GPTimer)

Übersicht

Der Quellcode des Modells des GRLIB-*Timers* (GPTIMER) befindet sich im Unterverzeichnis */models/gptimer* der Plattform (Anhang B). Das Modell besteht aus bis zu sieben konfigurierbaren Zählereinheiten, die im Takt eines Skalierregisters (*Prescaler*) dekrementiert werden (Abbildung 4.22). Die Zähler und alle zugehörigen Steuerregister werden im Konstruktor abhängig von Parametern dynamisch erzeugt. Die Funktion des Modells besteht in einem einfachen *Countdown*-Mechanismus. Sobald einer der Zähler den Nullwert unterschreitet (*Underflow*), wird ein Interrupt generiert. Zähler Sieben kann als *Watchdog* konfiguriert werden. Interrupts des *Watchdog*-Zählers werden über einen separaten Ausgabeport geleitet (*wdog*).

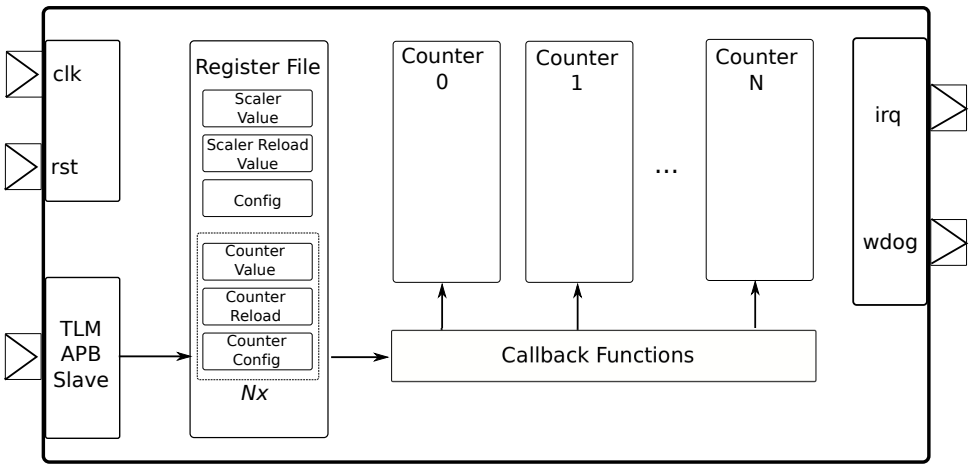


Abbildung 4.22: TL-Modell des *General Purpose Timer* (GPTimer)

TLM-Schnittstellen

Der GPTimer ist eine reine APB-Komponente. Das Modell erbt seine Busschnittstelle von der Basisklasse *APBDevice*. Der APB-*Slave* ist blockierend und direkt an die interne Registerbank gekoppelt. Darüber hinaus verfügt das Modell über einen Signaleingang zur Modellierung des Reset (*rst*), einen Ausgabevektor zur Generierung der Interrupts (*irq*) und, wie bereits erwähnt, einen Port für das *Watchdog*-Signal (*wdog*). Der Systemtakt kann mit Hilfe einer Schnittstellenfunktion (*set_clock*) gesetzt werden. Dazu erbt GPTimer die *SoCRocket*-Basisklasse *CLKDevice*.

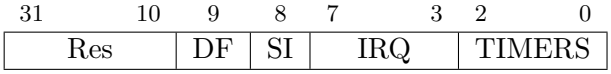
Steuerregister

Der GPTimer verfügt über eine Registerbank, die direkt an die APB-*Slave*-Schnittstelle angebunden ist. Die Registerbank dient der Konfiguration des Moduls und enthält alle zur Operation der bis zu sieben Zähler erforderlichen Zähl- und Laderegister. Die Register werden relativ zur APB-Basisadresse (*paddr/pmask*) adressiert (Tabelle 4.9).

APB Adresse (Offset)	Register
0x00	Scaler Value
0x04	Scaler Reload Value
0x08	Configuration Register
0x0c	Unused
0x**n**0	Counter **n** Value Register
0x**n**4	Counter **n** Reload Register
0x**n**8	Counter **n** Configuration Register
0x**n**c	Unused

Tabelle 4.9: GPTimer - Übersicht Steuerregister

Die Konfiguration des Modells erfolgt in zwei Stufen. Globale Einstellungen, wie die Anzahl der Zähler oder die Nummer des Basisinterrupt werden im *Configuration Register* vorgenommen (Tabelle 4.10). Mit Hilfe des SI-Bits kann eingestellt werden, ob alle Zähler den selben oder unterschiedliche Interrupts auslösen. Im Falle unterschiedlicher Interrupts (SI=1) enthält das IRQ-Feld die Nummer des Basisinterrupts für Zähler Null. Die Interrupts für alle weiteren Zähler werden davon ausgehend in aufsteigender Reihenfolge zugewiesen ($ZählerN\ N_{IRQ} = IRQ + N$). Das DF-Feld kann beschrieben und gelesen werden, hat hier aber keine Funktion: Im RTL-Modell dient es dem Einfrieren der Zählerregister im Falle eines *Debug*-Stops. Auf TL-Ebene ist die Funktion nicht relevant.



Bit-Feld	Beschreibung	Unterstützt
DF	*Disable-Freeze* zum Einfrieren der Timers im Falle eines RTL Debug-Stops (dhalt)	nein
SI	*Separate Interrupts*: Gleicher oder unterschiedliche Interrupts für alle Zähler	LT, AT
IRQ	Nummer des Basisinterrupts	LT, AT
TIMERS	Anzahl der Zähler (1-7)	LT, AT

Tabelle 4.10: GPTimer - *Configuration Register*

Neben dem globalen *Configuration Register* hat jeder Zähler ein lokales *Counter Configuration Register* (Tabelle 4.11). Alle Felder dieses Registers, mit Ausnahme von *Debug Halt*, werden sowohl auf LT- als auch AT-Ebene unterstützt. Die einzelnen Zähler müssen zum Beginn der

Simulation per Software eingeschaltet werden (EN=1, IE=1). Durch das Setzen des CH-Feldes wird der Zähler mit seinem Vorgänger verkettet.

31 7	6	5	4	3	2	1	0
Res	DH	CH	IP	IE	LD	RS	EN

Bit-Feld	Beschreibung	Unterstützt
DH	Debug-Halt	nein
CH	Chaining: Mit vorherigem Zähler verketten	LT, AT
IP	Interrupt Pending	LT, AT
IE	Interrupt Enable	LT, AT
LD	Lade Zählerwert aus dem Reload-Register ins Value-Register	LT, AT
RS	Restart-Zähler neu starten	LT, AT
EN	Enable	LT, AT

Tabelle 4.11: GPTimer - *Counter Configuration Register*

Interrupt Generierung

Zur Generierung eines Interrupts werden die einzelnen Zähler des GPTimers im Takt des *Prescalers* heruntergezählt. Der *Prescaler* wird mit dem Systemtakt betrieben und aus dem *Prescaler Reload*-Register initialisiert. Die Zähler selbst werden bei *Underflow* auf den Wert des zugehörigen *Counter Reload*-Registers zurückgesetzt. Die Momentanwerte der Zähler befinden sich in den *Counter Value*-Registern. Zur Implementierung dieses Verhaltens hält das RTL-Modell mehrere Prozesse vor, die in regelmäßigen Abständen aktiviert werden. Bei jeder Aktivierung wird ein Registerwert (*Scaler Value*- oder *Counter Value*-Register) mittels einer Arithmetikschaltung dekrementiert. Das TL-Modell enthält keine derartigen Prozesse und Zählerschaltungen. Zur Erhöhung der Leistungsfähigkeit wird die Zeit bis zum nächsten zu erwartenden Interrupt im voraus berechnet. Mit Hilfe der *SystemC*-Anweisung *wait* wird der ausführende *Thread* dann bis zum Zeitpunkt des Interrupts zurückgestellt.

$$wait(Reg[ScalerReload] * T_{clock} * Reg[CounterReload])^1$$

Das Modell wird durch den *SystemC-Kernel* nur einmal pro Zählerintervall aktiviert. Wird eines der *Reload*-Register zur Laufzeit geändert, so werden im Verhalten Rückruffunktionen ausgelöst. Diese Veranlassen die dynamische Neuberechnung der Wartezeit (Funktion *GPCounter::calculate*). Die betrachtete Vereinfachung des Verhaltens hat zur Folge, dass die *Counter-Value*-Register der Zähler nicht fortlaufend aktualisiert werden. Lesezugriffe werden darum wiederum durch eine Rückruffunktion abgefangen (*GPCounter::value_read*). Die Funktion berechnet den Momentanwert des Registers aus der zeitlichen Differenz zwischen Zugriff und Zurücksetzen (*Reload*) des entsprechenden Zählers.

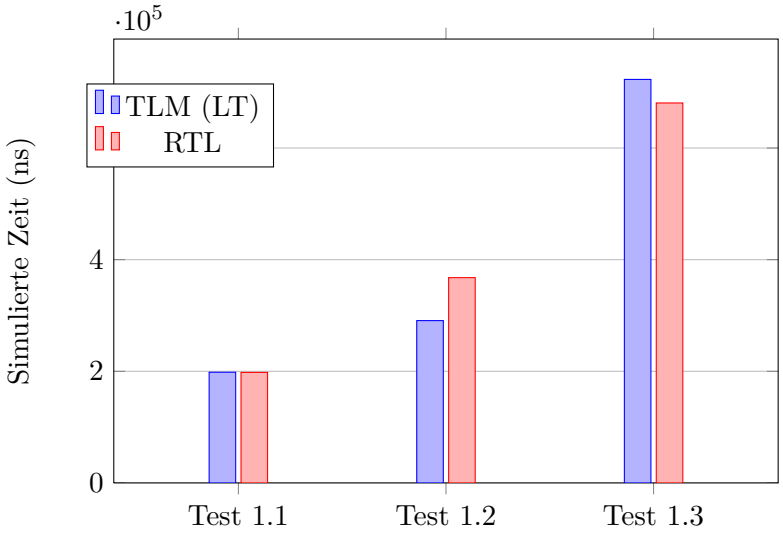
$$Reg[ScalerValue] = Reg[ScalerReload] - (T_{now} - T_{ScalerReload})/Reg[PrescalerReload]$$

Zur Vergrößerung der Zählerverzögerung können zwei oder mehrere Zähler miteinander verkettet werden. Befinden sich ein Zähler im Verkettungsmodus (CH=1), so wird dieser nicht im Takt des *Prescalers* sondern durch das Zurücksetzen des vorherigen Zählers dekrementiert. Wie im RTL-Modell lösen verkettete Zähler einen Interrupt aus, wenn die IE-Felder der entsprechenden *Counter Configuration*-Register nicht gelöscht wurden.

1 Vereinfachte Darstellung - *Threads* warten auf ein *SystemC*-Ereignis, dass in Anlehnung an die dargestellte Formel aktiviert und im Falle von Nebeneffekten abgebrochen oder neu berechnet werden kann (siehe Quell-Code).

Verifikation und Leistungsfähigkeit

Zur Überprüfung der Simulationsgenauigkeit und -geschwindigkeit wurde das GPTimer-TLM als Einzelkomponente unabhängig vom Rest des Systems getestet. Die Testumgebung besteht aus einem Stimulusgenerator, der das Modell über die APB-*Slave*-Schnittstelle programmiert und dessen Interruptausgänge überwacht. Abbildung 4.23 vergleicht die Simulationszeit des TL-Modelles mit denen der RTL-Referenz. Der Test kann zur Simulation beider Modelle wiederverwendet werden. Zur Ansteuerung des VHDLs werden entsprechende Transaktoren eingefügt. Der Test zeigt, dass die Simulationsgenauigkeit im Normalbetrieb, der Interruptgenerierung durch mehrere unabhängige Zähler, sehr hoch ist (nahe 100%). In Randfällen oder bei Nutzung von Spezialfunktionen können Fehler von bis zu 20% auftreten. Zusätzliche Testergebnisse und detaillierte Testbeschreibungen können [Sch12d] oder dem Quellcode (Anhang B) entnommen werden.



Test 1	Beschreibung
Test 1.1	Normalbetrieb - Zähler 0-6 und Watchdog generieren Interrupts
Test 1.2	Test ohne automatischem Zähler-Reset (Zähler 0-5)
Test 1.3	Chaining-Test (Zähler 0-4)

Abbildung 4.23: GPTimer / Test der Grundfunktionalität

4.3.2 Multi-Processor Interrupt Controller (IRQMP)

Übersicht

Der *SoCRocket*-IRQMP ist ein TL-Simulationsmodell des *Multi-Processor Interrupt Controller* aus der GRLIB. Der Quellcode des Modells befindet sich im Verzeichnis */models/irqmp* der Plattform (siehe Anlage B). Aufgabe des IRQMP ist die Priorisierung und Maskierung der Interrupts aller Komponenten des Systems. Es werden bis zu 16 CPUs unterstützt. Der Interrupt mit der jeweils höchsten Priorität wird entweder an eine oder mehrere CPUs weitergeleitet. Für Multiprozessorsysteme werden zwei Modi zur *Interrupt*-Behandlung bereitgestellt. In beiden Fällen wird der Interrupt an alle ausgewählten CPUs gesendet. Er kann nun durch eine CPU bestätigt und gelöscht werden. Dabei wird die *Interrupt Service Routine* nur einmal ausgeführt. Alternativ kann eine Bestätigung durch alle CPUs erzwungen werden. In diesem Fall wird der Interrupt durch jede der CPUs verarbeitet. Die Struktur und Funktionsweise des Modells sind in Abbildung 4.24 vereinfacht dargestellt und werden in den folgenden Abschnitten erläutert.

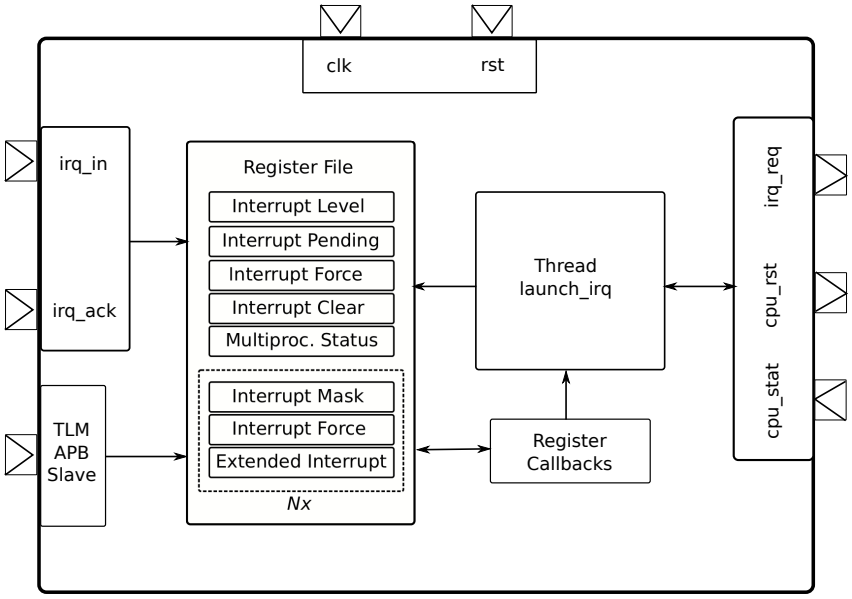


Abbildung 4.24: TL-Modell des *Multi-Processor Interrupt-Controller* (IRQMP)

TLM-Schnittstellen

Ähnlich wie der *General Purpose Timer* (GPTimer) ist IRQMP eine reine APB-Komponente. Die APB-*Slave*-Schnittstelle wird von der Basisklasse *APBDevice* geerbt. Alle internen Register sind in einer Bank zusammengefasst, die direkt mit dem *Socket* verbunden ist und über diesen gelesen und beschrieben werden kann. IRQMP erbt ebenfalls von *CLKDevice* und erhält dadurch eine Schnittstelle zur *Timing*-Annotation. Der *Reset*-Eingang (*rst*) und alle Interruptports sind als TL-*Signalsockets* implementiert. Die Interrupts der Peripheriekomponenten werden über den Eingabevektor *irq_in* (*infield*) eingelesen. Zur Verbindung mehrerer *Signalsockets* stellt das System die Funktion *connect* bereit (siehe Abschnitt 3.2.5). Um einen Ausgabesignalvektor mit einem Eingabevektor zu verknüpfen, müssen die entsprechenden Kanalnummern übergeben werden. Das folgende Kommando verbindet Interrupt drei des GPTimer mit Kanal fünf des IRQMP:

```
connect(gptimer.irq, irqmp.irq_in, 3, 5);
```

Hat die anzubindende Komponente nur einen Interruptausgang (*SignalKit::out*), so wird nur die Kanalnummer des IRQMP angegeben:

```
connect(socwire.irq, irqmp.irq_in, 6);
```

Interrupts werden über den *irq_req*-Ausgabevektor, je nach Einstellung, an eine oder mehrere CPUs weitergeleitet. Der Port hat den Datentyp *uint32_t* und transportiert die Kanalnummer des Interrupts. Die Bestätigungssignale (*Ackknowledgements*) der CPUs werden über den Eingangsvektor *irq_ack* eingelesen. Das Löschen von Interrupts erfolgt ausschließlich im IRQMP. Die Weiterleitung der Bestätigungssignale an die Peripheriekomponenten ist nicht erforderlich.

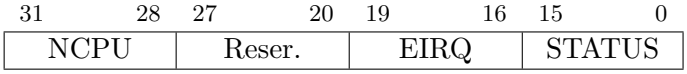
Steuerregister

Die Registerbank des IRQMP ist direkt an die APB-*Slave*-Schnittstelle angebunden. Alle Register wurden mit *GreenReg* modelliert. Die Komponente erbt hierzu die Basisklasse *gr_device*. Tabelle 4.12 listet die Steuerregister des IRQMP und zeigt deren Adressierung relativ zur APB-Basisadresse (*paddr/pmask*).

APB Adresse (Offset)	Register
0x00	Interrupt Level Register
0x04	Interrupt Pending Register
0x08	Interrupt Force Register
0x0c	Interrupt Clear Register
0x10	Multi-Processor Status Register
0x14	Broadcast Register
0x40 + 4**n**	Processor **n** Interrupt Mask Register
0x80 + 4**n**	Processor **n** Interrupt Force Register
0xc0 + 4**n**	Processor **n** Extended Interrupt Identification Register

Tabelle 4.12: GPTimer - Übersicht Steuerregister

Die Register sind sehr einfach strukturiert und enthalten mit wenigen Ausnahmen nur jeweils eine Bitmaske. Alle Felder/Masken werden sowohl im LT- als auch im AT-Modus unterstützt. Für jeden eingehenden Interrupt wird ein Bit im *Interrupt Pending*-Register gesetzt. Das *Interrupt Level*-Register spezifiziert die Priorität der einzelnen Interrupts. Durch Beschreiben des *Interrupt Clear*-Registers können Interrupts per Software gelöscht werden. Darüber hinaus verfügt jeder Prozessor über eigene *Interrupt Mask-*, *Interrupt Force-* und *Extended Interrupt Identification*-Register. Damit ist es möglich, Interrupts individuell zu unterdrücken oder zu erzwingen. Erweiterte Interrupts werden durch einen regulären Interrupt indirekt ausgelöst. Wird dieser reguläre Interrupt durch eine der CPUs bestätigt, so schreibt der IRQMP die Nummer des erweiterten Interrupts in das entsprechende *Extended Interrupt Identification*-Register. Durch einen Lesezugriff auf das Register kann die CPU die zugehörige *Interrupt Service*-Funktion identifizieren. Eine besondere Bedeutung kommt dem *Multi-Processor Status*-Register zu (Tabelle 4.13).



Bit-Feld	Beschreibung	Unterstützt
NCPU	Anzahl der CPUs im System - 1	LT, AT
EIRQ	Nummer des erweiterten Interrupts	LT, AT
STATUS	CPU-Status-Bits: pro CPU '1'= aus, '0'=ein	LT, AT

Tabelle 4.13: *IRQMP* - Multi-Processor Status Register

Das STATUS-Feld des Registers zeigt an, welche Prozessoren momentan eingeschaltet ('0') und welche ausgeschaltet ('1') sind. Durch das Ausschalten einer CPU wird diese in *Reset* versetzt. Beim Neustart des Systems sind alle CPUs mit Ausnahme von CPU 0 abgeschaltet. Diese ist dann als alleiniger *Master* für das Hochfahren aller anderen Prozessoren verantwortlich. Um diesen Mechanismus unterstützen zu können, wurde der LEON-ISS um ein Eingabesignal *run* und ein Ausgabesignal *status* erweitert. Beide Signale sind vom Typ Boolean und müssen bei der Konstruktion einer VP mit den Ports *cpu_rst* bzw. *cpu_stat* des IRQMP verbunden werden.

Interrupt Routing

Auf Hardwareebene werden Interrupts auf einem 32-Bit Interruptbus verteilt, der gemeinsam mit den Signalen des AMBA-Busses mit alle Komponenten des Systems verbunden wird. Die unteren 16 Bit (LSBs) dieses Busses werden mit regulären Interrupts assoziiert, die oberen 16 Bit sind für erweiterte (*extended*) Interrupts reserviert. In *SoCRocket* werden Signale durch einen eigens entwickelten Signalbaukasten propagiert (siehe Abschnitt 3.2.5). Lese- und Schreibvorgänge an Signalports lösen Rückruffunktionen innerhalb des IRQMP aus. Schickt eine Komponente einen Interrupt, so startet dies die Funktion *incoming_irq*. Dadurch wird der Interrupt in das

Interrupt Pending-Register oder gegebenenfalls in das *Interrupt Force*-Register übernommen. Alternativ kann ein Interrupt durch direktes Beschreiben des *Interrupt Pending*-Registers über den APB-Bus ausgelöst werden. In diesem Fall wird die *GreenReg*-Rückruffunktion *pending_write* aktiviert. Beide Funktionen, *incoming_irq* und *pending_write*, starten den *SystemC-Thread launch_irq*, der den größten Teil des Verhaltens der Komponente enthält. Bei jedem Aufruf von *launch_irq* wird der Interruptstatus jedes Prozessors neu berechnet. Dazu werden die angeschlossenen Prozessoren in einer Schleife beginnend mit der höchsten *Master-ID* durchlaufen. Der *Thread* vergleicht das *Interrupt Pending*-Register mit der jeweiligen Instanz des *Interrupt Mask*-Registers, unter Berücksichtigung der Einstellungen für *Broadcasting* und *Forcing*. Dadurch wird eine Maske gültiger, ausstehender Interrupts ermittelt. Aus dieser Maske wird der Interrupt mit der höchsten Priorität zur Weiterleitung ausgewählt. Die Priorisierung erfolgt in zwei Stufen. Die erste Stufe ist die Interruptebene (0 oder 1), die mit Hilfe des *Interrupt Level*-Registers bestimmt werden kann. Die zweite Stufe ist Kanalnummer. Die Kanalnummer ist nur für die Auswahl innerhalb der Interruptebene relevant. Die höchste Priorität hat der Interrupt der Stufe eins mit der höchsten Kanalnummer. Der *Thread launch_irq* beschreibt den Ausgabeport *irq_req* wie folgt:

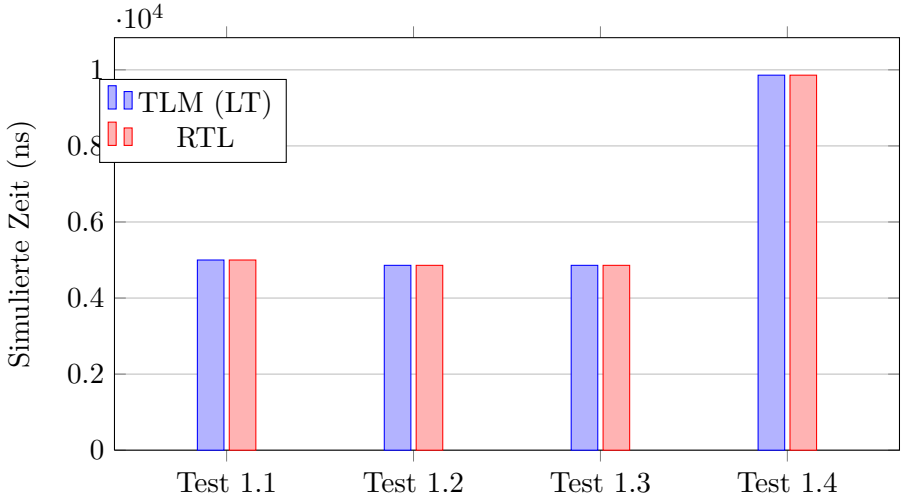
```
irq_req.write(1 << cpu, std::pair<uint32_t, bool>(number, on/off))
```

Mit Hilfe des Feldes *cpu* wird der Kanal des Ausgabevektors angewählt, *number* ist die Nummer des Interrupts, *on/off* bezieht sich auf das Ein- bzw. Ausschalten des Signals. Der *on/off*-Schalter ist dabei nur für RTL-Co-Simulation von Bedeutung und kann auf TL-Ebene ignoriert werden. Neben *Interrupt Pending* sind auch alle anderen Register der Registerbank mit Rückruffunktionen für Schreiboperationen ausgestattet, die ebenso *launch_irq* aktivieren. Der *Thread launch_irq* modelliert somit kombinatorische Logik und kann als sensitiv zur Registerbank betrachtet werden. Diese Art der Modellierung, sollte zur Optimierung der Simulationsgeschwindigkeit auf TL-Ebene, wenn möglich, vermieden werden. Struktur und Funktionsweise des IRQMP lassen jedoch nur wenig Raum zur Abstraktion.

Verifikation und Leistungsfähigkeit

Wie alle Komponenten der Modellbibliothek wurde der IRQMP zunächst als alleinstehende Komponente in einer speziell zugeschnittenen Testumgebung verifiziert (siehe Abschnitt 3.8). Die Testumgebung besteht aus einem Stimulusgenerator, der den IRQMP über seine APB-*Slave*-Schnittstelle programmiert und die Interrupteingänge steuert. Darüber hinaus überwacht die Testumgebung die Interruptausgänge und die Steuerregister des IRQMP, um die korrekte Abarbeitung der eingehenden Interrupts und deren Bearbeitungsreihenfolge zu überwachen. Die Testumgebung kann zur Co-Simulation des RTL-Referenzmodells wiederverwendet werden.

Die Testergebnisse für die Grundfunktionalität des IRQMP sind in Abbildung 4.25 zusammengefasst. In Phase 1 (Test 1.1) werden alle normalen und erweiterten Interrupts einzeln ausgelöst und abgearbeitet. Danach (Test 1.2) werden mehrere Interrupts mit unterschiedlichen Prioritäten gleichzeitig gestartet. Phase 3 (Test 1.3) testet die Weiterleitung einzelner Interrupts an mehrere *Master* (*Multi-Cast*). Phase 4 (Test 1.4) beschäftigt sich speziell mit der Priorisierung und Verarbeitung erweiterter Interrupts. Da die APB-*Slave*-Schnittstelle ausschließlich blockierende Kommunikation verwendet, ist keine Unterscheidung zwischen LT- und AT-Modus erforderlich. Die Simulationsergebnisse des TL-Modells entsprechen in allen Fällen der RTL-Referenz. Auf Grund des hohen Überhangs (*Overhead*) der Testumgebung und der relativ niedrigen Komplexität des Modelles können keine Aussagen zur Simulationsgeschwindigkeit getroffen werden. In den hier vorgestellten Tests simulierten RTL- und TL-System annähernd gleich schnell.



Test 1	Beschreibung
Test 1.1	Einzelinterrupts (1-15 normal, 16-31 erweitert) + Clearing
Test 1.2	Test der Priorisierung (2 Hoch- und 3 Niederprioritäts-IRQs)
Test 1.3	Multi-Cast Test (IRQs 1-15 (außer 4))
Test 1.4	Test der erweiterten Interrupts (Priorität und Clearing)

Abbildung 4.25: IRQMP / Test der Grundfunktionalität

Test	Beschreibung	TLM (LT)	RTL (ns)
2	Test des IRQ-Broadcastings	19580	19580
3	Test mit abgeschalteten erweiterten Interrupts	19580	19580
4	Prozesser-Interrupt-Schnittstelle (start/stop/reset)	-	-

Tabelle 4.14: IRQMP / Übersicht weitere TLM-Tests

4.3.3 Kombinierter PROM/I/O/SRAM/SDRAM Speichercontroller (MCTRL)

Übersicht

Der Quellcode des TL-Modells des GRLIB-Speichercontrollers (MCTRL) befindet sich im Unterverzeichnis */models/mctrl* der Plattform (Anhang B). MCTRL kontrolliert ein Speicheruntersystem bestehend aus bis zu vier Speichertypen: PROM, I/O, SRAM und SDRAM. Alle Speicher sind über eine AHB-*Slave*-Schnittstelle angebunden. Die Speicher selbst wurden auf rein funktionale Weise implementiert. Verzögerungszeiten und Dauer von Zugriffen werden allein im MCTRL modelliert. Die Steuerregister des MCTRL wurden mit *GreenReg* erstellt. Die Registerbank ist an einen APB-*Slave* gekoppelt. Abbildung 4.26 zeigt Aufbau und Struktur des Modelles.

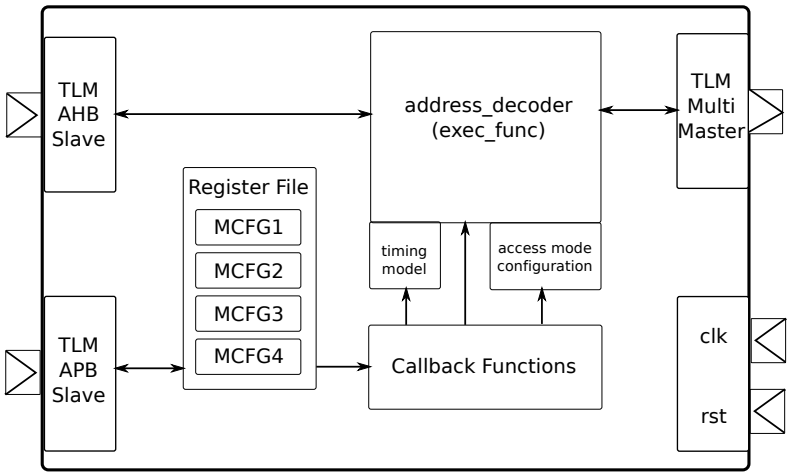


Abbildung 4.26: TL-Modell des Speichercontrollers (MCTRL)

TLM-Schnittstellen

Der MCTRL verfügt über eine AHB-*Slave*- und eine APB-*Slave*-Schnittstelle, die durch die Vererbung von Basisklassen eingebunden werden. Der AHB-*Slave* empfängt Kommandos vom System (der CPU) und leitet diese an die angeschlossenen Speicher weiter. Dies erfolgt über einen TLM-*Multi-Socket*, der bis zu vier generische Speicher (siehe Abschnitt 4.3.4) binden kann. Der APB-*Slave* dient einzig dem Zugriff auf die integrierten Steuerregister. Der *Reset*-Eingang ist als TL-Eingabeport implementiert. Ein Taktsignal existiert auf TL-Ebene nicht. Durch Vererbung von der Basisklasse *CLKDevice* kann jedoch eine Taktperiode festgelegt werden, die dann als Grundlage zur Berechnung aller Verzögerungen dient.

Steuerregister

Alle Steuerregister des MCTRL sind mit *GreenReg* modelliert. Dazu erbt das Modell von der Bibliotheksbasisklasse *gr_device*. Dies geschieht auf indirekte Weise durch Ableitung von *AHBSlave* unter Angabe von *gr_device* als *Template*-Parameter, wodurch eine Doppelvererbung von *sc_module* vermieden werden kann.

```
public AHBSlave<gs::reg::gr_device>
```

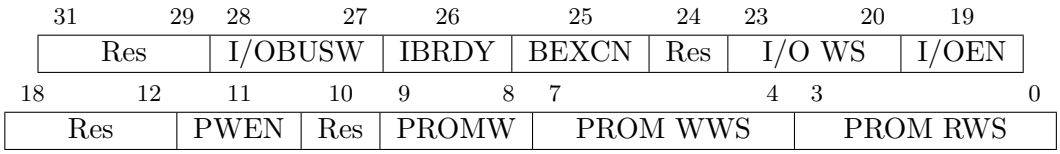
Die vier Steuerregister des MCTRL werden in den APB-Adressraum abgebildet. Die APB-Basisadresse (*paddr/pmask*-Paar) wird mit Hilfe von Konstruktorparametern eingestellt. Der *Offset* der Register ist fest und entspricht dem Vorbild des RTL-Modells (Tabelle 4.15).

APB Adresse (Offset)	Register
0x00	MCFG1 (PROM und I/O)
0x04	MCFG2 (SRAM/SDRAM)
0x08	MCFG3 (SDRAM Einstellungen)
0x0c	MCFG4 (Energiespareinstellungen (*Power*)

Tabelle 4.15: *MCTRL*-Übersicht Steuerregister

Tabelle 4.16 zeigt die Bitfelder im Steuerregister MCFG1. Alle Einstellung, mit Ausnahme von BEXCN, werden auf LT- und AT-Ebene unterstützt. Die Signalisierung von Übertragungsfehlern erfolgt mit Hilfe des *Response*-Statusfeldes der TLM-*Payload*. Die Konfiguration der Busweiten (I/OBUSW, PROMW) fließt in die Berechnung des *Timings* ein, hat aber darüber hinaus keinen funktionalen Einfluss. Das heißt, für die Verzögerungsberechnung eines Datentransfers zu einem 16-Bit-PROM wird im Vergleich zu einem 32-Bit-PROM die doppelte und im Vergleich zu einem 8-Bit-PROM die halbe Zeit angenommen. Die Bus-*Ready*-Signale IBRDY in MCFG1 und

RBRDY in MCFG2 werden implizit modelliert. Wie bereits erwähnt, wird die Verzögerung der Speicherzugriffe unabhängig von den verbundenen Speichermodellen vollständig im Controller berechnet. Sind Bus-*Ready*-Signale konfiguriert, wird die durch den Speicher zurückgegebene Transferzeit nicht wie im Normalfall ignoriert, sondern in die Berechnung der Gesamtverzögerung einbezogen. Dadurch hat der Entwickler die Möglichkeit, durch einen Speicher zusätzlich eingefügte Wartezeit einfach darzustellen.



Bit-Feld	Beschreibung	Unterstützt
I/OBUSW	I/O Bus-/Zugriffsbreite (8, 16 oder 32 Bit)	LT, AT
IBRDY	I/O Bus Ready	LT, AT
BEXCN	Bus Error Enable	nein
I/O WS	I/O Wartezyklen (0 - 15)	LT, AT
I/OEN	Ermöglicht Zugriff auf den I/O-Speicher	LT, AT
PWEN	Erlaubt Schreibzugriffe auf PROM-Speicher	LT, AT
PROMW	PROM Bus-/Zugriffsbreite (8, 16 oder 32 Bit)	LT, AT
PROM WWS/RWS	Wartezyklen für PROM Schreib-/Lesezugriff	LT, AT

Tabelle 4.16: MCFG1 - Steuerregister

Der Aufbau von MCFG2 ist in Tabelle 4.17 dargestellt. MCFG2 dient der Steuerung des SRAM- und SDRAM-Zugriffs. Einige den SDRAM betreffende Parameter wurden abstrahiert. Dies betrifft unter anderem den SDRAM-*Refresh* (SDRF, TRFC), dessen Einfluss auf die Simulationszeit als gering eingeschätzt werden kann. Durch die Abstraktion wird ein *SystemC-Thread* eingespart. Zur Modellierung des *Auto-Refresh* müsste ein solcher *Thread* einmal pro *Refresh*-Intervall aktiviert werden, eventuell offene Speicherbänke schließen und alle Zugriffe für einen Zeitraum von zwei bis drei Takten (TRP-Bereich) blockieren. Darüber hinaus wird der SDRAM-*Page Burst Modus* nicht unterstützt (SDPB). Für einen 32-Bit-Speicherbus werden alle SDRAM-Bursts als 8 Takte lang und für einen 64-Bit-Speicherbus (D64) als vier Takte lang angenommen. Alle anderen Einstellung wurden entsprechend der RTL-Vorlage übernommen. Dies umfasst die Konfiguration des Adressraumes, Wartezyklen für den SRAM-Zugriff, sowie SDRAM-Zeilen- und Spaltenaktivierungszeiten.

31	30	29 27	26	25 23	22 21	20 19	18	17	16
SDRF	TRP	SDTRFC	TCAS	SDBANKSZ	SDCOLSZ	SDCMD	D64	RES	MS

15	14	13	12 9	8	7	6	5 4	3 2	1 0
Res	SE	SI	RAM BANKSZ	Res	RBRDY	RMW	RAMW	RAM WWS	RAM RWS

Bit-Feld	Beschreibung	Unterstützt
SDRF	SDRAM Refresh	nein
TRP	Row Aktivierung – Precharge Timing	nein
TRFC	Refresh Cycle Timing	nein
TCAS	SDRAM-Zeile öffnen (Column Activation)	LT, AT
SDBANKSZ	SDRAM-Bankgröße (4 - 512MB)	LT, AT
SDCOLSZ	SDRAM Anzahl Spalten pro Zeile (256 - 4096)	LT, AT
SDCMD	Feld für Direkt-Kommandos (z.B. REFRESH)	nein
D64	64-Bit-SDRAM-Datenbus	LT, AT
SDPB	SDRAM Page Burst – Lesezugriff liefert ganze Speicherseite (ansonsten 8x Burst)	nein
SE	SDRAM Enable (5. SRAM Bank abschalten)	LT, AT
SI	SRAM Disable	LT, AT
RAM BANKSZ	Größe der RAM-Bänke (8kB - 256MB)	LT, AT
RBRDY	RAM Ready Enable	LT, AT
RMW	Read-Modify-Write Enable	LT, AT
RAMW	RAM-Busweite (8, 16 oder 32 Bit)	LT, AT
RAM WWS/RWS	RAM-Schreib-/Lesewartezyklen	LT, AT

Tabelle 4.17: MCFG2 - Steuerregister

Steuerregister MCFG3 und MCFG4 enthalten hauptsächlich Spezialeinstellungen zum Speicherschutz, wie Speicher-EDAC, *Reed-Solomon-Coding* und Kontrollbits für Testzwecke auf RT-Ebene. Eine Ausnahme bildet das PMODE-Feld in MCFG4, mit dessen Hilfe verschiedene Zugriffsmodi zur Energieeinsparung einstellbar sind.

Dekodierung der Adresse (Speicherwahl)

Die zentrale Aufgabe des MCTRL besteht in der Weiterleitung von Transaktionen von der AHB-*Slave*-Schnittstelle zu den angeschlossenen Speichern. Dazu müssen Adressen, gemäß der vorgegebenen Einstellung des Adressraumes, dekodiert werden. Der hierfür erforderliche Dekodierer ist vollständig Teil des TL-Modells und wird nicht wie im RTL-System in den AHB-Controller ausgelagert [Gai10]. Die Initialisierung des Dekodierers erfolgt dynamisch zu Beginn der Simulation mit Hilfe der *SystemC*-Funktion *start_of_simulation*. Die Funktion iteriert über alle am *Socket mem* gebundenen Speicher und extrahiert deren Konfigurationsinformation. Dazu wird die Programmierschnittstelle der Bibliotheksbasisklasse *mem_device* verwendet, die durch alle Simulationsspeicher implementiert werden muss (Abschnitt 3.3.3). Für jeden identifiziertem Speicher erzeugt das Modul ein PNP-Basisadressregister (BAR0-3), dass dann durch MCTRL exportiert werden kann. Der Adressbereich und die relative Anordnung der Speicher werden durch den MCTRL bestimmt. Der PROM-Adressbereich ergibt sich aus dem Parameterpaar *romaddr/rommask*. Der Parameter *romaddr* beinhaltet die oberen 12 Bit (MSBs) der Basisadresse. Die Maske bestimmt die Größe des Speicherbereichs ($=(2^{12} - rommask)$ MByte). PROM kann byteweise adressiert werden und hat eine Adressweite von 32 Bit. Der Speicherbereich wird zu gleichen Teilen auf zwei Bänke aufgeteilt. Für die Modellierung auf TL-Ebene ist dies unerheblich, da dadurch keine die Zugriffszeit verändernden Effekte auftreten. Der I/O-Adressbereich berechnet sich in ähnlicher Weise. Hier werden die Parameter *ioaddr* und *iomask* zugrunde gelegt. Es gibt jedoch keine Unterteilung in Speicherbänke. Der I/O-Speicher ist flach. Etwas schwieriger gestaltet sich die Konfiguration von SRAM und SDRAM. Der Adressbereich beider Speichertypen

wird durch das Parameterpaar *ramaddr/rammask* bestimmt. Die Partitionierung ist jedoch von den Einstellungen in MCFG2 abhängig. Mit Hilfe der Bitfelder SE und SI lassen sich verschiedene SRAM, SDRAM oder SRAM & SDRAM Mischkonfigurationen realisieren. Es werden bis zu fünf SRAM- und zwei SDRAM-Bänke dekodiert. Sind SDRAM und SRAM konfiguriert (SE=1, SI=0), so werden die SDRAM-Bänke in die obere Hälfte des RAM-Bereiches abgebildet, die untere Hälfe verteilt sich gleichmäßig auf vier SRAMs. Wird SRAM abgeschaltet (SI=1), so verschiebt sich der SDRAM in die untere Hälfte des RAMs. Durch das Abschalten von SDRAM (SE=0, SI=0) wird der obere RAM-Bereich auf eine fünfte SRAM-Bank abgebildet. Grafik 4.27 verdeutlicht einige der möglichen Einstellungen. Der Adressdekodierer des TL-Modelles überprüft die RAM-Einstellungen bei jedem Aufruf und passt sich entsprechend an. Dadurch kann der Speicher, ähnlich wie im RT-Modell, zur Laufzeit rekonfiguriert werden.

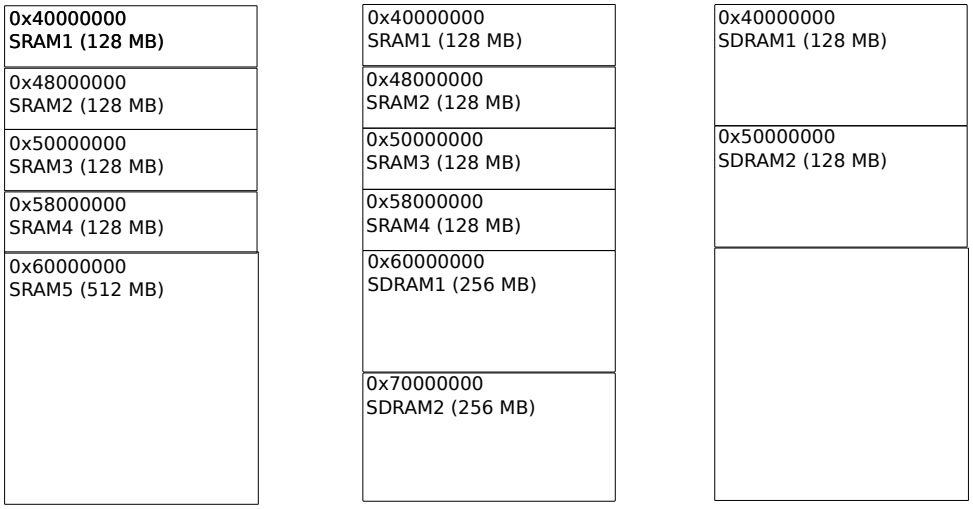


Abbildung 4.27: Partitionierung des Adressraumes am MCTRL

Das Verhalten des Modells ist größtenteils in die Rückruffunktion *exec_func* der AHB-Busschnittstelle eingebettet und weist keine Unterschiede bezüglich der Abstraktionsstufe auf. Im LT-Modus wird *exec_func* unmittelbar aus der blockierenden Transportfunktion *b_transport* aufgerufen. Die AT-Busschnittstelle ruft *exec_func* nach *BEGIN_REQ* auf. Dem Verhaltensteil der Komponente werden das TLM-*Payload*-Objekt und ein Verzögerungszeiger übergeben. Letzterer dient der Rückgabe der für den Transfer geschätzten Wartezyklen. Die Länge der Datenphase berechnet die Busschnittstelle selbstständig auf Grundlage der Burstweite und -länge. Nach Übernahme des *Payload*-Objektes ruft *exec_func* den Adressdekodierer auf. Dieser ist in Funktion *get_ports* implementiert und liefert bei erfolgreicher Zuordnung ein Objekt vom Typ *MEMPort* zurück. Wenn für den Zugriff kein Speicher gefunden werden kann, generiert MCTRL einen Fehler vom Typ tlm::TLM_ADDRESS_ERROR_RESPONSE. Dies führt zum Abbruch der Transaktion. Anderenfalls wird die Transaktion auf ihre Zugriffseigenschaften geprüft. Dazu sind abhängig vom adressierten Speicher mehrere Tests erforderlich. Es wird zum Beispiel kontrolliert, ob die geforderte Zugriffsweite mit dem Speicher kompatibel ist, ob der Speicherbereich beschreibbar ist (PROM) oder ob zusätzliche RMW-Zyklen eingefügt werden müssen. Ist eine der Bedingungen nicht erfüllt, so erzeugt das Modul eine Fehlermeldung (tlm::TLM_GENERIC_ERROR_RESPONSE) und bricht die Transaktion ab.

SDRAM-Zugriffsmodi

Der MCTRL unterstützt verschiedene Energiesparmodi für den Zugriff auf den SDRAM, die über das PMODE-Feld des Registers MCFG4 angewählt werden können:

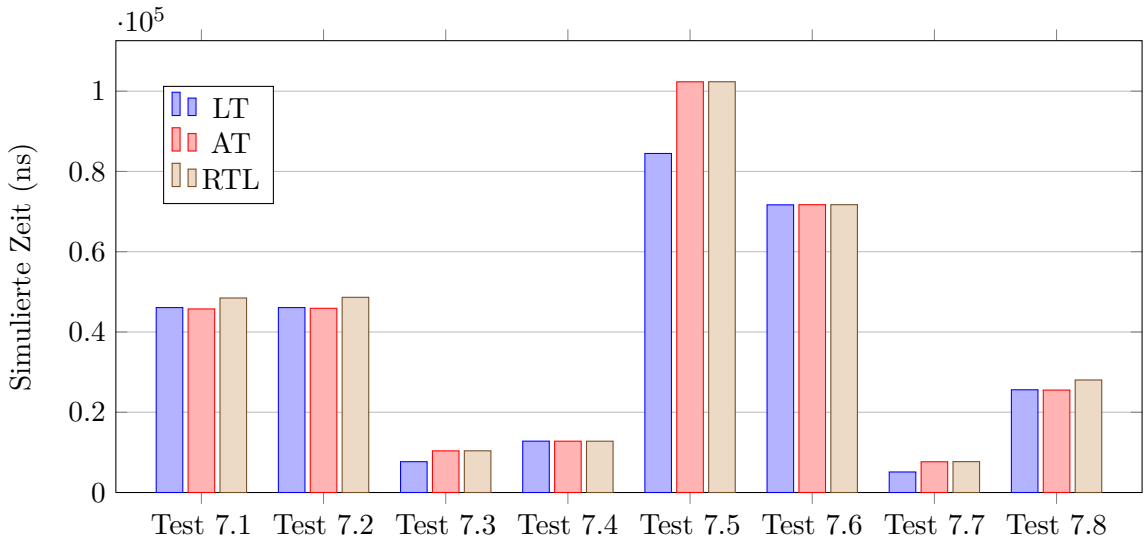
- *Power Down*

- *Partial Array Self-Refresh*
- *Self-Refresh*
- *Deep Power Down*

Im TL-Modell wird eine Änderung des Zugriffsmodus durch eine dynamische Anpassung der Wartezyklen oder zusätzliche Zugriffskontrollen dargestellt. Beim Wechsel in den *Power Down*-Modus werden die Eingabe- und Ausgabepuffer des SDRAMs nach 16 Leerlaufzyklen abgeschaltet. Das Wiedereinschalten verursacht eine Verzögerung von einem Takt, der auf die Wartezyklen der Transaktion aufgeschlagen wird. Ein Speicher wird in den *Self Refresh Mode* versetzt, um gespeicherte Daten über die Abschaltung des Systems hinaus zu erhalten. Daher sind im *Self Refresh* keine Zugriffe zu erwarten. Der MCTRL generiert eine Warnung, da diese aber theoretisch nicht ausgeschlossen sind. *Partial Array Self Refresh* kann im *Power Down*-Modus durch Beschreiben des PASR-Feldes in MCFG4 zusätzlich aktiviert werden. Dadurch wird ein Teil des SDRAMs abgeschaltet und unmittelbar gelöscht. Das TL-Modell realisiert dieses Verhalten mit Hilfe einer *Payload*-Erweiterung, die an den generischen Speicher (GENMEM) weitergeleitet wird. Der Speicher löscht den betreffenden Bereich daraufhin selbstständig. Im *Deep Power Down*-Modus wird der SDRAM komplett abgeschaltet und verliert dadurch seinen Inhalt. Alle Zugriffe verursachen eine Fehlermeldung (tlm::TLM_ADDRESS_ERROR_RESPONSE).

Verifikation und Leistungsfähigkeit

Der MCTRL wurde mit Hilfe zweier unterschiedlich ausgebauter Testumgebungen verifiziert. Die Testergebnisse sind in Abbildung 4.28 und Tabelle 4.18 zusammengefasst. Für Tests 1-4 wurde ein alleinstehender Stimulusgenerator verwendet, der sich direkt mit dem AHB-*Slave* und dem APB-*Slave* des MCTRL verbindet. Tests 5-8 integrieren den AHB-Bus (AHBCTRL) und die AHB/APB-Busbrücke (APBCTRL), wodurch viel der zum Test der Verbindungsstrukturen entwickelten Infrastruktur wiederverwendet werden konnte. Dies betrifft unter anderem das Transaktionsmanagement (*Payload Pools*), den automatischen Abgleich von Simulationsergebnissen und die Generierung von Zufallstransaktionen für speziell definierte Speicherbereiche. Durch die Integration des Busmodells wird ebenfalls die Co-Simulation des RTL-Referenzmodelles erleichtert, da so die Transaktoren zum Anschluss eines RTL-*Slaves* an einen TL-Bus wiederverwendet werden können. Zur Durchführung der Referenzsimulationen wurden die PROM-, I/O und SRAM-Schnittstellen des MCTRL an ein generisches SRAM-Modell mit parametrisierbarer Anzahl an Wartezyklen gebunden. Zur Simulation von SDRAM wurde ein *Micron*-Simulationsspeicher aus der GRLIB verwendet (*Micron mt481c16m16a2*). Abbildung 4.28 verdeutlicht die Simulationsgenauigkeit für unterschiedliche Standardoperationen. Die Genauigkeit beider TL-Modi ist sehr hoch. Das AT-Modell erreicht im Durchschnitt 95% und das LT-Modell 92%. Details der Testabfolge können dem Quellcode (siehe Anhang B) und [Sch12d] entnommen werden. Die Beeinflussung der Testergebnisse durch das in die Testumgebung integrierte AMBA-Bussystem wird als gering eingeschätzt, da für den AHBCTRL in Konfiguration mit einem *Master* für sowohl LT- als auch AT-Modus eine Genauigkeit von nahe 100% nachgewiesen werden konnte. Genaue Aussagen zur Simulationsgeschwindigkeit lassen sich auf Grund des *Overheads* der Testumgebung nicht treffen. Die hier vorgestellten Tests simulierten auf LT-Ebene 50 mal schneller und auf AT-Ebene 30 mal schneller als das RTL-Modell.



Test 7	Beschreibung
7.1	PROM Schreiben (256x)
7.2	PROM Lesen (256x)
7.3	I/O Schreiben (256x)
7.4	I/O Lesen (256x)
7.5	SRAM Schreiben mit RMW (256x Word, 512x Half, 1024x Byte)
7.6	SRAM Lesen mit RMW (256x Word, 512x Half, 1024x Byte)
7.7	SDRAM Schreiben (256x)
7.8	SDRAM Lesen (256x)

Abbildung 4.28: MCTRL / Test mit 32-Bit PROM, I/O, SRAM und SDRAM

Test	Beschreibung	LT (ns)	AT (ns)	RTL (ns)
1, 2, 3, 4	Testen der Grundfunktionalität in mit verschiedenen Zugriffsmodi:			
	- Lesen und Schreiben mit 32-Bit-Speicherweite, PROM Write-Enable, kein RMW	84380	64000	- -
	- Lesen und Schreiben mit 16-Bit-Speicherweite und 64-Bit-SDRAM-Bus	132950	64000	-
	- Lesen und Schreiben mit 8-Bit Speicherbreite	220720	48000	-
	- 32-Bit SRAM Read-Modify-Write (RMW)	18420	16000	-
	- 16bit SRAM Read-Modify-Write (RMW)	20640	16000	-
	- 8-Bit SRAM Read-Modify-Write (RMW)	33920	16000	-
	- PROM read-only und I/O abgeschaltet	8690	8000	-
	- Test des SDRAM Self-Refresh-Modus	17530	16000	-
	- Test des SDRAM-Power-Down-Modus	17220	16000	-
	- Test des SDRAM-Deep-Power-Down-Modus	16000	16000	-
5	Test der Konfiguration mit 8-Bit-PROM, 32-Bit-SDRAM (kein SRAM):			
	- PROM-Bereich lesen und schreiben	1788440	1802750	1824600
	- SDRAM lesen und schreiben	215040	232630	251510
6	Test der Konfiguration mit 32-Bit-PROM, 32-Bit-SRAM (kein SDRAM):			
	- PROM lesen und schreiben	92160	91540	97007
	- SRAM lesen und schreiben (RMW)	309760	327570	327600
8	Test mit 8-Bit-PROM und 32-Bit-Mobile-SDRAM:			
	- PROM lesen und schreiben	1790320	1790320	1825480
	- SDRAM lesen und schreiben (Subword)	215160	215160	251630
	- SDRAM-Zugriff im Power-Down Modus	250900	250900	-

Tabelle 4.18: AHB-Bus / Übersicht weitere TLM-Tests

4.3.4 Generischer Speicher (GENMEM)

Der generische Speicher (GENMEM) dient der funktionalen Modellierung der durch den Speichercontroller (MCTRL) unterstützten Speichertypen. Das Modell verfügt über einen TLM-*Slave-Socket*, der mit dem *Multi-Socket* des MCTRL (Abb. 4.26) verbunden werden kann. GENMEM erbt die *SoCRocket*-Basisklasse *mem_device* und erhält dadurch Attribute anhand deren es als SRAM, SDRAM, I/O oder PROM identifiziert werden kann (siehe Abschnitt 3.3.3). Unabhängig von den Einstellungen für die Anzahl, Größe und Weite der Speicherbänke stellt GENMEM aus funktionaler Sicht einen flachen Speicher dar. Der Speichertyp und dessen Eigenschaften sind nur für den MCTRL relevant und werden dort zur Abschätzung der Zugriffszeit berücksichtigt.

Zur Modellierung von Energiesparfunktionen oder Systemneustarts ist es gegebenenfalls erforderlich, flüchtige Speicher (z.B. SDRAM) ganz oder teilweise zu löschen. GENMEM wertet dazu die ignorierbare *Payload*-Erweiterung *erase_ext* aus. MCTRL setzt die Erweiterung zur Modellierung des *Deep Power Down*-Modus und des *Partial Error Self Refresh*-Modus ein.

4.3.5 On-Chip SRAM (AHBMEM)

AHBMEM ist ein Simulationsmodell der Komponente AHBRAM aus der GRLIB und stellt einen einfachen SRAM-Speicher mit einer AHB-*Slave*-Schnittstelle dar. Das Modell erbt von der Basisklasse *AHBSlave*. Lese- und Schreibzugriffe werden direkt in der Rückruffunktion (*exec_func*) der Busschnittstelle ausgeführt. Der Quellcode des Modells befindet sich im Verzeichnis *models/ahbmem* der Plattform. AHBMEM verfügt über keine Steuerregister. Speichergröße und

AHB-Adressbereich (*haddr/hmask*) werden mit Hilfe von Konfigurationsparametern eingestellt. Der eigentlich Speicher ist als C++-*std::map* implementiert und eignet sich damit auch zur Abbildung großer, dünn besetzter Speicherbereiche. AHBMEM erbt die Basisklasse *CLKDevice*, verfügt aber über keinen *Reset*-Eingang.

4.3.6 UART (APBUART)

Übersicht

Das *SystemC*-Modell des APBUART erzeugt eine UART-Schnittstelle, die auf einen TCP-Port des *Host*-Systems abgebildet wird. Dadurch wird eine flexible Weiterverarbeitung der Ein- und Ausgabedaten außerhalb der Simulationsumgebung ermöglicht. Die Struktur des Modells ist in Abbildung 4.29 dargestellt. Der Quellcode des APBUART befindet sich im Verzeichnis */models/apbuart* der VP. Zur Implementierung des *tcpio*-Treibers wird die Bibliothek *Boost.Asio* verwendet. Diese übernimmt ebenfalls die Modellierung der Ein- und Ausgabepuffer, die anders als im RT-Modell als unendlich groß angenommen werden. Da die so modellierte serielle Schnittstelle nur virtuell existiert, kann ebenfalls auf die Modellierung der Baudrate verzichtet werden.

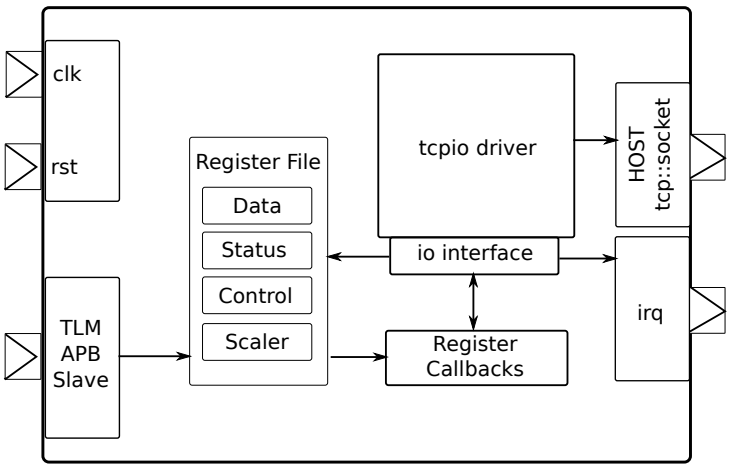


Abbildung 4.29: TL-Modell der UART-Schnittstelle (APBUART)

TLM-Schnittstellen

Die Komponente ist mit dem System über eine APB-*Slave*-Schnittstelle verbunden. Dazu erbt APBUART die Basisklasse *APBSlave*. Wie fast alle anderen Modelle in *SoCRocket* wird APBUART darüber hinaus von der Basisklasse *CLKDevice* abgeleitet und erhält dadurch eine Schnittstelle zur Annotation des Zeitverhaltens. Der *Reset*-Eingang (*rst*) und der Interruptausgang (*irq*) sind als TL-Signalports implementiert.

Steuerregister

Die UART-Schnittstelle wird durch eine Registerbank gesteuert, die über den APB-Bus beschrieben und gelesen werden kann. Eine Übersicht der Steuerregister befindet sich in Tabelle 4.19.

APB Adresse (Offset)	Register
0x00	UART Data Register
0x04	UART Status Register
0x08	UART Control Register
0x0c	UART Scaler Register

Tabelle 4.19: *APBUART*-Übersicht Steuerregister

Das Modell dient hauptsächlich der Ankopplung an das *Host*-System und damit dem Austausch von Ein- und Ausgabedaten. Daher werden verschiedene das Zeitverhalten und Fehlerkorrekturmechanismen betreffende Einstellungen abstrahiert. Das *Scaler*-Register ist auf TL-Ebene funktionslos, kann aber über den APB-Bus beschrieben und gelesen werden. Das gleiche gilt für viele Felder des *Control*-Registers und des *Status*-Registers. Im *Control*-Register werden lediglich Bits 0-3 zum Ein- und Ausschalten von Sender und Empfänger, sowie der damit verbundenen Sende- und Empfangsinterrupts unterstützt. Das *Status*-Register signalisiert das Anliegen von Daten in Bit Null (DR-Data Ready). Die Anzahl der im Empfänger gepufferten Bytes kann dem *Receiver FIFO Count*-Feld (Bits 31:26) entnommen werden. Dadurch wird die Datenabfrage durch *Polling* ermöglicht.

Ansteuerung der Ein-/Ausgabefunktionen

Durch die Abstraktion der Sende- und Empfangsebene kann das APBUART-Modell sehr einfach und flexibel eingesetzt werden. Der Sender ist immer bereit. Es entstehen keine Stall- oder Wartezyklen. Ausgabedaten werden über die APB-Schnittstelle byteweise ins *Data*-Register übernommen. Von dort werden sie mittels einer Register-Rückruffunktion an den *tcpio*-Treiber weitergegeben. Der Sendeinterrupt wird unmittelbar ausgelöst. Das Vorhandensein von Lesedaten wird dem Nutzer mit Hilfe des *Status*-Registers oder durch einen Empfängerinterrupt mitgeteilt. Die *Boost.Asio*-Bibliothek stellt einen TCP-*Socket* zur Anbindung eines *Host*-Terminals bereit. Die Verbindung wird mittels der Funktion *makeConnection* hergestellt. Diese wird zu Beginn der Simulation blockierend gestartet und gibt die Nummer des TCP-Ports aus:

```
apbuart: UART waiting for connection on port: <PORT NUMBER>

telnet localhost <PORT NUMBER>
```

5 Systementwurf mit SoCRocket

In den vorangegangenen Kapiteln wurde eine Methodik zur Entwicklung leistungsfähiger standardoffener Simulationsmodelle eingeführt (Abschnitt 3). Die abgeleiteten Ansätze wurden zum Aufbau einer flexibel erweiterbaren *SystemC*/TLM-Modellbibliothek für den Luft- und Raumfahrtbereich eingesetzt. Die dazu verwendeten Techniken wurden in Abschnitt 4 anhand konkreter Beispiele verdeutlicht. Im vorliegenden Kapitel wird beschrieben, wie die Modellbibliothek mit Hilfe der *SoCRocket*-Infrastruktur zur Konstruktion von Virtuellen Prototypen eingesetzt werden kann [Sch14]. Darüber hinaus wird der konkrete Einsatz des Systems zur Realisierung mehrstufigen Architekturexploration und zur Entwicklung hardwarenaher Software anschaulich beschrieben.

5.1 Entwurfsfluss zur Konstruktion Virtueller Prototypen

Der *SoCRocket*-Entwurfsfluss (*Design Flow*) (Abb. 5.1) baut auf der in Kapitel 2.4 beschriebenen allgemeinen ESL-Methodik auf und stellt eine Minimalinfrastruktur zur effizienten Konstruktion von Virtuellen Prototypen bereit. Die Besonderheit besteht in der technischen Umsetzung zur Vereinfachung und Beschleunigung der Erkundung des Entwurfsraumes, sowie der Entwicklung hardwarenaher Software. Für den Explorationsfall ist es erforderlich Simulations- und Analyseergebnisse automatisch erfassen und Parameter auf einfache, nicht-intrusive Weise variieren zu können. Außerdem sollten sich Hardware und Software selbstständig aufeinander anpassen, da zum Beispiel die Änderung des Speicheraufbaus Änderungen des Programmeintrittspunktes oder der Position von Programmsegmenten (z.B. *text*, *data*) im Hauptspeicher nach sich zieht. Zur Entwicklung von hardwarenaher Software muss es möglich sein, komplexe Interaktionen zwischen Komponenten zur Analyse nachzuvollziehen.

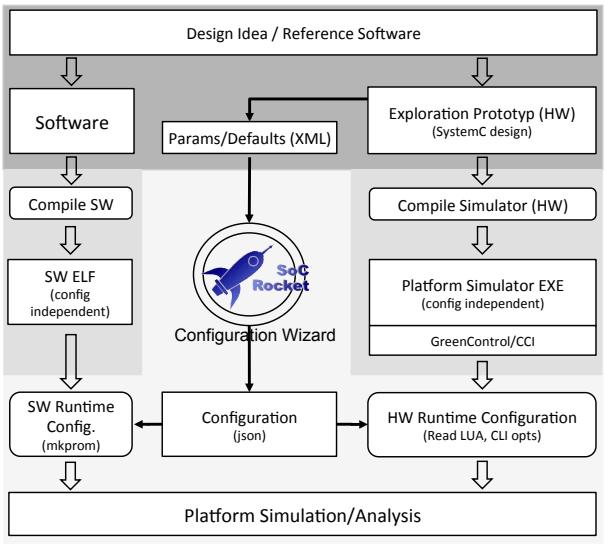


Abbildung 5.1: SoCRocket-Entwurfsfluss zur Systemkonstruktion

5.1.1 Entwurfseintritt/Partitionierung

Die in diesem Kapitel beschriebene Methodik umfasst den Weg von einer Referenzsoftware zu einem Virtuellen Prototypen und orientiert sich damit von einer höheren zu einer niedrigeren

Abstraktion (*Top-Down*). Ein anfänglicher Partitionierungsschritt entscheidet, welche Teile der Funktionalität in Software implementiert werden und wie die initiale Hardware zur Ausführung dieser Software aussehen könnte. Dieser Schritt ist in *SoCRocket* manuell und kann auf Grund der relativ eng gefassten Architekturtemplates und beschränkten Auswahl an Standardkomponenten im Raumfahrtbereich (Abschnitt 1.3) *noch* der Erfahrung des Entwicklers überlassen werden. Zur Unterstützung der Entscheidungsfindung werden in der Praxis häufig *Profiling*-Werkzeuge wie *GNU gprof* eingesetzt [bin14], mit denen besonders aufwendige Anwendungsteile identifiziert werden können. Die entsprechenden Funktionen werden dann als Kandidaten zur Implementierung in einem Beschleunigermodul markiert und können mit Hilfe der in Kapitel 3 beschriebenen Methodik in die Modellbibliothek integriert werden. Ein Beispiel zur Überführung von Funktionen aus einer Referenzsoftware in Explorationskomponenten kann Abschnitt 5.3 entnommen werden. Eine ausführliche Anleitung wurde darüber hinaus zur Arbeit mit Studenten im Rahmen der Veranstaltung *VLSI-Design II* an der TU Braunschweig entworfen [Sch13a]. Ziel des Experimentes ist die Beschleunigung einer JPEG-Bildkompression durch Verschiebung der DCT-Transformation und weiterer Softwareteile in Hardwarebeschleuniger.

Ergebnisse des Partitionierungschritts sind eine initiale Software und ein Explorationsprototyp (EP), der Bibliothekskomponenten und potentielle Beschleunigungsfunktionen in Abhängigkeit von Explorationsparametern instantiiert. Der EP kann als *SystemC*-Klasse oder als *SystemC*-Programmfunktion (*sc_main*) modelliert werden. Komplexität und Struktur des EP sind variable und abhängig vom Einsatzzweck. So können einfache EPs zum Erstellen von *Unit*-Tests verwendet werden. Das System besteht in diesem Fall nur aus der zu testenden Komponente und einem Testmodul. Ein weiterer einfacher Anwendungsfall wäre ein schneller Simulator zur Unterstützung der Softwareentwicklung, bestehend aus der LEON2/3-Integereinheit (Abschnitt 4.1) und direkt verbundenen Simulationsspeichern. Die Komplexität kann beliebig bis hin zu Multiprozessorsystemen mit variabler Anzahl an CPUs erweitert werden. Ein Beispiel für ein solches System wird in Abschnitt 6.1 erläutert (*leon3mp*). Darüber hinaus können mit Hilfe von Netzwerkroutern Simulatoren für massiv parallele Systeme konstruiert werden. *SoCRocket* integriert dazu ein *SpaceWire*-TLM der Europäischen Raumfahrtagentur [ESA14] und eine *SoCWire*-Busbrücke. Weitere Komponenten einer modernen Hochleistungs-*NoC*-Architektur befinden sich zurzeit in der Entwicklung (Abschnitt 7.2).

Ein *SoCRocket*-EP unterscheidet sich von einem normalen *SystemC*-Programm durch die bedingte Instantiierung und Verbindung von Komponenten. Die dafür erforderlichen Parameter werden durch eine standardoffene Konfigurations-*Middleware* verwaltet (*GreenControl* – Abschnitt 3.5.2) und sind in einem globalen Namensraum organisiert. Die Parameter können durch externe Werkzeuge zur Laufzeit gelesen und beschrieben werden. Dadurch wird eine dynamische Rekonfiguration des Systems ermöglicht. Abbildung 5.2 verdeutlicht das Konzept der bedingten Instantiierung von Komponenten an einem Beispiel.

```
1  ...
2  // Parameterbaum aufbauen
3  gs::gs_param_array p_conf("conf");
4  gs::gs_param_array p_system("system", pconf);
5  gs::gs_param_array p_bus("bus", pconf);
6
7  // Systemparameter: Anzahl der CPUs
8  gs::gs_param<unsigned int> p_ncpu("ncpu", 1, p_system);
9
10 // Modellparameter: Bus-Modus (z.B. Arbitrierung)
11 gs::gs_param<bool> p_bus_mode("bus_mode", 1, p_bus);
12
13 // Businstanz mit Modellparameter
14 bus my_bus(p_bus_mode);
15
16 // Prozessoren erzeugen
17 for (int i=0; i<ncpu; i++) {
18    processors[i] = new cpu(i);
19    // Prozessoren am Bus anschliessen
20    processor[i].out(my_bus.in);
21 }
22 ...
```

Abbildung 5.2: Bedingte Instanziierung von Komponenten im Explorationsprototyp

In den Zeilen 3 - 11 wird ein Parameterbaum aufgebaut. Die Wurzel des Baumes ist das Parameterfeld *p_conf*. Diesem werden Parametercontainer zur Aufnahme von Systemparametern (*p_system*) und Modellparametern zugeordnet (Bsp. *p_bus*). Die Systemparameter wirken sich auf die Konstruktion des Gesamtsystems aus. So werden in Abhängigkeit des Parameters *processors* ein oder mehrere Prozessoren instantiiert (Zeile 18) und an den Bus verbunden (Zeile 20). Modellparameter bestimmen das Verhalten einer einzelnen Komponente und werden entweder direkt implementiert oder, wie im Beispiel gezeigt, auf Konstruktorparameter abgebildet (Zeile 14). Die letztere Variante eignet sich ebenfalls zur Integration von Modellen von Drittanbietern, da diese oft nicht in Form von Quellcode zur Verfügung stehen.

Zur Schnittstellenbeschreibung für externe Werkzeuge werden die Konfigurationsparameter in eine XML-Beschreibung exportiert. Die so entstehenden *Template*-Dateien bestehen aus vier Sektionen. Ein vereinfachtes Beispiel ist in Abbildung 5.3 dargestellt. Die ersten zwei Sektionen sind optional. Sektion 1 (*description*) enthält eine allgemeine textuelle Beschreibung des Systems, die zur Information des Nutzer am Bildschirm angezeigt werden kann. Sektion 2 (*instructions*) kapselt systemspezifische Instruktionen und Anweisungen. Hier können zum Beispiel Ladeoptionen für Binärdateien, Informationen zu Simulationsausführung und -ablauf oder Beschreibungen von Analyseparametern untergebracht werden. Die dritte Sektion (*system*) dient der Beschreibung der Systemparameter. Wie bereits erwähnt wirken sich Systemparameter im Gegensatz zu Modellparametern auf das gesamte System aus. Typische Systemparameter sind neben der Anzahl der verwendeten Prozessoren, der Systemtakt (*Clock*) und das Abstraktionsniveau. Darüber hinaus könnte die Anzahl und die Verschaltung von Bussegmenten oder im Falle eines *Network-on-Chips* die Topologie und die Anzahl der Knoten beschrieben werden. Die vierte und letzte Sektion der *Template*-Datei beschreibt Modelle und Modellparameter.

```
1  <template name="Beispielsystem">
2  <description>                                          -- Sektion 1
3    Ein Beispielsystem
4  </description>
5  <instructions>                                         -- Sektion 2
6    <h1> Simulation wie folgt starten </h1>
7    <h1> Binaerdateien wie folgt laden  </h1>
8    ...
9  </instructions>
10 <option var="conf" name="Systemkonfiguration">
11   <option var="system" name="Systemparameter">        -- Sektion 3
12     <option var     = "ncpu" \
13             name    = "Anzahl der Prozessoren" \
14             type    = "int" \
15             default = "1" \
16             range   = "1-16" \
17             hint    = "keine" "/>
18             ...
19   </option>
20   <option var="bus" name="Ein Busmodell">             -- Sektion 4
21     <option var     = "rrobin" \
22             name    = "Round-robin Arbitrierung" \
23             type    = "bool" \
24             default = "false" "/>
25             ...
26   </option>
27 </option>
```

Abbildung 5.3: SoCRocket - XML-Parameterbeschreibung

Es ist zu erkennen, dass die Sektionen 3 und 4 den zuvor beschriebenen Parameterbaum hierarchisch nachbilden. Entsprechend der gegebenen Verschachtelung kann die Anzahl der Prozessoren mit Hilfe des Pfades: *conf/system/ncpu* beschrieben werden. Der Arbitrierungsparameter des Busses befindet sich an der Adresse *conf/bus/rrobin*. Für jeden System- oder Modellparameter muss neben der Parameterbezeichnung (*var*) eine Kurzbezeichnung (*name*) definiert werden. Weitere Pflichtfelder sind der Datentyp, die Standardeinstellung (*default*) und der Wertebereich (Zeilen 12-17). Darüber hinaus können beliebige weitere Felder hinzugefügt werden.

5.1.2 Systemkonfiguration

Hardware (EP) und Software können nun zunächst parameterunabhängig kompiliert werden. Wie in Abbildung 5.1 dargestellt, erfolgt die Konfiguration zu Beginn der Simulation. Die System- und Modellparameter des EP werden dazu mit Hilfe der *GreenControl-Middleware* initialisiert. Die entsprechenden Werte werden aus einer Konfigurationsdatei in JSON-Sprache eingelesen oder der Simulation als Kommandozeilenparameter übergeben. Alle Parameter die nicht durch einen dieser Mechanismen gesetzt werden, behalten die im EP spezifizierte Standardeinstellung. Die besondere Eignung von JSON zur Darstellung von Systemkonfigurationen wurde bereits in Abschnitt 3.5.2 beschrieben. Die einfache Schlüssel/Wert-Darstellung ist für Menschen gut lesbar und kann durch Maschinen ebenso leicht verarbeitet werden. Für einfache Systeme mit wenigen Konfigurationsparametern, wie zum Beispiel *Unit*-Tests, können somit Konfigurationen schnell von Hand geschrieben werden. Verschiedene Beispiele befinden sich im Verzeichnis *templates* des Systems (Anhang B). Für komplexere Systeme wie den in Abschnitt 6.1 beschriebenen LEON3MP ist Werkzeugunterstützung erforderlich. *SoCRocket* stellt dafür den *Configuration Wizard* (CW) bereit. Dieser verwendet die aus dem EP extrahierte XML-Parameterbeschreibung als Eingabe und bietet dem Nutzer eine einfache graphische Oberfläche zum Editieren, Laden und Speichern von Konfiguration (JSON-Dateien) an.

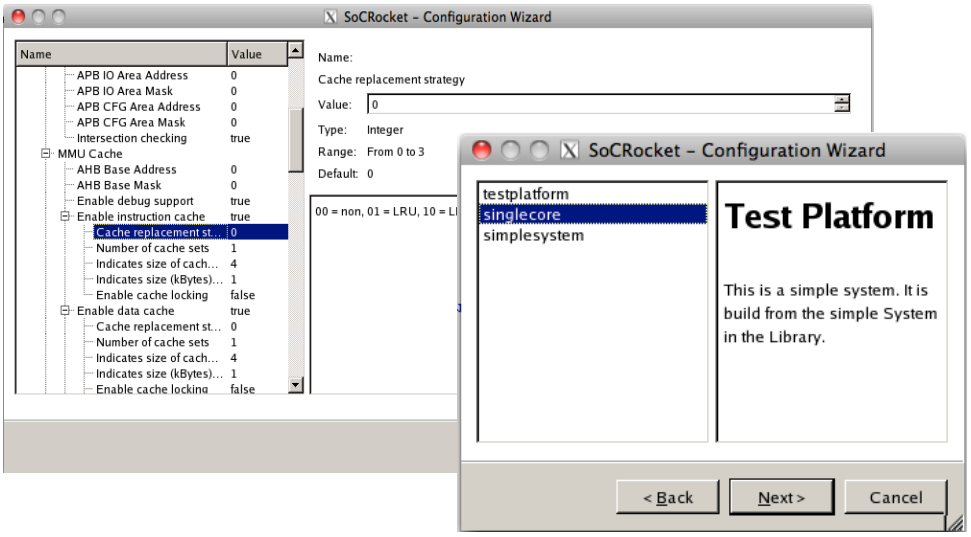


Abbildung 5.4: SoCRocket - *Configuration Wizard*

Die im *Configuration Wizard* vorgenommenen Einstellungen überführen den EP in einen Virtuellen Prototypen. Darüber hinaus wirken sich die Einstellungen abhängig von Anwendungszweck und Zielsystem auf die Software aus. Zum Beispiel kann in einem Einprozessorsystem der Programmeintrittspunkt automatisch an die im Speichercontroller eingestellte SDRAM-Startadresse verschoben werden. In Mehrprozessorsystemen wie dem in Abschnitt 6.1 beschriebenen LEON3MP muss darauf geachtet werden, dass sich die Software-*Images* der einzelnen CPUs im Speicher nicht überlagern. Dafür sind verschiedene Ansätze denkbar. So kann die Position von Programmsegmenten (z.B. *text*, *data*) für jede CPU mit Hilfe von Explorationsparametern explizit festgelegt werden. Ein weiterer gängiger Ansatz ist die Zerteilung des Speichers in Segmente gleicher Größe. Die Zuordnung von CPUs und Software-*Images* erfolgt in diesem Fall durch Multiplikation des *Offsets* (Segmentgröße) mit der Prozessor-ID. In allen beschriebenen Fällen steuern die Explorationsparameter das *Linken* des Codes und damit die Erzeugung eines oder mehrerer ausführbarer Programme. Je nach Umfang der erforderlichen Einstellungen können die Parameter dem *Linker* direkt oder mit Hilfe eines *Linker*-Kontrollskripts übergeben werden. Der *Linker* in der GNU *Compiler Collection* (GCC) [GNU14] stellt zur Markierung des Beginns von Text- bzw. Datensegment die Kommandozeilenoptionen *-Ttext* und *-Tdata* bereit. Im folgenden Beispiel werden mit Hilfe des *Sparc-Cross-Linkers* der Beginn des Programms *hello* auf die Adresse 0x40000000 und der Beginn der Daten auf die Adresse 0x80000000 verschoben:

```
sparc-elf-gcc -Ttext=0x40000000 -Tdata=0x80000000 hello.o -o hello.sparc
```

Linker-Kontrollskripts erlauben weit umfangreichere Einstellungen. Ein Beispiel für ein einfaches Kontrollskript ist in Abbildung 5.5 dargestellt.

```
1 SECTIONS
2 {
3         . = 0x40000000;
4         .text : { *(.text)}
5         . = 0x80000000;
6         .data : { *(.data)}
7         ...
8 }
```

Abbildung 5.5: GCC-Linker / Beschreibung des Speicher-Layouts

Das Kontrollskript beschreibt auf einfache Weise die gewünschte Speicherstruktur. Zeilen drei und vier bewirken, dass alle Programmsegmente (*Text*) der zu linkenden Objektdateien

hinter Adresse 0x40000000 angeordnet werden. Gleiches gilt für die Datensegmente und Adresse 0x80000000 (Zeilen 5-6). Die Übergabe der Explorationsparameter an den *Linker* oder das *Linker*-Kontrollskript erfolgt in *SoCRocket* mit Hilfe von Systemvariablen die durch das *Build*-System aus den Konfigurationsdateien (JSON) initialisiert werden. Die so erzeugten Programme können zum Simulationsbeginn direkt gestartet oder zuvor in einem ROM verpackt werden. Dazu können proprietäre Werkzeuge wie das von *Aeroflex Gaisler* angebotene *mkprom2* [gai13] verwendet werden. Die Erzeugung eines ROMs erlaubt die detaillierte Simulation des *Boot*-Prozesses. Mit Hilfe der integrierten Initialisierungsroutinen dekomprimiert der Prozessor die Daten und kopiert alle Programmsegmente an die vorgesehenen Speicheradressen.

5.1.3 Simulation und Analyse

Nach der Definition und Kompilierung des EP, der Erzeugung ein oder mehrerer Systemkonfigurationen und der Kompilierung der auszuführenden Software ist das System zur Simulation bereit. Die Simulation selbst stellt ein ausführbares Programm dar und wird vom *Build*-System im Unterverzeichnis *build/platforms* abgelegt (siehe Anhang B). Abhängig vom zu simulierenden System müssen unterschiedliche Parameter übergeben werden. Im Falle des in Abschnitt 6.1 beschriebenen LEON3MP sind dies Name und Position der Software-*Images* zur Initialisierung von RAM und ROM. Zusätzlich können eine JSON-Konfigurationsdatei und Initialwerte zum Überschreiben einer beliebigen Anzahl von Explorationsparametern übergeben werden:

```
./leon3mp.platform   --jsonconfig test.json
                     --option conf.mctrl.prom.elf=default.prom
                     --option conf.mctrl.ram.sdram.elf=hello.sparc
                     --option conf.ahbctrl.rrobin=true
```

Das Beispiel verwendet die Konfigurationsdatei *test.json*. Das ROM wird mit der Binärdatei *default.prom* und der RAM mit *hello.sparc* initialisiert. Abschließend wird der AHB-Bus für *Round-Robin*-Arbitrierung konfiguriert. Zusätzliche Kommandozeilenparameter können durch den Befehlen *–help* eingeblendet werden. So ist es zum Beispiel möglich, mit Hilfe der Option *-a* Aufrufparameter an ein auf dem Prozessorsimulator laufendes Programm zu übermitteln.

Unabhängig vom Explorationsprototypen werden zu Beginn der Simulation für alle Modellinstanzen Konfigurationsberichte ausgedruckt (*stdout*). Dafür muss das System mit *Verbosity-Level report* oder höher konfiguriert werden (Abschnitt 3.7.5). Die Konfigurationsberichte werden in den Konstruktoren der Modelle erzeugt und geben dem Nutzer eine visuelle Rückmeldung über die vorgenommenen Einstellungen. Abbildung 5.6 zeigt einen Konfigurationsbericht für den AHB-Bus im LEON3MP.

```
1  ****************************************************************
2  @0 s /0 (ahbctrl): Info:  * Created AHBCTRL with following parameters:
3  @0 s /0 (ahbctrl): Info:  * ioaddr/iomask: fff/fff
4  @0 s /0 (ahbctrl): Info:  * cfgaddr/cfmask: ff0/ff0
5  @0 s /0 (ahbctrl): Info:  * rrobin: 0
6  @0 s /0 (ahbctrl): Info:  * split: 0
7  @0 s /0 (ahbctrl): Info:  * defmast: 0
8  @0 s /0 (ahbctrl): Info:  * ioen: 0
9  @0 s /0 (ahbctrl): Info:  * fixbrst: 0
10 @0 s /0 (ahbctrl): Info:  * fpnpen: 1
11 @0 s /0 (ahbctrl): Info:  * mcheck: 1
12 @0 s /0 (ahbctrl): Info:  * pow_mon: 1
13 @0 s /0 (ahbctrl): Info:  * abstractionLayer (LT = 8 / AT = 4):  4
14 ****************************************************************
```

Abbildung 5.6: Statischer Konfigurationsbericht für AHBCTRL

Alle hier dargestellten Parameter sind statisch und sollten zur Simulationszeit nicht geändert werden. Die meisten von ihnen haben einen direkten Bezug zu den *Generics* der VHDL-Modelle aus der GRLIB. Eine Ausnahme bilden Systemparameter, die eine direkte oder indirekte Unterstützung durch das Modell voraussetzen. Im vorliegenden Beispiel sind dies *pow_mon* zur Aktivierung der *Power Monitoring*-Funktion (Abschnitt 3.6) und *abstractionLayer* zur Auswahl des Abstraktionsniveaus in der Darstellung der Buskommunikation.

Darüber hinaus generieren die Simulationsmodelle unmittelbar vor Beginn der Simulation Berichte mit dynamischen Einstellungen. Dazu wird die *SystemC*-Funktion *start_of_simulation* verwendet. Dynamische Konfigurationsberichte beziehen sich auf Einstellungen, die erst durch die Bindung der Komponenten im System verfügbar werden. Dies umfasst in vielen Fällen *Routing*-Tabellen, zu deren Aufbau verteilte PNP-Register ausgelesen werden müssen. Abbildung 5.7 zeigt einen dynamischen Konfigurationsbericht für den AHBCTRL in einer einfachen Testkonfiguration.

```
1  ****************************************************************
2  @0 s /0 (top.ahbctrl): Info: * DECODER INITIALIZATION
3  @0 s /0 (top.ahbctrl): Info: * ------------------------
4  @0 s /0 (top.ahbctrl): Info: * SLAVE name: top.apbctrl
5  @0 s /0 (top.ahbctrl): Info: * BAR0 MSB addr: 0x800 and mask: 0xfff
6  @0 s /0 (top.ahbctrl): Info: * BAR1 not used.
7  @0 s /0 (top.ahbctrl): Info: * BAR2 not used.
8  @0 s /0 (top.ahbctrl): Info: * BAR3 not used.
9  ****************************************************************
10 @0 s /0 (top.ahbctrl): Info: * SLAVE name: top.mctrl
11 @0 s /0 (top.ahbctrl): Info: * BAR0 MSB addr: 0x000 and mask: 0xe00
12 @0 s /0 (top.ahbctrl): Info: * BAR1 MSB addr: 0x200 and mask: 0xe00
13 @0 s /0 (top.ahbctrl): Info: * BAR2 MSB addr: 0x400 and mask: 0xc00
14 @0 s /0 (top.ahbctrl): Info: * BAR3 not used.
15 ****************************************************************
16 @0 s /0 (top.ahbctrl): Info: * MASTER name: top.testbench
17 @0 s /0 (top.ahbctrl): Info: * BAR0 not used.
18 @0 s /0 (top.ahbctrl): Info: * BAR1 not used.
19 @0 s /0 (top.ahbctrl): Info: * BAR2 not used.
20 @0 s /0 (top.ahbctrl): Info: * BAR3 not used.
21 ****************************************************************
```

Abbildung 5.7: Dynamischer Konfigurationsbericht für AHBCTRL

Der AHB-Bus erlaubt die Bindung von maximal 16 *Master*- und 16 *Slave*-Komponenten. Jede dieser Komponenten darf bis zu vier einzeln dekodierbare Speicherbereiche ausweisen. Der im Beispiel dargestellte Bus wird durch eine *Testbench* gesteuert, die den einzigen angeschlossenen *Master* darstellt. *Slaves* sind die AHB/APB-Busbrücke (APBCTRL / Zeilen 4-8) und der Speicher-Controller (MCTRL) mit drei registrierten Subkomponenten (Zeilen 11 - 14).

Für die Überwachung und den Eingriff in den Simulationsablauf stehen dem Nutzer verschiedene Werkzeuge zur Verfügung. Wie bereits in Abschnitt 3.7.3 beschrieben, können Systemparameter durch eine Analyse-API aufgezeichnet und modifiziert werden. Mit Hilfe von *GreenAV* werden auf einfache Weise Rückruffunktionen für Wertänderungen definiert und *Traces* für beliebige Parameter angelegt. Es ist ebenfalls möglich, Transaktionsverläufe in *Message Sequence Charts* aufzuzeichen (Abschnitt 3.7.4). Da *SoCRocket*-VPs reine *C++* Programme darstellen, können Simulationen auch komplett aus einem *Debugger* (z.B. GDB) heraus gestartet werden. Dadurch lassen sich Haltepunkte (*Breakpoints*) in kritischen Prozessen und Überwachungspunkte (*Watchpoints*) für Register und Speicherinhalten realisieren. Dieses Vorgehen eignet sich besonders für die Integration neuer Komponenten, setzt allerdings detaillierte Kenntnisse des Gesamtsystems voraus.

Zum Abschluss der Simulation generieren die instantiierten Simulationsmodelle je einen Simulationsbericht. Ähnlich den zuvor beschriebenen Konfigurationsberichten, werden diese ab

Verbosity-Level report ausgegeben. Die Erzeugung der Berichte wird individuell in jedem Modell mit Hilfe des *SystemC*-Funktion *end_of_simulation* ausgelöst. Die enthaltene Information ist stark von der Art des Modelles abhängig. Abbildung 5.8 zeigt ein Beispiel für den AHB-Bus AHBCTRL.

```
1  @400 us (ahbctrl): Report:  ****************************************
2  @400 us (ahbctrl): Report:  * AHBCtrl Statistic:
3  @400 us (ahbctrl): Report:  * --------------------
4  @400 us (ahbctrl): Report:  * Successful Transactions: 32000
5  @400 us (ahbctrl): Report:  * Total Transactions:      32000
6  @400 us (ahbctrl): Report:  *
7  @400 us (ahbctrl): Report:  * Simulation cycles: 40000
8  @400 us (ahbctrl): Report:  * Idle cycles: 7999
9  @400 us (ahbctrl): Report:  * Bus utilization: 0.800025
10 @400 us (ahbctrl): Report:  * Maximum arbiter waiting time:
11                             15000 cycles
12 @400 us (ahbctrl): Report:  * Master with maximum waiting time: 0
13 @400 us (ahbctrl): Report:  * Average arbitration time / transaction:
14                             8.499 cycles
15 @400 us (ahbctrl): Report:  *
16 @400 us (ahbctrl): Report:  * AHB Master interface reports:
17 @400 us (ahbctrl): Report:  * Bytes read: 64000
18 @400 us (ahbctrl): Report:  * Bytes written: 64000
19 @400 us (ahbctrl): Report:  ****************************************
```

Abbildung 5.8: Simulationsbericht für AHBCTRL

Die Statistik zeigt, dass in 32000 Transaktionen je 64000 Bytes gelesen und geschrieben wurden. Der Bus war 80% des Testzeitraumes durch einen *Master* besetzt (Zeile 9). Die durchschnittliche Wartezeit am Bus betrug 8.5 Takte, wobei mindestens eine Transaktion 15000 Takte blockiert wurde. Andere Simulationsmodelle, wie das Cachesystem der CPU generieren eine Statistik, die Aufschluss über die Anzahl an *Hits* und *Misses* in den verschiedenen Speicherbänken gibt. Der Interruptcontroller fasst zusammen, welcher Interrupt wie oft ausgelöst wurde und der Speichercontroller berechnet die Anzahl der Zugriffe per Speicherbereich (ROM, I/O, SRAM, SDRAM) und die in den verschiedenen Energiesparmodi verbrachte Simulationszeit.

Ein Überblick über alle in den verschiedenen Simulationsmodellen implementierten Parameter und die Struktur der daraus erzeugten Berichte kann dem SoCRocket *Analysis Capability Report* entnommen werden [Sch12a]. Wie auch die Konfigurationsparameter wurden alle Leistungszähler und Statistikparameter an die *GreenControl-Middleware* angeschlossen und stehen somit zur einfachen maschinellen Weiterverarbeitung zur Verfügung.

5.2 Architekturexploration

5.2.1 Mehrstufiger Explorationsansatz

Durch die beschriebene nicht-intrusive Konfigurierbarkeit von Hardware und Software und den durch die Analyse-API gegebenen freien Zugriff auf Simulationsstatistiken und Modellparameter, bietet *SoCRocket* ideale Voraussetzungen zur Architekturexploration. Wie in Abbildung 5.9 dargestellt kann dafür eine beliebige Explorationslogik verwendet werden.

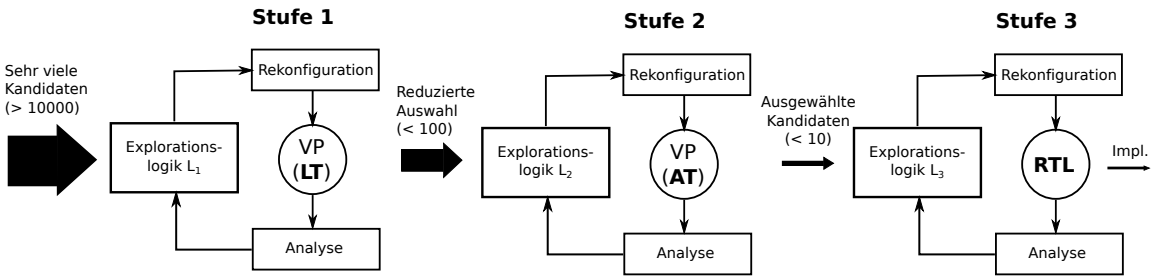


Abbildung 5.9: Mehrstufige Architekturexploration

Die Explorationslogik startet die Erkundung mit einer initialen Konfiguration (JSON-Datei), übergibt diese an das System und startet die Simulation. Am Ende des Laufes werden alle relevanten Kennzahlen mit Hilfe der Analyse-API extrahiert. Auf Grundlage der so gewonnenen Erkenntnisse kann nun die Konfiguration modifiziert und die Simulation neu gestartet werden. Im einfachsten denkbaren Fall stellt die Explorationslogik einen manuellen Nutzereingriff dar. Dabei entscheidet der Entwickler, auf Grundlage seiner Erfahrung, welche Veränderungen im System vorgenommen werden. Eine weitere Möglichkeit besteht in der vollständigen Durchsuchung (*exhaustive search*) des Entwurfsraumes. In diesem Fall durchläuft die Explorationslogik alle denkbaren Konfigurationen. Da die Anzahl der möglichen Kombinationen in Abhängigkeit der in Betracht gezogenen Parameter schnell sehr groß werden kann, ist dieses Vorgehen nur für kleinere Systeme mit beschränkter Simulationszeit praktikabel. Da alle Kombinationen unabhängig voneinander getestet werden, besteht jedoch die Möglichkeit der Parallelisierung. Für komplexere Systeme ist eine strukturiertere Vorgehensweise erforderlich. Dafür sind verschiedene Ansätze denkbar. Taghavi, Pimentel und Thompson stellen in [Tag09] eine auf der Auswertung einer Baumstruktur basierende Lösung vor. Durch den Ausschluss von Optimierungsgruppen (Ästen) wird der Suchraum effizient eingeschränkt. Sehr weit verbreitet ist der Einsatz von Heuristiken basierend auf *Simulated Annealing* [Tal06]. Ebenfalls denkbar ist die Verwendung von Techniken zum Maschinenlernen. Ein Beispiel zum Einsatz von neuronalen Netzen zur Steuerung von Entwurfsraumerkundungen kann [Ozi08] entnommen werden.

SoCRocket bietet darüber hinaus die Möglichkeit, den Entwurfsraum unter Einsatz der verschiedenen unterstützten Abstraktionsniveaus hierarchisch zu durchsuchen. Wie in Abbildung 5.9 dargestellt, wird dazu in Stufe 1 ein abstrakter Prototype mit lose modelliertem Zeitverhalten (*Loosely Timed* (LT)) eingesetzt. Mit Hilfe des schnellen LT-Prototypen kann die anfänglich enorm hohe Anzahl an Implementierungskandidaten schnell reduziert werden. Da alle *SoCRocket*-Simulationsmodelle durch das Setzen eines Systemparameters in den näherungsweise akkuraten AT-Modus gebracht werden können, ist der Aufwand zum Übergang in Stufe 2 minimal. Mit Hilfe der nun höheren zeitlichen Genauigkeit werden die vielversprechendsten Kandidaten aus Stufe 1 erneut simuliert. Das Ergebnis ist eine geringe Anzahl ausgewählter Konfigurationen (<5), die zur Ermittlung des finalen Implementierungskandidaten in der dritten und abschließenden Stufe auf RT-Ebene untersucht werden. Die Wahl der Explorationslogik sollte an die in den jeweiligen Stufen zu bewältigenden Aufgaben angepasst werden. Für Stufe 1 bietet sich die Nutzung einer Heuristik an, die den Suchraum aktiv einschränkt. Die daraus hervorgehende reduzierte Auswahl an Konfiguration kann in der Regel vollständig durchsucht werden (Stufe 2). Die wenigen zur Erprobung auf RT-Ebene ausgewählten Systeminstanzen (Stufe 3) können manuell parametrisiert werden.

5.2.2 Multispektrale/Hyperspektrale Bildkompression

Zur Demonstration des mehrstufigen Explorationsansatzes wurde eine verlustfreie multispektrale/hyperspektrale Bildkompression auf das in Abschnitt 6.1 beschriebene Multiprozessorsystem abgebildet. Der Algorithmus wurde durch das *Consultative Commitee for Space Data Systems* (CCSDS) standardisiert (Standard 123 [CCS12]) und stellt eine typische Anwendung für die *Payload*-Prozessoren an Bord wissenschaftlicher Satelliten dar. Derartige Systeme sind oft in

ihren Ressourcen beschränkt und daher auf eine optimale Konfiguration angewiesen, die für eine gegebene Siliziumfläche und ein gegebenes *Power*-Budget die höchstmögliche Leistung liefert.

Wie bereits erwähnt dient der betrachtete Algorithmus der Kompression hyperspektraler Bilder. Dabei handelt es sich um dreidimensionale Datensätze bestehend aus zwei räumlichen und einer spektralen Dimension. Ein hyperspektrales Bild kann somit als ein Stapel von Einzelbildern betrachtet werden, welche die selbe Szene in unterschiedlichen Bereichen des elektromagnetischen Spektrums darstellen. Der Algorithmus basiert auf einer adaptiven linearen vorhersagenden (*predictive*) Kompression, die den *Sign*-Algorithmus, *Local Mean Estimation* und Subtraktion zur Filteranpassung einsetzt. Die Vorhersagewerte (*prediction residuals*) werden abschließend mit einem *sample*-adaptiven *Golomb-Rice-Encoder* kodiert. Eine ausführliche Beschreibung des Algorithmus kann [Mag09] entnommen werden. Die Implementierung des Algorithmus in der Sprache *C* wurde durch die ESA bereitgestellt und für Explorationszwecke teilweise parallelisiert (Entropieencoder). Das primäre Optimierungsziel ist die Auffindung einer Konfiguration mit hoher Ausführungsgeschwindigkeit, bei möglichst geringer durchschnittlicher Leistungsaufnahme. Das Ziel soll mit möglichst geringem Hardwareeinsatz erreicht werden (sekundäres Ziel). Als Optimierungsparameter wurden die Anzahl der Prozessoren und verschiedene Cacheparameter ausgewählt (Tabelle 5.1), da diese die Systemkosten (z.B. *Power*, Fläche) und die Leistungsfähigkeit maßgeblich beeinflussen.

Parameter	Beschreibung	Bereich
system.ncpu	Anzahl der LEON3 CPUs im System	1, 2, 4, 6
conf.mmu_cache.ic.sets	Anzahl der *Instruction-Cache*-Bänke	1, 2, 3, 4
conf.mmu_cache.ic.setsize	Größe der *Instruction-Cache*-Bänke	1, 2, 4, 8, 16 kB
conf.mmu_cache.dc.sets	Anzahl der *Data-Cache*-Bänke	1, 2, 3, 4
conf.mmu_cache.dc.setsize	Größe der *Data-Cache*-Bänke	1, 2, 4, 8 kB

Tabelle 5.1: Optimierungsparameter für Entwurfsraumerkundung

Es ergeben sich 1280 Parameterkonfigurationen, die der ersten Explorationsstufe übergeben werden. Der dafür erforderliche Aufwand ist beherrschbar. Die Simulationszeit für die vollständige Durchsuchung des Entwurfsraumes beträgt 16 Stunden. Auf dem verwendeten Server (4x *Quad-XEON*, 3.4 GHz, 32GB RAM) konnten 16 Simulationen parallel ausgeführt werden. Dies entspricht einer durchschnittlichen Simulationsdauer von 45 Sekunden. Über die Hälfte dieser Zeit werden für das Booten des Betriebsystems (RTEMS) und die Initialisieren des Arbeitsspeichers benötigt. Zur Validierung der HW/SW-Schnittstelle wird die Anwendung gemeinsam mit dem OS und den Nutzerdaten zum Simulationsbeginn aus dem ROM geladen und in die den verschiedenen Prozessoren zugeordneten RAM-Bereiche verschoben. Eingabe ist ein hyperspektrales Bild mit drei Frequenzbändern zu je 256 x 256 Pixeln (768 kB), das im RTEMS-Dateisystem (IMFS) abgespeichert wird. Die dafür und zum *Booten* erforderliche Zeit ist konstant und wird bei der Betrachtung der Simulationsergebnisse ignoriert. In RTEMS wird für Prozessor 0 ein *Master-Thread* angelegt. Dieser berechnet zunächst die Residuen und startet im Anschluss *Slave-Threads*, die sich auf den Prozessoren $1 - N$ befinden. Dies geschieht durch Beschreiben eines Start-*Flags* in einem gemeinsam genutzten Speicherbereich. Dort wird ebenfalls eine zentrale Datenstruktur zur Beschreibung der Eingabedaten und des Bearbeitungsfortschritts abgelegt. Die Datenstruktur ist durch Semaphoren geschützt. Durch konkurrierenden Zugriff können die *Slave*-Prozessoren Datenblöcke zur Weiterverarbeitung markieren und anschließend die Entropiekodierung starten. Nach erfolgreicher Kodierung werden die komprimierten Daten in einem Ausgabefeld gespeichert. Die Bearbeitung wird fortgesetzt, bis alle Datenblöcke abgearbeitet sind. Danach werden die *Slaves* an einer Barriere synchronisiert. Abschließend fasst der *Master* die Ausgabedaten zusammen, speichert das komprimierte Bild im Dateisystem ab und beendet das Programm. Der Quellcode befindet sich im Verzeichnis *software/hyperspectral* der Plattform (Anhang B).

Abbildung 5.10 zeigt die Simulationsergebnisse für die erste Stufe der Exploration. Der Graph stellt die simulierte Zeit in einer logarithmischen Skala über der durchschnittlichen Leistungsaufnahme dar. Wie zu erwarten gruppieren sich die Ergebnisse entsprechend der Anzahl der

eingesetzten Prozessoren in vier Cluster. Die höchste Simulationsgeschwindigkeit wird durch das System mit sechs Prozessoren und maximaler Cachegröße erreicht. Mit vier Bänken zu je 16 kB Instruktionscache und vier Bänken zu je 8 kB Datencache benötigt die Simulation 466 ms, nimmt dabei im Durchschnitt jedoch mehr als 23 Watt Leistung auf. Das schnellste *Quad*-System bewältigt die Aufgabe in 544 ms mit 15,4 Watt. Einem Geschwindigkeitsverlust von 17% steht dabei eine Verringerung der Leistungsaufnahme um 33% gegenüber. Ein System mit zwei Prozessoren und maximaler Cachekonfiguration beendet die Simulation in 705 ms. Die durchschnittliche Leistung beträgt hier jedoch nur 7,9 Watt. Im Vergleich zum *Quad*-System entspricht dies einem Geschwindigkeitsverlust von 30%, gegenüber einer Senkung der durchschnittlichen Leistung von fast 49%. Das beste Ergebnis des *Single*-Prozessors liegt bei 1350 ms und 4,5 Watt. Er benötigt also fast 90% mehr Zeit als der *Dual*-Prozessor. Die Leistungsaufnahme ist dabei jedoch 43% geringer[1]. Die von den Ergebnisclustern gelösten Punkte mit hoher Simulationszeit gehören zu Konfigurationen mit sehr kleinem Instruktions- und/oder Datencaches. Ein *Single*-Prozessor mit *Direct-Mapped Caches* zu je 1kB benötigt für die Simulation fast 27 Sekunden. Das beschriebene schnellste System mit sechs Prozessoren ist 58x schneller. Die gewählten Explorationsparameter eröffnen also einen sehr weiten Spielraum zur Bestimmung angemessener Implementierungsoptionen. Zur endgültigen Auswahl kann eine Kostenfunktion zum Einsatz kommen, die neben Abarbeitungszeit und Leistungsaufnahme auch den Hardwareaufwand (z.B. Fläche) und weitere Faktoren einschließt. Da es sich um einen hypothetischen Anwendungsfall handelt, für den keine spezifischen Anforderungen vorliegen, erfolgt die Auswahl *ad-hoc*. Im vorliegenden Beispiel erscheinen die in Abbildung 5.10 markierten Punkte (Auswahl: Rechteck) vielversprechend, da sie eine verhältnismäßig hohe Simulationsgeschwindigkeit, bei niedriger durchschnittlicher Leistung versprechen. Außerdem ist der Hardwareaufwand im Vergleich zum *Quad*-System nur ungefähr halb so groß.

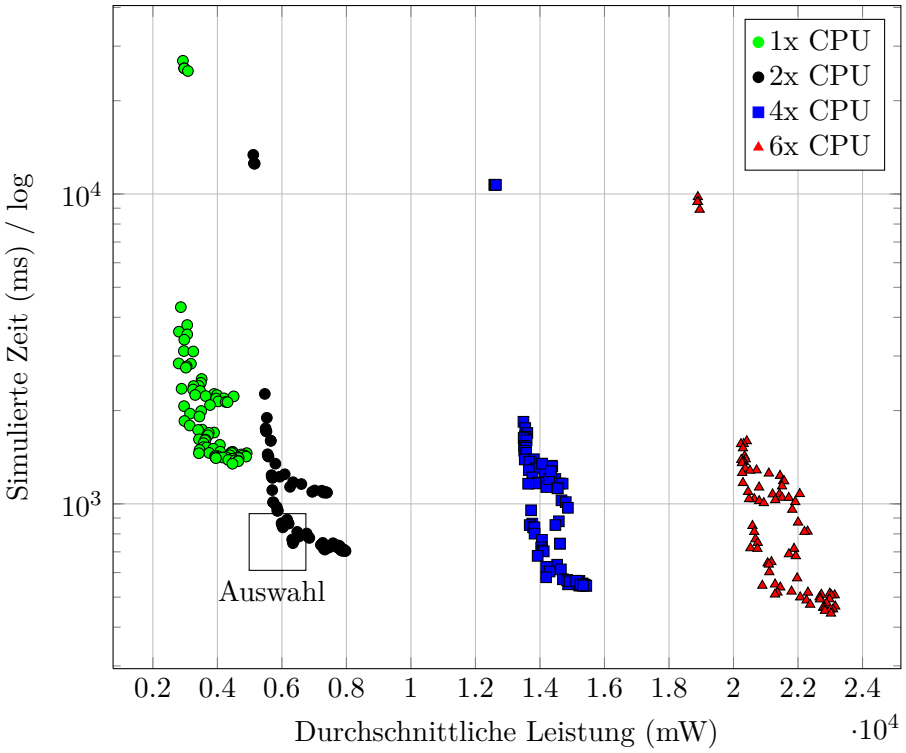


Abbildung 5.10: Explorationsergebnisse der Stufe 1 (LT - Simulation)

Die ausgewählten 12 Konfigurationen dienen als Eingabe für die zweite Explorationsstufe.

1 Ergebnisse für den *Single*-Prozessor sind *out-of-the-box*. Die vergleichsweise hohe Beschleunigung zwischen *Single*- und *Multi*-Prozessoren kann teilweise auf Kontrollflussoptimierungen zurückgeführt werden.

Das System wird durch das Setzen eines Konfigurationsparameters in den AT-Modus mit höherer Simulationsgenauigkeit geschaltet. Eine erneute Kompilierung ist nicht erforderlich. Die Simulationen können gleichzeitig gestartet werden und sind nach vier Stunden abgeschlossen. Dies entspricht einer Simulationsdauer von 20 min pro Simulation. Der Laufzeitunterschied zwischen LT- und AT-Modus beträgt somit Faktor 25. Die Simulationsergebnisse sind in Abbildung 5.11 dargestellt. Eine Übersicht der ausgewählten Konfigurationen und eine Gegenüberstellung der Ergebnissen befindet sich in Tabelle 5.2.

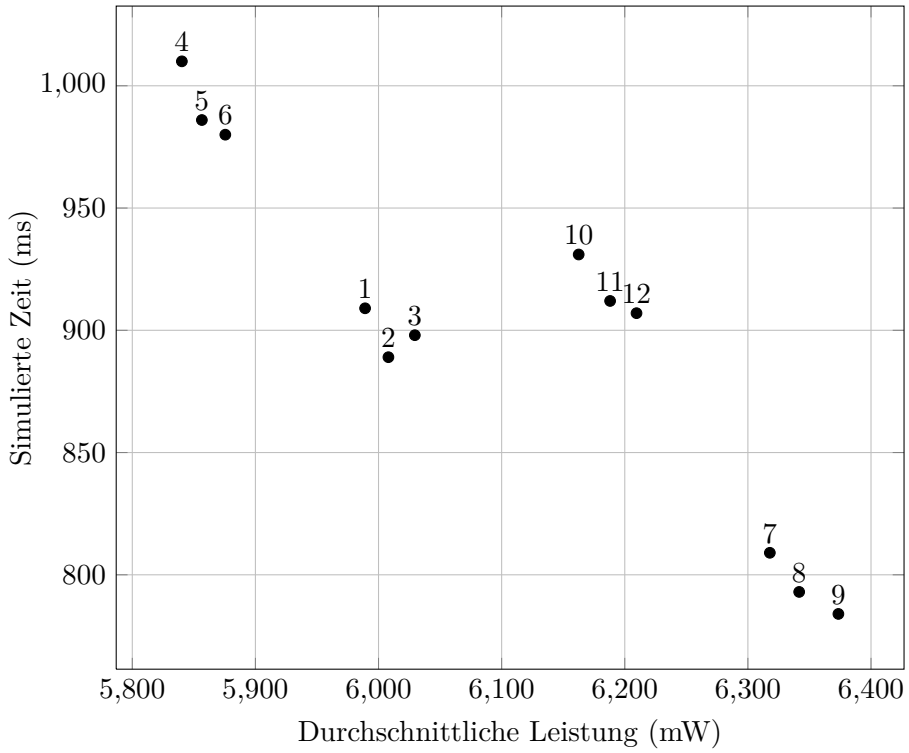


Abbildung 5.11: Explorationsergebnisse der Stufe 2 (AT - Simulation)

Aus den Ergebnissen ist ersichtlich, dass innerhalb der in der engeren Auswahl befindlichen *Dual*-Prozessoren die Größe und Konfiguration des Instruktionscaches den größten Einfluss auf Ausführungsgeschwindigkeit und Leistungsaufnahme haben. Die zwölf Systeme zerfallen in vier Dreiergruppen mit gleichem Instruktions- aber unterschiedlichen Datencaches.

Nr.	Konfiguration	Stufe 1 - LT Zeit (ms)	Stufe 2 - AT Zeit (ms)
1	p:2 i:4x1kB d:2x2kB	866	909
2	p:2 i:4x1kB d:3x4kB	851	889
3	p:2 i:4x1kB d:4x8kB	839	898
4	p:2 i:2x2kB d:2x2kB	977	1010
5	p:2 i:2x2kB d:3x4kB	949	986
6	p:2 i:2x2kB d:4x8kB	951	980
7	p:2 i:2x4kB d:2x2kB	767	809
8	p:2 i:2x4kB d:3x4kB	747	793
9	p:2 i:2x4kB d:4x8kB	760	784
10	p:2 i:1x8kB d:2x2kB	889	931
11	p:2 i:1x8kB d:3x4kB	869	912
12	p:2 i:1x8kB d:4x8kB	863	907

Tabelle 5.2: Ausgewählte Konfigurationen - *LT* vs. *AT*

5.3 Entwicklung von HW/SW-Schnittstellen (Beispiel CFDP)

Durch die vollständige Sichtbarkeit auf alle Hardware- und Softwarekomponenten eignet sich *SoCRocket* in besonderem Maße zur Entwicklung von HW/SW-Schnittstellen. Die dazu empfohlene Vorgehensweise wird im vorliegenden Abschnitt anhand eines Beispieles erläutert.

5.3.1 Übersicht: CFDP-Transaktionsmanager

Die Modellierung von HW/SW-Schnittstellen wurde in einem weiteren gemeinsamen Projekt mit der ESA erprobt. Ziel war die Entwicklung eines Hardwarebeschleunigers und entsprechender Steuerungssoftware zum effizienten Transfer von Dateien zwischen Satelliten und einer Bodenstation[1]. Das Projekt wurde vom Autor dieser Arbeit gemeinsam mit Prof. Harald Michalik konzipiert und beantragt. An der technischen Ausführung war der Autor nur beratend beteiligt, was eine objektive Einschätzung des Aufwandes ermöglicht. Abbildung 5.12 zeigt einen Ausschnitt der implementierten Funktionalität.

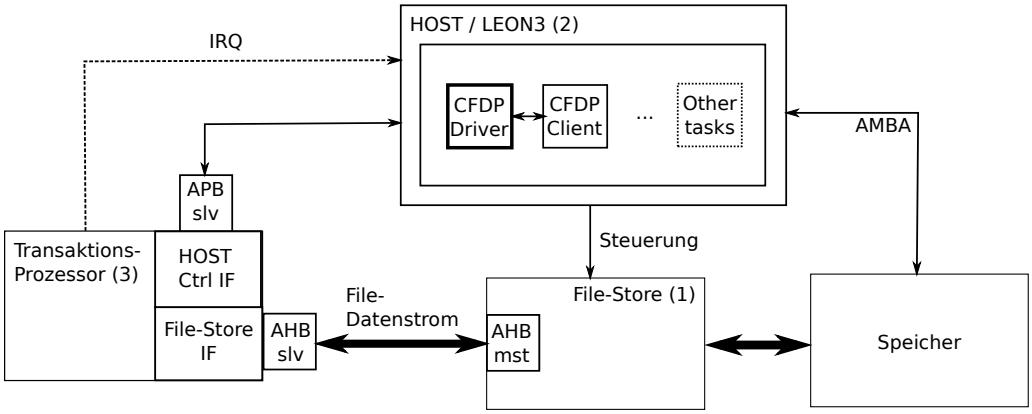


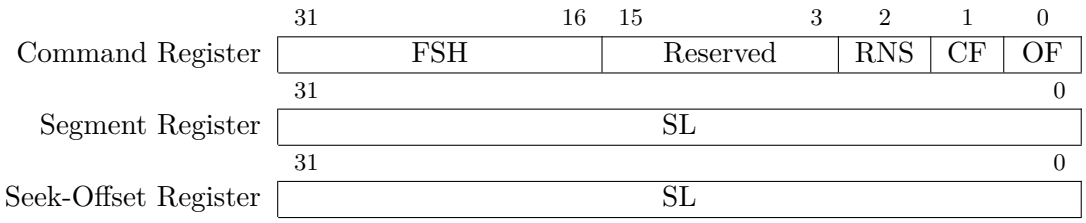
Abbildung 5.12: CFDP - Transaktionsprozessor

Der *Filestore* (FS) (1) ist ein Speicher, der Massendaten (z.B. Hyperspektrale Bilder) vorhält. Diese können über eine beliebige *Streaming*-Schnittstelle ausgegeben werden. Normalerweise erfolgt die Ausgabe in Richtung des *Host*-Prozessors (2), der die Daten dann gemäß des in [Con07] beschriebenen Protokolls partitioniert, kapselt und an die Telemetrie zur Übertragung weiterleitet. Der *Host* wird dabei sehr stark belastet. Zur Beseitigung dieses Engpasses wird ein Transaktion-Prozessor (TP) (3) eingeführt. Der TP implementiert das CFDP-Protokoll in Hardware. Anfragen (*Requests*) der Bodenstation werden dem *Host* mit Hilfe einer Steuerungsschnittstelle und eines Interrupts angezeigt. Der *Host* realisiert die Ansteuerung des FS und kann somit die Konsistenz des Systems wahren. Der Datenfluss läuft am *Host* vorbei, direkt vom FS zum TP, wodurch dieser erheblich entlastet wird.

5.3.2 Entwurf der *HW*-Schnittstelle

Zur Entwicklung der HW/SW-Schnittstelle wurde zunächst ein neues *SoCRocket*-Modul erzeugt. Das Modul erbt eine AHB-*Slave*-Schnittstelle von der Basisklasse *ahb_slave*, eine APB-*Slave*-Schnittstelle von *apb_device* und einen Signalausgang von *Signalkit* (siehe Abschnitt 3). Daraufhin wurde am APB-Sockel durch Vererbung von *gr_device* ein speicheradressierbares Registerfile aufgebaut. Wie in Abbildung 5.3 dargestellt besteht dieses aus drei Registern, die den Beginn des Adressraumes der Komponente bilden.

1 *Technical Support for ESA IP Cores: Development of CCSDS File Delivery Protocol IP* - ITT AO/1-6939/11/NL/JK



Bit-Feld	Beschreibung
OF	Datei öffnen (*Open File*)
CF	Datei schließen (*Close File*)
RNS	Nächstes Dateisegment lesen (*Read Next Segment*)
FSH	*File Store Handle*
SL	Segment-Länge
SEEK	*Seek*-Position innerhalb der Datei

Tabelle 5.3: CFDP - *HOST Ctrl IF* im Transaktions-Prozessor

Abbildung 5.13 zeigt den zum Aufbau der *Host*-Schnittstelle erforderlichen *GreenReg*-Code am Beispiel des *Command_Registers*. Alle weiteren Register werden in gleicher Weise erstellt. Das Registerfile *mr* wird automatisch am APB-Sockel registriert. Da die Schnittstelle nur zur Übermittlung von Befehlen an den Prozessor verwendet werden soll, müssen keine Rückruffunktionen deklariert werden.

```
1  mr.create_register(
2     "fs_request_command",                          // Name
3     "Command Register",                            // Beschreibung
4     mbase_address + FS_REQUEST_COMMAND,            // Offset
5     gs::reg::STANDARD_REG | gs::reg::SINGLE_IO | \\
6     gs::reg::SINGLE_BUFFER | gs::reg::FULL_WIDTH,  // Zugriffsmodi
7     0x0,                                           // Initialisierung
8     0xFFFFFFFF,                                    // Schreibmaske
9     32,                                            // Weite in Bits
10    0                                              // Lock-Modus
11 );
```

Abbildung 5.13: HOST Ctrl IF des CFDP Transaktionsprozessors

Zur vorläufigen Modellierung des Verhaltens wurde zu TP ein *Thread* hinzugefügt, der typische Operationen in der gewünschten zeitlichen Abfolge generiert. Im vorliegenden Fall umfasst dies Befehle zum Öffnen und Schließen von Dateien, sowie zum Lesen von Datensegmenten unterschiedlicher Länge von unterschiedlichen Positionen (*Seek*). Zur Ausgabe eines Befehls werden zuerst die Steuerregister gesetzt, danach wird ein Interrupt ausgesendet. Der Erfolg der Operation kann an der AHB-Schnittstelle (*File-Store IF*), direkt in der Rückruffunktion *exec_func*, beobachtet werden.

5.3.3 Erprobung mit *Unit*-Testumgebung

Zur Erprobung der initialen Hardwareschnittstelle bietet sich die Nutzung der in Abschnitt 3.8.1 beschriebenen Testumgebung, speziell der direkten Lese- und Schreibschnittstelle (*direct r/w*) an. Die Testkomponente für den *TP* dient als Platzhalter für den Prozessor und den *FS* (Abb. 5.14).

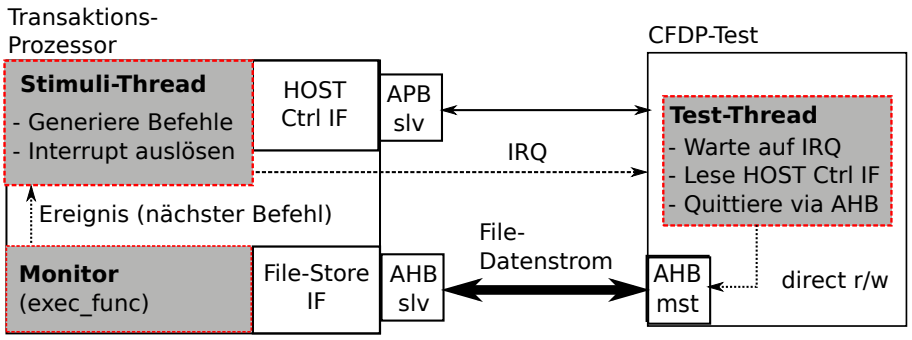


Abbildung 5.14: Testkomponente für HW-Schnittstelle des Transaktionsprozessors

Neben der AHB-*Master*-Schnittstelle, die sowohl auf das *HOST Ctrl IF* als auch auf das *File-Store IF* des TP zugreifen kann, erhält die Testkomponente einen *SignalKit*-Sockel zum Empfang von Interrupts. Die am Signaleingang registrierte *Handler*-Funktion (Test-*Thread*) liest die Steuerregister des TP ein und löst abhängig von den gelesenen Werten eine Operation am AHB-*Master* aus. Im einfachsten denkbaren Fall handelt es sich dabei um einen einzelnen Transfer, der in seiner *Payload* eine Quittung für den empfangenen Befehl transportiert. Es ist aber auch möglich, die vom TP empfangenen Operationen auf das Dateisystem des *Host*-Computers auszulagern, wodurch ein realistischer Datenstrom vom FS zum TP simuliert werden kann.

Durch den *Unit*-Test bildet sich Sicherheit bezüglich der neu entwickelten HW-Schnittstelle heraus. Aufgrund der Einfachheit des Systems können Fehler leicht erkannt und erste Optimierungen vorgenommen werden.

5.3.4 Entwurf der Softwareschnittstelle

Nach erfolgreichem *Unit*-Test kann die im Test-*Thread* implementierte Funktionalität in Software überführt werden. Dazu wird das in 5.12 dargestellte System, wie in Abschnitt 5.1 beschrieben, in *SoCRocket* aufgebaut und zur Simulation vorbereitet. Es hat sich als vorteilhaft herausgestellt, zunächst eine initiale Version der Software ohne Betriebssystem (*Bare-Metal*) zu entwickeln. Dadurch können eventuell auftretende Fehler leichter lokalisiert werden. *Boot*-Code für *Bare-Metal*-Systeme wird durch die Bibliothek bereitgestellt (Anhang B - */software/prom*). Zur Ansteuerung des Interruptcontrollers können die in GRLIB enthaltenen Assemblerroutinen verwendet werden (*asm-leon/irq.h*). Das Beispiel in Abbildung 5.15 veranschaulicht den Aufbau des Codes.

```
1  // Container fuer CFDP-Filestore-Request
2  typedef struct {
3    uint32_t of : 1, cf : 1, rns : 1, reserved : 13, fsh : 16;
4    uint32_t segment_length;
5    uint32_t seek_location;
6  } cfdp_filestore_request_t;
7  ...
8  void cfdp_bare_metal_interrupt_handler (int irq) {
9
10   cfdp_filestore_request_t filestore_request;
11   ...
12   switch(irq) {
13     ...
14     case CFDP_IRQ_FILESTORE_ACCESS:
15
16     // APB Register von HOST CTRL IF lesen
17     ahb_read(TP_HOST_CTRL_IF_ADDR, &filestore_request.command, 4);
18     ahb_read(TP_HOST_CTRL_IF_ADDR+4, &filestore_request.segment_length, 4);
19     ahb_read(TP_HOST_CTRL_IF_ADDR+8, &filestore_request.seek_location, 4);
20
21     // Quittung erzeugen und and AHB Filestore IF senden
22     ahb_write(TP_FILE_STORE_IF_ADDR, gen_receipt(filestore_request), 4);
23     break;
24 ...
25 }
```

Abbildung 5.15: Handhabung von Filestore-Request im Bare-Metal System

In Zeilen 2-6 wird eine Datenstruktur aufgebaut, welche die Steuerregister der *HOST-Ctrl*-Schnittstelle vollständig abbildet. Mit der in Zeile 3 verwendeten Notation lassen sich Bitfelder für den Direktzugriff einfach darstellen. Darüber hinaus sollten Konstanten für die Adressen der anzusteuernden Register und Bitmasken zum Extrahieren von Datenfeldern (z.B. *File Open/Close*) deklariert werden. Der Interrupt-*Handler* ist vereinfacht ab Zeile 8 dargestellt. Er kann mit Hilfe der Methode *catch_interrupt* aus *leon-asm/irq.h* beim System registriert werden. Die Identifikationsnummer des Interrupts wird der Funktion als Parameter übergeben (*irq*). Wurde der Interrupt vom TP generiert (CFDP_IRQ_FILESTORE_ACCESS), so werden mit Hilfe der Testschnittstelle *direct r/w* alle Register des *HOST Ctrl IF* ausgelesen (Zeilen 17 - 19). Die Ausgabe der Antwort erfolgt in Zeile 22. Die Generierung der Quittung könnte ähnlich wie dargestellt mit Hilfe einer Funktion (*gen_receipt*) erfolgen. Der LEON3-Simulator unterstützt Quellcode-*Debugging*, was die Programmierung stark erleichtert. Dazu muss lediglich ein GDB-*Stub* erzeugt und dem *Tool-Manager* der Integereinheit übergeben werden. Im in Abschnitt 6.1 beschriebenen Beispielsystem LEON3MP geschieht dies durch das Setzen des Systemparameters *conf.gdb.en*.

Der letzte Schritt zur Fertigstellung der HW/SW-Schnittstelle ist die Integration der Software mit dem Betriebssystem. Im Rahmen des Projektes CFDP wurde RTEMS eingesetzt. RTEMS organisiert den Zugriff auf periphere Hardwareschnittstellen mit Hilfe eines Managermoduls. Der I/O-Manager stellt dem Programmierer einen einheitlichen Mechanismus zum Zugriff auf alle registrierten Geräte bereit. Die Adressierung erfolgt, ähnlich wie in Linux, mit Hilfe von *Major*- und *Minor*-Identifikationsnummern. Der Gerätetreiber kann über die *Plug & Play*-Einstellungen des Busses (Abschnitt 4.2.1), die Basisadressen der Schnittstellen ermitteln und transparent für den Nutzer umleiten. Um einen entsprechenden Treiber zu registrieren, muss die Software eine Initialisierungsroutine bereitstellen, die aus der RTEMS-Hauptfunktion (*rtems_task Init* aufgerufen werden kann. Die Initialisierungsfunktion ruft die Funktion *rtems_io_register_driver* des I/O-Managers auf und übergibt dieser eine Treiberadresstabelle (*cfdp_io_operations*) und eine *Major*-Nummer (*cfdp_major*):

```
status = rtems_io_register_driver(0, &cfdp_io_operations, &cfdp_major);
```

Die Treiberadresstabelle ordnet den Basisfunktionen des IO-Managers spezielle Funktionen des Treibers zu:

```
rtems_driver_address_table cfdp_io_operations = {
  .initialization_entry =  cfdp_init,
  .open_entry =            cfdp_open,
  .close_entry =           cfdp_close,
  .read_entry =            cfdp_read,
  .write_entry =           cfdp_write,
  .control_entry =         cfdp_ioctl,
};
```

Der Zugriff auf das *HOST Ctrl IF* des TP erfordert keine Initialisierung, sofern die Basisadresse der Schnittstelle als konstant betrachtet werden kann. Außerdem muss die Schnittstelle weder speziell zur Nutzung geöffnet oder im Anschluss geschlossen werden. Die Funktionen *cfdp_init*, *cfdp_open* und *cfdp_close* werden daher leer implementiert. Für den Zugriff ist einzig die Funktion *cfdp_ioctl* erforderlich. Die Funktionen *cfdp_read* und *cfdp_write* werden zur Kommunikation mit dem FS benötigt. Neben der Installation des Treibers erfordert die Integration der Software die Registrierung eines Interrupt-*Handlers*. RTEMS stellt dafür die Methode *set_vector* bereit:

```
set_vector(cfdp_interrupt_handler, cfdp_vector_number, 1);
```

Der Interrupt-*Handler* ist ähnlich wie im *Bare-Metal*-System aufgebaut, nur dass zum Zugriff auf die externen Schnittstellen nun der I/O-Manager von RTEMS verwendet wird (Abbildung 5.16).

```
1  rtems_isr cfdp_interrupt_handler(rtems_vector_number vector) {
2
3    // Container fuer I/O-Operationen
4    cfdp_ioctl_t control;
5
6    control.command = CFDP_REGISTER_READ;
7    control.address = CFDP_INTERRUPT_STATUS;
8    rtems_io_control(cfdp_major, cfdp_minor, &control);
9
10   switch(control.data) {
11
12     case (CFDP_IRQ_FILESTORE_ACCESS):
13
14     // Lese alle Register des TP Host Ctrl IF
15     control.address = CFDP_FILESTORE_REQUEST_COMMAND;
16     rtems_io_control(cfdp_major, cfdp_minor, &control);
17     ...
18     // Quittung erzeugen und senden
19     rtems_io_write(cfdp_major, cfdp_minor, gen_receipt(control.data));
20     break;
21 ...
22 }
```

Abbildung 5.16: Handhabung von Filestore-Request in RTEMS

Als Container für I/O-Operationen kann im Treiber ein beliebiger Datentyp deklariert werden, dieser wird dann von den Schnittstellenfunktionen des I/O-Managers als Argument akzeptiert. Der Container *cfdp_ioctl_t* (Zeile 4) enthält ein Kommando-, ein Adressen- und ein Datenfeld.

In Zeilen 6-7 wird die Ursache des Interrupts ermittelt. Das dafür erforderliche Register gehört nicht zum *HOST Ctrl IF* und wurde im TP nachträglich eingefügt. Handelt es sich um einen Befehl zum Zugriff auf den FS (Zeile 12), werden ähnlich wie im *Bare-Metal*-System *Command Register*, *Segment Register* und *Seek-Offset Register* ausgelesen. Der Codeausschnitt verdeutlicht dies am Beispiel des *Command Registers* (Zeilen 15 - 16). Im Anschluss wird wie zuvor mit der Funktion *gen_receipt* eine Quittung zur Übersendung an den TP generiert (*rtems_io_write* - Zeile 19).

5.3.5 Aufwandsschätzung

Durch die beschriebene Vorgehensweise konnten die Schnittstellen des CFDP-Transaktionsprozessors und die zu deren Ansteuerung erforderliche Systemsoftware in kurzer Zeit definiert und erprobt werden. Das Verfahren verspricht einen hohen Produktivitätsgewinn, da essentielle Softwarekomponenten bereits vor der Verfügbarkeit von Hardware entwickelt und getestet werden können. Außerdem ist es auf Grundlage der Simulationsergebnisse möglich, Hardwareschnittstellen frühzeitig zu optimieren. Durch die verhältnismäßig geringe Komplexität waren nur wenige Iterationen nötig. Der Entwurf der Hardwareschnittstelle, des *Unit*-Tests und des Systems mit *Bare-Metal*-Software konnten nach einwöchiger Einarbeitungszeit in *SoCRocket*, durch einen Entwickler innerhalb von zwei Tagen ausgeführt werden. Die Integration mit RTEMS erforderte weitere zwei Tage, wobei ein Großteil dieser Zeit zur Einarbeitung in das Betriebssystem benötigt wurde.

6 Anwendungsbeispiel: VP LEON2/3MP

6.1 Übersicht

Einer der wichtigsten in *SoCRocket* integrierten Explorationsprototypen ist der LEON2/3MP. Wie in Abschnitt 1.3 beschrieben, wurde das entsprechende RTL-Architekturtemplate durch die Firma *Aeroflex/Gaisler* entwickelt und bildet die Grundlage für verschiedene in der europäischen Raumfahrt eingesetzten Datenverarbeitungssysteme, wie den *Dual*-Prozessor GR712RC oder den LEON3FT-RTAX. Das System integriert alle in Kapitel 4 beschriebenen Kernkomponenten zur Modellierung robuster eingebetteter Systeme und unterstützt die in Kapitel 5 beschriebene Infrastruktur zum Entwurf Virtueller Prototypen. Wie in Abbildung 6.1 vereinfacht dargestellt besteht der LEON2/3MP aus ein oder mehreren Prozessoren, die an einen zentralen AHB-Bus (AHBCTRL) angeschlossen sind.

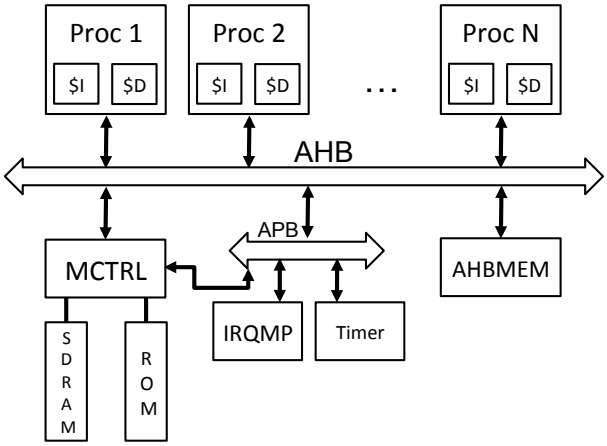


Abbildung 6.1: Der Explorationsprototyp LEON2/3MP

Die Anzahl der Prozessoren ist wie im RTL-System auf 16 beschränkt. Speicherzugriffe werden durch den Speichercontroller MCTRL realisiert. Die Modellbibliothek stellt generische Speicher bereit, die als ROM, I/O, SRAM oder SDRAM konfiguriert werden können. Peripheriekomponenten wie Interruptcontroller (IRQMP), Timer (GPTimer) und UART (APBUART) werden mit Hilfe der AHB/APB-Brücke (APBCTRL) angesteuert.

6.2 Implementierungsdetails

Der Explorationsprototyp (EP) besteht aus nur einer Datei, wird direkt in der *SystemC*-Hauptfunktion (*sc_main*) konstruiert und befindet sich im Unterverzeichnis *./platforms/leon3mp* der VP. Zum besseren Verständnis der Funktionsweise werden hier einige Details der Implementierung näher beschrieben. Es empfiehlt sich, diese Sektion gemeinsam mit dem Quellcode zu lesen. Auf einen vollständigen Abdruck wird aus Platzgründen verzichtet.

Der LEON2/3MP ist ein normales *SystemC*-Programm und entsprechend konstruiert. Das Programm instantiiert jedoch verschiedene Infrastrukturkomponenten, welche die Konfiguration, Simulation und Analyse des Systems erleichtern. Aufgrund des einfachen Aufbaus ist der LEON2/3MP einfach erweiterbar und kann auch als Vorlage zur Implementierung eigener Prototypen verwendet werden. Der EP LEON2/3MP besteht aus drei Sektionen: einer Kurzbe-

schreibung, dem *Include*-Abschnitt und der *SystemC*-Hauptfunktion, die sich wiederum in vier Unterabschnitte gliedert.

1. **Kurzbeschreibung:** Die ersten Zeilen der Datei enthalten eine Kurzbeschreibung des Prototypen, die Revisionsnummer und einen Abdruck der *ESA Special License*. Das System darf zum gegenwärtigen Zeitpunkt frei genutzt, kopiert, modifiziert und weiterverteilt werden.

2. ***Include*-Abschnitt:** Jede Komponente der Modellbibliothek wird in einem separaten *SystemC-Header* beschrieben. Diese *Header* werden in *sc_main* inkludiert, um die entsprechenden Klassen im System bekannt zu machen. Beispiele für Komponentenklassen sind *ahbctrl.h* (AHB-Bus), *gptimer.h* (*General Purpose Timer*) oder *mmu_cache.h* (CPU-Cachesystem). Des Weiteren werden verschiedene C/C++-*Header* geladen, die den Zugriff auf Container der *Standard Template Library* (z.B. *vector.h*), die Systemzeit (z.B. *sys/time.h*) und *Boost*-Funktionen (z.B. *boost/program_options.hpp*) ermöglichen. Wie in jedem *SystemC*/TLM-Programm müssen darüber hinaus die *Header systemc.h* und *tlm.h* inkludiert werden.

```
1  // SECTION 2 - Includes
2  #include <systemc.h>                 // Mandatory headers
3  #include <tlm.h>
4   ...
5  #include <iostream>                  // Standard C/C++ headers
6  #include <vector>
7   ...
8  #include <greencontrol/config.h>     // Infrastruktur headers
9  #include <json_parser.h>
10 #include <exec_loader.hpp>
11  ...
12 #include "ahbctrl.h"                 // Model headers
13 #include "mmu_cache.h"
14  ...
```

Abbildung 6.2: LEON2/3MP Prototyp - *Include*-Abschnitt

3. ***SystemC*-Hauptfunktion (*sc_main*)**: Die Hauptfunktion instantiiert verschiedene Helfer- und Infrastrukturklassen mit deren Hilfe Kommandozeilenparameter ausgewertet, Binärdateien (SPARC-Software) geladen und Systemkonfigurationen (JSON-Dateien) eingelesen werden können. Im Anschluss wird der globale Namensraum zur Adressierung der Systemparameter (*GreenControl-Middleware*) aufgebaut. Danach wird die in Abschnitt 5.1.1 beschriebene bedingte Instanziierung und Bindung von Komponenten durchgeführt. Das Ende der Datei bildet ein Abschnitt zur Simulationskontrolle.

   a) **Konfiguration und Parameterübergabe:** Der Auswertung der Kommandozeilenparameter kommt in sofern besondere Bedeutung zu, da mit ihrer Hilfe nicht nur einfache Optionen ausgewertet, sondern alle Systemparameter überladen werden können. Dazu werden die angegebenen Parameter mit Hilfe des *Boost*-Kommandozeilenparsers (*boost::program_option::command_line_parser*) zunächst in eine globale Datenstruktur überführt (*boost::program_options::variables_map*) und geordnet. Eine Liste aller Kommandozeilenoptionen kann durch den Aufruf der Simulation mit dem Kommando *–help* generiert werden:

```
tschuster@diebels:$ ./build/platforms/leon3mp/leon3mp.platform --help

        SystemC 2.2.0 --- Mar 23 2012 15:15:32
    Copyright (c) 1996-2006 by all Contributors
```

```
                    ALL RIGHTS RESERVED

SoCRocket -- LEON3 Multi-Processor Platform

Usage: ./build/platforms/leon3mp/leon3mp.platform [options]

Options:
  --help                              Shows this message
  -j [ --jsonconfig ] arg             Name of the configuration file
                                      (default: config.json)
  -o [ --option ] arg                 Additional configuration options
                                      (for overloading system parameters)
  -a [ --argument ] arg               Arguments for software running on
                                      the processors
  -l [ --listoptions ]                Shows a list of all options/parameters
  -f [ --listoptionsfiltered ] arg    Shows a filtered list of
                                      all options/parameters
  -s [ --saveoptions ] arg            Save options to JSON-file
                                      (default: config.json)
```

Der Parameter *-j* ermöglicht die Übergabe einer JSON-Konfigurationsdatei. Der Name der Datei wird an den integrierten Parameterparser weitergereicht (siehe *common/json_parser.h/cpp*):

```
jsonreader->config(json.string().c_str)
```

Der JSON-Parser greift direkt auf die Konfigurations-*Middleware* zu und initialisiert alle Parameter, für die eine Beschreibung vorhanden ist. Im Anschluss werden alle auf der Kommandozeile übergebenen Parameter manuell überladen. Dazu durchsucht das Programm die durch den *Boost*-Kommandozeilenparser aufgebaute Parameterliste. Gültige Parameterinitialisierungen haben das Format *–option parameter_name=value* und können der Konfigurations-*Middleware* mit Hilfe der Funktion *set_init_value* übergeben werden:

```
mApi->setInitValue(parameter_name, value);
```

Die Initialisierung mit Kommandozeilenparametern hat somit höhere Priorität als die Konfigurationsdatei. Beide Methoden überschreiben jedoch die bei der Parameterinstanzierung festgelegten Standardwerte. Zusätzliche Informationen zur Konfigurations-*Middleware* können den Abschnitten 3.5.2 und 3.7.3 entnommen werden. Eine detaillierte Beschreibung aller Schnittstellenfunktionen befindet sich in [Sch11].

b) **Aufbau des Namensraumes (*GreenControl Namespace*):** Im nächsten Schritt wird mit Hilfe von *GreenControl* der globale Namensbaum zur Adressierung von Konfigurations- und Analyseparametern aufgebaut. Wie bereits in Abschnitt 5.1.1 beschrieben, wird dabei zwischen Modell- und Systemparametern unterschieden. Die Wurzel des Namensbaumes im LEON2/3MP bildet das Parameterfeld *p_conf*. Diesem werden je ein weiteres Parameterfeld zur Aufnahme allgemeiner Systemparameter (*p_system*) und Steuerung der Generierung von Simulationsberichten (*p_report*) zugeordnet:

```
gs::gs_param_array p_conf("conf");
gs::gs_param_array p_system("system", p_conf);
gs::gs_param_array p_report("report", p_conf);
```

Die Systemparameter des LEON2/3MP sind in Tabelle 6.1 zusammengefasst.

Parameter	Beschreibung	Standardeinst.
p_system_at	Abstraktionsebene der Simulation (LT oder AT)	LT
p_system_ncpu	Anzahl der Prozessoren im System	1
p_system_clock	Taktrate in ns	10
p_system_osemu	Binärdatei mit Symbolen zur OS-Emulation[1]	keine OS-Emulation
p_system_log	Logdatei erzeugen (Name)	kein Logfile
p_system_gdb_en	GDB-Stubs einschalten	nein
p_system_gdb_port	Basisport des Debuggers (CPU 0)	1500
p_report_timing	*Timingmonitor* erzeugt Bericht für alle Simulationsphasen	ja
p_report_power	*Powermonitor* erzeugt Bericht zum Energieverbrauch	nein

Tabelle 6.1: Systemparameter im LEON2/3MP

Modellparameter werden zur besseren Übersicht gemeinsam mit den zugehörigen Simulationsmodellen instanziiert. Für jedes Modell wird dazu ein individuelles Parameterfeld erzeugt und ähnlich wie *p_system* oder *p_report* direkt unter *p_conf* im Parameterbaum integriert.

c) **Bedingte Instanziierung/Bindung von Komponenten:**

Im LEON2/3MP werden Simulationsmodelle abhängig von System- und Modellparametern bedingt instantiiert. Dadurch kann auf einfache Weise und ohne erneutes Kompilieren eine große Anzahl verschiedener Prototypen generiert und erkundet werden. Konfigurationsparameter werden direkt in *sc_main* erzeugt und auf die Konstruktorparameter des entsprechenden Modelles abgebildet. Diese entsprechen zu weiten Teilen dem Vorbild der Hardwareimplementierung aus der GRLIB (*Generics* in VHDL). Zunächst wird das aus dem AHB-Bus und der AHB/APB-Busbrücke bestehende AMBA-*Interconnect* aufgebaut (Abschnitt 4.2). Auszüge der Instanziierung sind in Abbildung 6.3 dargestellt.

```
1  // AHBCTRL
2  // Konfigurationsparameter
3  gs::gs_param_array p_ahbctrl("ahbctrl", p_conf);
4  gs::gs_param<unsigned int> p_ahbctrl_ioaddr("ioaddr", 0xFFF, p_ahbctrl);
5  gs::gs_param<unsigned int> p_ahbctrl_iomask("iomask", 0xFFF, p_ahbctrl);
6  ...
7  // Instanziierung
8  AHBCtrl ahbctrl("ahbctrl",
9     p_ahbctrl_ioaddr, // The MSB address of the I/O area
10    p_ahbctrl_iomask, // The I/O area address mask
11    ...
12    ambaLayer         // Abstraktionsebene (LT oder AT)
13 );
14
15 // Taktannotation (clock_device)
16 ahbctrl.set_clk(p_system_clock, SC_NS);
17
18 // APBCTRL
19 // Konfigurationsparameter
20 gs::gs_param_array p_apbctrl("apbctrl", p_conf);
21 gs::gs_param<unsigned int> p_apbctrl_haddr("haddr", 0x800, p_apbctrl);
22 ...
23 // Instanziierung
24 APBCtrl apbctrl("apbctrl",
25    p_apbctrl_haddr,  // The 12 bit MSB address of the AHB area.
26    ...
27    ambaLayer         // Abstraktionsebene (LT oder AT)
28 );
29
30 // Taktannotation (clock_device)
31 apbctrl.set_clk(p_system_clock, SC_NS);
32
33 // Bindung an AHBCTRL
34 ahbctrl.ahbOUT(apbctrl.ahb);
```

Abbildung 6.3: LEON2/3MP Prototyp - Instanziierung des AMBA-*Interconnects*

Die Modellparameter des AHBCTRL werden in den Zeilen 3-5 erzeugt. Im Anschluss wird der Bus instantiiert (Zeilen 8-13). Dabei werden die Modellparameter an den Konstruktor übergeben. Außerdem wird dem Konstruktor die aus dem Systemparameter *p_system_at* (Tabelle 6.1) abgeleitete Variable *ambaLayer* zugewiesen, mit der das Abstraktionsniveaus des Modelles eingestellt werden kann. Zur Übergabe des Systemtaktes wird in Zeile 16 die Funktion *set_clk* der von der Bibliotheksbasisklasse *clock_device* geerbten *Timing*-Schnittstelle aufgerufen. Die Instanziierung des APBCTRL erfolgt auf ähnliche Art und Weise (Zeilen 18-31). Das Modell muss jedoch als *Slave* an den Sockel *ahbOUT* von AHBCTRL gebunden werden (Zeile 34). Im Anschluss werden der *Multi-Processor Interrupt Controller* (IRQMP), der *General Purpose Timer* (GPTimer) und der UART (APBUART) generiert. All diese Komponenten sind APB-*Slaves* und werden an die AHB/APB-Busbrücke gebunden. Das Beispiel verdeutlicht dies anhand des APBUART:

```
apbctrl.apb(apbuart->bus)
```

Der IRQMP wird in jedem Fall erzeugt, da er für den Betrieb des Systems zwingend erforderlich ist. Die Instanziierung des GPTimer hängt dagegen vom Parameter *p_gptimer_en* ab. Befindet sich der GPTimer im System, so werden seine Ausgänge automatisch an die entsprechenden Eingänge des IRQMP verbunden. Dazu kommt die

in Abschnitt 3.2.5 beschriebene Methode zur Modellierung von Signalkommunikation zum Einsatz. Die Anzahl der UARTs im System ist frei definierbar. Dazu muss die Identifikationsnummer (ID) und der gewünschte Typ der jeweiligen Schnittstelle auf der Kommandozeile oder in der Konfigurationsdatei übergeben werden. Die Definition des folgenden Parameters erzeugt einen UART mit der ID 2 vom Typ 1:

```
conf.uart.2.type=1
```

Gegenwärtig sind zwei UART-Typen verfügbar: ein Null-*Device* (Typ 0) und ein TCP-*Client* (Typ 1). Die ID dient im Falle des TCP-*Clients* als *Offset* der Port-Adresse (2000 + ID). Alle auf diese Weise erzeugten UARTs können durch ihre *Plug & Play*-Register am Bus eindeutig identifiziert werden. Dadurch ist es möglich, jedem Prozessor in einem Mehrprozessorsystem eine exklusive Ausgabeschnittstelle zuzuordnen. Die nächste im Quellcode instantiierte Komponente ist der Speichercontroller MCTRL. Er verfügt sowohl über eine AHB- als auch eine APB-Schnittstelle und muss somit an beide Busse gebunden werden:

```
ahbctrl.ahbOUT(mctrl.ahb);
apbctrl.apb(mctrl.apb);
```

Darüber hinaus muss der Speichersockel des MCTRL (*mem*) mit mindestens einem Speicher bestückt werden. Der LEON2/3 erzeugt standardmäßig vier Simulationsspeicher zur Modellierung von ROM, I/O, SRAM und SDRAM. Ob diese Speicher effektiv genutzt werden, hängt von den individuellen Adresseinstellungen ab. Der MCTRL hält für jeden Speichertyp ein AHB-Adresse/Maske-Paar bereit (z.B. *p_mctrl_prom_addr/p_mctrl_prom_mask*). Außerdem können Speicherbereiche durch das Beschreiben der Steuerregister des MCTRL ein- und ausgeschaltet, sowie unterschiedlich angeordnet werden (siehe Abschnitt 4.3.3). Davon unabhängig ist es möglich, jeden Speicherbereich mit einer Binärdatei zu initialisieren. Dafür wird der im *Trap*-LEON-Simulator (IU) enthaltene ELF-*Loader* eingesetzt. Bei Verwendung der proprietären Werkzeugkette von *Aeroflex Gaisler*, kann das mit *mkprom2* erzeugte *Image* zum Simulationsbeginn in das ROM geladen werden. Die integrierten *Low-Level*-Routinen entpacken den Code, kopieren ihn in den RAM-Bereich und springen anschließend zur Hauptfunktion. Alternativ kann ein mit einem SPARC-Compiler erzeugtes Programm direkt in den RAM geladen werden. In diesem Fall muss das ROM mit einem Boot-Code initialisiert werden, der einige grundsätzliche Initialisierungen vornimmt und den Programmeintrittspunkt anspringt. Eine Vorlage, die in fast allen Fällen unverändert übernommen werden kann, befindet sich im Verzeichnis *./software/prom/default*. Ein ROM-*Image* kann mit dem Parameter *p_mctrl_prom_elf* übergeben werden. RAM-*Images* für SRAM oder SDRAM werden mit Hilfe von *p_mctrl_ram_sram_elf* oder *p_mctrl_ram_sdram_elf* geladen. Das Prinzip der Instanziierung und Bindung von Prozessoren in Abhängigkeit des Systemparameters *p_system_ncpu* wurde bereits in Abschnitt 5.1.2 (Abbildung 5.2) auf generische Weise erläutert. Die im LEON2/3MP gegebene Implementierung ist in sofern aufwendiger, da für jeden Prozessor ein vollständiges Cachesystem (*mmu_cache*) instantiiert wird. Jeder Prozessor wird über zwei TLM-Sockel an das Cachesystem und jedes Cachesystem über einen AHB-*Master*-Sockel an den AHB-Bus (AHBCTRL) gebunden. Außerdem werden die Interruptschnittstellen der Prozessoren mit dem Interruptcontroller und die *Snooping*-Ports der Caches mit dem AHBCTRL verbunden. Durch zusätzliche Einstellungen in diesem Codeabschnitt können die Startadressen der Simulatoren festgelegt, die GDB-Server zum Debuggen von auf den Prozessoren ausgeführter Software gestartet und die OS-Emulation aktiviert werden.

d) **Simulationskontrolle:** Der letzte Abschnitt der Datei enthält Funktionen zur Simulationskontrolle. Hier wird unter anderem *sc_start* aufgerufen. Darüber hinaus

instantiiert der Prototyp die Helferklasse *AHBProf*. Diese stellt eine reguläre Komponente dar, die über eine AHB-*Slave*-Schnittstelle mit dem Bus verbunden ist. Durch das Beschreiben des Adressbereiches von *AHBProf* kann der Nutzer die Simulation per Software steuern. Mögliche Aktionen sind die Beendigung der Simulation (*sc_stop*), das Starten und Stoppen von *Timern* zur Messung der Simulationszeit und die Generierung von Berichten. Die Beendigung der Simulation per Software ist besonders für Architekturexplorationen mit Betriebssystem von Bedeutung, da diese nach Abarbeitung eines *Tasks* keinen Rücksprung aus der Hauptfunktion ausführen, sondern bis zur Zuweisung einer neuen Aufgabe in einen Leerlauf-*Task* schalten. Des Weiteren wird im Simulationskontrollabschnitt der in 3.6 beschriebene *Power Monitor* instantiiert. Zur Aktivierung des *Power Monitors* im LEON2/3MP muss der Parameter *p_report_power* gesetzt werden. Je nach Bedarf können an dieser Stelle weitere Infrastrukturkomponenten zur Unterstützung von Debugging und Analyse eingebunden werden. Eine detaillierte Beschreibung der dafür in *SoCRocket* bereitgestellten Methoden befindet sich in Abschnitt 3.7.

6.3 Simulationsergebnisse

Zur Bestimmung der Simulationsgenauigkeit und -geschwindigkeit der Virtuellen Plattform wurde ein den Standardeinstellungen des LEON3MP aus der GRLIB[1] entsprechender Prototyp konfiguriert. Das System soll im LT- und im AT-Modus mit der RTL-Referenz verglichen werden. Die dafür erforderliche JSON-Datei befindet sich im Verzeichnis *./templates* (siehe Anhang B). Die wichtigsten Einstellungen und Anpassungen sind in Tabelle 6.2 zusammengefasst.

Komponente	VP-Parameter	Einstellung
SPARC v8 CPU	p_system_ncpu	1
	p_system_clock	50 MHz
I-Cache	ic.sets	1
	ic.setsize	2 (4kB)
	ic.linesize	8
D-Cache	dc.sets	1
	dc.setsize	2 (4kB)
	dc.linesize	8
AMBA-Bus	rrobin	1
MCTRL	sden	1
	prom.size	512 MB
	sdram.size	512 MB
	sram.size	1014 MB

Tabelle 6.2: Standardeinstellungen des LEON3MP

Der UART im RTL-System wurde mit Hilfe von *xconfig* für schnelle Simulation konfiguriert. In dieser Einstellung werden eingehende Daten (UART-Datenregister) sofort an die Standardausgabe weitergeleitet und der Transmitter ist immer zum Empfang neuer Daten bereit (*Transmitter-Ready-Signal* immer *high*). Dadurch wird sichergestellt, dass sich I/O-Operationen, wie zum Beispiel Testausgaben, nicht auf die Simulationszeit auswirken. Diese Interpretation des *Timings* entspricht der UART-Implementierung der VP, die ebenfalls das Verhältnis zwischen Systemtakt und Baudrate zur Steigerung der Simulationsgeschwindigkeit ignoriert.

Die verwendeten Benchmarks entstammen der *PowerStone*-Suite [Sco98] und befinden sich im Verzeichnis *software/trapgen*. Tabelle 6.3 fasst die enthaltene Funktionalität kurz zusammen. Alle Benchmarks wurden von Dateioperationen befreit. Dazu wurden die Eingabedaten in den

1 Version GRLIB-GPL-1.3.7-b4144

Quellcode inkludiert. Außerdem wurden die Programme mit Hilfe des Schalters SHORT_BENCH im Umfang beschränkt, um auf dem RTL-Referenzsystem in angemessener Zeit simuliert werden zu können.

Benchmark	Beschreibung
FIR2	FIR-Filter von Texas Instruments
Engine	Berechnung des Luftstromes aus einem 8 Bit weitem A/D-Wert und der Umdrehungsgeschwindigkeit eines Motors
CRC	Berechnung und Kontrolle von CRC-Testsummen (10000 Iterationen)
DES	Verschlüsselungsalgorithmus
FFT	Fast Fourier Transformation
JPEG	Bildkompression
Hanoi	Lösen: Türme von Hanoi
Quick	Quicksort Sortieralgorithmus

Tabelle 6.3: Benchmarks Überblick

Mit Hilfe von BCC und *mkprom2* wurden unkomprimierte ROM-*Images* und entsprechender Bootcode erstellt:

```
// 1. Kompilieren und linken mit Sparc-Cross-Compiler
sparc-elf-gcc -msoft-float -O2 benchmark.c -o benchmark.exe

// 2. Unkomprimiertes Boot-Image generieren (erzeugt prom.out)
mkprom2 -rmw -nocomp -msoftfloat -v -ramsize 1024 benchmark.exe

// 3. Boot-Image für RTL-Simulation in Motorola-SREC transformieren
sparc-elf-objcopy --output-target=srec prom.out prom.srec
```

Die Simulationen im TLM- und im RTL-System verwenden das gleiche Programm-*Image*, ohne jegliche Modifikationen. Die Prozessoren booten aus dem ROM, kopieren das Programm in den RAM und führen dieses im Anschluss aus. Vor dem Sprung zur Hauptfunktion werden die Caches aktiviert. Dabei wird der Instruktionscache durch Beschreiben des *Cache Control Registers* in den *Burst Fetch*-Modus versetzt. Im Falle eines *Read Miss* wird so ein inkrementeller Burst variabler Länge erzeugt, der die Cachezeile ausgehend von der aktuellen Adresse neu befüllt. Der Datencache erzeugt automatisch Bursts fester Länge zum Laden kompletter Cachezeilen. Dieses Verhalten wurde in *GRLIB* ab Version 1.3.0 eingeführt. Nach Abarbeitung des Programmes verlässt der Prozessor die Hauptfunktion (*main*) und terminiert mit einer ungültigen Instruktion, die eine Ausnahme (*Exception*) auslöst.

Simulationsgenauigkeit

Die in Abbildung 6.4 dargestellten Simulationsergebnisse schließen den *Boot*-Vorgang des Prozessors ein. Diese Einstellung wurde gewählt, da während des Startvorganges besonders viele Spezialoperationen, wie *Bypass*-Zugriffe bei abgeschalteten Caches oder Kopieroperationen, die in kurzer Zeit eine hohe Anzahl an Busoperationen erzeugen, auftreten. Alle Tests wurden auf einem Server mit *Xeon*-Prozessoren, 3.5 GHz Taktfrequenz und 32 GB RAM (6933 BogoMips) unter Linux (CentOS 2.6.32-220.17.1.el6.x86_64) durchgeführt.

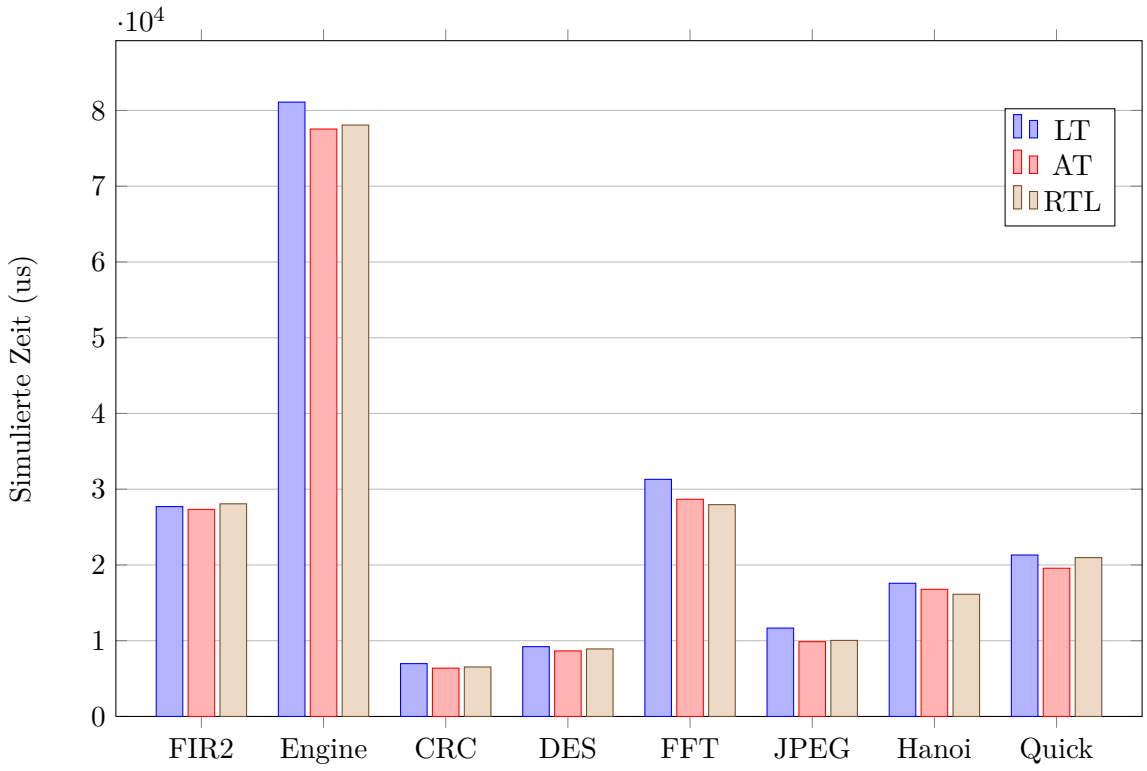


Abbildung 6.4: Absolute Benchmark-Simulationszeiten

Es ist ersichtlich, dass sowohl LT- als auch AT-System eine sehr hohe Genauigkeit aufweisen. Im AT-Modell beträgt die größte Abweichung 6.68% (Quick) und im LT-Modell 11.95% (FFT). Die Höhe des Fehlers zwischen LT- und AT-Modus ist nicht korrelliert. Maxima und Minima treten für unterschiedliche Benchmarks auf. In Ausnahmefällen, wie zum Beispiel FIR2, erreicht das LT-Modell eine höhere Genauigkeit als das AT-Modell. Dies ist durchaus möglich, wenn die für das Zeitverhalten im LT-Modus getroffenen Abschätzungen der Instruktionsabfolge des untersuchten Programmes sehr gut entsprechen. Für den praktischen Einsatz des Systems sind jedoch nicht der absolute Fehler einzelner Benchmarks, sondern die Verlässlichkeit der Simulationszeiten von Bedeutung. Diese kann durch den durchschnittlichen prozentualen Fehler aller Tests oder durch die statistische Standardabweichung ausgedrückt werden. Der **durchschnittliche prozentuale Fehler** beträgt **3.03% im AT-Modus** und **7.04% im LT-Modus**. Zur Ermittlung der Standardabweichung wurden die einzelnen Simulationszeiten mit Hilfe der Referenzergebnisse aus der RTL-Simulation normiert. Dadurch ergeben sich für den **LT-Modus Erwartungswerte von 1.06 und für den AT-Modus von 0.99**. Daraus leitet sich für den LT-Modus eine Varianz von 0.0028 und eine **Standardabweichung von 0.053** ab. Die Varianz des AT-Modelles beträgt 0.0011 und die **Standardabweichung 0.033**, was einer wesentlich höheren statistischen Verlässlichkeit entspricht.

Aus den Erwartungswerten folgt, dass das LT-System tendenziell längere und damit pessimistischere Abschätzungen der Simulationszeit liefert. Grund dafür ist die geringere inherente Parallelität der Modelle, insbesondere die zeitliche Entkopplung des Prozessorsimulators und die vereinfachte Darstellung von AMBA-Bustransfers (siehe Abschnitt 3.2.2). Durch ein geringeres Maß an Parallelität addieren sich Verzögerungen entlang des Pfades einer Transaktion, die sich andererseits in unabhängigen *Threads* überlagern. Die Entwicklung des Fehlers über der Simulationszeit, ist in Abbildung 6.5 anhand der JPEG-Kompression exemplarisch dargestellt. Während die Simulationszeit des AT-Modell zu jeder Zeit sehr genau der RTL-Referenz entspricht, liefert das LT-Modell eine zu pessimistische Abschätzung. Der größte Anteil des Fehler bildet sich zwischen Instruktionen 70000 und 90000. Im LT-Modell entspricht dies dem Simulationszeitraum von 300 μs bis 550 μs. Die in Abbildungen 6.6-6.9 dargestellten Histogramme für Instruktions- und Datenzugriffe des Prozessors verdeutlichen, dass in diesem Zeitraum große

Mengen an Daten in den Speicher geschrieben werden. Im gleichen Zeitraum gibt es jedoch fast keine Lesezugriffe. Gründe für die Abweichung an dieser Stelle sind die zeitliche Entkopplung des Prozessorsimulators und die vereinfachte Modellierung des Schreibpuffers im LT-Modus der AHB-*Master*-Schnittstelle des Cachesystems (Abschnitt 4.1). Da der Prozessorsimulator der Systemzeit zur Erzielung einer höheren Simulationsgeschwindigkeit vorauseilt, werden die durch die Schreiboperationen im übrigen System akkumulierten Verzögerungen zum Zeitquantum des Prozessors addiert, anstatt am Schreibpuffer parallel verbraucht zu werden.

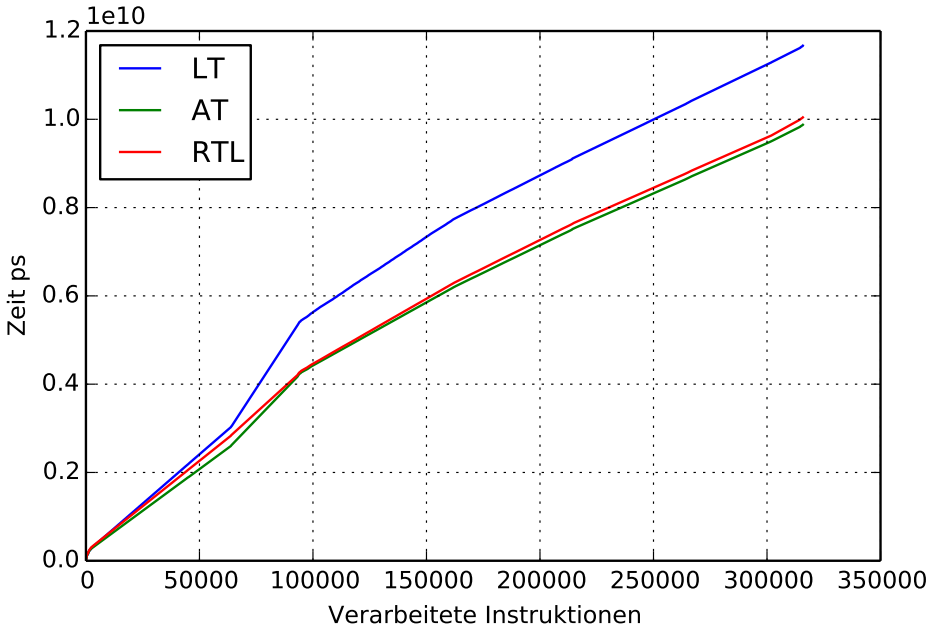


Abbildung 6.5: JPEG - Simulationsverlauf

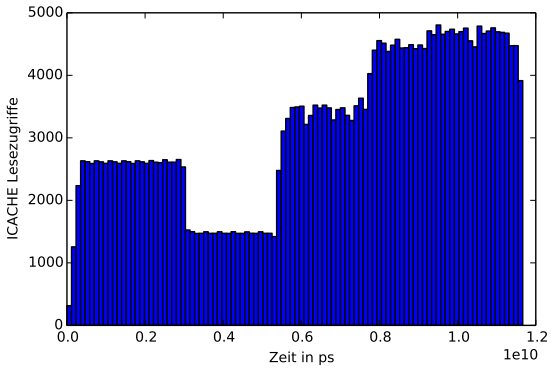


Abbildung 6.6: JPEG - Histogramm für ICACHE-Lesezugriffe

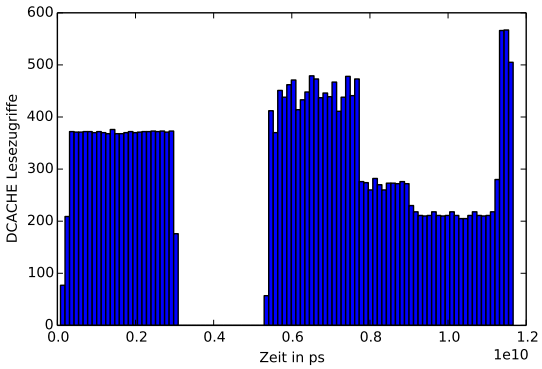


Abbildung 6.7: JPEG - Histogramm für DCACHE-Lesezugriffe

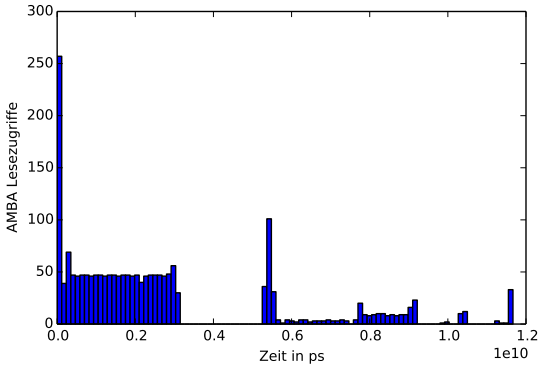


Abbildung 6.8: JPEG - Histogramm AHB-Lesezugriffe

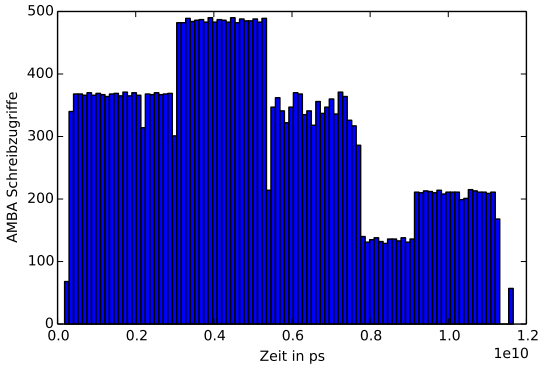


Abbildung 6.9: JPEG - Histogramm für AHB-Schreibzugriffe

Ähnliche auf den abstrakteren Modellierungsstil der LT-Modelle zurückzuführende Effekte können den Simulationsverläufen der übrigen Benchmarks entnommen werden. Die entsprechenden Grafiken sind in Anhang C beigelegt.

Simulatorgeschwindigkeit

Reduzierte Parallelität verringert die Anzahl an Kontextumschaltungen im *SystemC*-Kern, der zur Ausführung einer Simulation erforderlich ist und steigert dadurch die Simulationsgeschwindigkeit. Dies wird in Abbildung 6.10 deutlich. Die dargestellten Messwerte entsprechen der reinen Programmlaufzeit. Zeiten für den Start und die Initialisierung des Simulators wurden nicht aufgenommen.

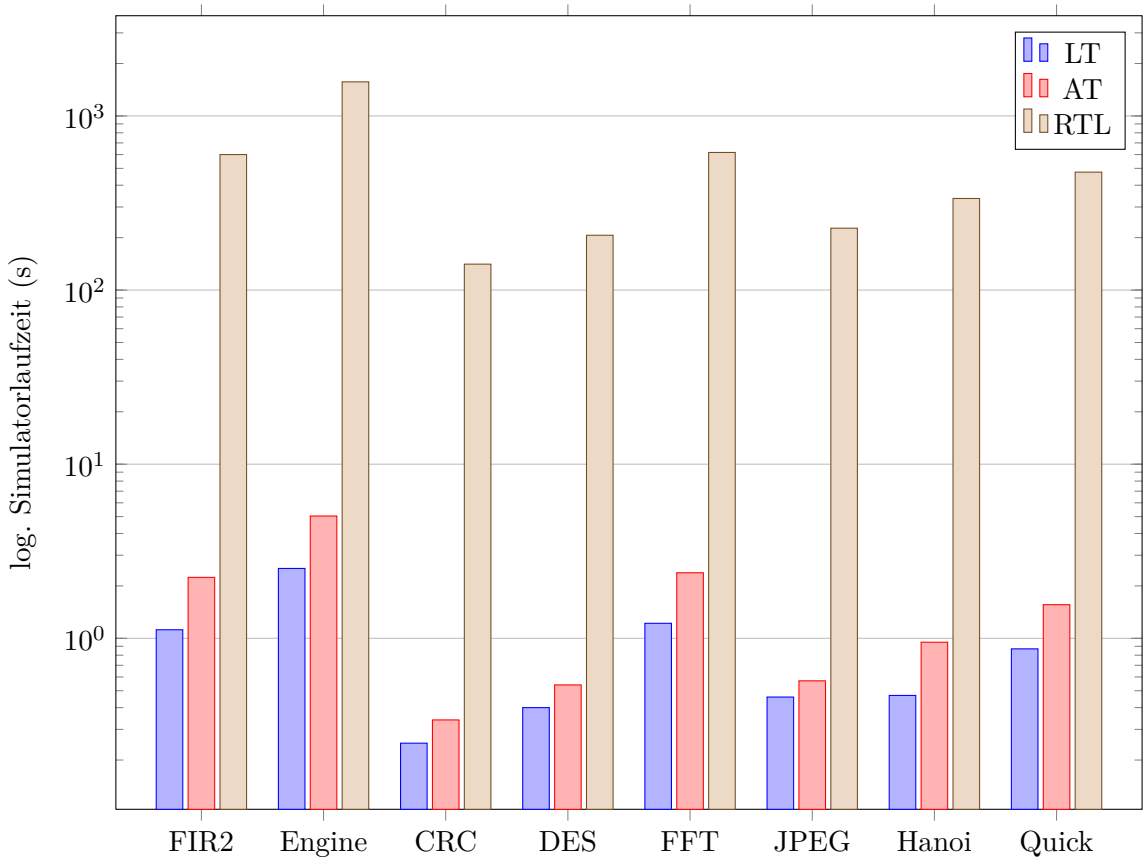


Abbildung 6.10: Simulatorlaufzeit (Echtzeit)

Das LT-System simuliert im Durchschnitt 561 mal schneller als die RTL-Referenz. Der

Geschwindigkeitsgewinn für das AT-System beträgt Faktor 335. In beiden Modi zeigen Hanoi die größte (LT: 712, AT: 549) und FIR2 die kleinste Beschleunigung (LT: 588, AT: 297). Dies ist die höhere Anzahl an Cache-*Misses* und Schreibzugriffen in Hanoi zurückzuführen. In Hanoi werden insgesamt 416709 Instruktionen verarbeitet, die 18642 Bustransaktionen auslösen. In FIR2 ist die Gesamtanzahl der Instruktionen wesentlich höher (1203873), trotzdem sind nur 9975 Bustransaktionen erforderlich. Durch den höheren Anteil an Bustransaktionen verlängert sich die durchschnittliche Pfadlänge der Transaktionen und das Potential zur Beschleunigung gegenüber den niedrigeren Abstraktionsstufen steigt.

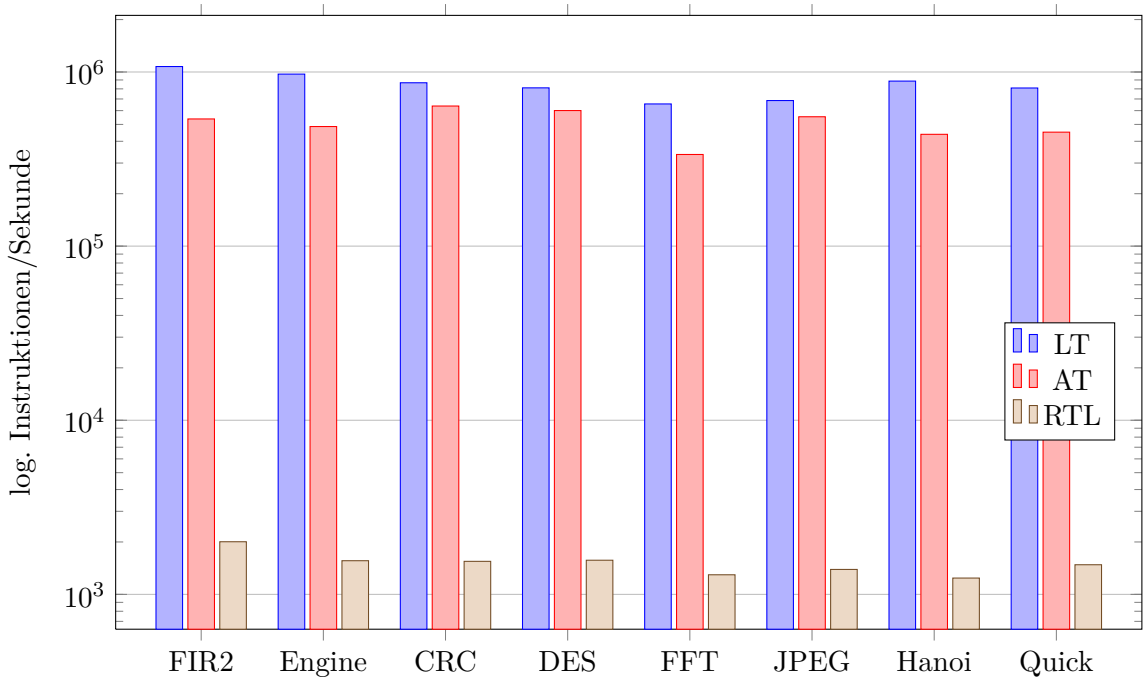


Abbildung 6.11: Simulatorgeschwindigkeit

Wie in Abbildung 6.11 dargestellt, beträgt die Geschwindigkeit der RTL-Referenzsimulation im Durchschnitt ca. 1500 Instruktionen pro Sekunde. Das **AT-Simulationsmodell** verarbeitet im Durchschnitt **505145** und das **LT-Simulationsmodell 845279 Instruktionen pro Sekunde**. Die höchste Geschwindigkeit, mit 1074890 Instruktionen pro Sekunde wurde im Test FIR2 gemessen. Das Ergebnis kann wiederum durch den verhältnismäßig geringen Anteil an Bustransaktionen erklärt werden. Der für Cache-*Hits* durch Transaktionen im System zurückzulegende Weg ist sehr kurz. Dadurch können viele Transaktionen in kurzer Zeit verarbeitet werden.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die flexible erweiterbare Virtuelle Plattform *SoCRocket* entworfen, implementiert und evaluiert. Das System ermöglicht die Simulation von SoCs basierend auf industrietypischen Modulen mit einer Geschwindigkeit von bis zu einer Million Instruktionen pro Sekunde bei nahezu taktzyklischer Genauigkeit. Derzeit implementiert sind u.a. LEON2/3-Cores, diverse Kommunikationsbausteine und Speicherinfrastruktur. Die flexible Schnittstellenarchitektur erlaubt die jederzeitige Erweiterung existierender oder Aufnahme neuer Komponenten. Im Vergleich zu RTL-Implementierungen simulieren *SoCRocket*-Prototypen bis zu 500x schneller und weisen dabei einen durchschnittlichen prozentualen Fehler von nur 3% auf. *SoCRocket* eröffnet damit neue bisher unbekannte Möglichkeiten für Architekturexploration und den Entwurf hardwarenaher Software für robuste eingebettete Systeme.

7.1 Zusammenfassung

Virtuelle Plattformen sind Softwareimplementierungen prozessorbasierter Systeme und einer der momentan vielversprechendsten Ansätze zur Überwindung der Produktivitätsprobleme bei der Entwicklung komplexer Hardware und Software. Trotz langjähriger Forschungsaktivitäten und vereinzelter Erfolge, speziell im Mobilfunkbereich, werden Virtuelle Plattformen durch die Industrie bisher nur schleppend adaptiert. Ein Beispiel dafür ist die europäische Raumfahrt. Potentielle Gründe sind (1) die hohen Kosten für die Umstellung existierender *Workflows* und Anschaffung neuer Werkzeuge, (2) der Mangel an geeignetem Personal mit Expertise für abstrakte Entwurfsmethoden, (3) der Mangel an verlässlichen Simulationsmodellen und (4) die eingeschränkte Wiederverwendbarkeit abstrakter Modelle aufgrund fehlender Standards. Die vorliegende Arbeit adressiert diese latenten Probleme vor dem speziellen Hintergrund von Raumfahrtanwendungen durch die Entwicklung neuer offener Simulationsmodelle und Werkzeuge zur Konstruktion, Simulation und Analyse von Virtuellen Plattformen (1), vereinfachte Entwurfsmethoden zur Konstruktion von Modellen und Systemen auf unterschiedlichen Abstraktionsniveaus (2), umfassende Verifikation und Genauigkeitsuntersuchungen bezüglich der entwickelten Werkzeuge, Modelle und Techniken (3) und den konsequenten Einsatz standardisierter und standardoffener Lösungen, sowie deren gezielte Weiterentwicklung (4).

Zum Erreichen dieser Ziele wurden zunächst grundlegende Aspekte zur Konstruktion von Simulationsmodellen mit *SystemC*/TLM identifiziert und in Hinblick auf den Stand der Technik untersucht. Dazu zählen Aufbau und Struktur von Modellen, TLM-Schnittstellen, Modellierung von Speicherelementen und Verhalten, Verwaltung und Handhabung von Metadaten, sowie Modellierung des Energieverbrauchs, Debugging, Analyse und Verifikationsunterstützung.

Der Schlüssel zur erfolgreichen abstrakten Modellierung von Komponenten und Systemen liegt in der Entscheidung, welche Implementierungsdetails zur Steigerung der Simulationsgeschwindigkeit bei abschätzbarem Einfluss auf die Simulationsgenauigkeit vereinfacht oder weggelassen werden können. Diese Entscheidung erfordert Erfahrung und in vielen Fällen Kreativität und kann daher nicht vollständig automatisiert werden. Es ist jedoch durchaus möglich, die Grundfunktionalität eines Systems im Sinne von Busschnittstellen, Konfigurationsinformation, Speicher, Zeitverhalten oder Energieverbrauch in einheitlichen Basisklassen zu bündeln. Darauf aufbauend wurde ein generisches Grundgerüst zur Konstruktion von TL-Komponenten entwickelt. Das Gerüst gliedert sich in vier Sektionen: Kommunikation, Verhalten, Speicher und Metadaten. Die Struktur entsteht implizit durch Vererbung von Bibliotheksbasisklassen und bildet die Grundlage für alle im Rahmen dieser Arbeit entwickelten Modelle und Werkzeuge.

Ein erheblicher Teil der Schwierigkeit beim Entwurf von Simulationsmodellen besteht in der Abstraktion der Bus- und Signalkommunikation. Zur Modellierung von Datenübertragung und der Gewährleistung von Interoperabilität zwischen IP-Modellen stellt der TLM2.0-Standard ein generisches Basisprotokoll zur Realisierung näherungsweise akkurater Kommunikation bereit. Dieses Basisprotokoll definiert Transaktionen auf Grundlage generischer *Payload*-Objekte, ignorierbare Erweiterungen für Sekundärinformationen und vier Protokollphasen (*Timing*-Punkte) zur Modellierung einfacher zweiphasiger (*Request/Response*) *handshake*-basierter Punkt-zu-Punkt-Übertragungen. Moderne SoC-Kommunikationsprotokolle erfordern jedoch *Split*-Transaktionen oder mehrere unabhängige Kanäle (z.B. AXI: Lese-/Schreibadresse, Lese-/Schreibdaten, Schreib-*Response*), um geringe Latenz und hohen Durchsatz zu garantieren. Diese Protokolle können nicht auf einfache Weise interoperabel mit gleichzeitig akkuratem *Timing* modelliert werden: Hier müssen gegebenenfalls mehrere Kanäle in einer Schnittstelle gebündelt werden und es sind darüberhinaus Erweiterungen des Transaktionsobjektes an sich oder der vier Phasen des Standardprotokolls erforderlich. In der vorliegenden Arbeit wird eine Methode zur Modellierung derartiger Protokolle mit Hilfe des TLM-Standardprotokolls vorgestellt, die ohne zusätzliche Synchronisationspunkte und die Opferung zeitlicher Genauigkeit auskommt. Grundidee ist der Ausgleich des Intertransaktions-*Timings* durch die Auswertung später *Stall*-Bedingungen, welche nicht durch die Transaktion selbst aber durch ihre Nachfolger berücksichtigt werden können. Die Methodik wird am Beispiel der AMBA-Protokollfamilie erläutert.

Des Weiteren wird ein neues Konzept zur Realisierung von Signalkommunikation vorgestellt. Signalkommunikation in Virtuellen Plattformen wird gewöhnlich durch *SystemC*-Signale realisiert. Diese sind sehr hardwarenah, erfordern Synchronisation auf Zugriffsbasis, skalieren schlecht und sind somit für schnelle Simulation auf Transaktionsebene ungeeignet. Der präsentierte Ansatz kombiniert die hardwarenahe Handhabung von *SystemC*-Signalen mit dem schnellen auf direkten Funktionsaufrufen beruhenden Konzept von TLM.

Einer der ersten und wichtigsten Schritte zur Erstellung eines Plattformprototypen ist die Modellierung aller direkt und indirekt adressierbaren Speicherelemente. Trotz der Essenzialität dieses Problems existiert bis heute kein allgemein akzeptierter Standard. Eine Alternative zu existierenden proprietären Lösungen ist das *Open Source*-Projekt *GreenReg*, das auf einem von *Intel* entwickelten *Framework* zur Beschreibung von Registern aufbaut. Für die Integration in *SoCRocket* wurde *GreenReg* zur Verbindung mit den verwendeten AMBA/TLM-*Sockets* angepasst. Dadurch können Registerbänke direkt an *Sockets* registriert und ohne zusätzlichen Aufwand gelesen und beschrieben werden. Nach bestem Wissen des Autors ist *SoCRocket* die erste VP, die *GreenReg* zur Modellierung von Registern in allen Komponenten konsequent und konsistent einsetzt.

Die größte Herausforderung bei der abstrakten Modellierung von SoC-Komponenten besteht in der Darstellung des Verhaltens. Dazu können unterschiedliche Hochsprachen und verschiedene *Models of Computation* (MoCs) eingesetzt werden. Gängige Varianten sind Datenflussgraphen, *Discrete Event*- oder *Continuous Time*-Beschreibungen, Kahn-Prozessmodelle oder synchrone/reaktive Modelle. Für den Entwurf von SoC-Komponenten für Virtuelle Plattformen setzt sich jedoch zunehmend die funktionale Beschreibung mit *SystemC* durch. Die Grundvoraussetzung für die Akzeptanz von *SoCRocket* in der europäischen Raumfahrt ist die Verfügbarkeit geeigneter Simulationsmodelle. Zum Aufbau einer entsprechenden Modellbibliothek mussten daher wichtige Kernkomponenten, die bisher nur als VHDL-Code verfügbar waren, mit Hilfe von *SystemC* beschrieben werden. In dieser Arbeit wurde dazu eine Methodik entwickelt, die ausgehend von der Analyse der Spezifikation und der VHDL-Beschreibung die systematische Konstruktion abstrakter Modelle ermöglicht. *SoCRocket*-Modelle entstehen durch Vererbung von Bibliotheksbasisklassen und anschließende Zuordnung von Funktionsabläufen zu Schnittstellen oder Speicherelementen.

Ein weiterer bisher in der Standardisierung vernachlässigter Aspekt zur Konstruktion von TL-Simulationsmodellen ist die Verwaltung und Handhabung von Metadaten. Darunter versteht man alle Informationen, die der Einstellung, Beschreibung und Konfiguration von Modellen dienen oder zu deren Analyse annotiert werden. Einer der erfolgversprechendsten Ansätze auf

diesem Gebiet ist die im Vorfeld dieser Arbeit durch Christian Schröder an der TU Braunschweig entwickelte *GreenControl-Middleware*. *GreenControl* organisiert die Parameter eines Systems in einem globalen Namensraum und macht diese mit Hilfe einer universellen Schnittstelle für beliebige Werkzeuge zugänglich. *SoCRocket* nutzt und erweitert *GreenControl* für die dynamische Instanziierung von Komponenten. Durch die vorgenommen Erweiterungen können Konfigurationen in JSON-Sprache zur Laufzeit geladen werden. *SoCRocket*-VPs können daher ohne erneutes Kompilieren komplett rekonfiguriert werden. Dadurch wird insbesondere in der Architekturexploration wertvolle Simulationszeit eingespart. *GreenControl* fließt in den geplanten *Accellera*-Standard für *Control, Configuration & Inspection* (CCI) ein, wurde aber in seiner bisherigen Form noch nicht produktiv eingesetzt. *SoCRocket* nimmt hier daher eine Vorreiterrolle ein.

Neben dem Zeitverhalten gewinnt die frühzeitige Abschätzung des Energieverbrauchs immer mehr an Bedeutung. Für die Entwicklung entsprechender Methoden stellen neben der Verifikation, Konsistenz und Geschwindigkeit die größten Probleme dar. Bisher wurden unterschiedlichste Ansätze verfolgt. Ein Standard existiert nicht. Als standardoffene Lösung wird in *SoCRocket* eine an *GreenControl*/CCI gekoppelte Methode vorgeschlagen. Dazu wird jedes Modell mit einer global adressierbaren Parameterschnittstelle ausgestattet. Die Energieberechnung erfolgt auf Grundlage normalisierter Energie- und Leistungswerte, die zum Beginn der Simulation mit Hilfe der aktuellen Konfiguration skaliert werden. Das Simulationsmodell zählt Zugriffe auf Speicher, Verbindungs- und Unterkomponenten und errechnet daraus – nur bei Bedarf – die durchschnittliche Leistungsaufnahme innerhalb eines Zeitintervalls. Die Berechnung erfolgt mit Hilfe von *Callback*-Funktionen, die bei Zugriff eines Werkzeugs auf die Parameterschnittstelle ausgelöst werden. Abhängig von der Frequenz der Zugriffe können dadurch Profile zum Energieverbrauch mit unterschiedlicher Auflösung generiert werden.

Zur Unterstützung von Debugging und Analyse werden vordringlich existierende Werkzeuge eingesetzt. Alle entwickleten Modelle unterstützen das TLM2.0 *Debug Transport Interface* (DTI). Mit Hilfe des DTIs können die im Adressbereich sichtbaren Speicherelemente ohne Verzögerung und Kontextwechsel direkt gelesen und beschrieben werden. Dieser Umstand wird u.a. durch den Prozessorsimulator zur Emulation eines Betriebssystems genutzt. Dadurch können im frühen Entwicklungsstadium Funktionen des Laufzeitsystems auf den *Host* ausgelagert und beschleunigt werden. Des Weiteren wird, über die *Tracing*-Fähigkeiten von *SystemC* hinaus, *GreenAV* unterstützt. Mit *GreenAV* lassen sich *GreenReg*-Register und *GreenControl*-Parameter einfach in Listen oder *Waveforms* speichern. Um strukturiertere Terminalausgaben zu erhalten, wurde eine Methodik zur Ausgabeformatierung und -filterung entwickelt. Das *SoCRocket-Verbosity Kit* kombiniert die natürliche Handhabbarkeit von *C++-Output Streams* mit dem *Reporting*-Mechanismus von *SystemC*. Außerdem wurde mit *mscgen* ein Werkzeug zur Erzeugung von *Message Sequence Charts* integriert. Dadurch können Transaktionsverläufe im System nachvollzogen und visualisiert werden.

Ein weiterer wichtiger Aspekt Virtueller Plattformen ist die Verifikationsunterstützung. Standardisierte Methoden wie UVM konnten bisher noch nicht erfolgreich auf Systemebene transportiert werden. Zur Verifikation von *SoCRocket* wurde daher eine auf existierenden *SystemC*/TLM-Methoden aufbauende Testumgebung implementiert. Die Umgebung besteht aus einer generischen Testklasse, die jedes Modul des Systems ansteuern kann. Zur Implementierung konkreter Tests werden drei Schnittstellen bereitgestellt: direktes Lesen- und Schreiben, zufälliges Lesen- und Schreiben, sowie gepuffertes Lesen- und Schreiben. Tests können zur Simulation von Modellen auf unterschiedlichen Abstraktionsniveaus wiederverwendet werden. Zur Co-Simulation von RTL-Modellen werden Adapterklassen eingefügt, die TL-Transaktionen auf zyklengenaue Signale abbilden.

Auf Grundlage der beschriebenen Konzepte und Lösungen wurde eine ganzheitliche Methodik zur Entwicklung von TL-Simulationsmodellen und -systemen erstellt, die nahezu alle Aspekte Virtueller Plattformen, mit wenigen Ausnahmen, wie Hochsprachensynthese und HW/SW-Partitionierung, umfasst und praktisch beschreibt. Die Methodik erlaubt die Modellierung von Hardwarekomponenten durch Vererbung von Bibliotheksbasisklassen, die essentielle Funktio-

nen kapseln. Dadurch kann sich der Entwickler auf die Beschreibung von Funktionalität und Zeitverhalten konzentrieren, wodurch die Produktivität gesteigert wird. Zur Demonstration der entwickelten Methodik wurden Kernkomponenten einer im europäischen Raumfahrtsektor maßgeblichen Hardware-Bibliothek modelliert und in einer Virtuellen Plattform zur Erzeugung von Systemmodellen integriert. Die Modellpalette umfasst einen LEON2/3-Prozessorsimulator mit Cachesystem und *Memory Management Unit*, der aus einer Integereinheit entwickelt wurde. Außerdem entstanden ein Busmodell für AMBA 2.0, ein Speichercontroller und verschiedene Peripheriekomponenten wie *General Purpose Timer* (GPTimer) und *Multi Processor Interrupt Controller* (IRQMP). Aufbau, Struktur und Besonderheiten zur effizienten Modellierung dieser Komponenten werden in der Arbeit ausführlich beschrieben. Die Komponenten wurden zunächst in *Unit*-Tests verifiziert und anschließend in einem Systemprototypen integriert. Der Prototyp wurde dann für verschiedene Abstraktionsstufen konfiguriert und mit einem in VHDL beschriebenen RISC-Referenzentwurf (LEON3MP) verglichen. Dabei konnte äußerst hohe, nahezu der RTL-Synthese entsprechende, Genauigkeit erzielt werden, obwohl das System keine getakteten Prozesse enthält. Das System mit **losem *Timing* (LT) und blockierender Kommunikation** ist im Durchschnitt **561 mal schneller als die RTL-Referenz** und weist eine **durchschnittliche *Timing*-Abweichung von 7,04%** auf. Das System mit **näherungsweise akkuratem *Timing* (AT) und nicht-blockierender Kommunikation** ist **335 mal schneller**. Die durchschnittliche ***Timing*-Abweichung beträgt hier nur noch 3,03%**, was einer Standardabweichung von 0.033 und damit einer sehr hohen statistischen Sicherheit entspricht. Die verschiedenen Abstraktionsniveaus können zur Realisierung mehrstufiger Architekturexplorationen eingesetzt werden. Dies wird am Beispiel einer **hyperspektralen Bildkompression** verdeutlicht, die auf ein Mehrprozessorsystem abgebildet wurde. Die dynamische Rekonfigurierbarkeit des Systems ermöglicht es, einen **Entwurfsraum aus 1280 Prototypen in weniger als 24 Stunden vollständig zu durchsuchen**. Das System hat sich ebenfalls als sehr effizient zur Modellierung hardwarenaher Software erwiesen. Neben der höheren Simulationsgeschwindigkeit im Vergleich zum Entwurf auf RT-Ebene sind dabei Sichtbarkeit und Modifizierbarkeit aller involvierten Komponenten von besonderer Bedeutung. Dies wird anhand eines praktischen Beispiel zum Entwurf der Hardware/Software-Schnittstelle eines Beschleunigers für Dateitransfers verdeutlicht, der aufbauend auf *SoCRocket* in Zusammenarbeit mit der ESA an der TU Braunschweig entwickelt wurde.

SoCRocket ist zum Zeitpunkt der Fertigstellung dieser Arbeit eine **vollständige Virtuelle Plattform die der Europäischen Raumfahrtindustrie durch die ESA als Referenz für Entwurfsmethodik auf Systemebene angeboten wird**:

`http://www.esa.int/Our_Activities/Space_Engineering/Microelectronics/SoCROCKET_Virtual_Platform_-_SystemC`

Das entwickelte *Framework* ist jedoch nicht auf den Raumfahrtbereich beschränkt und wurde exemplarisch für eine Vielzahl von Anwendungen umgesetzt. In der Folge werden einige auf *SoCRocket* aufbauende aktuelle und geplante Forschungsaktivitäten beschrieben.

7.2 Weiterführende Arbeiten

7.2.1 Laufende und geplante Aktivitäten

CCSDS File Delivery Protocol (CFDP)

Sören Michalik, Sönke Michalik und Rolf Meyer von der Abteilung Technische Informatik der TU Braunschweig (C3E) arbeiten an einem Hardwaremodul für die Beschleunigung von Dateitransfers zwischen Satelliten und Basisstationen (ITT AO/1-6939/11/NL/JK). *SoCRocket* kommt dabei zur Entwicklung eines Virtuellen Prototypen zur Architekturexploration und zur Entwicklung von Treibersoftware für RTEMS OS zum Einsatz.

Next Generation Multi-Processor (NGMP)

Die Firma *Terma* arbeitet auf Grundlage von *SoCRocket* an TL-Modellen für den in Abschnitt 1.3 beschriebenen *Next Generation Multi-Processor* (NGMP) [ter14] (ITT AO/1-7150/12/NL/LvH). Dabei entstehen ein *Level 2*-Cache für LEON-CPUs, eine IOMMU, ein DDR2-Controller mit EDAC, ein Interruptcontroller mit erweitertem ASMP, ein *SpaceWire-Codec* mit RMAP und verschiedene generische Speichermodelle.

Embedded Multi-Core Systems for Mixed Criticality Applications in Dynamic and Changeable Real-Time Environments (EMC2)

Im Projekt EMC2, bearbeitet von Jan Wagner und Rolf Meyer (beide C3E), findet *SoCRocket* Einsatz als Plattform zur Modellierung virtueller Prototypen (WP4: Multicore Hardware Architectures and Concepts, T4.6: Virtual Platform/Prototype, Validation). Hierbei liegt der Schwerpunkt der Arbeiten auf der vereinfachten Konfiguration bei gleichzeitig erhöhter Flexibilität. Im Zuge dieser Arbeiten soll *SoCRocket* ein größeres Anwendungsfeld erschlossen werden. Der entsprechende *Task* des Arbeitspakets dieses höchst ambitionierten Projekts wird von der TU Braunschweig/C3E geleitet.

Reconfigurable ROS-based Resilient Reasoning Co-Operating Platform (R5-COP)

Jan Wagner (C3E) arbeitet im Rahmen von R5-COP und seiner hiermit verknüpften Dissertation an Methoden zur Analyse der Zuverlässigkeit und Fehlerfestigkeit eingebetteter Systeme. Dabei wird die *SoCRocket*-Infrastruktur um Komponenten zur Fehlerinjektion, Fehlerverfolgung und Fehlerkorrektur erweitert.

Parallelsimulation von Multiprozessorsystemen

Sven Horsinka (C3E) beschäftigt sich mit der Parallelisierung von *SystemC* zur Beschleunigung von Multiprozessorsysteme. Dabei sollen in einem gemeinsamen NPI-Projekt mit der ESA verschiedene Mechanismen zur Synchronisation verteiler Systemsimulationen evaluiert werden. Die Ergebnisse werden in *SoCRocket* integriert, um die Simulationsgeschwindigkeit im Hinblick auf zukünftige massiv parallele Systeme im Raumfahrtbereich zu steigern.

7.2.2 Mögliche zusätzliche Erweiterungen

Trotz des Umfanges der vorliegenden Arbeit deckt *SoCRocket* nicht alle Aspekte des Entwurfs auf Systemebene ab. In diesem Abschnitt werden einige mögliche zukünftige Entwicklungsmöglichkeiten aufgezeigt.

Systempartitionierung für Multiprozessoren

Der in Abschnitt 5.1 vorgestellte Entwurfsfluss setzt die manuelle Partitionierung des Systems in Hardware und Softwarekomponenten voraus. Für Systeme mit höherer Parallelität und Heterogenität ist dies nicht praktikable. Es müssen daher Lösungen gefunden werden, welche Anwendungssoftware und Laufzeitsystem stärker in die Architekturexploration integrieren. Verschiedene aktuelle Entwicklungen auf diesem Gebiet wurden bereits in Abschnitt 2.2 vorgestellt.

High-Level-Synthese

Um die praktische Anwendbarkeit von *SoCRocket* sicherzustellen, wurden in dieser Arbeit Kernkomponenten zur Konstruktion von Weltraum-DPUs auf Systemebene entwickelt. Dabei handelte es sich vordringlich um existierende RTL-Bausteine, die sich hoher Wiederverwendung erfreuen. Zur Erweiterung des Systems um neue Komponenten, sollte durch die Integration von

Synthesewerkzeugen ein direkter Pfad zur Implementierung auf RT-Ebene geschaffen werden (Abschnitt 2.3.3).

Intelligente Explorationlogik

Der vorgestellte mehrstufige Ansatz zur Architekturexploration (Abschnitt 5.2) erlaubt die Rekonfiguration von Systemparametern zur Laufzeit. Dadurch können große Entwurfsräume in kurzer Zeit durchsucht werden. Die Auswahl der Implementierungskandidaten erfolgt mit Hilfe einer Explorationslogik. Da die vollständige Durchsuchung von Entwurfsräumen nur selten praktikabel ist, sollten hier zusätzliche Lösungen erarbeitet werden (z.B. *Simulated Annealing*, Neuronale Netze).

Erweiterung und Verifikation der Energiemodelle

Die im Rahmen der Arbeit entwickelten Energiemodelle konnten nicht vollständig verifiziert werden. Hauptgrund war das Fehlen realistischer Speichermodelle. Darüber hinaus wurden alle Daten aus einem generischen 90nm-ASIC-*Design Kit* extrahiert. Um dem Anwender weiterreichende Abwägungen zu ermöglichen, sollten mehrere realistische ASIC- und FPGA-Zieltechnologien untersucht werden.

AXI-Integration

Über AMBA 2.0 hinaus (AHB und APB) wurde eine Methode zur akkuraten Modellierung von AXI-Schnittstellen vorgestellt (Abschnitt 3.2.4). Diese wurde bereits in Zusammenarbeit mit *Cadence* im Einsatz für Punkt-zu-Punkt-Verbindungen untersucht [Sch13b]. AXI wird bisher noch nicht im Zusammenhang mit LEON-Prozessoren eingesetzt. Daher war bisher keine Integration in *SoCRocket* möglich. In Hinblick auf die wachsenden Bandbreitenanforderungen im Raumfahrtbereich, sollte AXI schnellstmöglich implementiert werden.

Network-on-Chip

Die bereitgestellten Komponenten erlauben die Modellierung einfacher busbasierter Mehrprozessorsysteme, die sich gegenwärtig im europäischen Raumfahrtbereich durchsetzen. Für zukünftige massiv-parallele Systeme (z.B. Macspace CPU, IDANOC) sollte das System zur Modellierung von NoCs erweitert werden. Dazu müssen entsprechende Router entwickelt und die Modellierung von packetorientierter Kommunikation auf TL-Ebene untersucht werden.

7.3 Schlussbetrachtung

Die Forschung für *SoCRocket* und die anschließende praktische Umsetzung haben dem Autor große Freude bereitet. Es ist gelungen, ein in sich geschlossenes System zu schaffen. Die bereits jetzt große Zahl an Folgeprojekten und weiterführenden Aktivitäten bestätigt die Relevanz der geleisteten Arbeit. *Electronic System Level Design* und speziell Virtuelle Plattformen werden in der Zukunft weiter an Bedeutung gewinnen. Der Autor ist fest davon überzeugt, dass sich diese Techniken in allen Einsatzgebieten Eingebetteter Systeme unumgänglich durchsetzen werden. Für viele der beschriebenen aktuellen Schwierigkeiten existieren Lösungen oder vielversprechende Lösungsansätze (Abschnitt 3). Es ist jetzt wichtig, dieses Wissen zu konsolidieren und in allgemein gültigen Standards zu überführen.

Wir befinden uns auf dem richtigen Weg!

Literaturverzeichnis

[And12] Andersson, Jan; Hjorth, Magnus; Habinc, Sandi und Gaisler, Jiri: Development of a functional prototype of the quad-core NGMP space processor, in: *Data Systems in Aerospace (DASIA)*

[Ang06] Angiolini, F.; Ceng, Jianjiang; Leupers, R.; Ferrari, F.; Ferri, C. und Benini, L.: An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration, in: *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, Bd. 1, S. 1–6

[AS14] Ali Shah, Syed Abbas; Wagner, Jan; Schuster, Thomas und Berekovic, Mladen: Power and area estimation for heterogeneous multi-processor embedded systems based on adaptable and extendable ASIPs, in: *Power And Timing Modeling, Optimization and Simulation (PATMOS)*

[Aus02] Austin, T.; Larson, E. und Ernst, D.: SimpleScalar: an infrastructure for computer system modeling. *Computer* (2002), Bd. 35(2): S. 59–67

[Ayn09] Aynsley, John: *TLM-2.0 User Manual*, OSCI - Open SystemC Initiative (2009)

[Bai05] Bailey, Brian; Martin, Grant und Anderson: *Taxonomies for the Development and Verification of Digital Systems*, Springer Science + Business Media (2005)

[Bai07] Bailey, Brian; Grant, Martin und Piziali, Andrew: *ESL Design and Verification - A Prescription for Electronic System Level Methodology*, Morgan Kaufmann, 2 Aufl. (2007)

[Bar13] Bari, N.; Mani, G. und Berkovich, S.: Internet of Things as a Methodological Concept, in: *Computing for Geospatial Research and Application (COM.Geo), 2013 Fourth International Conference on*, S. 48–55

[Bel09] Beltrame, G.; Fossati, L. und Sciuto, D.: ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (2009), Bd. 28(12): S. 1857–1869

[Ben05] Benini, Luca; Bertozzi, Davide; Bogliolo, Alessandro; Menichelli, Francesco und Olivieri, Mauro: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI signal processing systems for signal, image and video technology* (2005), Bd. 41(2): S. 169–182, URL http://dx.doi.org/10.1007/s11265-005-6648-1

[Bla10] Black, D. C.; Donavan, J.; Bunton, B. und Keist, A.: *SystemC: From the Ground Up*, Springer, 2nd Aufl. (2010)

[Bli98] Blickle, T.; Teich, J. und Thiele, L.: System-Level Synthesis Using Evolutionary Algorithms. *Design Automation for Embedded Systems* (1998), Bd. 3(1): S. 23–58, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0031704808&partnerID=40&md5=7d716829f7650dae2ed64e56aed6433c, cited By (since 1996)61

[Bou06] Bougard, Bruno; Novo, David; Naessens, Frederik; Hollevoet, Lieven; Schuster, Thomas; Glassee, Michael; Dejonghe, Andy und Van der Perre, Liesbet: A scalable programmable baseband platform for energy-efficient reactive Software Defined Radio, in: *Prodeedings of the International Conference on Cognitive Radio Oriented Wireless Networks (Crowncom)*

[Bou12] Bougard, Bruno und Schuster, Thomas: SyncPro2 - Programmable Device for Software Defined Radio Terminal, US Patent: US 2013 0173884 A1 (2012)

[Bou13] Bouhadiba, Tayeb; Moy, Matthieu und Maraninchi, Florence: System-level modeling of energy in TLM for early validation of power and thermal management, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 1609–1614

[Bro10] Brooks, C.; Lee, E.A und Tripakis, S.: Exploring models of computation with Ptolemy II, in: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, S. 331–332

[Buc91] Buck, J. T.; Ha, S.; Lee, E. A. und Messerschmitt, D. G.: Ptolemy: A Mixed-Paradigm Simulation/Prototyping Platform in C++, in: *Proceedings C++ At Work Conference*, Santa Clara, CA

[Cas14] Castilhos, G.; Wachter, E.; Madalozzo, G.; Erichsen, A; Monteiro, T. und Moraes, F.: A framework for MPSoC generation and distributed applications evaluation, in: *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, S. 408–411

[CCS12] CCSDS - The Consultative Commitee for Space Data Systems: *Draft Recommendation for Space Data System Standards: Lossless Multispectral and Hyperspectral image compression*, CCSDS 123.0-R-1 Aufl. (2012)

[Che12] Chen, Weiwei; Han, Xu und Domer, R.: Out-of-order parallel simulation for ESL design, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, S. 141–146

[Che13] Chen, Weiwei und Domer, Rainer: Optimized out-of-order Parallel Discrete Event Simulation using predictions, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 3–8

[Che14] Chen, Zhimiao; Wang, Yifan; Liao, Lei; Zhang, Ye; Aytac, A; Muller, J.H.; Wunderlich, R. und Heinen, S.: A SystemC Virtual Prototyping based methodology for multi-standard SoC functional verification, in: *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, S. 1–6

[Com10] Committee, Design Automation Standards: *IEEE Standard 1685 for IP-XACT, Standard Structure for Packaging, Integration, and Reusing IP within Tool Flows*, IEEE, New York (2010)

[Con07] The Consultative Committee for Space Data Systems: *CCSDS File Delivery Protocol (CFDP)*, 727.0-b-4 Aufl. (2007)

[cri05] criticalblue: *White Paper: Boosting Software Processing Performance With Coprocessor Synthesis*, www.CriticalBlue.com (2005)

[Den06] Densmore, D.; Passerone, R. und Sangiovanni-Vincentelli, A.: A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers* (2006), Bd. 23(5): S. 359–373, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-33947101971&partnerID=40&md5=ddc133c35e282d5f760cb9473738a33d, cited By (since 1996)80

[Der10] Derudder, Veerle; Bougard, Bruno; Couvreur, Aissa; Dewilde, Andy; Dupont, Steven; Folens, Luc; Hollevoet, Lieven; Naessens, Frederik; Novo, David; Raghavan, Praveen; Schuster, Thomas; Stinkens, Kurt; Weijers, Jan-Willem und Van der Perre, Liesbet: A 200+Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs, in: *Transactions of the International Solid State Circuits Conference (ISSCC)*

[Dha05] Dhanwada, N.; Lin, I.C. und Narayanan, V.: A power estimation methodology for systemC transaction level models, in: *Proceedings of the 3rd IEEE/ACM/IFIP*

international conference on Hardware/software codesign and system synthesis, ACM, S. 142–147

[DM86] De Man, H.; Rabaey, J.; Six, P. und Claesen, L.: Cathedral-II: A Silicon Compiler for Digital Signal Processing. *Design Test of Computers, IEEE* (1986), Bd. 3(6): S. 13–25

[DM90] De Micheli, G.; Ku, D.; Mailhot, F. und Truong, T.: The Olympus synthesis system. *Design Test of Computers, IEEE* (1990), Bd. 7(5): S. 37–53

[DM93] De Micheli, G.: Extending CAD tools and techniques. *Computer* (1993), Bd. 26(1): S. 85–87

[Dom12] Domer, R.; Chen, Weiwei und Han, Xu: Parallel discrete event simulation of Transaction Level Models, in: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 227–231

[Eck14] Ecker, W.; Velten, M.; Zafari, L. und Goyal, A: The metamodeling approach to system level synthesis, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–2

[Ern93] Ernst, R.; Henkel, J. und Benner, T.: Hardware-software cosynthesis for microcontrollers. *Design Test of Computers, IEEE* (1993), Bd. 10(4): S. 64–75

[Eur11] European Space Agency (ESA): *Technical Dossier on Data Systems and Onboard Computers, TEC-EDD/2011.109/GM* (2011)

[Fil07] Filion, L.; Cantin, M.-A.; Moss, L.; Aboulhamid, E. M. und Bois, G.: Space Codesign: A SystemC Framework for Fast Exploration of Hardware/Software Systems. *Design and Verification Conference and Exhibition (DVCON)* (2007)

[Fin10] Fingeroff, M.: *High-Level Synthesis Blue Book*, Xlibris Corporation (2010)

[Fis14] Fischer, B.; Cech, C. und Muhr, H.: Power modeling and analysis in early design phases, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–6

[Fos10] Fossati, Luca: TLM2.0 Standard into Action: Designing Efficient Processor Simulators. *Proceedings - 19th IP based electronic conference and exhibition (IP-SoC)* (2010)

[Fos13] Fossati, Luca; Schuster, Thomas; Meyer, Rolf und Berekovic, Mladen: SoCRocket: A Virtual Platform for SoC Design, in: *Data Systems in Aerospace (DASIA)*

[Ful14] Fuller, David: System design challenges for next generation wireless and embedded systems, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–1

[Fum14] Fummi, F.; Lora, M.; Stefanni, F.; Trachanis, D.; Vanhese, J. und Vinco, S.: Moving from co-simulation to simulation for effective smart systems design, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–4

[Gai02] Gaisler, J.: A portable and fault-tolerant microprocessor based on the SPARC v8 architecture, in: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, S. 409–415

[Gai10] Gaisler, Aeroflex: *GRLIB IP Core User's Manual* (2010)

[Gaj83] Gajski, D.D. und Kuhn, R.H.: Guest Editors' Introduction: New VLSI Tools. *Computer* (1983), Bd. 16(12): S. 11–14

[Gaj94] Gajski, D. und Ramachandran, L.: Introduction to high-level synthesis, in: *Design and Test of Computers*

[Gla12] Gladigau, J.; Haubelt, C. und Teich, J.: Model-Based Virtual Prototype Acceleration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (2012), Bd. 31(10): S. 1572–1585

[Goo13] Goossens, S.; Kuijsten, J.; Akesson, B. und Goossens, K.: A reconfigurable real-time SDRAM controller for mixed time-criticality systems, in: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, S. 1–10

[Gra13] Grammatikakis, M.D.; Papagrigoriou, A; Petrakis, P. und Kornaros, G.: Monitoring-Aware Virtual Platform Prototype of Heterogeneous NoC-Based Multicore SoCs, in: *Digital System Design (DSD), 2013 Euromicro Conference on*, S. 497–504

[Gra14] Graf, S.; Glass, M.; Teich, J. und Lauer, C.: Multi-variant-based design space exploration for automotive embedded systems, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–6

[Gro00] Group, RASSP Terminology Working: www.eda.org/rassp, in: *VHDL Modeling Terminology and Taxonomy*

[Gru12] Gruttner, K.; Hartmann, P.A; Hylla, K.; Rosinger, S.; Nebel, W.; Herrera, F.; Villar, E.; Brandolese, C.; Fornaciari, W.; Palermo, G.; Ykman-Couvreur, C.; Quaglia, D.; Ferrero, F. und Valencia, R.: COMPLEX: COdesign and Power Management in PLatform-Based Design Space EXploration, in: *Digital System Design (DSD), 2012 15th Euromicro Conference on*, S. 349–358

[Gün11] Günzel, Robert: *Taktgenaue Bus-Simulation mit der Transaction-Level-Modellierung*, Dissertation, Technische Universität Braunschweig (2011)

[Gup93] Gupta, R.K. und De Micheli, G.: Hardware-software cosynthesis for digital systems. *Design Test of Computers, IEEE* (1993), Bd. 10(3): S. 29–41

[Han98] Handy, Jim: *The cache memory book*, Academic Press, 2nd Aufl. (1998)

[Hel13] Helmstetter, Claude; Cornet, Jerome; Galilee, Bruno; Moy, Matthieu und Vivet, Pascal: Fast and accurate TLM simulations using temporal decoupling for FIFO-based communications, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 1185–1188

[Hen02] Henkel, J. und Li, Yanbing: Avalanche: an environment for design space exploration and optimization of low-power embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* (2002), Bd. 10(4): S. 454–468

[Hen05] Henia, R.; Hamann, A.; Jersak, M.; Racu, R.; Richter, K. und Ernst, R.: System level performance analysis - The SymTA/S approach, Bd. 152, S. 148–166, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-19344371097&partnerID=40&md5=ca0237cc1bb7589f224c6958d7d6f060, cited By (since 1996)130

[Hof02] Hoffmann, A.; Meyr, H. und Leupers, R.: *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers (2002)

[Inc] Inc., Cadence: http://www.cadence.com/products/sd/silicon_compiler, in: *C-to-Silicon Compiler*

[Inc92] Inc., SPARC International: *The SPARC Architecture Manual - Version 8* (1992)

[Inc14] Inc, Wind River: *Simics: Product Note* (2014)

[Ini05] Initiative, Accellera Systems: *IEEE 1666 - 2005 Standard Language Reference Manual* (2005)

[Jov08] Joven, J.; Font-Bach, O.; Castells-Rufas, D.; Martinez, R.; Teres, L. und Carrabina, J.: xENoC - An eXperimental Network-On-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC Architectures, in: *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, S. 141–148

[Kes12] Kesel, Frank: *Modellierung von digitalen Systemen mit SystemC*, Oldenbourg Verlag München (2012)

[Kle11] Kleine, Etienne: Activity based power modeling for high-level virtual platform prototyping, in: *Diplomarbeit, Friedrich-Schiller-Universität Jena*

[Le13] Le, Hoang M.; Grosse, Daniel und Drechsler, Rolf: Scalable fault localization for SystemC TLM designs, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 35–38

[Leb08] Lebreton, H. und Vivet, P.: Power Modeling in SystemC at Transaction Level, Application to a DVFS Architecture, in: *IEEE Computer Society Annual Symposium on VLSI*, IEEE, S. 463–466

[Lee05] Lee, Young-Ran; Cho, Sang-Young und Lee, Jeong-Bae: The design a virtual prototyping based on ARMulator, in: *Computer and Information Science, 2005. Fourth Annual ACIS International Conference on*, S. 387–390

[Lei13] Lei, Li; Xie, Fei und Cong, Kai: Post-silicon conformance checking with virtual prototypes, in: *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, S. 1–6

[Lie97] Liem, C.; Nacabal, F.; Valderrama, C.; Paulin, P. und Jerraya, A.: System-on-a-chip cosimulation and compilation. *IEEE Design and Test of Computers* (1997), Bd. 14(2): S. 16–24, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0031123958&partnerID=40&md5=bf65f8f856187f984e16c2519d2c5f78, cited By (since 1996)25

[Lim11] Limited, ARM: *AMBA Specification*, 2.0 Aufl. (2011)

[Lin08] Lin, Ye-Jyun; Chen, Yi-Jung; Huang, Chin-Chie; Lin, Tzu-Ching; Chi, Jaw-Wei und Yang, Chia-Lin: TunableVP: A Tunable Virtual Platform for easy SoC design space exploration, in: *SoC Design Conference, 2008. ISOCC '08. International*, Bd. 01, S. I–250–I–251

[Lu13] Lu, Kun; Muller-Gritschneder, Daniel und Schlichtmann, Ulf: Analytical timing estimation for temporally decoupled TLMs considering resource conflicts, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 1161–1166

[Mag02] Magnusson, P.S.; Christensson, M.; Eskilson, J.; Forsgren, D.; Hallberg, G.; Hogberg, J.; Larsson, F.; Moestedt, A und Werner, B.: Simics: A full system simulation platform. *Computer* (2002), Bd. 35(2): S. 50–58

[Mag09] Magli, E.: Multiband Lossless Compression of Hyperspectral Images. *Geoscience and Remote Sensing, IEEE Transactions on* (2009), Bd. 47(4): S. 1168–1178

[Mar09] Marculescu, R.; Ogras, U.Y.; Peh, L.-S.; Jerger, N.E. und Hoskote, Y.: Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2009), Bd. 28(1): S. 3–21, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-66549114708&partnerID=40&md5=42bbf79bfd57483d4240c8c44cadc401, cited By (since 1996)196

[Mel10] Mello, A; Maia, I; Greiner, A und Pecheux, F.: Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, S. 606–609

[Mey10] Meyer, Rolf: *Methode zur effizienten Entwicklung von TLM2.0 Modellen mit hoher Simulationsgenauigkeit*, Diplomarbeit, Technische Universität Braunschweig (2010)

[Moo06] Moore, Gordon E.: Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE* (2006), Bd. 11(5): S. 33–35

[Moy13] Moy, Matthieu: Parallel programming with SystemC for loosely timed models: A non-intrusive approach, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 9–14

[Mue12] Mueller, W.; Becker, M.; Elfeky, A und DiPasquale, A: Virtual prototyping of Cyber-Physical Systems, in: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 219–226

[Mur08] Muralimanohar, N.; Balasubramonian, R. und Jouppi, N.P.: Architecting Efficient Interconnects for Large Caches with CACTI 6.0. *Micro, IEEE* (2008), Bd. 28(1): S. 69–79

[Ng07] Ng, Anthony; Weijers, Jan-Willem; Glassee, Michael; Schuster, Thomas; Bougard, Bruno und Van der Perre, Liesbet: ESL Design and HW/SW Co-Verification of high-end Software Defined Radio Platform, in: *Proceedings of the International Conference on Hardware/Software Co-Design and System Synthesis*

[Nov08] Novo, David; Schuster, Thomas; Bougard, Bruno; Lambrechts, Andy; Van der Perre, Liesbet und Francky, Catthoor: Energy-performance Exploration of a CGA-based SDR processor. *Journal on Signal Processing Systems* (2008)

[Oet14] Oetjens, J.-H.; Bannow, N.; Becker, M.; Bringmann, O.; Burger, A; Chaari, M.; Chakraborty, S.; Drechsler, R.; Ecker, W.; Gruttner, K.; Kruse, T.; Kuznik, C.; Le, H.M.; Mauderer, A; Müller, W.; Müller-Gritschneder, D.; Poppen, F.; Post, H.; Reiter, S.; Rosenstiel, W.; Roth, S.; Schlichtmann, U.; von Schwerin, A; Tabacaru, B.-A und Viehl, A: Safety evaluation of automotive electronics using Virtual Prototypes: State of the art and research challenges, in: *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, S. 1–6

[Ozi08] Ozisikyilmaz, B.; Memik, G. und Choudhary, A.: Efficient system design space exploration using machine learning techniques, in: *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, S. 966–969

[Pie13] Pierre, L. und Bel Hadj Amor, Z.: Automatic refinement of requirements for verification throughout the SoC design flow, in: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, S. 1–10

[Pin06] Pinto, C.A; Beric, A; Singh, S.P. und Farfade, S.: HiveFlex-Video VSP1: Video Signal Processing Architecture for Video Coding and Post-Processing, in: *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, S. 493–500

[Poc11] Pockrandt, M.; Herber, P. und Glesner, S.: Model checking a SystemC/TLM design of the AMBA AHB protocol, in: *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, S. 66–75

[Pre08] Prechtl, Peter und Burkhard, Franz-Peter: *Metzler Lexikon Philosophie: Begriffe und Definitionen*, Bd. 3, J.B. Metzler (2008)

[Pul11] Pulka, A; Golly, L. und Milik, A: SystemC hardware-software design and simulation platform based on AMBA bus, in: *Mixed Design of Integrated Circuits and Systems (MIXDES), 2011 Proceedings of the 18th International Conference*, S. 644–649

[Red05] Redant, S.; Marec, R.; Baguena, L.; Liegeon, E.; Soucarre, J.; Van Thielen, B.; Beeckman, G.; Ribeiro, P.; Fernandez-Leon, A. und Glass, B.: Radiation test results on first silicon in the design against radiation effects (DARE) library. *Nuclear Science, IEEE Transactions on* (2005), Bd. 52(5): S. 1550–1554

[Ret14] Rethinagiri, S.-K.; Palomar, O.; Unsal, O.; Cristal, A; Ben-Atitallah, R. und Niar, S.: PETS: Power and energy estimation tool at system-level, in: *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, S. 535–542

[Rey09] Reyes, V.: Refinement and reuse of TLM 2.0 models: The key for ESL success, in: *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, S. 102–105

[Rol12] Roloff, S.; Hannig, F. und Teich, J.: Approximate time functional simulation of resource-aware programming concepts for heterogeneous MPSoCs, in: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 187–192

[Sch] Schröder, Christian; Klingauf, Wolfgang und Günzel, Robert: *Green Analysis and Visibility User's Guide*, TU-Braunschweig, Dept. E.I.S.

[Sch06a] Schirner, G. und Domer, R.: Quantitative Analysis of Transaction Level Models for the AMBA Bus, in: *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, Bd. 1, S. 1–6

[Sch06b] Schuster, Thomas; Novo, David; Bougard, Bruno; Derudder, Veerle; Hoffmann, Andreas und Van der Perre, Liesbet: Subword Parallel VLIW Architecture Exploration for Multi-Mode Software Defined Radio, in: *Proceedings of the Workshop on Signal Processing Systems (SIPS)*

[Sch07] Schuster, Thomas; Bougard, Bruno; Raghavan, Praveen; Priewasser, Robert; Novo, David; Van der Perre, Liesbet und Francky, Catthoor: Design of a Low Power Pre-Synchronization ASIP for Multi-mode SDR Terminals, in: *Proceedings of the International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS)*

[Sch11] Schröder, Christian: *Konfigurations-Interoperabilität von Hardware-Software Modellen in SystemC*, Dissertation, TU-Braunschweig (2011)

[Sch12a] Schuster, Thomas und Meyer, Rolf: *SoCRocket - Analysis Capability Report*, C3E - TU Braunschweig (2012)

[Sch12b] Schuster, Thomas und Meyer, Rolf: *SoCRocket - IP User Manual & Development Document*, C3E - TU Braunschweig (2012)

[Sch12c] Schuster, Thomas und Meyer, Rolf: *SoCRocket - Power Modeling Report*, C3E - TU Braunschweig (2012)

[Sch12d] Schuster, Thomas und Meyer, Rolf: *SoCRocket - Verification and Performance Document*, C3E - TU Braunschweig (2012)

[Sch13a] Schuster, Thomas; Meyer, Rolf und Berekovic, Mladen: *Skript zum Praktikum Advanced VLSI Design II*, TU-Braunschweig (2013)

[Sch13b] Schuster, Thomas; Sauer, Christian und Berekovic, Mladen: Timing-precise modeling of contemporary bus interfaces for SystemC components by only using TLM2.0's generic base protocol, in: *Cadence User Conference, CDNLive*

[Sch14] Schuster, T.; Meyer, R.; Buchty, R.; Fossati, L. und Berekovic, M.: SoCRocket - A virtual platform for the European Space Agency's SoC development, in: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, S. 1–7

[Sco98] Scott, Jeff; Lee, Lea Hwang; Arends, John und Moyer, Bill: Designing the Low-Power M*CORE Architecture (1998)

[Sha14] Shan, Tang; Ziyuan, Zhu und Yongtao, Su: System-level design methodology enabling fast development of baseband MP-SoC for 4G small cell base station, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–6

[Sin12] Sinha, R.; Prakash, A und Patel, H.D.: Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs, in: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 455–460

[Som14] Sommer, C.; Hagenauer, F. und Dressler, F.: A networking perspective on self-organizing intersection management, in: *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, S. 230–234

[Str09] Streubühr, M.; Gladigau, J.; Haubelt, C. und Teich, J.: Efficient approximately-timed performance modeling for architectural exploration of MPSoCs, in: *Specification Design Languages, 2009. FDL 2009. Forum on*, S. 1–6

[SV01] Sangiovanni-Vincentelli, A. und Martin, G.: Platform-based design and software design methodology for embedded systems. *Design Test of Computers, IEEE* (2001), Bd. 18(6): S. 23–33

[Swa12] Swan, Stuart und Cornet, Jerome: Beyond TLM 2.0: New Virtual Platform Standards Proposals from ST and Cadence, Presented at NASCUG at DAC; Jun 18, 2012; http://www.nascug.org/events/18th/beyond_tlm20_6-6-2012.pdf (2012)

[Tag09] Taghavi, T.; Pimentel, A.D. und Thompson, M.: System-level MP-SoC design space exploration using tree visualization, in: *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, S. 80–88

[Tal06] Talarico, C.; Rodriguez-Marek, E. und Koh, Min-Sung: Multi-objective design space exploration methodologies for platform based SOCs, in: *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, S. 7 pp.–359

[Tei07] Teich, J. und Haubelt, C.: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*, Springer-Verlag, Berlin, 2nd Aufl. (2007)

[Tha14] Thanner, Manfred: Virtual prototype life cycle in automotive applications, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–1

[Thi00] Thiele, Lothar; Chakraborty, Samarjit und Naedele, Martin: Real-time calculus for scheduling hard real-time systems, Bd. 4, S. IV–101–IV–104, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0033682521&partnerID=40&md5=72c21df7dab7fd159d737ae6fefa8ad7, cited By (since 1996)132

[Tho13] Thomas, Pierre-Xavier; Martin, Grant; Heine, David; Moolenaar, Dennis und Kim, James: Configurability in IP subsystems: Baseband examples, in: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, S. 163–168

[Tob10] Tobias, Lange: Analyse von High-Level Synthese Werkzeugen zur Implementierung von Signalverarbeitungsalgorithmen auf FPGA, in: *Diplomarbeit, TU-Braunschweig*

[Tra11] Trautner, Roland: ESA's Roadmap for Next Generation Payload Data Processors, in: *Data Systems in Aerospace (DASIA)*

[Uba14] Ubal, R.; Schaa, D.; Mistry, P.; Gong, Xiang; Ukidave, Y.; Chen, Zhongliang; Schirner, G. und Kaeli, D.: Exploring the heterogeneous design space for both performance and reliability, in: *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, S. 1–6

[VR96] Van Rompaey, K.; Verkest, D.; Bolsens, I. und De Man, H.: CoWare-a design environment for heterogeneous hardware/software systems, in: *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European*, S. 252–257

[Wal89] Walker, R.A. und Thomas, D.E.: Behavioral transformation for algorithmic level IC design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (1989), Bd. 8(10): S. 1115–1128

[Wan12] Wang, Chen-Chieh; Lo, Sheng-Hsin; Liu, Yao-Ning und Chen, Chung-Ho: NetVP: A system-level NETwork Virtual Platform for network accelerator development, in: *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, S. 249–252

[Wei14] Weinstock, J.H.; Schumacher, C.; Leupers, R.; Ascheid, G. und Tosoratto, L.: Time-decoupled parallel SystemC simulation, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–4

[You07] Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, in: *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, S. 23–34

[Yun12] Yun, Dukyoung; Kim, Sungchan und Ha, Soonhoi: Relaxed synchronization technique for speeding-up the parallel simulation of multiprocessor systems, in: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, S. 449–454

[Ziv96] Zivojnovic, Vojin und Meyr, Heinrich: Compiled HW/SW co-simulation, S. 690–695, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0029708449&partnerID=40&md5=6609d4871d8220f2b47924025026e0dc, cited By (since 1996)23

[Zuo14] Zuolo, L.; Zambelli, C.; Micheloni, R.; Galfano, S.; Indaco, M.; Di Carlo, S.; Prinetto, P.; Olivo, P. und Bertozzi, D.: SSDExplorer: A virtual platform for fine-grained design space exploration of Solid State Drives, in: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, S. 1–6

Internetquellen

[acc13] Accellera Systems Initiative, http://www.accellera.org (2013)

[Ace14] Acellera: SystemC Configuration, Control & Inspection (CCI) Working Group, http://www.accellera.org/activities/committees/systemc-cci/ (2014)

[atm14] Atmel - Rad Hard FPGAs, http://www.atmel.com/products/other/space_rad_hard_ics/rad_hard_fpgas.aspx (2014)

[bin14] GNU Binutils, http://www.gnu.org/software/binutils/ (2014)

[cal14] Calypto CatapultC, http://calypto.com/en/products/catapult/overview/ (2014)

[car14] Carbon Design Systems - White Papers, http://www.carbondesignsystems.com/virtual-prototype-white-papers/ (2014)

[cot14] HP-Labs COTson Simulator, http://cotson.sourceforge.net (2014)

[Cro14] Crockford, Douglas: JavaScript Object Notation (JSON), http://www.json.org (2014)

[cto14] Cadence C-to-Silicon Compiler, http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx (2014)

[dou14] Doulos SystemC/TLM2.0 Online Tutorials, http://www.doulos.com (2014)

[ecl14] Eclipse, http://www.eclipse.org (2014)

[ESA13] ESA: SoCRocket Virtual Platform - SystemC IP, http://www.esa.int/TEC/Microelectronics/SEMW9XK1QAH_0.html (2013)

[ESA14] ESA: SpaceWire standaard for high-speed links and networks onboard spacecrafts, http://www.spacewire.esa.int/content/Home/HomeIntro.php (2014)

[eur14] Europractice, http://www.europractice.com (2014)

[fas14] ARM Fast Models, http://www.arm.com/products/tools/models/fast-models/index.php (2014)

[gai13] Aeroflex Gaisler AB, http://www.gaisler.com (2013)

[gco14] GCOV - Test Coverage Program, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (2014)

[gdb14] GDB GNU Debugger, http://www.gnu.org/software/gdb/ (2014)

[GNU14] GNU: GNU Compiler Collection (GCC), http://gcc.gnu.org (2014)

[gre13] GreenSoCs, http://www.greensocs.com (2013)

[gtk14] GTKWave Multi-Format Waveform Viewer, http://gtkwave.sourceforge.net (2014)

[hip14] HiPEAC Netzwerk, http://www.hipeac.net/ (2014)

[imp14] Impulse Accelerated Technologies, ImpulsC, http://www.impulseaccelerated.com (2014)

[Inc13] Inc., Synopsys: SystemC Modeling Library (SCML) Source Kit, http://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi (2013)

[inc14a] Cadence Incisive Enterprise Simulator, http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx (2014)

[Inc14b] Inc., Cadence: Cadence Virtual System Platform (VSP), http://www.cadence.com/products/sd/virtual_system/pages/default.aspx (2014)

[Inc14c] Inc., MathWorks: Matlab / Simulink, http:http://www.mathworks.de/ (2014)

[Inc14d] Inc., Synopsys: Synopsys Platform Architect, http://www.synopsys.com/Systems/ArchitectureDesign/Pages/PlatformArchitect.aspx (2014)

[inn14] Synopsys Innovator, http://www.synopsys.com/Systems/VirtualPrototyping/Pages/default.aspx (2014)

[itr14] International Technology Roadmap for Semiconductors - Chapter Design, http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf (2014)

[mac] MacSpace collaborative research and development project, http://www.macspace.eu

[McT14] McTernan, Michael: mscgen - Message Sequence Chart Renderer, https://code.google.com/p/mscgen/ (2014)

[MJ14] Mason-Jones, Craig: JSON4LUA Parser, http://json.luaforge.net (2014)

[oB04] of Bologna, University: MPARM Architectural Simulator, http://www-micrel.deis.unibo.it/sitonew/research/mparm.html (2004)

[ope14] OpenCores, http://opencores.org/ (2014)

[ovp13] Imperas Open Virtual Platform (OVP), http://www.imperas.com (2013)

[pro14] Synopsys Processor Designer, http://www.synopsys.com/systems/blockdesign/processordev/pages/default.aspx (2014)

[que14] Mentor Questa Verifikationsumgebung, http://www.mentor.com/products/fv/questa/ (2014)

[res13] RESP - MPSoC Simulation Platform, resp-sim.googlecode.com (2013)

[sim14] AMD SIMNOW, http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/ (2014)

[soc13] SoCLib project website, www.soclib.fr (2013)

[spa14] SpaceCodesign, http://www.spacecodesign.com (2014)

[syn13] Synopsys, Synphony C, http://www.synopsys.com/Systems/BlockDesign/HLS (2013)

[Sys14] Systems, Carbon Design: TLM AMBA Modeling Kit, https://portal.carbondesignsystems.com/Model/Carbon/TLM-2.0-AMBA (2014)

[ter14] Terma GmbH Darmstadt, http://www.terma.com (2014)

[tlm14] TLMCentral, http://www.dr-embedded.com/tlmcentral/servlet/catalog (2014)

[tra] TrapGen Repository, https://code.google.com/p/trap-gen/

[tri14] Trimaran Prozessorsimulator, http://www.trimaran.org (2014)

[uni13] UNISIM: UNIted SIMulation environment, unisim.org (2013)

[vec] Online C++ Referenz: std::vector, http://www.cplusplus.com/reference/vector/vector/

[waf14] WAF Build-System, https://code.google.com/p/waf/ (2014)

Abkürzungsverzeichnis

ADL	Architecture Definition Language
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASI	Address Space Identifier
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
AT	Approximately Timed
AXI	Advanced eXtensible bus interface
BFM	Bus Functional Model
CCR	Cache Control Register
CCSDS	Consultative Commitee for Space Data Systems
COTS	Common Off The Shelf
CPS	Cyber Physical System
CFDP	CCSDS File Delivery Protocol
CPU	Central Processing Unit
DPU	Data Processing Unit
DSE	Design Space Exploration
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
EDAC	Error Detection And Correction mechanism
EP	Exploration Prototype
ESA	European Space Agency
ESL	Electronic System Level
ESLD	Electronic System Level Design
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU is Not Unix
HW	Hardware
IC	Integrated Circuit
IF	Interface
IO	Input/Output
IP	Intellectual Property
IU	Integer Unit
ISS	Instruction Set Simulator
JSON	JavaScript Object Notation
ITRS	International Technology Roadmap for Semiconductors
LISA	Language for Instruction Set Architecture
LT	Loosely Timed

MIPS	Millions Instructions Per Second
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
NGMP	Next Generation Multi-Processor
NoC	Network on Chip
OS	Operating System
PM	Power Monitor
PNP	Plug and Play
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
ROM	Read Only Memory
RISC	Reduced Instruction Set Computer
RMW	Read Modify Write
RTEMS	Real-Time Executable for Multi-Processor Systems
RTL	Register Transfer Level
SRAM	Synchronous Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SW	Software
TL	Transaction Level
TLB	Translation Lookaside Buffer
TLM	Transaction Level Modeling
TU	Technische Universität
UML	Unified Modeling Language
VAT	Virtual Address Tag
VHDL	Very high speed Hardware Description language
VP	Virtual Platform
VPI	Virtual Platform Infrastructure
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

A SoCRocket Installation und Kommandoübersicht

SoCRocket wird durch die Europäische Raumfahrtagentur vertrieben [ESA13] und als GIT-*Repository* durch die TU-Braunschweig zum *Download* bereitgestellt. Für den Zugriff ist eine Registrierung erforderlich. Weitere Informationen finden sich hier:

https://projects.c3e.cs.tu-bs.de/socrocket/

Das *Build*-System von SoCRocket beruht auf *waf* [waf14] und wurde durch Rolf Meyer verfasst. Der Vollständigkeit halber werden hier die wichtigsten Kommandos kurz erläutert, ohne auf Implementierungsdetails einzugehen. Zur Auflistung aller verfügbaren Kommandos kann im Wurzelverzeichnis des Systems *./waf -h* aufgerufen werden.
Kommandos in *waf* haben folgende Form:

```
waf [command] [options]
```

Liste verfügbarer Kommandos:

```
build     : Kompiliert das Gesamtsystem oder ein ausgewähltes Ziel
clean     : Säubert das Projekt (Objektfiles löschen, etc.)
configure : Konfiguriert das System
coverage  : Ermittelt die Testabdeckung im System mit "lcov/gcov"
dist      : Erzeugt ein Archiv (tar) zur Redistribution des Systems
distcheck : Überprüft das mit "dist" erzeugte Archiv auf Kompilierbarkeit
docs      : Erzeugt die Dokumentation des Quellcodes mit "doxygen"
list      : Listet alle Ziele (z.B. Tests, Plattformen) auf,
            die mit "build" kompiliert werden können
generate  : Startet den Configuration Wizard
```

Nach erfolgreichem *Download* muss das System konfiguriert und kompiliert werden. Zur Konfiguration wird das Kommando *configure* verwendet.

```
./waf configure
```

Das *Build*-System informiert den Nutzer über erforderliche und fehlende Softwarepakete. Nach Abschluss der Konfiguration kann SoCRocket kompiliert werden.

```
./waf build
```

Eine vollständige Kompilierung umfasst alle integrierten Komponenten, Tests und Plattform-Prototypen. Der Vorgang kann daher längere Zeit in Anspruch nehmen (ca. 1 Stunde). Im Anschluss stehen die erzeugten Tests und Systemsimulationen im Unterverzeichnis *build* als ausführbare Programme bereit.

SoCRocket wurde unter *Red Hat Enterprise Linux 5* entwickelt, ist aber auch in anderen *Linux* Distributionen und *Mac OS X* lauffähig. Ausführliche Installationsanleitungen und eine Liste der erforderlichen Abhängigkeiten kann [Sch12b] entnommen werden.

B Verzeichnisstruktur und Files

Dieser Abschnitt gibt einen Überblick über die zum System gehörigen Files und Verzeichnisse. Aus Gründen der Übersichtlichkeit ist es nicht möglich das Gesamtsystem in einem einzigen Verzeichnisbaum darzustellen. Es wird daher zuerst eine Gesamtübersicht der wichtigsten Verzeichnisse präsentiert. Im Anschluss wird deren Inhalt einzeln dargestellt und erklärt.

B.1 Gesamtübersicht

```
\SoCRocket
├── adapters ..... RTL Co-Simulationsadapter
├── build ..... Verzeichnis für Binärdateien
├── common ..... Allgemeine Modellinfrastruktur (z.B. Endianesskonversion)
├── contrib ..... Patches für Boost und GreenSoCs
├── doc ..... Dokumentation
├── models ..... TLM Simulationsmodelle
│   ├── ahbctrl ..... AHB Bus Controller
│   ├── apbctrl ..... AHB2APB Busbrücke und Controller
│   ├── mctrl ..... LEON Speichercontroller
│   ├── memory ..... Generischer Speicher IO/ROM/SRAM/SDRAM
│   ├── gp_timer ..... General Purpose Timer
│   ├── irqmp ..... Multi-Prozessor Interrupt Controller
│   ├── mmu_cache ..... LEON3 Cache Sub-System
│   ├── extern ..... Extern entwickelte Modelle (z.B. LEON IU, SpaceWire)
│   └── utils ..... Bibliotheksbasisklassen
├── platforms ..... Plattforminstanzen
├── signalkit ..... TL Signal Toolbox
├── software ..... LEON Software und Tests
├── templates ..... Templates zur Generierung von Plattforminstanzen
└── waf ..... Build- und Kontrollsystem
```

B.2 adapters

Das Verzeichnis *Adapters* enthält alle für die Bibliothek entwickelten Co-Simulationstransaktoren. Die Datei *ahb_adapter_types.h* definiert SystemC-Datentypen zur Abbildung von VHDL-Busschnittstellen. Der Adapter *tlmcpu_rtlcache_transactor.h/cpp* dient dem Anschluss des Prozessorsimulators (Integereinheit/IU) an den VHDL-*Cache* der *GRLIB*. Die Funktionen der übrigen Adaptoren erklären sich aus deren Namen. So enthalten zum Beispiel die Dateien *ahb_rtlbus_tlmslave_transactor.h/cpp* einen Adapter zum Anschluss des RTL-AHB-Busses and eine Komponente mit TLM-*Slave*-Schnittstelle. Aufbau und Funktionsweise von Adapterklassen wird in Abschnitt 3.8 beschrieben.

```
\adapters
├── ahb_adapter_types.h
├── ahb_rtlbus_tlmslave_transactor.h/cpp
├── ahb_rtlmaster_tlmbus_transactor.h/cpp
└── ahb_tlmbus_rtlslave_transactor.h/cpp
```

```
├── ahb_tlmmaster_rtlbus_transactor.h/cpp
├── apb_tlmbus_rtlslave_transactor.h/cpp
├── tlmcpu_rtlcache_transactor.h/cpp
└── wscript
```

B.3 build

Dieses Verzeichnis ist nach dem *Checkout* leer und dient der Speicherung von Objektfiles und ausführbaren Programmen. Die Position der Binärdateien nach dem Kompilieren entspricht der relativen Position des zugehörigen Quellcodes in den Verzeichnissen *models* oder *platforms*.
z.B. Das Kompilieren des Tests *ahbctrl.1.lt.test* aus dem Verzeichnis *./**models**/ahbctrl/tests/* mit dem Kommando *./waf build –target=ahbctrl.1.lt.test* erzeugt ein ausführbares Programm *ahbctrl.1.lt.test* im Verzeichnis *./**build**/models/ahbctrl/test*.

B.4 common

Das Verzeichnis *common* enthält von verschiedenen Modellen gemeinsam genutzte Infrastrukturkomponenten. Hier finden sich unter anderem der in Abschnitt 3.7 beschriebene MSC-*Logger*, zum Darstellung von Transaktionsabfolgen mit *Message Sequence Charts* (*msclogger.h/cpp*), der *Timing Monitor* zur Überprüfung des Transaktionszeitverhaltens (*timingmonitor.h/cpp*) und der Prototyp des in Abschnitt 3.6 eingeführten *Power*-Monitors (*powermonitor.h/cpp*). Ebenfalls in *common* befinden sich Hilfsfunktionen zur Konvertierung der *Endianess* (*vendian.h/cpp*) und Verwaltung von *Debug*-Ausgaben.

```
\common
├── msclogger.h/cpp
├── powermonitor.h/cpp
├── timingmonitor.h/cpp
├── vendian.h/cpp
├── verbose.h/cpp
└── wscript
```

B.5 contrib

Das *contrib* Verzeichnis enthält Patches und Beiträge zu externen Softwarepaketen.

```
\contrib
├── grambasockets
├── greenreg-4.0.0-writemask.patch
├── greensocs-4.0.0.patch
└── systemc-2.2_boost.patch
```

B.6 doc

Dieses Verzeichnis dient der Aufnahme der *Doxygen*-Dokumentation und ist nach dem *Checkout* leer. Die Dokumentation kann mit dem Kommando *./waf docs* erzeugt werden.

B.7 models/ahbctrl

Dieses Verzeichnis enthält den Quellcode der AHB-Bus TLM-IP (AHBCTRL) und alle zu dessen Verifikation entwickelten Tests. Das Modell kann mit dem Kommando *./waf build –target=ahbctrl*

kompiliert werden. Eine detaillierte Beschreibung des AHBCTRL befindet sich in Abschnitt 4.2.1.

```
\ahbctrl
├── ahbctrl.h/cpp
├── tests
│   ├── test1
│   │   ├── testbench.h/cpp
│   │   ├── top_lt/at/rtl.cpp
│   │   └── ahbctrl_wrapper.vhd
│   ├── ..
│   └── test9
└── wscript
```

B.8 models/apbctrl

Dieses Verzeichnis enthält den Quellcode der AHB/APB-Busbrücke TLM-IP (APBCTRL) und alle zu dessen Verifikation entwickelten Tests. Das Modell kann mit dem Kommando *./waf build –target=apbctrl* kompiliert werden. Eine detaillierte Beschreibung des APBCTRL befindet sich in Abschnitt 4.2.2.

```
\apbctrl
├── apbctrl.h/cpp
├── tests
│   ├── test1
│   │   ├── testbench.h/cpp
│   │   └── top_lt.cpp
└── wscript
```

B.9 models/mctrl

Dieses Verzeichnis enthält den Quellcode der Speicher-*Controller* TLM-IP (MCTRL) und alle zu dessen Verifikation entwickelten Tests. Das Modell kann mit dem Kommando *./waf build –target=mctrl* kompiliert werden. Eine detaillierte Beschreibung des MCTRL befindet sich in Abschnitt 4.3.3.

```
\mctrl
├── mctrl.h/cpp
├── tests
│   ├── test1
│   │   ├── testbench.h/cpp
│   │   └── top_lt/at/rtl.cpp
│   ├── ..
│   └── test10
└── wscript
```

B.10 models/memory

Dieses Verzeichnis enthält eine *array*-basierte (*arraymemory.h/cpp*) und eine *hashmap*-basierte Implementierung des mit dem Speichercontroller (MCTRL) zu verwendenden generischen Speichers. Anwendung und Funktionsweise werden in Abschnitt 3.3.3 beschrieben.

```
\memory
```

```
├── arraymemory.h/cpp
├── mapmemory.h/cpp
├── ext_erase.h
└── wscript
```

B.11 models/gptimer

Dieses Verzeichnis enthält den Quellcode der *Timer* TLM-IP (GPTIMER) und alle zu dessen Verifikation entwickelten Tests. Das Modell kann mit dem Kommando *./waf build –target=gptimer* kompiliert werden. Eine detaillierte Beschreibung des GPTIMER befindet sich in Abschnitt 4.3.1.

```
\gptimer
├── gpcounter.h/cpp
├── gptimer.h/cpp
├── tests
│   ├── gptimer_wrapper.vhd
│   ├── test1
│   │   ├── testbench.h/cpp
│   │   └── top_lt/rtl.cpp
│   └── test2
└── wscript
```

B.12 models/irqmp

Dieses Verzeichnis enthält den Quellcode der *Interrupt Controller* TLM-IP (IRQMP) und alle zu dessen Verifikation entwickelten Tests. Das Modell kann mit dem Kommando *./waf build – target=irqmp* kompiliert werden. Eine detaillierte Beschreibung des IRQMP befindet sich in Abschnitt 4.3.2.

```
\irqmp
├── irqmp.h/cpp
├── tests
│   ├── irqmp_wrapper.vhd
│   ├── test1
│   │   ├── testbench.h/cpp
│   │   └── top_lt/rtl.cpp
│   ├── ..
│   └── test4
└── wscript
```

B.13 models/mmu_cache

Dieses Verzeichnis enthält den Quellcode des Speicheruntersystems des LEON2/3 Prozessorsimulators (MMU_CACHE) und alle zu dessen Verifikation entwickelten Tests. Die Komponenten von MMU_CACHE sind in Unterverzeichnis *lib* zusammengefasst und können optional instantiiert werden. Aufbau und Funktionsweise des MMU_CACHE werden in Abschnitt 4.1 beschrieben. Das *Top-Level* des Moduls befindet sich in den Files *mmu_cache.h/cpp*. Die Kompilierung erfolgt mit Hilfe des Befehls *./waf build –target=mmu_cache*.

```
\mmu_cache
└── lib
```

```
    ├── cache_if.h
    ├── mem_if.h
    ├── dcio_payload_extensions.h/cpp
    ├── icio_payload_extensions.h/cpp
    ├── dvectorcache.h/cpp
    ├── ivectorcache.h/cpp
    ├── vectorcache.h/cpp
    ├── localram.h/cpp
    ├── mmu_cache.h
    ├── mmu_cache_if.h
    ├── mmu.h/cpp
    ├── mmu_if.h
    ├── nocache.h/cpp
    ├── tlb_adapter.h
    └── typ_adapters.vhd
├── tests
    ├── test1
    │   ├── mmu_cache_wrapper.vhd
    │   ├── testbench.h/cpp
    │   └── top_lt/at/rtl.cpp
    ├── ..
    └── test9
└── wscript
```

B.14 models/extern

Im Verzeichnis *models/extern* befinden sich Simulationsmodelle, die nicht im Rahmen dieser Arbeit entwickelt, aber zur Integration in *SoCRocket* angepasst wurden. Dies sind die LEON2/3 Integereinheit (LEON3) und die ihr zugrunde liegende *Trap*-Bibliothek (*trap-gen*), sowie das *SpaceWire*-Simulationsmodell der ESA.

```
\extern
├── LEON3
├── SpaceWire
├── trap-gen
└── wscript
```

B.15 utils

Das Verzeichnis *utils* enthält alle für *SoCRocket* neu entwickelten Bibliotheksbasisklassen. Die Klassen *ahbdevice*, *ahbmaster*, *ahbslave* und *apbdevice* dienen als Ausgangspunkt zur Modellierung von AMBA-Buskomponenten. Mit Hilfe von *memdevice* werden Eigenschaften zum Einsatz von Simulationsspeichern mit dem Speichercontroller MCTRL beschrieben. Die Klasse *clkdevice* wird verwendet, um alle *SoCRocket*-Modelle mit einer einheitlichen *Timing*-Schnittstelle auszustatten. Weitere Bibliotheksbasisklassen werden aus externen Datenpaketen (z.B. *GreenReg*) importiert. Das Konzept der Modellierung mit Bibliotheksbasisklassen wird in Abschnitt 3.1 erläutert.

```
\utils
├── ahbdevice.h/cpp
├── ahbmaster.h/cpp/tpp
├── ahbslave.h/cpp/tpp
└── apbdevice.h/cpp
```

```
├── clkdevice.h/cpp
├── memdevice.h/cpp
└── wscript
```

B.16 platforms

Dieses Verzeichnis ist für aus *SoCRocket*-Komponenten zusammengesetzte Virtuelle Plattformen reserviert. Nach dem *Checkout* befindet sich hier der in Abschnitt 6.1 beschriebene LEON3MP-Explorationsprototyp. Das *Top-Level* des Systems ist in *sc_main* beschrieben. Das Skript *json.lua* dient dem Laden der Konfiguration zum Simulationsbeginn. Die Standardkonfiguration ist in *config.json* gegeben. Diese kann beliebig überschrieben oder gegen eine der Konfigurationen aus dem Verzeichnis *templates* ausgetauscht werden.

```
\platforms
└── leon3mp
    ├── sc_main.cpp
    ├── config.json
    ├── json.lua
    └── wscript
```

B.17 signalkit

Dieses Verzeichnis enthält die Basismodule für das in Abschnitt 3.2.5 beschriebene Konzept zur TL-Signalkommunikation.

```
\signalkit
├── signalkit_h
│   ├── adapter.h
│   ├── base.h
│   ├── connect.h
│   ├── ifs.h
│   ├── infield.h
│   ├── in.h
│   ├── inout.h
│   ├── module.h
│   ├── out.h
│   └── selector.h
├── signalkit.h
└── wscript
```

B.18 software

Im Verzeichnis *Software* sind alle Programme und Softwaretests für den LEON2/3-Prozessorsimulator zusammengefasst. Das Unterverzeichnis *grlib_tests* enthält Kopien von Tests aus der GRLIB-Hardwarebibliothek, die ohne Änderungen in *SoCRocket* ausgeführt werden können.

```
\software
├── fft64
├── grlib_tests
├── mibench
├── hyperspectral
└── prom
```

```
├── rtems
├── trapgen
└── wscript
```

B.19 templates

Das Verzeichnis *templates* enthält eine Parameterbeschreibung (leon3mp.tpa) für den LEON3MP-Multiprozessorsimulator (Abschnitt 6.1 und verschiedene Beispielkonfigurationen für den Betrieb mit unterschiedlicher Anzahl an Prozessoren (z.B. leon3mp.singlecore.json). Parameterbeschreibung und Konfigurationen können im *Configuration Wizard* geladen und modifiziert werden.

```
\templates
├── leon3mp.tpa
├── leon3mp.singlecore.json
├── leon3mp.dualcore.json
├── leon3mp.tricore.json
├── leon3mp.quadcore.json
├── leon3mp.pentacore.json
├── leon3mp.hexacore.json
├── leon3mp.heptacore.json
└── leon3mp.octocore.json
```

C Simulationsergebnisse

Dieser Anhang enthält zusätzliche detaillierte Simulationsergebnisse für die in Abschnitt 6.3 beschriebenen Benchmarks. Wie bereits am Beispiel der JPEG-Kompression verdeutlicht, wurden alle Tests, zur Bestimmung von Simulationsgenauigkeit und -geschwindigkeit, sowohl auf der Virtuellen Plattform (LT- und AT-Modus), als auch auf dem RTL-Referenzmodell ausgeführt.

C.1 FIR2 - Simulation

Abbildung C.1 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den FIR2-Benchmark. Abbildungen C.2 - C.5 enthalten die Zugriffsstatistiken für Caches und Systembus.

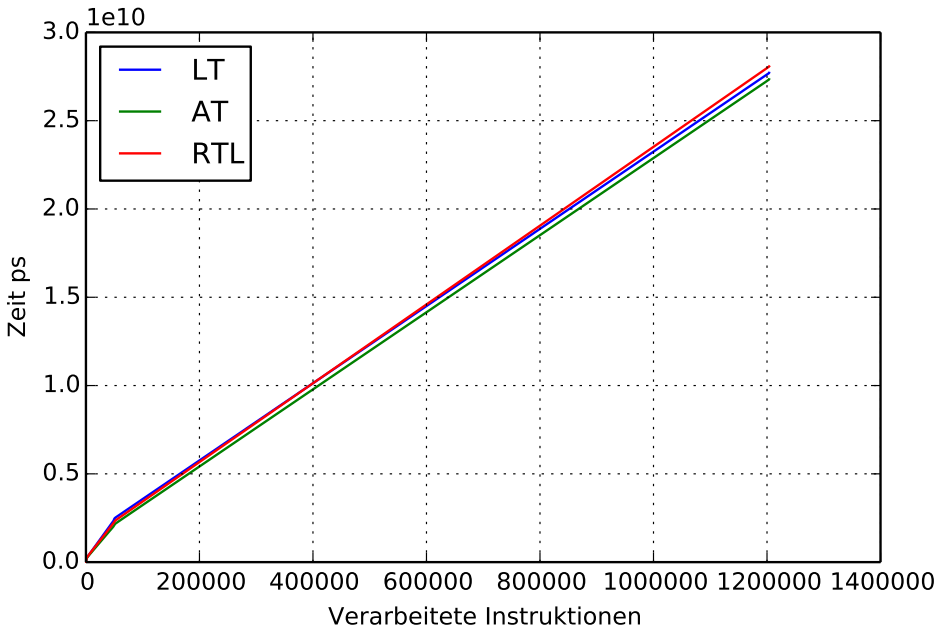


Abbildung C.1: FIR2 - Simulationsverlauf

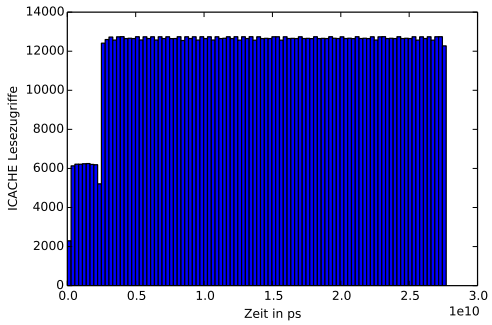


Abbildung C.2: FIR2 - Histogramm für ICACHE-Lesezugriffe

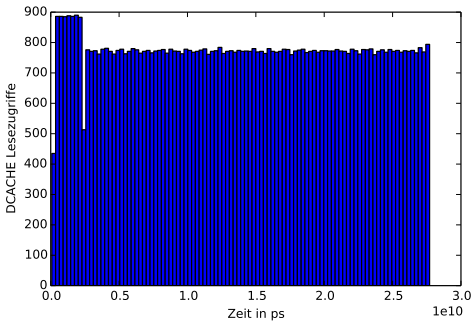


Abbildung C.3: FIR2 - Histogramm für DCACHE-Lesezugriffe

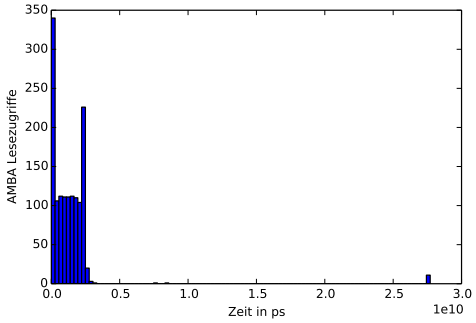


Abbildung C.4: FIR2 - Histogramm AHB-Lesezugriffe

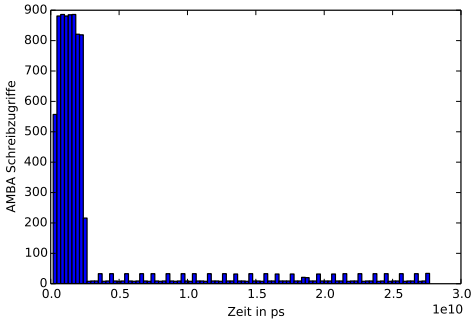


Abbildung C.5: FIR2 - Histogramm für AHB-Schreibzugriffe

C.2 ENGINE - Simulation

Abbildung C.6 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den *Engine*-Benchmark. Abbildungen C.7 - C.10 enthalten die Zugriffsstatistiken für Caches und Systembus.

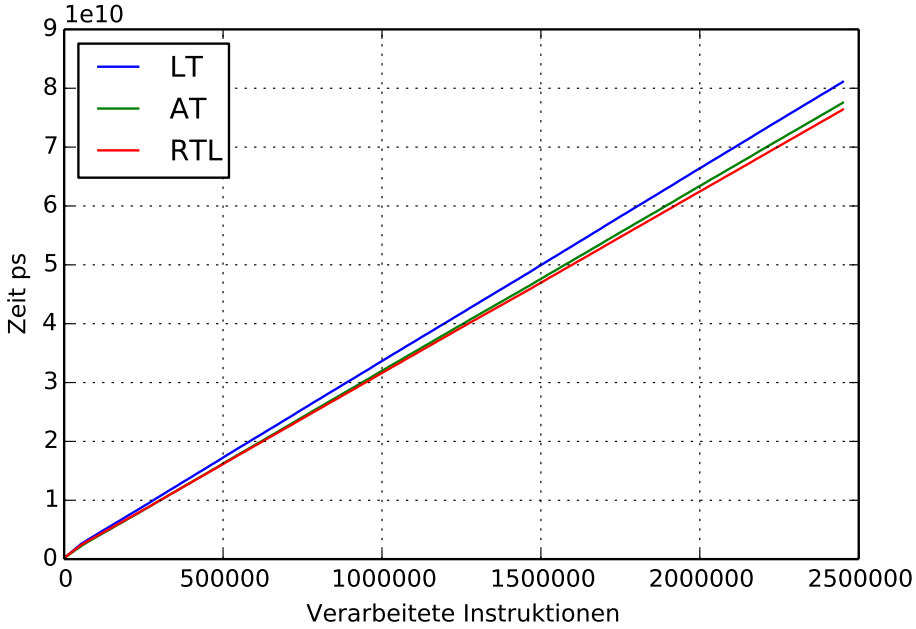


Abbildung C.6: engine - Simulationsverlauf

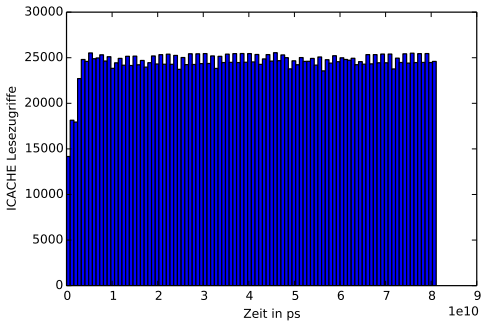


Abbildung C.7: engine - Histogramm für ICACHE-Lesezugriffe

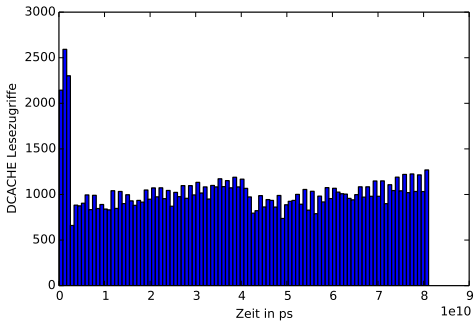


Abbildung C.8: engine - Histogramm für DCACHE-Lesezugriffe

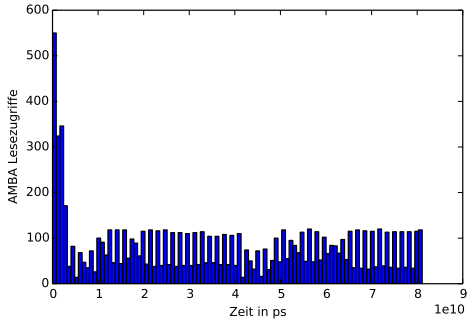


Abbildung C.9: engine - Histogramm AHB-Lesezugriffe

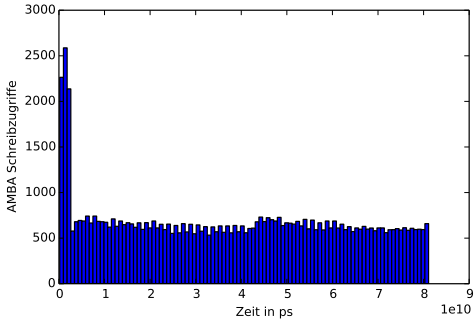


Abbildung C.10: engine - Histogramm für AHB-Schreibzugriffe

C.3 CRC - Simulation

Abbildung C.11 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den CRC-Benchmark. Abbildungen C.12 - C.15 enthalten die Zugriffsstatistiken für Caches und Systembus.

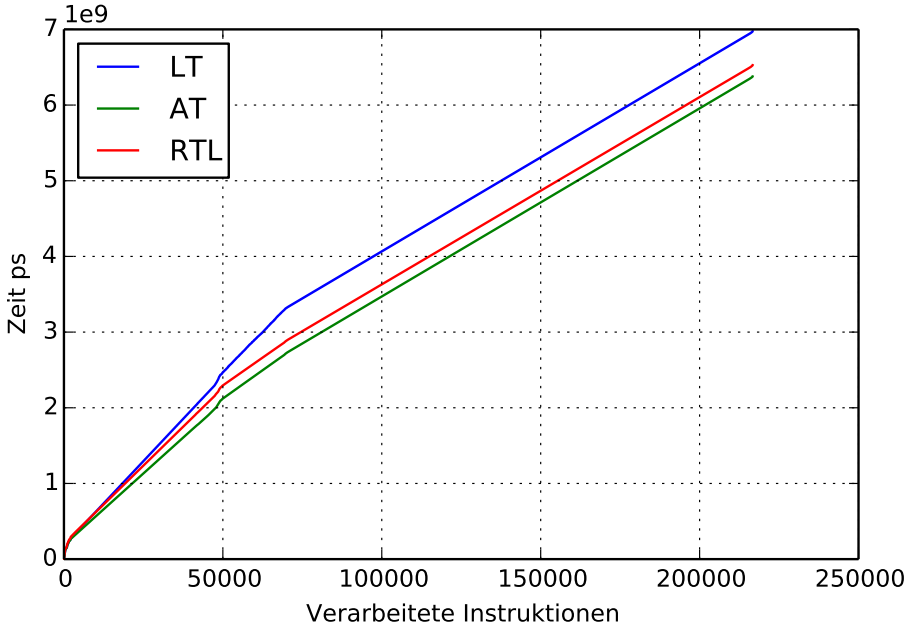


Abbildung C.11: CRC - Simulationsverlauf

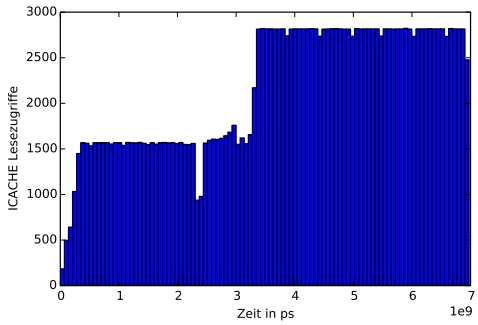


Abbildung C.12: CRC - Histogramm für ICACHE-Lesezugriffe

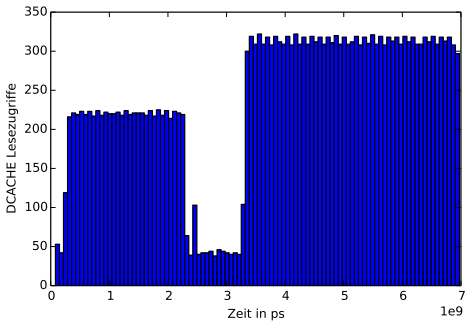


Abbildung C.13: CRC - Histogramm für DCACHE-Lesezugriffe

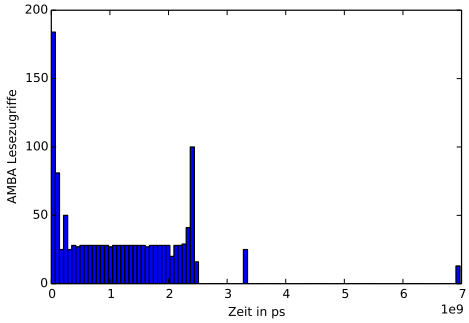


Abbildung C.14: CRC - Histogramm AHB-Lesezugriffe

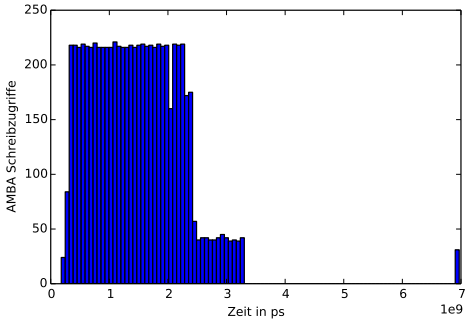


Abbildung C.15: CRC - Histogramm für AHB-Schreibzugriffe

C.4 DES - Simulation

Abbildung C.16 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den DES-Benchmark. Abbildungen C.17 - C.20 enthalten die Zugriffsstatistiken für Caches und Systembus.

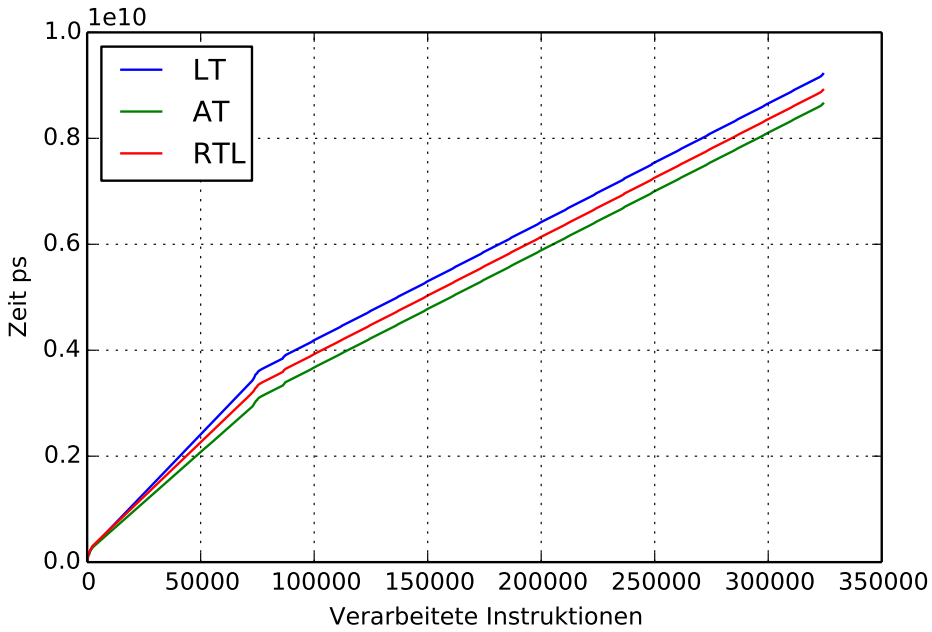


Abbildung C.16: DES - Simulationsverlauf

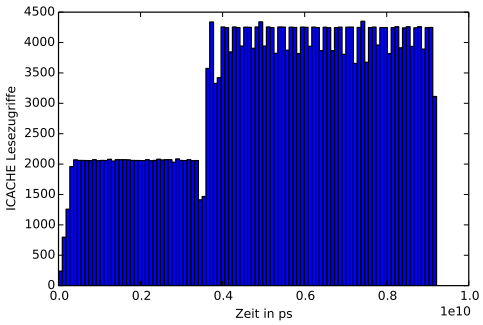


Abbildung C.17: DES - Histogramm für ICACHE-Lesezugriffe

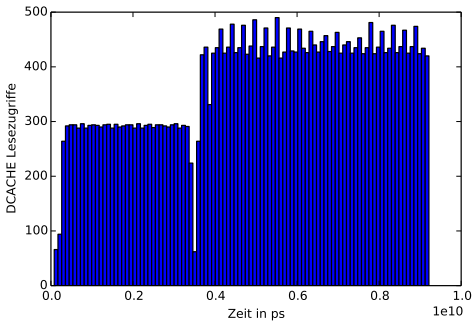


Abbildung C.18: DES - Histogramm für DCACHE-Lesezugriffe

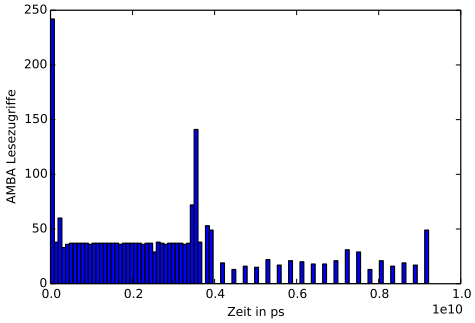


Abbildung C.19: DES - Histogramm AHB-Lesezugriffe

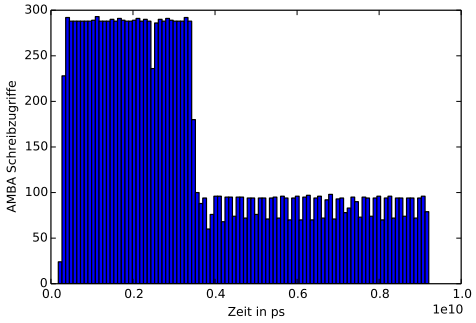


Abbildung C.20: DES - Histogramm für AHB-Schreibzugriffe

C.5 FFT - Simulation

Abbildung C.21 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den FFT-Benchmark. Abbildungen C.22 - C.25 enthalten die Zugriffsstatistiken für Caches und Systembus.

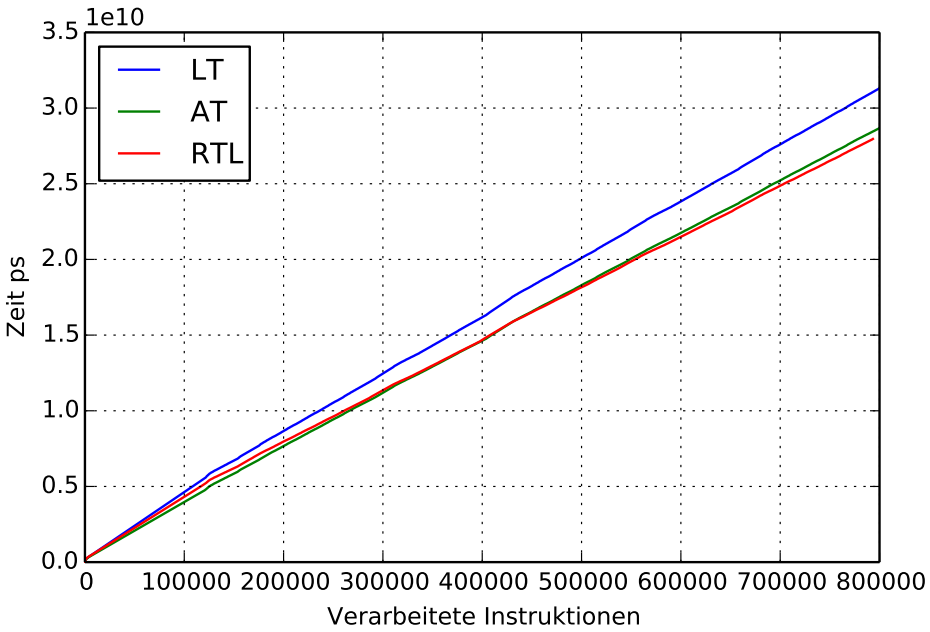


Abbildung C.21: FFT - Simulationsverlauf

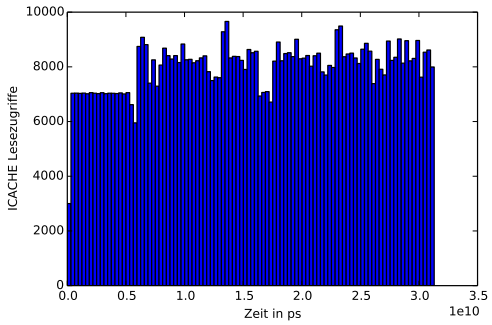


Abbildung C.22: FFT - Histogramm für ICACHE-Lesezugriffe

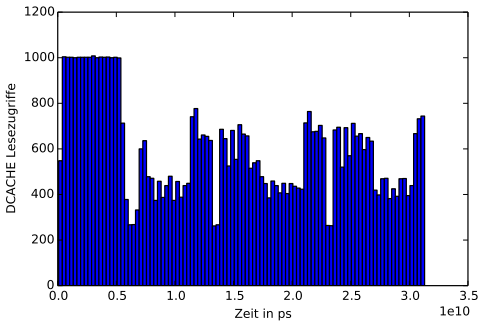


Abbildung C.23: FFT - Histogramm für DCACHE-Lesezugriffe

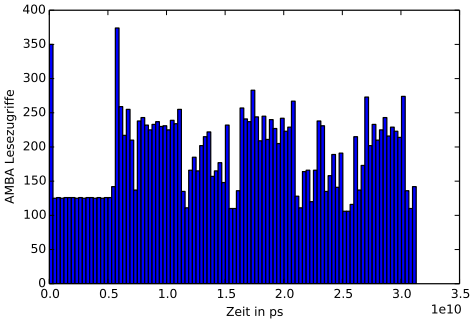


Abbildung C.24: FFT - Histogramm AHB-Lesezugriffe

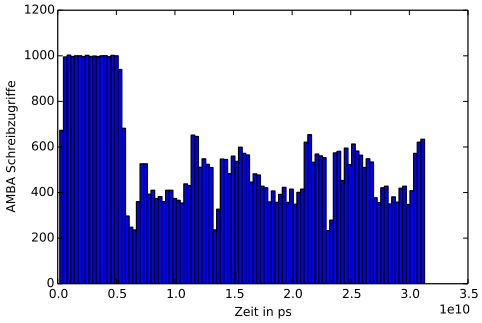


Abbildung C.25: FFT - Histogramm für AHB-Schreibzugriffe

C.6 Hanoi - Simulation

Abbildung C.26 zeigt die Entwicklung der Simulationszeit relativ zur Anzahl der verarbeiteten Instruktionen für den Hanoi-Benchmark. Abbildungen C.27 - C.30 enthalten die Zugriffsstatistiken für Caches und Systembus.

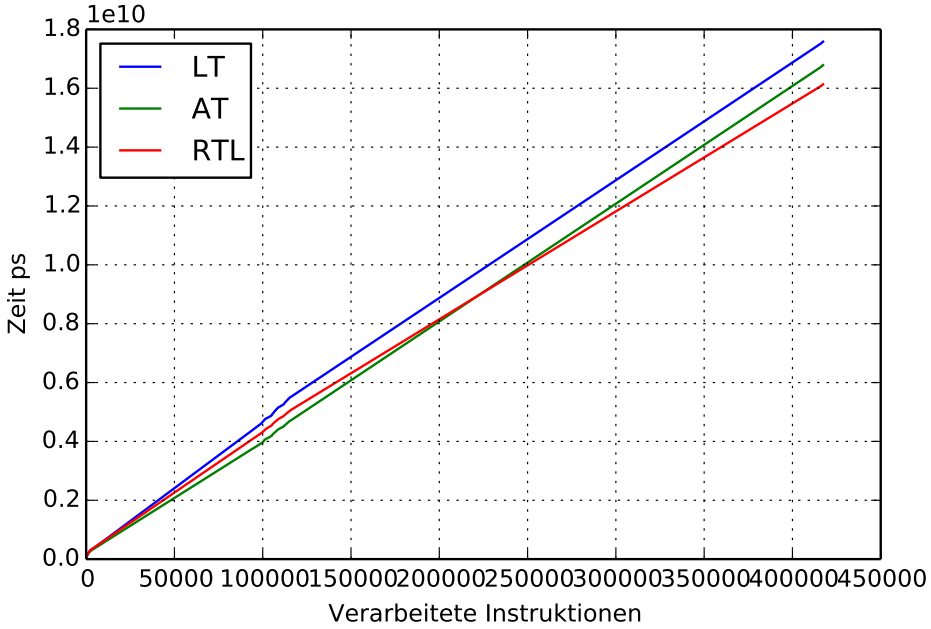


Abbildung C.26: Hanoi - Simulationsverlauf

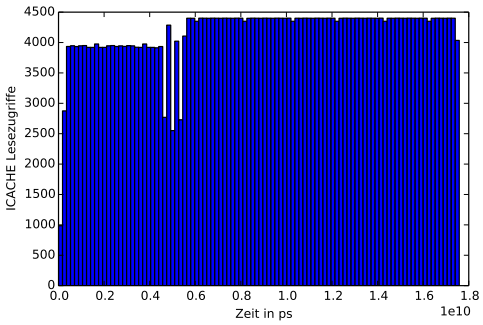


Abbildung C.27: Hanoi - Histogramm für ICACHE-Lesezugriffe

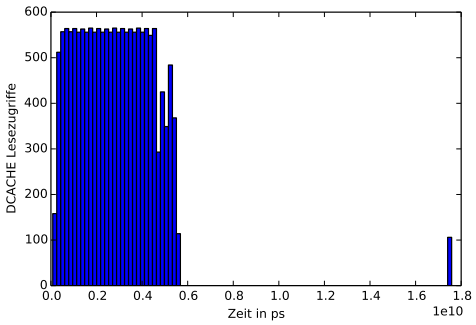


Abbildung C.28: Hanoi - Histogramm für DCACHE-Lesezugriffe

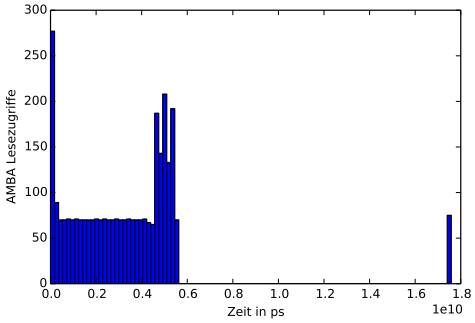


Abbildung C.29: Hanoi - Histogramm AHB-Lesezugriffe

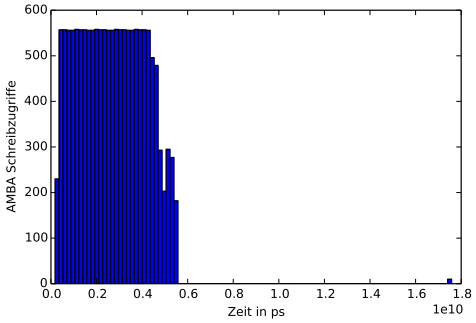


Abbildung C.30: Hanoi - Histogramm für AHB-Schreibzugriffe

D Lebenslauf von Thomas Schuster

Persönliche Daten	
	Geboren am 28.03.1976 in Räckelwitz (Sachsen) Verheiratet
Schulausbildung	
1982 - 1992	Realschulabschluss
1992 - 1995	Ausbildung zum Einzelhandelskaufmann
1997 - 2000	Abitur am Abendgymnasium Bautzen
Hochschulstudium	
2000 - 2005	Studium der Informationssystemtechnik an der Technischen Universität Dresden Schwerpunkte: Architektur Verteilter Systeme (Informatik) Nachrichtentechnik (Elektrotechnik) Diplomarbeit am *Interuniversity MicroElectronic Center* (IMEC) in Belgien Thema: *Architecture Exploration for Coarse Grained Reconfigurable Architectures* Abschluss als Diplom Ingenieur
Berufstätigkeit	
1995 - 2000	Einzelhandelskaufmann, Flamme GmbH & Co KG
2005 - 2007	Forschungsingenieur, IMEC, Belgien Entwicklung von Hardware- und Softwarekomponenten für *Software-Defined-Radio*
2007 - 2010	Wissenschaftlicher Mitarbeiter am Intel Stiftungslehrstuhl für *VLSI Design*, Institut für Datentechnik und Kommunikationsnetze, Technische Universität Braunschweig (Prof. Dr.-Ing. Mladen Berekovic)
2011	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Rechnerarchitektur und Kommunikation, Institut für Informatik, Friedrich-Schiller-Universtität Jena (Prof. Dr.-Ing. Mladen Berekovic)
2012 - 2014	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Technische Informatik (C3E), Institut für Theoretische Informatik, Technische Universtität Braunschweig (Prof. Dr.-Ing. Mladen Berekovic)