Ein Betriebssystem für konfigurierbare Hardware

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herrn Dipl.-Inf. Dipl.-Phys. David Krutz
geboren am 14.12.1976 in Schwedt

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Beate Meffert
2. Prof. Dr. Klaus Müller-Glaser
3. Prof. Dr. Reinhold Orglmeister

eingereicht am:	5. Juli 2006
Tag der mündlichen Prüfung:	21. November 2006

Abstract

This work investigates the possibility of describing a hardware design independent of special hardware. This is realized with the concept of an operating system. The re-use of operating system modules reduces the time of development and also increases the reliability. Additionally, the change of a development platform has no influence on the application algorithm anymore. In order to apply the concept of an operating system special constraints have to be fulfilled by the hardware description language, which is not supported by VHDL. For that reason a structure compiler has been developed. The structure compiler connects the application program with the operating system modules and produces a VHDL program, which can be used to simulate or to program the FPGA with the typical VHDL development tools.

In the progress of developing the operating system concept for reconfigurable hardware it was realized that such a concept can only be used in connection with a design methodology for heterogeneous systems. In this work a design methodology based on a declarative language represented as signal flow graph is discussed.

The operating system concept for reconfigurable hardware was tested on different FPGA boards. For these cards an operating system was developed. The operating system contains modules for the communication with the PC over different interfaces as well as modules for accessing different exterior peripheries, i.e. memory. Additionally, the integration of processors as part of the configurable hardware within the operating system concept was investigated. For the verification of the structure compiler and the operating system modules some examples have been developed. The operating system concept for configurable hardware was also applied in different projects.

Keywords:
heterogeneous system, operating system, VHDL, FPGA

Zusammenfassung

In dieser Arbeit wird die Möglichkeit der Unterstützung des Hardwareentwurfs mit VHDL durch ein Hardwarebetriebssystem untersucht. Durch die Wiederverwendung von Betriebssystemmodulen sollen die Entwicklungszeit verkürzt, die Nachnutzbarkeit von Entwürfen verbessert und die Zuverlässigkeit erhöht werden. Um ein Betriebssystemkonzept umzusetzen, müssen spezielle Anforderungen an die Programmiersprache gestellt werden. Diese werden von VHDL nicht erfüllt. Daher wird ein Strukturcompiler vorgestellt, der unter Beibehaltung der Syntax der Sprache VHDL den zusätzlichen Anforderungen gerecht wird. Der Strukturcompiler verbindet das Anwendungsprogramm mit den Betriebssystemmodulen und erzeugt daraus ein VHDL-Programm, das mit den typischen FPGA-Entwicklungswerkzeugen simuliert oder synthetisiert werden kann.

Bei der Entwicklung des Betriebssystems für konfigurierbare Hardware hat sich herausgestellt, dass sich dieses nur eingebettet in ein Gesamtkonzept für den Entwurf von heterogene Systeme sinnvoll anwenden lässt. Deshalb wird in dieser Arbeit eine Methode für die Entwicklung von heterogenen Systemen auf Basis eines Signalflussgraphen diskutiert.

Angewendet wurde das Betriebssystemkonzept auf verschiedenen FPGA-Karten, sowohl käuflich erworbene als auch Eigenentwicklungen. Das für diese Karten erstellte Betriebssystem umfasst dabei Module zur Kommunikation zwischen FPGA und PC sowie zur Anbindung verschiedener externer Peripheriegeräte, wie z.B. Speicher. Es wurde ebenfalls untersucht wie Prozessoren als Bestandteil der konfigurierbaren Hardware in das Betriebssystemkonzept integriert werden können. Im Rahmen dieser Arbeit wurden auch viele Beispielanwendungen untersucht. Diese wurden einerseits zum Testen des Strukturcompilers und der Betriebssystemmodule benutzt. Andererseits fand das Betriebssystemkonzept für konfigurierbare Hardware auch Anwendung in verschiedenen Projekten.

Schlagwörter:
heterogenes System, Betriebssystem, VHDL, FPGA

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Beim Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) gibt es eine Vielzahl an Projekten aus den verschiedensten Anwendungsbereichen, welche die Entwicklung von heterogenen Systemen zur Signalverarbeitungen beinhalten (Kleinsatellit Bird [BBG+04], Verkehrsszenenanalyse [DRK+03], Digitale Luftbildkamera ADS40 [FSW00], Panoramakamera Eyescan [SKR+01]).

Alle diese Systeme besitzen einen ähnlichen Aufbau. Dieser besteht aus Sensorelementen mit entsprechender analoger Vorverarbeitung und Ansteuerung, einer digitalen Vorverarbeitung, aufgebaut aus einem oder mehreren FPGA[1] und einem oder mehreren Prozessoren in Form eines DSP[2] oder eines Standard-Prozessors zur Endverarbeitung. Aufgrund dieses Aufbaus werden solche Systeme auch als heterogene Systeme bezeichnet. Die Heterogenität bezieht sich dabei sowohl auf das Vorhandensein von Hardware- und Softwarekomponenten als auch auf die Verbindung von mechanischen, optischen und elektronischen Elementen und die Verknüpfung von analogen mit digitalen Komponenten. Entwicklungen für solche Systeme sind immer verbunden mit einem hohen Maß an Komplexität. Dies rührt einerseits aus der Anzahl der verschiedenen Komponenten her, andererseits auch aus der Heterogenität selbst.

Der Vorteil eines heterogenen Systems liegt in der optimalen Anpassung des Systems an die äußeren Bedingungen. So haben Prozessoren den Vorteil, universell einsetzbar und leicht programmierbar zu sein. Sie besitzen jedoch den Nachteil, dass sie eine komplexere Einbindung in ein Gesamtsystem und damit höhere Kosten verursachen und aufgrund der fehlenden bzw. eingeschränkten Parallelität einen geringeren Datendurchsatz haben. Anwenderspezifische Hardwarelösungen besitzen diese Beschränkung nicht, mit dem Nachteil, dass die Entwicklung umfangreicher und damit auch kostspieliger

[1] Field Programmable Gate Array
[2] Digital Signal Processor

ist. Ein weiterer Vorteil einer Hardwarelösung liegt in der besseren Anpassung an die Systemeigenschaften, was wiederum Einsparung an z.B. Platz und Energie bedeutet. Durch Verbindung beider Lösungen in einem heterogenen System können die Vorteile der homogenen Systeme vereint werden, um eine optimale Lösung für ein Gesamtsystem zu finden.

Bei der Betrachtung von heterogenen Systemen spielen die Begriffe des Entwurfs, des Modells und der Simulation eine wichtige Rolle. Unter Modellierung wird die Erzeugung einer formalen Beschreibung eines Systems oder Subsystems verstanden. Diese kann sowohl mathematisch durch Formeln als auch algorithmisch durch Prozeduren erfolgen. Im ersten Fall kann z.B. eine Beschreibung der physikalischen Einbettung des Systems zugrunde liegen, durch welche die Eigenschaften des Modells definiert sind. Bei der algorithmischen Beschreibung steht eine Verhaltensbeschreibung im Vordergrund, d.h. die Reaktion des Systems oder Subsystems auf äußere Stimuli.

Der Prozess des Entwurfs umfasst die Definition eines Systems oder Subsystems. Gewöhnlich beinhaltet dies die Definition von einen oder mehreren Modellen des Systems. Um zu einem Modell zu gelangen, wird bei komplexen Systemen meist die Top-Down-Methode favorisiert. Ausgehend von einer allgemeinen Beschreibung des Systems findet eine kontinuierliche Verfeinerung der Submodelle statt bis die gewünschte Funktionalität des Systems erreicht ist und die Randbedingungen erfüllt werden. Eine spezielle Klasse von Modellen werden Simulationen genannt. Dabei handelt es sich um ausführbare Modelle.

1.1 Einordnung der Arbeit

Für den Entwurf heterogener Systeme auf abstrakter (algorithmischer) Ebene stehen verschiedene Methoden und Werkzeuge zur Verfügung, z.B. Matlab/Simulink [Gro01], Ptolemy [HLL+03]. Ein Nachteil dieser Systeme ist, dass sie keinen vollständigen Entwurf von der Algorithmusbeschreibung bis zu den ausführbaren Programmen für die verschiedenen Zielplattformen ermöglichen. Besonders im Hinblick auf die stark variierenden FPGA-Plattformen bieten sie keine Möglichkeit, die Entwicklungszyklen durch Wiederverwendung von Modulen für verschiedene Plattformen zu verkürzen.

Um die Lücke zu schließen, wird in dieser Arbeit ein Betriebssystemkonzept für VHDL[3] vorgestellt, das in Kombination mit anderen Entwicklungswerkzeugen den Entwurf vom Algorithmus zum System vereinfacht. Der große Vorteil eines solchen Systementwurfs liegt in der Wiederverwendbarkeit von Modulen. Gerade in der Signalverarbeitung gibt es eine Vielzahl

[3]Very High Speed Integrated Circuit Hardware Description Language

an Operationen, die bei verschiedenen Problemstellungen immer wieder benutzt werden können. Neben der Reduzierung der Entwicklungszeit für ein System und damit der Verkürzung der Time-To-Market, die heutzutage ein immer wichtigeres Kriterium ist, liegt ein weiterer Vorteil in der Erhöhung der Zuverlässigkeit.

Die hier vorgestellte Methode des Entwurfs für ein heterogenes System ist dabei nicht nur beschränkt auf Systeme zur Signalverarbeitung. Jedes System, das programmierbare Hardware, in Form z.B. eines FPGA, mit Prozessoren verknüpft, kann mit dieser Methode erfasst werden. Natürlich werden sich die Vorteile des Entwurfs an dem Grad der Wiederverwendung von Modulen messen lassen. Dabei wird es auch Systeme geben, bei denen sich der Einsatz dieser Methode nicht lohnt.

Bei der Entwicklung des Betriebssystemkonzepts hat sich herausgestellt, dass sich dieses nur eingebettet in ein Gesamtkonzept für heterogene Systeme sinnvoll anwenden lässt. Die Untersuchung von Methoden für den Entwurf heterogener Systeme sind Bestandteil vieler Forschungsprojekte [Bal97, HBK96, HLL+03, VCA94]. Diese befinden sich jedoch noch nicht in einem Stadium, in dem diese für Produktentwicklungen einsetzbar sind. Prinzipiell gibt es drei Hauptprobleme, die bei der Umsetzung eines Konzepts für heterogene Systeme gelöst werden müssen. Zum einen ist es die Beschreibung des Algorithmus. Hier gibt es die Möglichkeit, auf bereits vorhandenen Sprachen aufzubauen. Damit ist jedoch der Nachteil verbunden, dass für die Erzeugung der Programme für die konfigurierbare Hardware ein entsprechender Transcompiler benötigt wird. Das gleiche Problem tritt auf, wenn eine reine Hardwarebeschreibungssprache als Basis der Systembeschreibung benutzt wird. Die durch Transcompiler erzeugten Programme sind jedoch nicht optimal an die Zielplattform angepasst. Deshalb wird hier die Beschreibung des Systemalgorithmus auf Basis einer deklarativen Beschreibung geschehen. Die Darstellung der deklarativen Beschreibung erfolgt dabei in Form eines Graphen. Mit Hilfe der deklarativen Beschreibung wird nicht mehr ausgedrückt, wie bestimmte Ergebnisse berechnet werden sollen, sondern nur noch, welches Ergebnis gewünscht ist. Um eine Realisierung einer solchen Beschreibung in Form eines Programms zu erzeugen, muss zu jeder möglichen Zielplattform eine Realisierung der Operation vorhanden sein. Für die konfigurierbare Hardware wird sich zeigen, dass durch Benutzung von standardisierten Schnittstellen und eines Betriebssystems unabhängig von der Zielplattform nur eine Beschreibung jedes Operators notwendig ist.

Das zweite Problem besteht darin, eine Partitionierung des Systems zu finden, d.h. eine Zerlegung des Systemalgorithmus in Teilalgorithmen, die auf den verschiedenen Zielplattformen ausgeführt werden sollen. Dabei spielen die Randbedingungen des Systems eine große Rolle, wie z.B. Datendurchsatz,

Ressourcenverbrauch, Latenzzeiten.

Die Paritionierung kann automatisch oder manuell erfolgen. Automatische Lösungsgenerierungen werden im Bereich des Hardware-Software-Codesigns [Bal97, HBK96, VCA94] untersucht. Eine manuelle Partitionierung entspricht einem Try-and-Error-Verfahren zur Findung einer Lösung, die den Systembedingungen genügt. Dieses Verfahren ist jedoch nur dann sinnvoll anwendbar, wenn die Entwicklung einer Lösung für eine konkrete Zielplattform in kurzer Zeit durchgeführt werden kann. Durch die Einführung von standardisierten Betriebssystemschnittstellen zum Zugriff auf spezielle Komponenten einer Zielplattform kann der Schritt von der manuellen Partitionierung zum Programm für die Zielplattform automatisieren werden.

Die dritte große Herausforderung bei der Umsetzung eines Konzepts für heterogene Systeme ist die Benutzung der verschiedenen Verbindungen zwischen den Komponenten des Gesamtsystems. Neben der zusätzlichen Komplexität, die durch die Anzahl der Verbindungen hervorgerufen wird und Einfluss auf den Partitionierungsprozess hat, ergibt sich auch das Problem, wie die unterschiedlichen Verbindungen von einem Programm universell angesprochen werden. Auch hier helfen das Betriebssystemkonzept und die Definition von standardisierten Schnittstellen weiter. Sowohl für Prozessoren als auch für konfigurierbare Hardware müssen Schnittstellen definiert werden, die einen einheitlichen Zugang zu den Verbindungen gestatten. Für einen Prozessor werden diese Schnittstellen durch die Implementierung von Bibliotheken realisiert. Für konfigurierbare Hardware geschieht dies durch die Implementierung von Betriebssystemmodulen. Die Module sind dabei so ausgelegt, dass über einer physikalischen Verbindung beliebig viele logische Verbindungen aufgebaut werden können.

Wie anhand der erläuterten Probleme erkennbar ist, hilft ein Betriebssystemkonzept, die Komplexität der verschiedenen Plattformen und Verbindungen in den Griff zu bekommen. Ein Betriebssystem ist dabei durch die Betriebssystemschnittstellen definiert. Durch diese werden die konkreten Realisierungen der unterliegenden Hard- und Software abstrahiert. Für eine Prozessorplattform stellt dies keine Neuerung dar. Hier werden bereits seit Jahrzehnten standardisierte Schnittstellen benutzt. Diese beziehen sich auf den Zugriff auf Hardwarekomponenten (z.B. BIOS[4], VGA[5]), die Benutzung der Betriebssystemfunktionen (z.B. POSIX[6]), die Benutzung von Bibliotheken (z.B. STL[7]) oder die Benutzung einer Programmiersprache (z.B. C, Java).

[4]Basic Input Output System
[5]Video Graphics Array
[6]Portable Operating System Interface for UniX
[7]Standard Template Library

Auf Seite der konfigurierbaren Hardware ist dies so nicht ohne weiteres umsetzbar. So wird zwar von der Sprache VHDL die Wiederverwendung von VHDL-Programmen in verschiedenen FPGA-Bausteinen durch die Benutzung von VHDL-Bibliotheken gewährleistet. Sobald sich jedoch die externen Komponenten eines FPGA verändern, d.h. beispielsweise ein Speichermodul von SRAM[8] auf SDRAM[9] wechselt, zieht dies eine aufwendige Anpassung des VHDL-Programms nach sich.

Das Problem liegt dabei in der Definition der Hardwarebeschreibungssprachen. Dabei spielt es keine Rolle, ob VHDL oder andere Sprachen wie z.B. Verilog betrachtet werden. Beide besitzen nicht die Voraussetzungen, um ein Betriebssystem umzusetzen. Damit eine Sprache von einem Betriebssystemkonzept unterstützt wird, müssen zwei grundsätzliche Bedingungen gegeben sein [Tan92].

Zum einen muss das Prinzip des *Information Hiding* umsetzbar sein. Darunter wird verstanden, dass die Benutzung einer Schnittstelle unabhängig ist von der Art der Implementierung der Funktionalität der Schnittstelle. Die Implementierung ist für die Schnittstellenbenutzung unsichtbar. Nur auf diese Weise können die verschiedenen Hardwarerealisierungen von dem Anwendungsprogramm separiert werden. Dieses Prinzip wird aber durch die Definition der Modulschnittstellen verletzt. Da jedes externe Signal, auf das ein Modul zugreift, Bestandteil der Schnittstelle dieses Moduls sein muss, ist hier eine Umsetzung dieses Prinzip nicht möglich.

Eine zweite wichtige Eigenschaft, die umsetzbar sein muss, ist das Ressourcenmanagement, d.h. die Kontrolle, welche Instanz welche Ressource benutzt, die Bewilligung von Ressourcenanfragen und die Vermittlung bei Ressourcenkonflikten. Als Ressourcen kommen dabei sowohl interne als auch externe Strukturen in Frage, wie z.B. Speichermodule oder die Kommunikationswege. Damit ein Ressourcenmanagement möglich ist, muss das Betriebssystem in der Lage sein, gleichzeitig Kontrolle über mehrere gleichartige oder verschiedene Betriebssystemschnittstellen zu haben. In Softwarebetriebssystemen ist dies kein Problem, da Programme immer sequentiell ausgeführt werden und durch Nutzung von globalen Variablen Informationen über Schnittstellen global zur Verfügung gestellt werden können. In VHDL und Verilog ist dies nicht möglich. Es gibt keine Möglichkeit, Informationen ohne Signalzuweisungen, und damit Schnittstellenelementen, anderen Modulen zur Verfügung zu stellen. Mit der Komponenten-Architektur-Verknüpfung in VHDL wird immer nur eine Schnittstelle mit einem Modul verknüpfen. Um trotzdem ein Hardwarebetriebssystem umsetzen zu können, muss es eine

[8]Static Random Access Memory
[9]Synchronous Dynamic Random Access Memory

Möglichkeit geben, mehrere Betriebssystemschnittstellen mit einem Betriebssystemmodul zu verknüpfen.

1.2 Ziel der Arbeit

Um die Probleme des Ressourcenmanagements und der Verletzung des Information Hiding zur Umsetzung eines Betriebssystemkonzepts für Hardwarebeschreibungssprachen zu lösen, sollen eine Methode und ein unterstützendes Werkzeug entwickelt werden. Das Werkzeug soll ein Präcompiler sein, der ein VHDL-konformes Anwendungsprogramm mit Betriebssystemmodulen verknüpft und daraus ein VHDL-konformes Programm erzeugt, das mit den typischen VHDL-Werkzeugen weiterverarbeitet werden kann. Eine erste Version dieses Compilers wurde bereits in [Kru02][KW03] beschrieben. Im Rahmen der vorliegenden Arbeit soll der Compiler jedoch um zusätzliche Eigenschaften erweitert und die Einbettung in eine Methode zur Entwicklung von heterogenen Systemen untersucht werden.

Die entwickelte Methode soll auf verschiedenen FPGA-Karten und in verschiedenen Projekten angewendet werden. Dafür wird es nötig sein, ein Betriebssystem, d.h. eine Sammlung von Betriebssystemschnittstellen mit entsprechenden Betriebssystemmodulen zu definieren. Die Betriebssystemmodule müssen dabei den FPGA-Karten angepasst werden. Bei den Karten handelte es sich zum einen um die RC1000-PP Karte der Firma Celoxica [Cela, Celb]. Zum anderen sollen verschiedene vom DLR im Rahmen des Projekts OIS[10] [DRK+03] entwickelte Karten benutzt werden, die mit verschiedenen Xilinx-FPGA, mehreren Kommunikationsschnittstellen (PCI[11], USB[12], Ethernet) und mehreren Speicheranbindungen (SDRAM, SRAM) ausgestattet sind. Bei der Definition der Betriebssystemschnittstellen soll der Schwerpunkt auf der Anbindung der verschiedenen externen Komponenten liegen. Dafür wird es notwendig sein, Kommunikationsprotokolle zwischen FPGA und PC zu definieren.

Außerdem soll dieses Konzept auch im Rahmen des Projekts ModoS[13] [Sup03] angewandt werden. Hierbei handelt es sich um ein Projekt, das die Bestimmung der Position von frei beweglichen Objekten in Gebäuden zum Ziel hat. Der in einem FPGA umgesetzte Algorithmus soll dabei ein synchronisiertes Einlesen und eine Vorprozessierung der Daten von verschiedensten Sensorelementen, wie Kameras, Neigungssensoren und inertialen Messeinhei-

[10] Optische Informationssystems zur Verkehrsszenenanalyse und Verkehrslenkung
[11] Peripheral Component Interconnect
[12] Universal Serial Bus
[13] Multisensorielle Modellierung mittels referenzierter optischer Sensoren

ten ermöglichen.

1.3 Aufbau der Arbeit

Die Arbeit untergliedert sich in folgende Kapitel: In dem nachfolgenden Kapitel 2 werden die für die Arbeit erforderlichen Grundlagen beschrieben. Dazu zählt eine Einführung in die konfigurierbare Hardware, deren Untergliederung, Aufbau und Anwendungsgebiete und eine Einführung in die Hardwarebeschreibungssprache VHDL. Neben dem klassischen Entwurf mit einer Hardwarebeschreibungssprache werden auch alternative Entwurfsmethoden für heterogene Systeme im Rahmen einer Betrachtung des aktuellen Stands der Technik in Kapitel 3 aufgezeigt. Im Kapitel 4 werden die Probleme eines Entwurfs von heterogenen Systemen beschrieben und die Umsetzung im Rahmen dieser Arbeit. Gefolgt wird dieses Kapitel von einer theoretischen Betrachtung eines Lösungsansatzes zur Umsetzung eines Betriebssystemkonzepts in Hardwarebeschreibungssprachen. Im nachfolgenden Kapitel 6 wird auf die Funktionsweise und die Definition verschiedener Kommunikationsprotokolle eingegangen. Die konkrete Realisierung eines Werkzeuges zur Umsetzung des Betriebssystemkonzepts in VHDL, der so genannten Strukturcompiler, wird in Kapitel 7 erläutert. Im Kapitel 8 werden dann wichtige Aspekte beschrieben, die bei der praktischen Arbeit mit dem Strukturcompiler und der Methode des Systementwurfs beachtet werden müssen. Dazu werden die verschiedenen Hardwaresysteme vorgestellt, auf denen das System angewendet wurde, und einfache Beispiele, welche die Arbeit mit dem System aufzeigen. Das letzte Kapitel gibt dann eine kurze Zusammenfassung und einen Ausblick über zukünftige Entwicklungen im Bereich des Hardwarebetriebssystemkonzepts.

Kapitel 2

Konfigurierbare Hardware und ihre Beschreibung

In diesem Kapitel werden die Grundlagen zum Verständnis der Arbeit beschrieben. Dazu zählt eine Einführung in die konfigurierbaren Schaltkreise. Des Weiteren wird näher auf die Hardwarebeschreibungssprache VHDL eingegangen, die in dieser Arbeit als Beschreibungssprache für die konfigurierbare Hardware gewählt wurde.

2.1 Konfigurierbare Schaltkreise

Für die Realisierung eingebetteter Systeme stehen heutzutage unterschiedliche integrierte Schaltkreise (IC[1]) zur Verfügung. Diese weisen verschiedenste Vor- und Nachteile hinsichtlich ihres Preises, ihrer Komplexität, ihres Datendurchsatzes, ihrer Entwicklungszeit und ihres Ressourcenverbrauches auf. Die integrierten Schaltkreise können in drei Gruppen eingeteilt werden: die anwenderspezifischen Schaltkreise (ASIC[2]), die konfigurierbaren Schaltkreise und die universellen Schaltkreise (Abbildung 2.1).

Zu der letzten Kategorie gehören z.B. Mikroprozessoren und -controller, Speicher aber auch herstellerspezifische Kommunikationsbausteine, wie z.B. USB-Controller. Diese Schaltkreise sind dadurch gekennzeichnet, dass sie unabhängig von der konkreten Anwendung spezifiziert und produziert werden. Dadurch müssen sie sehr universell gehalten werden, d.h. eine hohe Komplexität aufweisen, was sich in einem erhöhten Ressourcenverbrauch, wie Strom, und einer größeren Chipfläche, aber auch in einer erhöhten Wärmeabgabe ausdrückt. Auf der anderen Seite werden diese Bausteine in einer sehr großen

[1] Integrated Circuit
[2] Application Specific Integrated Circuit

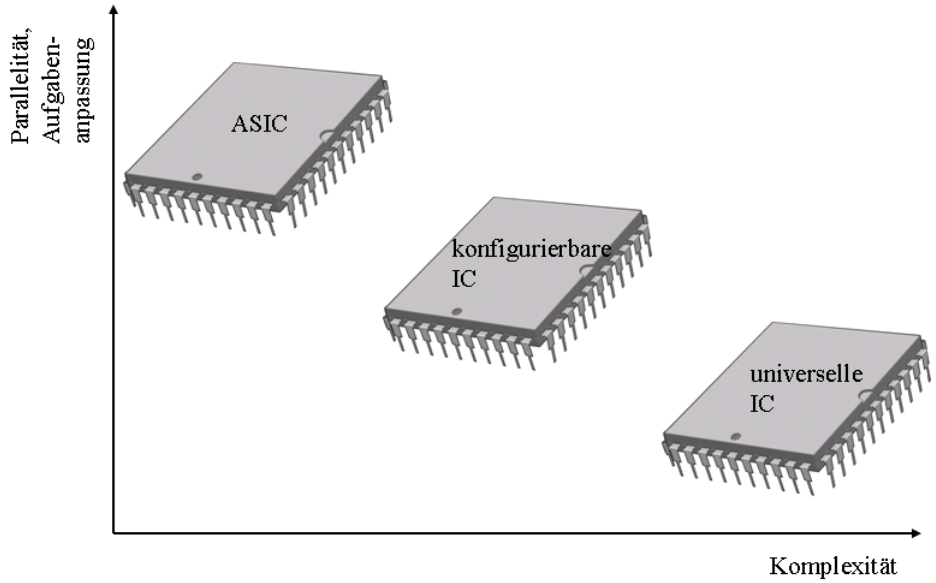

Abbildung 2.1: Schaltkreisgruppen

Anzahl produziert, sodass der Preis für den einzelnen Baustein gering ausfällt. Bedingt durch die große Anzahl an Nutzern liegt ein weiterer Vorteil in der sofortigen Verfügbarkeit und einer hohen Zuverlässigkeit.

Das genaue Gegenteil gilt für die Gruppe der ASIC. Hierbei werden die integrierten Schaltkreise für eine konkrete Anwendung konstruiert. Da sie auf die Aufgabenstellung abgestimmt sind, erreichen sie einen sehr hohen Datendurchsatz bei gleichzeitiger optimaler Ressourcenverwendung. Die Nachteile dieser Bausteine sind der hohe Entwicklungsaufwand und die daraus resultierende lange Entwicklungszeit bzw. die hohen Entwicklungskosten. Jede Änderung des Algorithmus zieht einen neuen Entwurf nach sich sowie die Produktion eines neuen Bausteins. Die hohen Entwicklungskosten können sich nur dann amortisieren, wenn diese Bausteine in einer großen Anzahl benutzt werden. Typische Vertreter sind Kommunikationsbausteine in Mobiltelefonen.

Die Vorteile beider Bausteinklassen werden in den konfigurierbaren integrierten Schaltkreisen vereint. Diese beinhalten eine große Anzahl an Logik- und Speicherblöcken. Die Verdrahtung dieser Blöcke untereinander wird jedoch nicht bei der Produktion festgelegt. Vielmehr geschieht dies im Nachhinein durch eine Programmierung dieses Bausteins. Dabei gibt es sowohl konfigurierbare Bausteine, die nur einmal programmiert werden können, als auch solche, bei denen die Programmierung auch mehrmals durchgeführt

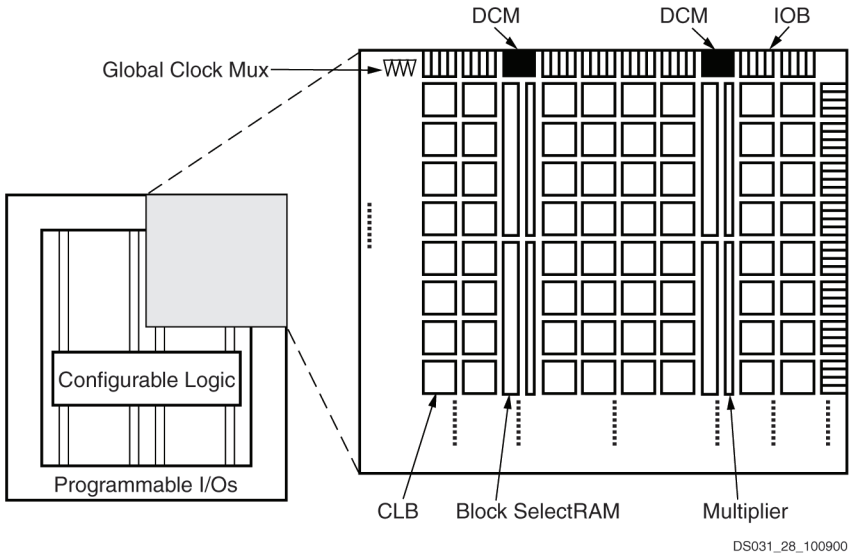

Abbildung 2.2: Blockschaltbild eines Xilinx-VirtexII-FPGA aus [Xil01]

werden kann. Durch ihre Programmierung sind diese Bausteine für die verschiedensten Aufgabengebiete einsetzbar. Aufgrund ihrer Struktur können konfigurierbare Schaltkreise nicht mit den hohen Frequenzen von modernen Prozessoren betrieben werden. Sie bieten aber die Möglichkeit, Informationen massiv parallel zu verarbeiten, sodass bei entsprechenden Anwendungen höhere Datendurchsätze als bei Prozessorsystemen erreicht werden. Auch bei der Integrationsdichte weisen sie einen um den Faktor 10 schlechteren Flächenbedarf gegenüber einem ASIC auf. Dem gegenüber stehen jedoch geringere Entwicklungskosten und kürzere Implementierungzeiten, da keine Hardwareentwicklung nötig ist und die Programmierung durch Simulationsprogramme unterstützt wird. Im Preis liegen konfigurierbare integrierte Schaltkreise um mehrere Größenordnungen über dem Preis von ASIC. Sie werden daher bevorzugt für die Prototypen-Entwicklung von ASIC eingesetzt. Typische Vertreter der konfigurierbaren Bausteine sind (C)PLD[3] und FPGA.

Bei den (C)PLD handelt es sich um Bausteine, die aus programmierbaren AND/OR-Matrizen, programmierbaren Rückkopplungen und Ein- und Ausgabeblöcken bestehen. Sie unterscheiden sich vom FPGA durch ihre einfachere Struktur sowie die dadurch hervorgerufene Vorhersagbarkeit der Signallaufzeiten und damit der maximalen Taktfrequenz.

FPGA bestehen ebenfalls aus frei programmierbaren Logikbausteinen. Je nach Typ und Hersteller beinhaltet ein FPGA verschiedenste Komponenten.

[3](Complex) Programmable Logic Device

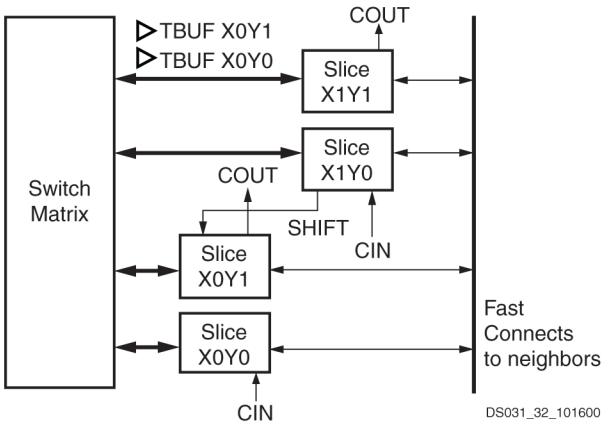

Abbildung 2.3: CLB eines Xilinx-VirtexII-FPGA aus [Xil01]

In Abbildung 2.2 ist die Blockstruktur für einen FPGA der Firma Xilinx vom Typ VirtexII dargestellt. Am Rand liegen die IOB[4], die für die Verbindung zu den externen Pins verantwortlich sind. Die Richtung der Verbindung (Eingang, Ausgang oder bidirektional) sowie die Definition des Standards (z.B. TTL, CMOS) sind, von bestimmten Einschränkungen abgesehen, frei programmierbar. Mit Hilfe der Switch-Matrix lassen sich die IOB untereinander und mit den anderen Komponenten verbinden. Die Pfade der Verbindungen werden über die Programmierung des FPGA festgelegt.

Im FPGA befinden sich die Matrixstrukturen aus CLB[5]. Diese sind auch mit der Switch-Matrix verbunden. Die Struktur eines CLB ist, wieder für den VirtexII, in Abbildung 2.3 dargestellt. Jeder CLB besteht aus vier Slices (Abbildung 2.4). Diese sind untereinander sowie mit der Switchmatrix und benachbarten CLB verbunden. Die CLB und die Slices ermöglichen die Umsetzung von kombinatorischem und sequentiellem Verhalten des FPGA. So beinhaltet jeder Slice zwei 4-bit-Lookup-Tabellen und zwei Speicherelemente. Um ein besseres zeitliches und räumliches Verhalten zu bekommen, ist es möglich, die Slices eines CLB zu übergeordneten Strukturen zusammenzufassen. Damit können schnelle Schieberegister, Lookup-Tabellen und große Register erzeugt werden. Diese Grundstruktur eines FPGA besitzen alle gängigen Arten, wenngleich die Ausbildung der Vernetzung und der Logikkomponenten von Typ zu Typ und Hersteller zu Hersteller unterschiedlich ist. Neben diesen Basiselementen können nun noch verschiedene andere Komponenten für Spezialaufgaben vorhanden sein. Dazu gehören Block-SelectRAM-

[4]Input/Output Blocks
[5]Configurable Logic Block

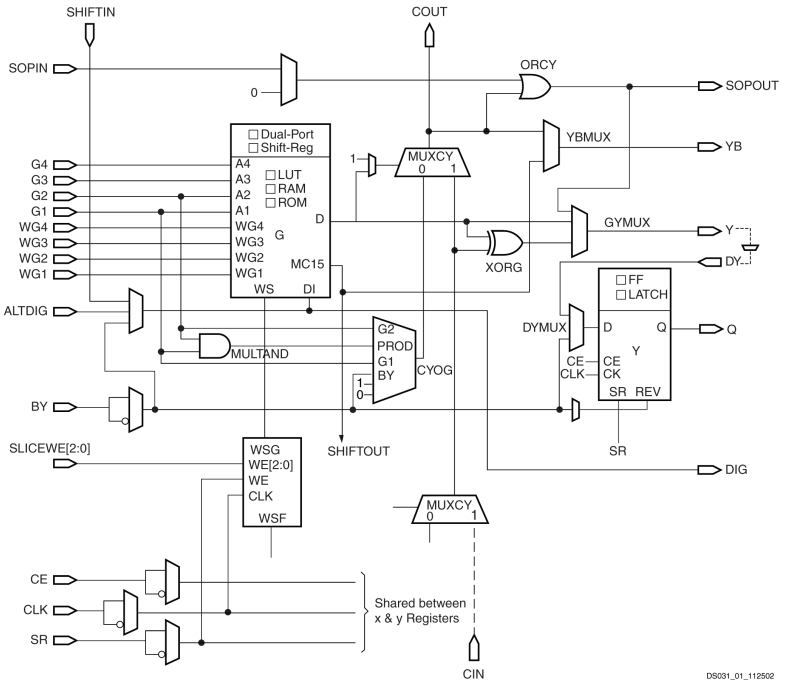

Abbildung 2.4: Slice eines Xilinx-VirtexII-FPGA aus [Xil01]

Speichermodule, welche die Umsetzung von großen Dual-Ported-Speicher im FPGA ohne Benutzung der Slices ermöglichen. Andererseits kann es auch spezielle Module geben, wie zum Beispiel Multiplizierer, welche die Umsetzung von Berechnungen vereinfachen und das zeitlichen Verhalten verbessern.

Als *State of the art* gelten heutzutage FPGA, die komplette Prozessorkerne integriert haben. Zu dieser Klasse von FPGA gehört z.B. der VirtexII-Pro [Xil03] mit seinen bis zu vier PowerPC-405-RISC-CPU-Kernen.

2.2 Hardwarebeschreibungssprache VHDL

Zur Programmierung der konfigurierbaren Bausteine gibt es verschiedene Möglichkeiten. Eine der gängigsten ist die Benutzung einer Hardwarebeschreibungssprache. Die beiden meist benutzten Sprachen sind Verilog und VHDL. In dieser Arbeit wird der Schwerpunkt auf der Verwendung der Sprache VHDL liegen. Wenngleich die Syntax und der Sprachumfang beider Sprachen stark voneinander abweichend sind, weisen beide Sprachen jedoch bzgl. des untersuchten Konzepts eines Hardwarebetriebssystems gleiche Strukturen auf und machen dieselben Modifikationen der Sprache erforderlich.

Die Entwicklung der Sprache VHDL reicht bis in die achtziger Jahre zurück. Zur Reduzierung der Kosten für die Entwicklung und Wartung militärischer Systeme suchte das amerikanische Verteidigungsministerium nach einer Sprache zur Dokumentation elektronischer Systeme. Im Jahre 1983 wurden die Firmen Intermetrics, IBM und Texas Instruments damit beauftragt, eine entsprechende Sprache mit passenden Werkzeugen zu entwickeln. Aufgrund der breiten Verwendung der Sprache ADA im amerikanischen Verteidigungsministerium sollte die neue Sprache an dieser angelehnt sein. Im August 1985 wurde eine erste Version (VHDL Version 7.2) freigegeben, die im Februar 1986 dem IEEE Konsortium zur Standardisierung übergeben wurde. Im Dezember 1987 wurde VHDL als IEEE 1076-1987 Standard übernommen (VHDL-87). Wegen fehlender Definitionen im Hinblick auf die Verwendung von mehrwertigen Signaltypen erfuhr der Standard eine Erweiterung in Form des IEEE Standards 1164 [Ins]. Da jeder IEEE-Standard alle paar Jahre einer Überarbeitung unterzogen wird, entstand im Jahre 1993 die Version IEEE 1076-1993, kurz VHDL'93 [MBG94]. In dieser Version wurde unter anderem die Sprache dahingehend überarbeitet, dass die Syntax konsistenter ist und die Namenskonventionen vereinfacht wurden. In der jüngsten Version des VHDL-Standards (VHDL-2003) wird die Sprache um die Möglichkeit der Einbindung von analogen Schaltelementen erweitert (VHDL-AMS[6]).

Die Standardisierung der Sprache VHDL bezieht sich jedoch nur auf die Syntax und die Semantik der Sprache. Sie definiert nicht die Umsetzung bzw. ihre Anwendung. Von Anfang an wurde Wert darauf gelegt, dass mit Hilfe der Sprache VHDL ein elektrisches System nicht nur beschrieben wird, sondern auch eine Simulation des Systems möglich ist. Dadurch kann jedes System erstmal einer Simulation unterzogen werden, bevor die Hardware realisiert wird. Die für die Simulation benötigten Sprachelemente von VHDL bewirken jedoch, dass nicht jedes mögliche VHDL-Programm synthetisierbar ist, d.h. auf einer Hardware umgesetzt werden kann. Es ist sogar möglich, Sprachkonstrukte zu benutzen, die weder synthetisierbar noch simulierbar sind. Dies liegt an der Unterspezifizierung der Sprache. Aus diesem Grund gibt es eine in IEEE P1076.6 [Ins00] festgelegte Untermenge von VHDL, die von einem Synthesecompiler verstanden wird.

Mit Hilfe der Sprache VHDL werden Modelle von Systemen oder Subsystemen beschrieben. Dafür stehen vier verschiedene Blockarten zur Verfügung: *Entities*, *Architectures*, *Configurations* und *Packages* [Ash98]. Die Sprache trennt die Schnittstellen von den entsprechenden Realisierungen. Die Schnittstellen werden durch die *Entities* definiert, die Realisierungen dagegen in den *Architectures*. Da es möglich ist, auch mehrere Realisierungen

[6]VHDL Analog and Mixed Signals [Ins03]

für eine Schnittstelle zur Verfügung zu stellen, kann mit Hilfe der *Configuration* die Zuordnung definiert werden. Als letzten möglichen Block gibt es *Packages*, welche die Erzeugung von Bibliotheken aus Konstanten, Typen, Komponentendeklarationen, Prozeduren und Funktionen ermöglichen.

2.2.1 Schnittstellenbeschreibung

Mit Hilfe der Schnittstellenbeschreibung (*Entity*) wird die Schnittstelle eines Modells, d.h. einer Komponente oder eines ganzen Systems beschrieben. Hier müssen alle Ein- bzw. Ausgänge des Modells angegeben sein. Es können auch weitere Vereinbarung festgelegt werden, die für alle Realisierungen der Schnittstelle erforderlich sind. Ein Beispiel für eine Schnittstelle ist:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test_entity is
    generic (
        generic1: integer;
        generic2: integer range 0 to 12
    );
    port (
        port1: out std_logic;
        port2: inout std_logic_vector(generic2-1 downto 0);
        port3: buffer integer
    );
end;
```

Am Anfang einer Schnittstellenbeschreibung befinden sich Referenzen zu Bibliotheken, in denen Typen definiert sind, die für die Schnittstelle benötigt werden. Anschließend gibt es zwei Klassen von Listen: eine Genericliste und eine Portliste. Die Portliste definiert die Menge an Signalen, die für das Modell benötigt werden. Alle Signale, auf die das Modell früher oder später zugreifen muss, müssen Bestandteil dieser Liste sein. Wie sich später noch zeigen wird, stellt das ein großes Problem für die Umsetzung eines Betriebssystems dar. Jedes Signal ist dabei mit einem Typ und einem Attribut versehen. Das Attribut legt dabei die Richtung des Informationsflusses des Signals fest. Zulässig sind *in* (Information von außen zum Modell, kann nur gelesen werden), *out* (Information vom Modell nach außen, kann nur geschrieben werden), *inout* (Information in beiden Richtungen, kann gelesen und geschrieben werden) und *buffer* (Informationen in beiden Richtungen, kann gelesen werden, aber nur von einer Quelle geschrieben).

In der Genericliste, lassen sich zusätzliche Parameter übergeben. Diese Parameter haben die Eigenschaft, dass sie zeitlich unveränderlich sind, jedoch bei jeder Verwendung des Modells mit anderen Werten versehen werden können. Sie sind dabei nicht nur auf die Architektur bezogen, sondern können sich auch auf die Schnittstellenbeschreibung beziehen. In diesem Fall kann der Parameter den Typ eines Ports festlegen. Ein typischer Anwendungsfall besteht darin, die Bit-Breite eines Signals als Parameter zu übergeben. Dadurch kann dieselbe Beschreibung des Modells für verschiedene Signalbreiten benutzt werden. Die Modellbeschreibung wird damit wiederbenutzbar. Theoretisch ist es auch möglich, mit Hilfe eines Parameters den Typ eines anderen Parameters festzulegen. Dies impliziert jedoch die Gefahr, zu nicht-definierten rekursiven Definitionen von Typen zu kommen. Es wird sich außerdem zeigen, dass für die Umsetzung des Betriebssystemkonzepts die Parameter zur Definition der Ports gewissen Einschränkungen unterliegen müssen.

2.2.2 Architektur

Die Architektur (*Architecture*) beinhaltet eine Beschreibung der Funktionalität einer Modells. Diese Beschreibung kann eine Verhaltensbeschreibung sein oder aber auch eine strukturelle Beschreibung. Es besteht sogar die Möglichkeit, beide Beschreibungen gleichzeitig zu benutzen. Ein Beispiel für eine Architektur ist:

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture test_arch of test_entity is

    signal test1: std_logic_vector(13 downto 0);
    signal test2: std_logic;

    component submodule is
        generic (
            genericA: integer
        );
        port (
            portA: in std_logic;
            portB: out std_logic
        );
    end component;
```

```
begin
    I_sub:submodule
        generic map (
            genericA => 12
        )
        port map (
            portA    => test2,
            portB    => test1(0)
        );
    test1(1)<=test1(0);

    P0:process(clock,reset)
    begin
        if (reset='0') then
            test2<='0';
        elsif (clock'event and clock='1') then
            test2<=not test2;
        end if;
    end process;

end;
```

Jede Architektur muss auf genau eine Schnittstellenbeschreibung verweisen. Es ist jedoch möglich, dass zu einer Schnittstellenbeschreibung mehrere oder aber auch keine Architekturbeschreibung existieren. Die Kombination aus Schnittstelle und Architektur wird im Weiteren als Modul bezeichnet. Ein Modul ist damit das Äquivalent einer Funktion oder einer Prozedur wie sie von Softwareprogrammiersprachen her bekannt sind.

Eine Architektur besteht aus einem Deklarationsteil und einem Anweisungsteil. Im Deklarationsteil können unter anderem neue Signale, Funktionen und Typen definiert werden. Das besondere an einer Hardwarebeschreibungssprache ist, dass die Anweisungen im nachfolgenden Anweisungsteil alle gleichzeitig ausgeführt werden. Im Gegensatz zu Softwarebeschreibungssprachen, wo die sequentielle Abarbeitung ein Programm definiert und eine Parallelität von Anweisungen nur virtuell durch Taskwechsel oder nur beschränkt in Multiprozessorsystemen möglich ist, werden Programme in einer Hardwarebeschreibungssprache durch ihre parallelen Anweisungen festgelegt. Die Funktionalität wird damit nicht in der zeitlichen Abfolge von Anweisungen, sondern in der räumlichen Anordnung von Operatoren festgelegt. Dieser gravierende Unterschied macht es auch so schwer bzw. unmöglich, Algorithmen

von einer Software- in eine Hardwarebeschreibungssprache und umgekehrt optimal zu transformieren. Trotzdem ist es möglich, eine sequentielle Abarbeitung in die Verhaltensbeschreibung der Modelle einfließen zu lassen. Dies wird durch die Benutzung von so genannten Prozessen (*Process*) erreicht. Um Prozesse jedoch auch synthetisieren zu können, müssen sie bestimmte Randbedingungen erfüllen [LWS94].

Eine besondere Klasse von Anweisungen stellen die Komponenten (*Components*) dar. Diese ermöglichen die Einbindung von anderen Modellen. Die Komponentendeklaration muss dabei mit der Schnittstellendefinition des einzubindenden Modells übereinstimmen. Durch Zuweisung aller Portelemente sowie aller Parameter kann auf die entsprechende Funktionalität der Komponente zurückgegriffen werden. Die konkrete Realisierung dieses Modells, d.h. die Architektur der Komponente, sollte an dieser Stelle keinen Einfluss auf das ursprüngliche Modell haben. Durch Einbindung einer Komponente, die mit der Schnittstellenbeschreibung des Modells übereinstimmt, besteht die Möglichkeit, eine rekursive Struktur zu erstellen.

2.2.3 Konfiguration

In der Konfiguration (*Configuration*) wird festgelegt, welche Architektur zu einer Schnittstellenbeschreibung zugeordnet werden soll. Diese Zuordnung bezieht sich auch auf mögliche Submodelle, die in der Architektur benutzt werden. Solange nur eine Architektur für jede Schnittstellenbeschreibung existiert, wird eine Konfiguration nicht benötigt. Ein Beispiel für eine Konfiguration ist:

```
configuration test_config of test_entity is
    for test_arch        -- verknüpfe Architektur test_arch
    end for;             -- mit Entity test_entity
end;
```

2.2.4 Paket

In einem Paket (*Package*) können alle Arten von Deklarationen gesammelt und so verschiedenen Modellen und Projekten zur Verfügung gestellt werden. Durch das Zusammenfassen von Deklarationen wird das Kodieren von Modellen und die Erhaltung der Konsistenz der Schnittstellen zwischen den Modellen erleichtert. Weiterhin verkürzt die Benutzung von Paketen die Kompilationszeit, da diese bei Veränderung der Modelle nicht neu übersetzt werden müssen. Innerhalb der Sprache VHDL gibt es zwei vordefinierte Pakete. Das Paket *STANDARD* definiert verschiedene Typen, Subtypen, Funktionen und

Operationen. Zu den Typen gehören z.B. *BOOLEAN*, *BIT* und *CHARACTER*. Dieses Paket wird in jedem Modell implizit eingebunden. Das Paket *TEXTIO* enthält Deklarationen von Typen und Funktionen, welche die formatierte Ein- und Ausgabe in Dateien ermöglichen. Typische Vertreter sind der Typ *TEXT* und die Funktionen *read* und *write*. Um die Deklarationen dieses Paketes zu benutzen, muss es explizit im Modell eingebunden werden.

2.2.5 Entwurf elektronischer Systeme

Die hohe Komplexität heutiger elektronischer Systeme erfordert beim Entwurf eine strukturierte Vorgehensweise. Dabei bietet sich ein Top-Down-Vorgehen an, d.h. eine schrittweise Strukturierung und Verfeinerung des Entwurfs, ausgehend von einer Spezifikation auf Systemebene. Am Ende eines Entwurfs stehen dann Layouts für Leiterplatten, Masken für die IC-Herstellung oder die Programmierdaten für konfigurierbare Bausteine. Üblicherweise wird der Entwurf nach drei Sichtweisen unterschieden: Verhalten, Struktur und Geometrie. Diese sind im Y-Diagramm von Gajski-Walker zusammengefasst (Abbildung 2.5). Zum einen sind hier die drei verschiedenen Sichtweisen des Entwurfs durch die drei Äste des Y dargestellt, zum anderen eine Zuordnung der verschiedenen Abstraktionsebenen, die durch die Kreise wiedergegeben wird. Mit der Entfernung vom Mittelpunkt steigt der Abstraktionsgrad. Innerhalb dieses Diagramms stellt sich ein Entwurf als eine Reihe von Transformationen sowohl zwischen den Abstraktionsgraden als auch zwischen den Sichtweisen dar.

Die Sprache VHDL unterstützt die Beschreibung und Verifikation von Systemen auf der algorithmischen Ebene, der Register-Transfer-Ebene sowie der Logikebene. Dabei werden die Verhaltensbeschreibung und die strukturelle Modellierung angewendet. Die beiden Sichtweisen der Modellierung spiegeln sich in der Architektur wider. Zum einen besteht die Möglichkeit, das Verhalten der Ausgänge des Modells in Abhängigkeit von den Eingängen und der Zeit auszudrücken. Auf der anderen Seite kann unter Ausnutzung der Komponenten das Modell aus anderen Submodellen zusammengesetzt und damit eine strukturelle Modellierung durchgeführt werden. Mit Hilfe der Generic-Parameter besteht sogar die Möglichkeit, eine rekursive strukturelle Beschreibung vorzunehmen. Die Sprache VHDL ist außerdem in der Lage, beide Sichtweisen in einem Modell gleichzeitig umzusetzen. Für einige Beschreibungssprachen ist in Abbildung 2.6 der Umfang der unterstützten Abstraktionsebenen angegeben.

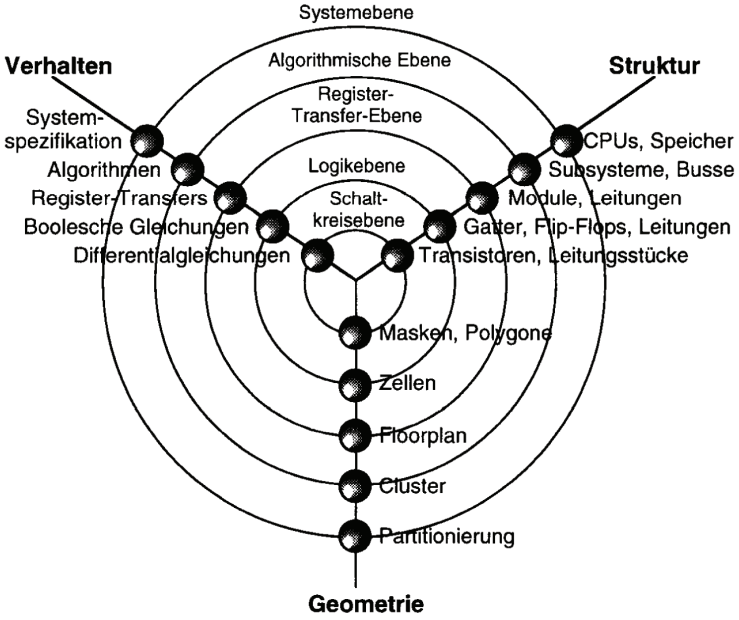

Abbildung 2.5: Y-Diagramm nach Gajski-Walker aus [WT85]

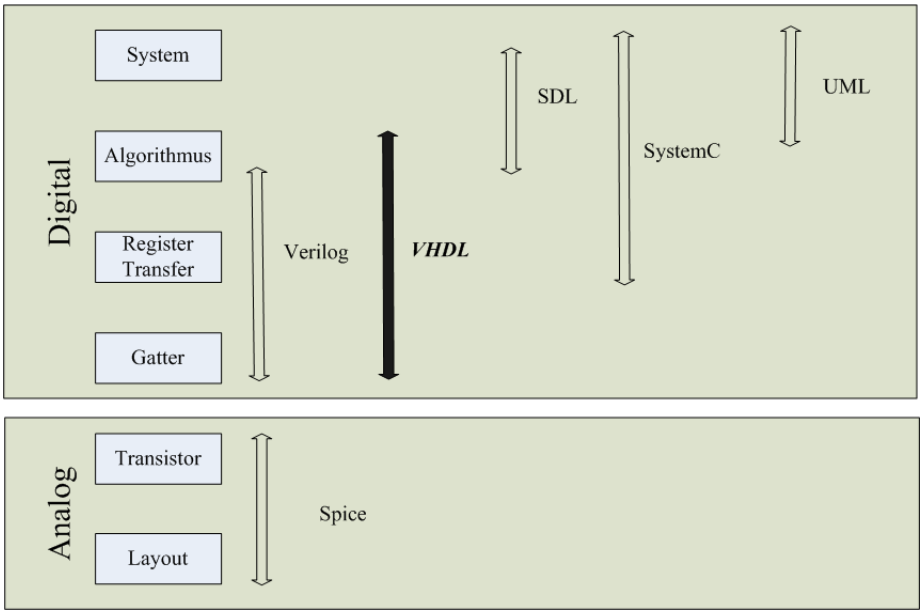

Abbildung 2.6: Abstraktionsgrad von Modellierungssprachen

2.2.6 Zustandsautomaten

Für die Definition von Betriebssystemschnittstellen wird es nötig sein, diese Hinsichtlich ihres zeitlichen Verhaltens zu betrachten. Dazu werden im Weiteren kurz die verschiedenen Arten der Implementierung eines Zustandsautomaten, des Moore-Automaten und des Mealy-Automaten, eingeführt.

Alle synthetisierbaren digitalen Modelle in VHDL können als endliche Automaten oder Zustandsautomaten aufgefasst werden. Sie werden damit beschrieben durch ein 6-Tupel:

$$\mathcal{A} = (Z, \Sigma, \Omega, \delta, \lambda, z_0) \tag{2.1}$$

mit:

- Z ist die Menge der Zustände, die das Modell annehmen kann. Diese Zustände müssen mit Speicherelementen realisiert werden. Im günstigsten Fall werden $\log_2(|Z|)$ Bits zur Speicherung der Zustände benötigt. In der praktischen Umsetzung wird die Anzahl an Bits aber meistens größer sein.

- Σ ist das Eingabealphabet. Es beschreibt die Menge an möglichen Kombinationen von Werten der Eingabesignale. Aufgrund der Endlichkeit der Anzahl der Eingabesignale und der Endlichkeit der Anzahl der Werte eines Eingabesignals, ist die Mächtigkeit des Eingabealphabets auch endlich.

- Ω ist das Ausgabealphabet. Es beschreibt die Menge an möglichen Kombinationen von Werten der Ausgangssignale. Genauso wie das Eingabealphabet hat auch das Ausgabealphabet eine endliche Mächtigkeit.

- δ ist die Übergangsfunktion der Zustände, d.h. $\delta : Z \times \Sigma \rightarrow Z$. In einem digitalisierten System handelt es sich um eine logische Funktion der Eingangssignale und des Zustandsspeichers.

- λ beschreibt die Ausgabe, d.h. $\lambda : Z \times \Sigma \rightarrow \Omega$. Bei dieser Funktion handelte es sich auch um eine logische Funktion.

- z_0 beschreibt den Startzustand, d.h. den Initialisierungswert des Speichers.

In Abhängigkeit von der Ausgabefunktion wird zwischen Moore- und Mealy-Automaten unterschieden. Im Falle, dass λ unabhängig von dem Eingabealphabet ist und damit die Ausgangssignale nur von dem aktuellen Zustand

abhängig sind, handelt es sich um einen Moore-Automaten. Andernfalls handelt es sich um einen Mealy-Automaten. Beide Automaten lassen sich ineinander transformieren, was jedoch Auswirkung auf die Anzahl der Zustände hat.

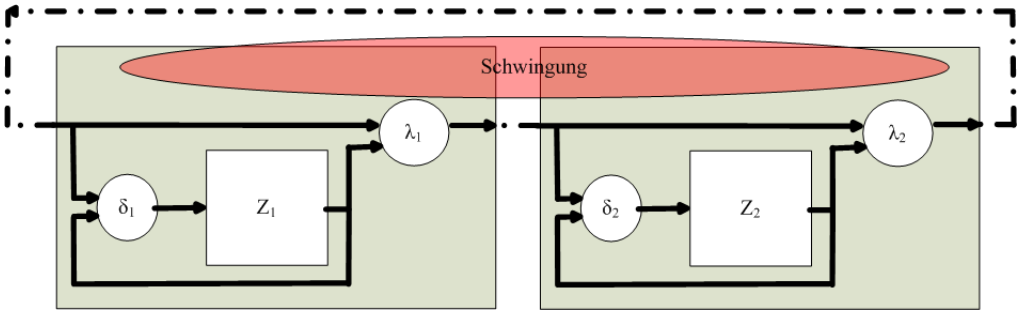

Abbildung 2.7: Undefinierte Zustände im Mealy-Automaten

Für ein elektronisches System spielt diese Unterscheidung noch in anderer Hinsicht eine große Rolle. Im Gegensatz zu einem idealen Automaten besitzen reale Systeme Laufzeiten. Die Taktfrequenz in einem System darf nicht größer sein als das Reziproke der maximalen Laufzeit eines Pfades in dem System. In einem Netzwerk aus Automaten ergibt sich die Laufzeit aus den Längen der Pfade zwischen den Zustandsspeichern. Im Falle eines Moore-Automaten ist diese Länge immer beschränkt durch die Funktion der Ausgabe des ausgebenden Automaten und die Übergangsfunktion des empfangenden Automaten. Aufgrund der Abhängigkeit der Ausgabefunktion vom Eingang kann sich im Falle eines Mealy-Automaten ein Pfad über beliebig viele Automaten des Netzwerkes ausdehnen. Daher werden bei einem Mealy-Automaten die Laufzeiten immer größer sein als bei einem entsprechenden Moore-Automaten. Ein weiterer Nachteil eines Mealy-Automaten ist, dass es zu undefinierten Zuständen bzw. Schwingungen kommen kann, wenn die Ein- bzw. Ausgänge zweier Automaten direkt miteinander verknüpft sind (Abbildung 2.7). Dieses Verhalten wird zur Konstruktion von asynchronen Automaten ausgenutzt [KP03]. In einem synchronen System ist dies jedoch unerwünscht. Der Vorteil von Mealy-Automaten liegt in der Reduzierung der Anzahl der Zustände und damit in einer Reduzierung des Speicherbedarfs. Außerdem bewirkt die direkte Beeinflussung der Ausgangssignale durch die Eingangsignale, dass die Latenzzeit der Antwort des Automaten reduziert werden kann. Bei einem Moore-Automat kann die Antwort erst nach einem Takt am Ausgang erscheinen.

2.2.7 Diskussion der Sprache VHDL

Wie jede Sprache besitzt auch VHDL viele Vor- und Nachteile. Zu den Vorteilen zählt, dass es mit Hilfe dieser Sprache möglich ist, ein elektrisches System sowohl zu modellieren als auch zu simulieren und zu synthetisieren. Diese drei Aufgaben sind außerdem auf verschiedensten Ebenen der Beschreibung lösbar. Durch die Standardisierung und Herstellerunabhängigkeit wird VHDL auch von vielen Programmen unterstützt.

Besonders beim Einstieg in die Benutzung von Hardwarebeschreibungssprachen stellt sich der Aspekt der Parallelisierung als Nachteil heraus. Ein Software-Entwickler ist gewohnt, einen Algorithmus sequentiell umzusetzen und zu programmieren. Bei der Hardwareprogrammierung müssen jedoch primär parallele Strukturen realisiert werden. Damit verbunden ist auch das Problem, dass sich VHDL-Modelle schwerer lesen lassen. Anweisungen können in VHDL aufgrund ihrer Parallelität in der Architektur in beliebiger Reihenfolge angegeben werden. Damit wird es aber auch schwerer, die Struktur des Modells und des damit verbundenen Algorithmus zu erkennen. Um das Problem zu lösen, muss eine strukturierte Vorgehensweise konsequent umgesetzt und das Modell so stark zerlegt werden, bis am Ende nur noch einfache Submodelle auftreten. Außerdem ist die Einhaltung von Regeln bei der Namensvergabe und der Implementierung vorteilhaft [Gas02, Sin94].

Ein weiterer Nachteil von Hardwarebeschreibungssprachen und damit auch von VHDL liegt darin, dass nicht auf die Funktionsbibliotheken von Softwareprogrammiersprachen zurückgegriffen werden kann. In jede gängige Programmiersprache können eine Vielzahl an Funktionen eingebunden werden, die in den letzten Dekaden ständig Verbesserungszyklen unterworfen waren und damit bzgl. der Ausführungszeit, des Ressourcenverbrauches und der Zuverlässigkeit optimiert wurden. Bei Hardwarebeschreibungssprachen existiert eine solche Basis noch nicht, weil dazu jede Funktion erstmal neu entwickelt und erprobt werden muss. Und dies erfordert zusätzliche Zeit und Kosten.

Während die bisher genannten Nachteile für alle Hardwarebeschreibungssprachen gelten, gibt es aber auch einige Nachteile, die sich speziell auf VHDL beziehen. Dazu zählt zum einen die sehr komplexe Semantik. Diese wurde bereits in der VHDL-93 Version überarbeitet, um die Interpretation zu vereinheitlichen. Auch die Syntax weist Konstrukte auf, die auf den ersten Blick nicht konsequent erscheinen (z.B. Benutzung des Semikolons). Des Weiteren stellt sich als Nachteil heraus, dass selbst für einfache Modelle die entsprechenden Quellen sehr schnell hunderte Zeilen lang sind.

Ein weiterer Nachteil liegt in der Unterstützung von Simulations- und Synthesemodellen. Nicht jede Sprachkonstruktion in VHDL lässt sich in einer

entsprechenden Hardware realisieren. Darüber hinaus unterstützen die Synthesewerkzeuge unterschiedliche Mengen an VHDL-Sprachkonstruktionen. Daher sollten in synthetisierbaren Modellen nur solche Sprachelemente benutzt werden, für die auch eine Unterstützung durch eine Vielzahl an Werkzeugen verfügbar ist.

Kapitel 3

Entwurfsmethoden für konfigurierbare Hardware

In diesem Abschnitt werden verschiedene Entwurfsmethoden für rekonfigurierbare Hardware vorgestellt, die den aktuellen Stand der Technik wiedergeben. Dazu wird zuerst der Entwurf mit der Hardwarebeschreibungssprache VHDL beschrieben. Die meisten Projekte aus der Produktentwicklung bedienen sich heutzutage dieser Methode. Danach werden alternative Entwurfsmethoden vorgestellt, die versuchen, die Nachteile des Entwurfs mit einer Hardwarebeschreibungssprache zu beseitigen. Die Auswahl der Methoden orientiert sich dabei primär an der Verwendung in Systemen der Signalverarbeitung.

3.1 Entwurf mit VHDL

Ein derzeit typischer Entwurf für konfigurierbare Hardware mit Hilfe der Hardwarebeschreibungssprache VHDL besteht aus den in Abbildung 3.1 gezeigten Schritten. Jeder Schritt wird durch ein spezielles Programm ausgeführt. Es gibt jedoch auch Programme, welche die einzelnen Schritte in einer Oberfläche integrieren und von dort die Subprogramme ansteuern. Zu dieser Klasse gehören FPGAdvantage [Men05] und ISE [Xil05b].

Ausgangspunkt ist ein Modell in der Sprache VHDL. Es kann mit Hilfe eines beliebigen Editors erstellt werden. Viele Editoren unterstützen heutzutage die Syntaxhervorhebung sowie die Erstellung von Sprachtemplates. Es gibt außerdem die Möglichkeit, sich spezielle VHDL-Modelle durch die Benutzung von graphischen Frontends erstellen zu lassen. Zum Beispiel unterstützt FPGAdventage die Erzeugung von Zustandsmaschinen aus einem Zustandsgraphen.

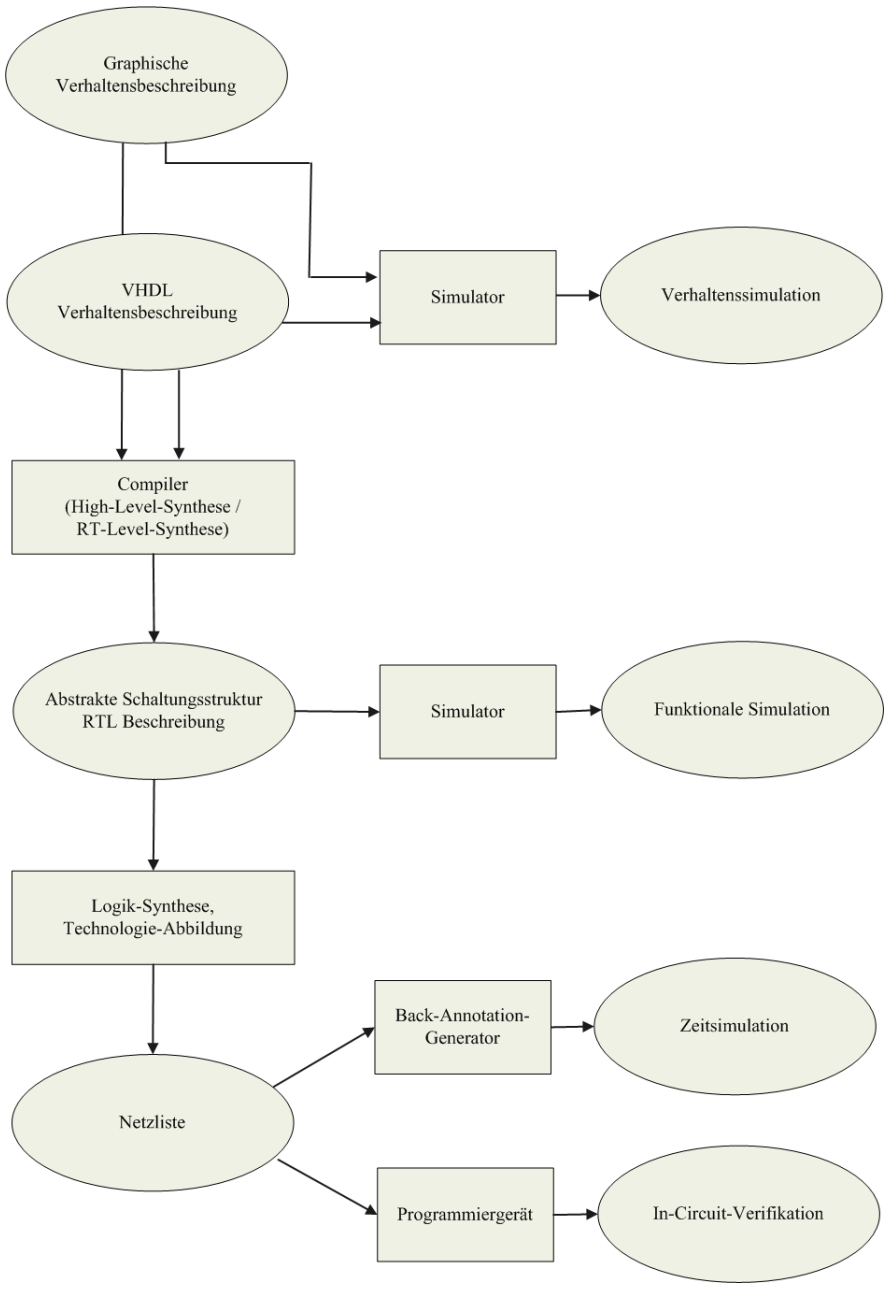

Abbildung 3.1: Hardware-Entwurfsschritte unter Verwendung von VHDL

Die so erzeugten VHDL-Modelle können dann mit einem Simulationsprogramm, wie z.B. ModelSim [htt06, Men03], simuliert werden. Um jedoch eine sinnvolle Simulation durchführen zu können, muss das Modell in eine Testumgebung (*testbench*) eingebettet werden. In dieser müssen alle extern eingehenden Signale als Stimuli erzeugt werden. Ein typisches Beispiel dafür sind die Signale für den Takt und das Reset. Ausgehende Signale können innerhalb der Testumgebung einer weiteren Überprüfung unterzogen werden. Um externe Hardware, wie z.B. Speicherbausteine, zu simulieren, bietet es sich an, entsprechende vom Hersteller gelieferte Simulationsmodelle der Hardware in die Testumgebung einzubauen. Diese Simulationsmodelle können dabei sowohl das logische Verhalten des kompletten Systems als auch das zeitliche Verhalten widerspiegeln. Die Simulation kann nicht nur auf das Gesamtsystem sondern auch auf Teilsysteme angewendet werden. Dies wird immer dann notwendig, wenn schon die Teilsysteme eine hinreichende Komplexität erreichen. Neben der Verifikation des Modells als Black-Box-Test, kann mit den Simulationsprogrammen auch eine Fehleranalyse durchgeführt werden. Es besteht die Möglichkeit, sich das zeitliche Verhalten aller Signale in dem Modell anzeigen zu lassen. Daneben wird auch die Benutzung von Breakpoints und eine schrittweise Abarbeitung der Simulation unterstützt. Damit ermöglichen heutige Simulationsprogramme alle Grundfunktionen, die für die Entwicklung und Verifikation von Modellen notwendig sind.

Neben der Simulation eignen sich Modelle auch zur Erzeugung einer abstrakten Schaltungsstruktur. Dieses geschieht mit einem Compiler, der eine High-Level-Synthese durchführt. Dabei wird eine Überführung des Modells in die Register-Transfer-Ebene vollzogen. Diese Ebene beschreibt den Datenfluss zwischen den Registern eines Schaltkreises. Zur Erzeugung einer Beschreibung in der Register-Transfer-Ebene werden verschiedene Schritte durchgeführt. Am Anfang findet eine Modulauswahl statt. Für z.B. eine Addition von Signalen müssen passende Addierermodule gewählt werden. Danach wird das Scheduling ausgeführt. Hierbei wird das Modell hinsichtlich paralleler Strukturen analysiert. Ein letzter Schritt besteht in der Allokation. Dabei wird bestimmt, wie viele Module eines Typs benötigt werden und welche Operation welches Modul benutzen darf. Bei den Modulen handelt es sich dabei um die Basiselemente der Register-Transfer-Ebene, wie z.B. Multiplizierer und Register. Weiterhin werden in dieser Synthesestufe auch verschiedene Programmtransformationen durchgeführt, z.B. das Abrollen von Schleifen oder das Auflösen von Prozedur- oder Funktionsaufrufen. Auch auf dieser Ebene der Beschreibung besteht die Möglichkeit, eine Simulation durchzuführen. Diese Simulation wird Funktionale Simulation genannt. Im Gegensatz zur Verhaltenssimulation (vgl. Abbildung 3.1) ist die Funktionale Simulation umfangreicher, da sich die Modellbeschreibung auf

einem niedrigeren Abstraktionsniveau befindet. Auch hier wird wieder eine Testumgebung benötigt. Es bietet sich natürlich an, die Testumgebung aus der Verhaltenssimulation zu benutzen.

Die Beschreibung auf diesem Level ist immer noch unabhängig von der Plattform, auf der das Modell realisiert werden soll. So kann dieses Modell zur Programmierung eines FPGA oder CPLD, aber auch zur Herstellung eines ASIC benutzt werden. Erst im nächsten Schritt, der Logik-Synthese, wird aus der abstrakten Schaltungsstruktur ein Ergebnisfile für die Zielplattform. Im Falle eines FPGA oder CPLD ist dies eine BIT-Datei bzw. eine MCS-Datei. Diese können zur Programmierung des FPGA oder eines PROM[1] benutzt werden. Deshalb wird der Compiler zur Logik-Synthese auch vom Hersteller der Zielplattform, wie dem FPGA-Hersteller, mitgeliefert. Die Logik-Synthese beinhaltet dabei die Funktionen des Übersetzens, des Abbildens, des Platzierens und des Routens. Daher werden diese Compiler auch PAR[2] genannt. Ausgehend von der abstrakten Schaltungsstruktur des vorhergehenden Syntheseschrittes werden die Elemente der Register-Transfer-Ebene auf die vorhandenen Strukturen der Zielplattform abgebildet. Danach müssen die Verbindungen zwischen den einzelnen Strukturen durch Routing ermittelt werden. Auf diesem Level der Synthese werden zusätzliche Informationen zur Zielhardware benötigt. Dabei kann es sich unter anderem um Informationen zur Beschreibung der Anbindung der externen Hardware handeln, d.h. um die Zuordnung der internen Signale zu den äußeren Pins und deren Standard, um Informationen zur Platzierung von bestimmten Strukturen oder um Informationen zu maximalen Signallaufzeiten. Diese werden in einem UCF[3] beschrieben.

Neben den UCF spielen auch EDN[4]-Dateien für die Synthese eine große Rolle. Dabei handelt es sich um Netzlisten im EDIF[5]-Format. Mit Hilfe dieser Dateien ist es möglich, Komponenten in ein Modell zu integrieren, die bereits eine feste Abbildung auf die Zielplattform besitzen. Ein typisches Beispiel dafür sind FIFO[6]-Komponenten. Zur Vereinfachung der Logik-Synthese und zur optimalen Ausnutzung vorhandener Ressourcen werden die resultierenden Strukturen aber nicht auf der abstrakten Schaltungsstruktur beschrieben. Vielmehr ist deren Umsetzung auf der Zielhardware in der Beschreibung integriert. Erzeugt werden EDN-Dateien mit einem Core-Generator. Dieser wird ebenfalls vom Hersteller der Zielplattform mitgeliefert.

[1] Programmable Read Only Memory
[2] Place-And-Route
[3] User Constraint File
[4] Electronic Data Interchange Format Netlist
[5] Electronic Data Interchange Format
[6] First in - First Out

Das Ergebnis der Logik-Synthese kann zur Programmierung oder zur Erstellung der Zielplattform-Hardware genutzt werden. Im Falle eines FPGA gibt es verschiedene Möglichkeiten, diesen zu programmieren. Einerseits gibt es FPGA, bei denen das Programm im Baustein selber gespeichert werden kann [Act06]. Andererseits gibt es auch FPGA, bei denen das Programm nach jedem Neustart oder Reset von außen neu vorgegeben werden muss. Bei diesen bietet es sich an, den FPGA mit Hilfe eines externen (EEP)ROM[7] zu programmieren. Dieser kann mit einem Brenner oder JTAG[8] verändert werden. Eine weitere Möglichkeit besteht darin, das Programm direkt von einer externen Schnittstelle zu laden, wie z.B. der PCI-Schnittstelle. Der Vorteile eines solchen Entwurfs liegt darin, dass während der Laufzeit des PC und des FPGA das Programm des FPGA ohne zusätzlichen externen Hardwareaufwand ausgetauscht werden kann. Bei den heutigen Größen eines FPGA dauert die Programmierung einige Millisekunden.

Zu Testzwecken kann aus dem Ergebnis der Logik-Synthese eine VHDL-Datei erzeugen werden. Diese Datei kann mit dem Simulator und einer Testumgebung wieder für eine Simulation herangezogen werden. Neben den funktionalen Elementen, die schon aus der Funktionalen Simulation bekannt sind, beinhaltet diese Simulation auch alle Zeitinformationen, die sich aus der konkreten Realisierung der Hardware ableiten lassen. Damit entspricht diese Simulation einer Zeitsimulation. Aufgrund der Ableitung der Simulationsdatei aus dem Ergebnis der Logik-Synthese heißt diese Simulation auch Back-Annotation-Simulation. Sie weist jedoch für das Debugging zwei Nachteile auf. Zum einen dauert die Simulation sehr lange, da sie auf einem sehr niedrigen Abstraktionslevel stattfindet. Zum anderen wird jegliche Struktur des Modells durch die Logik-Synthese und die Back-Annotation-Generierung zerstört. Alle Signale existieren in nur einer Architektur. Des Weiteren ist es schwierig, die neu auftretenden Signalnamen mit den Signalnamen in dem Originalmodell in Verbindung zu bringen. So kann es vorkommen, dass Signale überhaupt nicht mehr auftreten, da sie durch die Optimierung entfernt wurden. Andererseits können komplett neue Signale aus funktionalen Elementen des Modells entstehen. Daher eignet sich diese Art der Simulation bei großen Entwürfen nur zur Verifikation, nicht aber zum Debuggen.

Neben der Verifikation des Entwurfs mit Hilfe einer Simulation besteht auch die Möglichkeit, eine Verifikation auf der Zielplattform durchzuführen. Diese Methode wird auch In-Circuit-Verifikation genannt. Dabei werden innerhalb des Modells Komponenten untergebracht, die ähnlich einem Logikanalysator in der Lage sind, bestimmte Signalverläufe aufzuzeichnen [Car05].

[7] (Electrically Erasable Programmable) Read Only Memory
[8] Joint Test Action Group

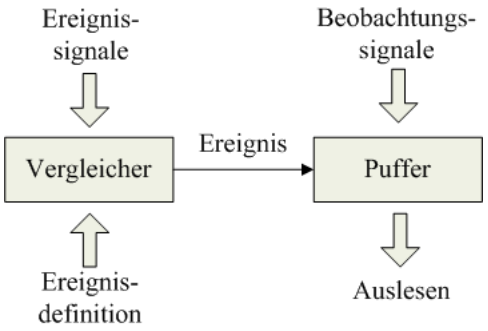

Abbildung 3.2: Aufbau einer In-Circuit-Komponente

Aufgebaut sind solche Komponenten aus zwei Teilen (Abbildung 3.2): einem Vergleicherblock und einem Puffer. Mit Hilfe des Vergleicherblocks können definierte Ereignisse detektiert werden. Die Definition der Ereignisse kann dabei statisch erfolgen oder aber während der Laufzeit durch den Benutzer vorgegeben werden. Falls ein entsprechendes Ereignis vorliegt, wird der Puffer dazu veranlasst, je nach Konfiguration das Signalverhalten vor, nach oder in der Umgebung des Ereignisses aufzuzeichnen. Zu einem späteren Zeitpunkt kann dann der Puffer ausgelesen und somit eine Rekonstruktion des Signalverhaltens durchgeführt werden. Die dem Puffer zugeführten Beobachtungssignale müssen dabei nicht mit den Ereignissignalen übereinstimmen, die dem Vergleicher zur Erkennung eines Ereignisses zugeführt werden. Als Auslesemedium wird meistens die JTAG-Schnittstelle benutzt. Für FPGA der Firma Xilinx wird diese Verifikation mit Hilfe des Werkzeuges ChipScope [Xil05a] ermöglicht.

3.2 IP-Cores

Ein IP[9]-Core ist eine Beschreibung für einen Hardware-Logikblock, der die Entwicklung von Produkten auf Basis von konfigurierbaren Schaltkreisen vereinfachen soll. Diese Cores werden einmal entwickelt und verifiziert und können dann wiederverwendet werden. IP-Cores lassen sich in drei Kategorien systematisieren: Hard-Cores, Firm-Cores und Soft-Cores.

Bei einem Hard-Core handelt es sich um ein Modell, dessen Beschreibung in der Layoutebene gegeben ist. Ein Hard-Core ist daher sehr gut für die Entwicklung von ASIC geeignet. Er besitzt ein definiertes zeitliches Verhalten und ist bereits optimal an die vorhandenen Ressourcen angepasst. Der

[9]Intellectual property

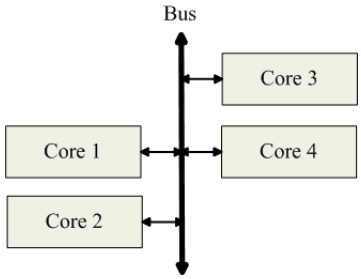

Abbildung 3.3: Anbindung von IP-Cores an einen Bus

Nachteil liegt in der schlechten Wiederverwendung in verschiedenen Arten von ASIC.

Mit Firm-Cores, auch Semi-Hard-Cores genannt, wird durch die Möglichkeit der Konfiguration der Cores eine größere Anzahl von Zielplattformen unterstützt. Soft-Cores sind diesbezüglich am flexibelsten. Sie existieren als Netzlisten oder sind in einer Hardwarebeschreibungssprache, wie VHDL, beschrieben. IP-Cores können durch Kauf erworben werden oder aber sie sind frei verfügbar durch Open-IP-Projekte, wie OpenCores [Ope06]. Die Quellen stehen dabei in den Sprachen VHDL oder Verilog zur Verfügung. Neben den eigentlichen Quellen gehören zu den entsprechenden Modellen auch eine Simulationsumgebung sowie entsprechende Dokumentationen.

Die Schnittstellen von IP-Cores sind sehr einfach gehalten. Sie orientieren sich meistens an entsprechenden Standards oder an existierenden elektronischen Realisierungen. Um die Interoperabilität zwischen verschiedenen Cores zu ermöglich, werden IP-Cores in den letzten Jahr verstärkt mit einer Verbindung zu Standardbussen, wie AMBA[10] [ARM99], WISHBONE [OPE02] oder CoreConnect [IBM99] ausgerüstet. Der Vorteil liegt in dem einfachen Aufbau eines eigenen Entwurfs unter Nutzung von IP-Cores (Abbildung 3.3). Nachteilig wirkt sich jedoch aus, dass jeder Datentransfer zwischen den Cores über den Bus geht, sodass dieser den Gesamtdatendurchsatz des Systems begrenzt. Außerdem muss selbst für einfache Cores die Anbindung an den teilweise sehr komplexen Bus erfolgen.

3.3 Handel-C

Zur Vereinfachung der Entwicklung von Programmen für FPGA wurde von der Firma Celoxica die Sprache Handel-C entwickelt [Cel02]. Dabei handelt es

[10] Advanced Microcontroller Bus Architecture

sich um eine Hardwarebeschreibungssprache, die auf der Syntax von ANSI-C aufbaut und diese um Sprachelemente zur Hardwarebeschreibung erweitert. Ein Beispielprogramm ist:

```
set intwidth = 16;

#define WIDTH 4

macro proc fir_filter(reset, input_data, output_data)
{
    ram int b[WIDTH + 1];
    ram int shift_reg[WIDTH];
    unsigned int ctr;
    int acc;
    int tmp;
    int help;

    for (ctr = 0; ctr <= WIDTH; ++ctr)
    {
        b[ctr<-3] = 1;
    }
    if (reset)
    {
        for (ctr = 0; ctr < WIDTH; ++ctr)
        {
            shift_reg[ctr<-2] = 0;
        }
        output_data = 0;
    }
    else
    {
        tmp = input_data;
        acc = tmp * b[0];
        for (ctr = 0; ctr < WIDTH; ++ctr)
        {
            acc += shift_reg[ctr<-2]*b[ctr<-3 + 1];
        }
        for (ctr = WIDTH - 1; ctr > 0; --ctr)
        {
            help = shift_reg[ctr<-2 - 1];
            shift_reg[ctr<-2] = help;
```

```
        }
        shift_reg[0] = tmp;
        output_data = acc;
    }
}
```

Vom ANSI-C-Standard werden jedoch keine externen Bibliotheksfunktionen unterstützt. Auch bei der Definition und Benutzung von Typen unterliegt Handel-C gewissen Einschränkungen. So sind Fließkommaformate (float, double), Aufzählungstypen, rekursive Funktionsdefinitionen sowie Typdefinitionen verboten. Um das Problem der rekursiven Funktionsaufrufe zu umgehen, besteht die Möglichkeit, Makros rekursiv zu definieren. Unterstützt werden Schleifen (While), Verzweigungen und arithmetische Berechnungen.

Bezüglich der Synthese sind bei der Umsetzung von Algorithmen in Handel-C verschiedenste Randbedingungen zu beachten. So muss bei jeder Variablendeklaration die Anzahl der Bits angegeben werden, durch welche die Variable repräsentiert wird. Zusätzlich kann auch die Ressource (externer/interner RAM[11], Register) angegeben werden, die der FPGA für die Realisierung der Variable benutzen soll. Durch das Schlüsselwort *par* müssen des Weiteren parallele Strukturen explizit definiert werden. Über das Schlüsselwort *delay* können zusätzliche Verzögerungselemente eingebaut werden. Dies ist notwendig, um die Daten zwischen den parallelen Strukturen zu synchronisieren.

Der Hauptgrund für die Benutzung von Handel-C liegt zum einen in der Wiederverwendung von Algorithmen aus dem Softwarebereich, zum anderen in der Vereinheitlichung der Programmiersprachen von Hard- und Software. Für kleine und einfache Projekte lassen sich damit auch schnell entsprechende Realisierungen für eine FPGA-Plattform von hardware-unerfahrenen Programmierern entwickeln [DV01, SB01]. Bei größeren Projekten werden jedoch schnell die Grenzen von Handel-C erreicht. Gerade bei der Verwendung von großen Feldern generiert Handel-C Lösungen, die mehr Ressourcen verlangen als ein FPGA zur Verfügung stellen kann.

Umso wichtiger ist der Schritt der Optimierung bzw. der Anpassung der Realisierung an die äußeren Bedingungen, wie Ressourcenverbrauch und Datendurchsatz. Dafür ist die Zerlegung des Programms in parallel ausführbare Strukturen erforderlich, wobei die Synchronität zwischen den Strukturen gerade im Hinblick auf Variablenzugriffe beachtet werden muss. In [DW03] werden verschiedene Optimierungen für einen Klassifikationsalgorithmus zur Erkennung von Bränden auf Basis von Satellitenbildern diskutiert. Es zeigt

[11] Random Access Memory

sich, dass gerade für umfangreiche Projekte der Vorteil der Wiederverwendung nicht den Nachteil der schlechten Realisierung im Hinblick auf Ressourcen und Datendurchsatz aufwiegen kann. Erst durch zusätzliche Optimierungen, die jedoch zeitaufwändig sind und ein Verständnis der zu erzeugenden Hardwarestrukturen voraussetzt, werden von Handel-C Lösungen generiert, die den äußeren Bedingungen genügen.

3.4 JHDL

In [BH98] wird die Hardwarebeschreibungssprache JHDL[12] beschrieben, die auf der Programmiersprache Java beruht. Zu der Sprache gehört eine Vielzahl an Werkzeugen, welche die Durchführung von Simulationen und Synthesen ermöglichen. Entwickelt wurde JHDL an der Brigham Young University und steht als frei verfügbare Software unter einer Open-Source-Lizenz zur Verfügung. Das Konzept von JHDL ist, die Beschreibung der Hardware durch den Gebrauch einer gängigen Software-Programmiersprache zu vereinfachen und damit einem Softwareprogrammierer den Zugang zu Hardwarebeschreibungen zu erleichtern. Das folgende Beispiel aus dem Tutorial [JHD04] zeigt, dass es sich bei JHDL nicht um einen Java-zu-HDL-Compiler handelt, d.h. einem Compiler, der ein beliebiges Java-Programm in eine entsprechende Hardwarerealisierung übersetzt.

```
// Import the base libraries for JHDL design
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class FullAdder extends Logic
{
    // This is cell's interface (ports)
    public static CellInterface[] cell_interface =
    {
        in("a", 1),
        in("b", 1),
        in("cin", 1),
        out("sum", 1),
        out("cout", 1)
    };

    // This is the constructor - it gets called when a
```

[12] Java Hardware Description Language

```
    // new FullAdder is desired
    public FullAdder(Node parent, Wire a, Wire b,
                     Wire cin, Wire sum, Wire cout)
    {

        // Since we extend Logic, always have to call
        // its constructor as the first thing
        super(parent);


        // Connect the wires passed in as parameters to
        // the constructor to
        // the ports from the CellInterface above.
        connect("a", a);
        connect("b", b);
        connect("cin", cin);
        connect("sum", sum);
        connect("cout", cout);

        // Build our logic as a collection of gates.
        or_o( and(a,b), and(a,cin), and(b,cin), cout );
        xor_o( a, b, cin, sum );

    }
}
```

Vielmehr werden durch Java-Objekte die Strukturen der Hardware beschrieben. Dies kann wieder, wie in VHDL, mit einer Struktur- oder einer Verhaltensbeschreibung geschehen. Mit Hilfe der mitgelieferten Entwicklungswerkzeuge können Programme simuliert bzw. für die Synthese Netzlisten erzeugt werden, deren Übersetzung dann mit den entsprechenden Synthesewerkzeugen der FPGA-Hersteller erfolgen kann.

Im Vergleich zu der Entwurfsmethode mit VHDL zeigen sich bei JHDL keine allzu großen Unterschiede. In beiden Methoden sind Simulationen und Synthesen möglich. In beiden Herangehensweisen muss sich der Entwickler mit einer neuen Syntax bzw. mit neuen Klassen auseinandersetzen. Auch ist bei beiden ein grundsätzliches Verständnis der Hardware vonnöten. Nur so lassen sich Realisierungen für den FPGA erzeugen, die auch den äußeren Bedingungen an Datendurchsatz und Platz genügen.

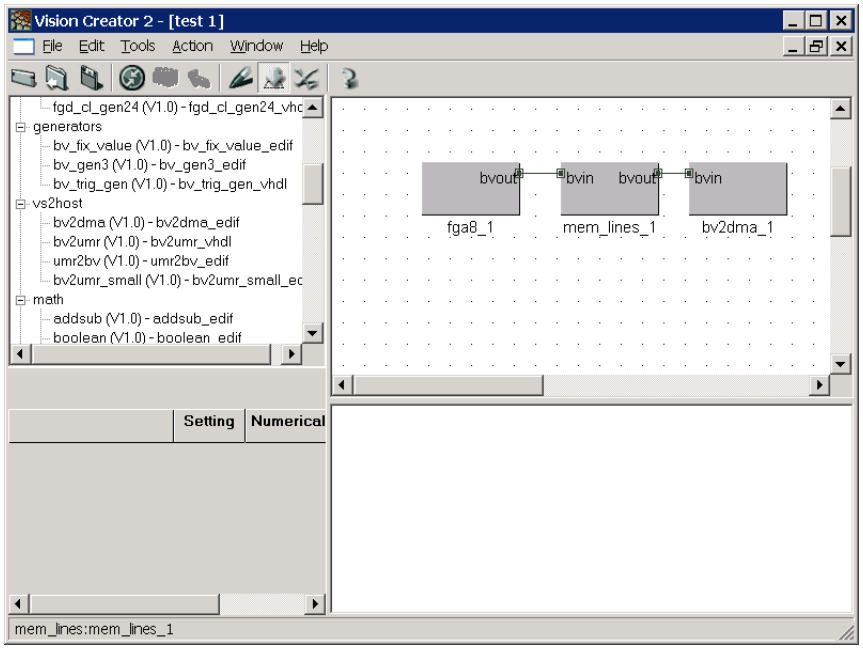

Abbildung 3.4: Graphische Oberfläche von VisionCreator

3.5 VisionCreator

VisionCreator [Pro04] ist eine Projektierungssoftware zur Bildverarbeitung. Diese Software gehört zu einem Paket der Firma ProDesign [Pro06] zusammen mit einer FPGA-PCI-Karte zur Bilderfassung und Auswertung. Die FPGA-Karten besitzen CameraLink-Eingänge [PA00] zur Verbindung mit entsprechenden Kameras, einen FPGA der Firma Xilinx zur Bildverarbeitung sowie verschiedene Speichermodule zur Aufnahme von Zwischenergebnissen. Die Verbindung zu einem PC erfolgt mit Hilfe der PCI-Schnittstelle und erreicht Datentransfers bis zu 264 MByte/s.

Die Software VisionCreator ermöglicht durch Nutzung einer umfangreichen Bildverarbeitungsbibliothek die schnelle Erstellung von Anwendungen für die Synthese. Die Beschreibung erfolgt durch einen Signalflussgraphen in einer graphischen Oberfläche (Abbildung 3.4). Daher sind bei der Erstellung von Anwendungen auf Basis der Bildverarbeitungsbibliothek keine Programmierkenntnisse einer Hardwarebeschreibungssprache erforderlich. Der durch die graphische Oberfläche erstellte Signalflussgraph kann mit VisionCreator in ein VHDL-Programm überführt werden, das mit den Synthesewerkzeugen des FPGA weiterverarbeitet wird.

Im Gegensatz zu den bisher beschriebenen Ansätzen der Hardwareent-

wicklung kann bei diesem Ansatz zwischen zwei Anwendergruppen unterschieden werden. Zum einen die hardwarenahen Entwickler der Bildverarbeitungsbibliothek und der Anpassung der FPGA-Plattformen, zum anderen die Entwickler der Anwendung. Nur die hardwarenahen Entwickler müssen Kenntnisse über die Hardware und die Hardwarebeschreibung besitzen.

Beim DLR wurde die Software VisionCreator in verschiedenen Projekten und Arbeiten eingesetzt. In [DRK+03] wird der Hardwareentwurf für einen Algorithmus beschrieben, der im Rahmen des Projekts OIS zur Objekterkennung entwickelt worden ist. In [Fle04] wird ein VHDL-Entwurf mit VisionCreator für das Labeln von Objekten beschrieben. Dabei zeigt sich, dass mit der graphischen Oberfläche eine schnelle Entwicklung von Anwendungen möglich ist. Voraussetzung ist jedoch, dass die entsprechenden Operationen bereits in der Bibliothek vorliegen und die FPGA-Karte unterstützt wird. Als nachteilig stellte sich heraus, dass keine Simulation der graphischen Beschreibung möglich ist. Ein Testen konnte daher nur auf einer entsprechenden FPGA-Plattform durchgeführt werden. Dabei konnte alternativ zu einer Kamera auch der PC als Datenquelle von Bildern genutzt werden. Hier stellte sich jedoch die Geschwindigkeit der Datenübertragung vom PC zum FPGA im Bereich von mehreren Sekunden für ein Bild als unzureichend heraus. Ein weiterer Nachteil ist, dass die Anzahl der Operationen, die in einer Anwendung verwendet werden können, primär davon abhängt, wie viele Speichermodule an den FPGA angeschlossen sind. Bei komplexen Anwendungen, wie z.B. dem Verkehrsmonitoring innerhalb des OIS-Projekts [HJP03], werden die Grenzen des Entwurfs mit VisionCreator deutlich.

Für den Entwickler der Bildverarbeitungsbibliothek ergaben sich ebenfalls Probleme, die kurz erläutert werden sollen. Zur Verbindung der Operationen, die im GUI als Blöcke dargestellt sind, wurde der BV-Bus definiert:

```
entity bv_bus_receiver
  generic (
    BV_DATA_BW  : INTEGER:= 8;        -- Breite des Busses
    -- weitere Generics
  )
  port (
    bv_clk      : IN  STD_LOGIC;      -- BV Clock
    bv_in_dat   : IN                  -- BV Daten
      STD_LOGIC_VECTOR(BV_DATA_BW-1 DOWNTO 0);
    bv_in_index : IN
      STD_LOGIC_VECTOR(7 DOWNTO 0);   -- Lfd. Nr des Bildes
    bv_in_en    : IN  bv_en_type;     -- BV Daten gültig
    bv_in_sync  : IN  bv_sync_type;   -- Sync Signale
```

```
    -- weitere Signale
  );
end;
```

Jede Kante in der Darstellung wird durch eine Menge von Signalen des BV-Busses dargestellt. Eine Operation kann nun eine beliebige Anzahl von Ein- und Ausgängen besitzen. Neben dem BV-Bus gibt es noch einen weiteren Bus, den so genannten UMR-Bus. Jede Operation ist mit einer entsprechenden Verbindung zum UMR-Bus ausgestattet, obwohl diese nicht explizit in der graphischen Darstellung angegeben ist. Über diesen Bus können den Operationen zusätzliche Parameter übermittelt werden. Ein Beispiel für die Parameter sind Threshold- oder Filterkoeffizienten.

Probleme treten nun bei der Entwicklung von neuen Operationen, z.B. einer Labelingoperation [Fle04], auf. Zum einen können Operationen nicht hierarchisch aus anderen, bereits definierten Operationen zusammengesetzt werden, d.h. eine strukturelle Definition ist nicht möglich. Auf der anderen Seite bedingt die Definition des BV-Busses Einschränkungen hinsichtlich der Operationen. Wie anhand der Definition des BV-Busses erkennbar ist, gibt es kein Rücksignal vom Empfänger zum Sender. Der Empfänger ist damit gezwungen, den eingehenden Datenstrom mit derselben Geschwindigkeit zu bearbeiten wie der Sender. Dieses kann aber, wie beim Labeling geschehen, zu eingeschränkten Lösungen führen.

Die Integration einer nicht von ProDesign hergestellten FPGA-Plattform ist über eine XML-Konfigurationsdatei möglich. In dieser muss dann eine Anpassung der FPGA-Karte an die in VisionCreator vorkommenden Standardmodule, wie z.B. die Speichermodule, stattfinden.

3.6 Ptolemy

Bei dem Projekt Ptolemy [HLL+03, LNW03] handelt es sich um ein Open-Source-Projekt, das an der Berkeley University of California entwickelt wurde, um das Modellieren, Simulieren und den Entwurf von heterogenen Systemen zu vereinfachen. Der Fokus liegt dabei auf eingebetteten Systemen, besonders jene, die analoge und digitale, Hardware und Software sowie elektronische und mechanische Komponenten miteinander verbinden. Viele Eigenschaften von Ptolemy sind für die Beschreibung von heterogenen Systemen verallgemeinerbar und werden in den nachfolgenden Kapiteln im Zusammenhang mit dem Betriebssystemkonzept für FPGA noch näher erläutert.

Die Beschreibung eines Systems oder Subsystems erfolgt in Ptolemy durch Graphen (Abbildung 3.5), die aus gerichteten Kanten und Knoten bestehen.

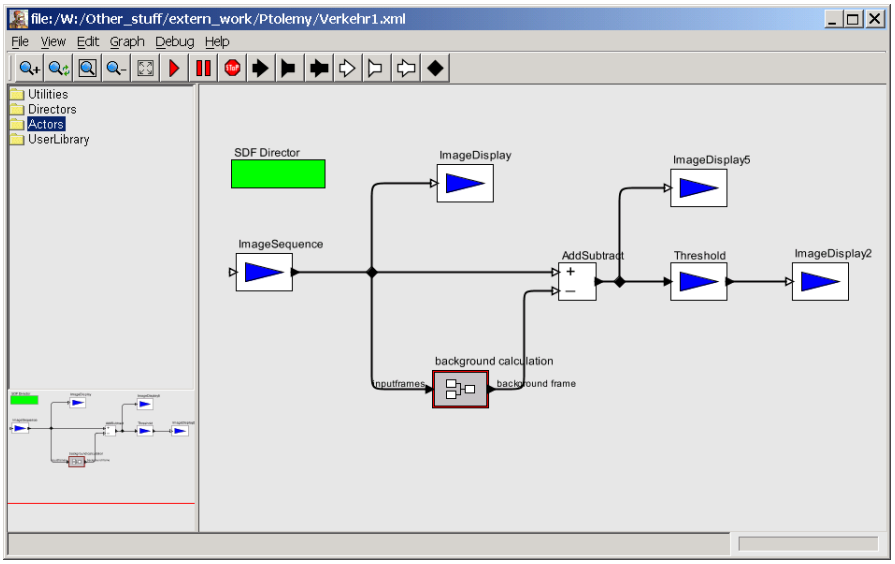

Abbildung 3.5: Graphische Oberfläche von Ptolemy

Die Knoten werden in Ptolemy auch Aktoren genannt. Die Interpretation des Graphen für die Modellierung wird durch so genannte Domänen definiert. So können z.B. Aktoren Operationen definieren, wohingegen Kanten die Verknüpfung der Operationen wiedergeben. Die Domäne legt die Art der Übertragung von Daten zwischen den Aktoren fest. In der CT[13]-Domäne entspricht eine Kante einem Signal, das zeit- und wertekontinuierlich definiert ist [Liu98]. Die Aktoren sind dann typischerweise algebraische Gleichungen oder Differentialgleichungen zwischen den Eingabe- und Ausgabesignalen. Zur Simulation dieses Modells müssen dann Lösungen, d.h. zeit- und wertekontinuierliche Funktionen gefunden werden, welche die Gleichungen erfüllen. Eine andere Domäne, bekannt aus der FPGA-Programmierung, stellt die Domäne der endlichen Zustandsautomaten (FSM[14]) dar. Hier entsprechen Aktoren Zuständen, wohingegen die Kanten mögliche Übergänge der Zustände repräsentieren.

Eine weitere wichtige Domäne ist die des Synchronen Datenflusses (SDF) [LM87]. Diese Domäne spielt eine große Rolle im Bereich der Signalverarbeitung. Zwischen den Aktoren werden die diskreten Datenströme synchron übergeben. Kein Element des Datenstroms kann dabei verloren gehen. Bei den Datenströmen kann es sich z.B. um Video- oder Audiodaten handeln. Als Beispiel sei wieder auf Abbildung 3.5 verwiesen. Gezeigt ist ein Algorithmus,

[13]Continous Time
[14]Finite State Machine

der im Projekt OIS [DRK+03] untersucht worden ist. Es handelt sich um eine einfache Signalverarbeitungskette zur Ermittlung von bewegten Objekten in einer Bildsequenz. In dieser Kette gibt es Aktoren zur Signalverarbeitung (*Threshold*, *background calculation*) und Aktoren für die Interaktion mit dem Anwender (*ImageSequence*, *ImageDisplay*). Diese dienen zum Laden einer

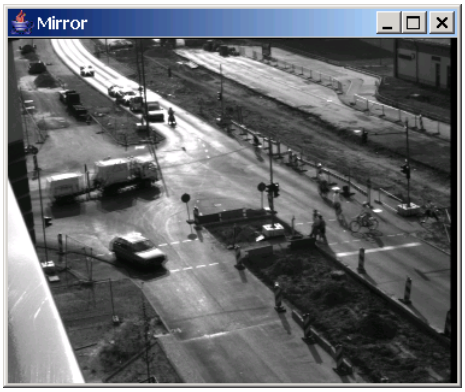

Abbildung 3.6: Originalbild

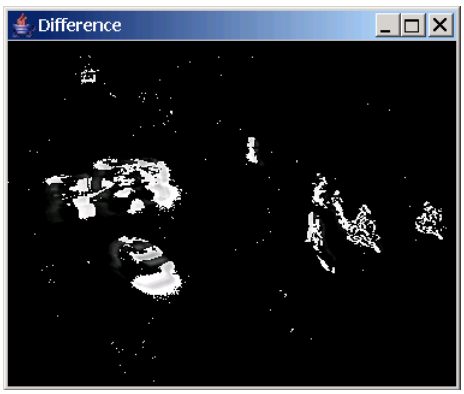

Abbildung 3.7: Differenz des Originalbildes zum geschätzten Hintergrund

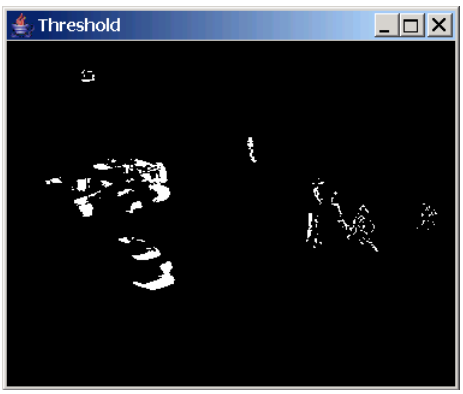

Abbildung 3.8: Binarisierung des Differenzbildes

Bildsequenz bzw. zum Visualisieren von Zwischen- und Endergebnissen (Abbildung 3.6, 3.7, 3.8). Anhand dieses Beispiels wird noch ein anderer wichtiger Aspekt bei der Beschreibung von heterogenen Systemen deutlich: die Benutzung einer hierarchischen Beschreibung. Der Operator *background calculation* ist selbst wieder als Modell definiert und aus anderen fundamentalen Aktoren aufgebaut (Abbildung 3.9).

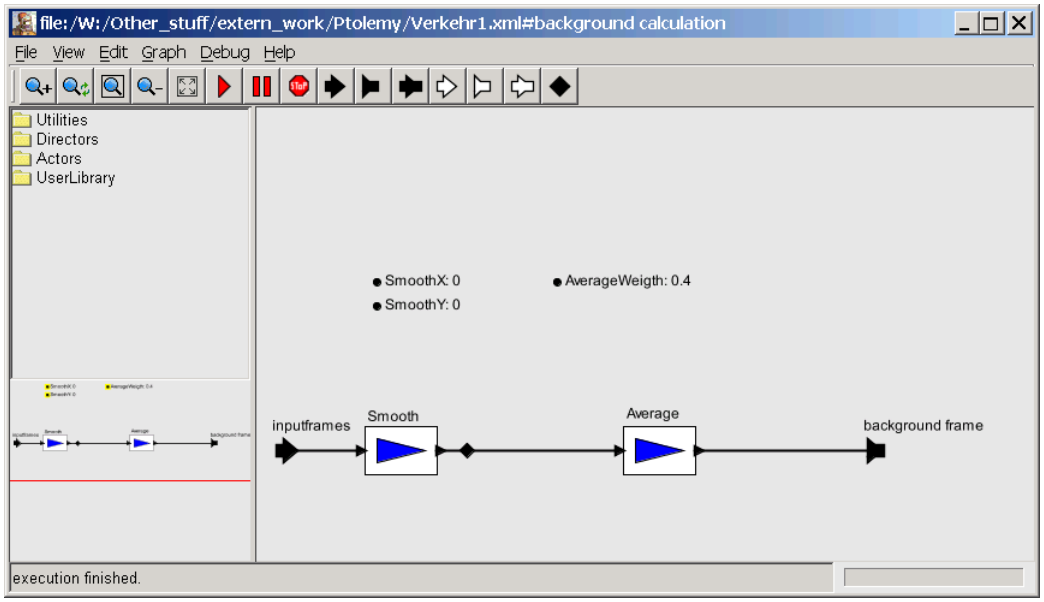

Abbildung 3.9: Hierarchischer Aufbau des Aktors *background calculation* aus den Aktoren *smooth* und *average*

Im Ptolemy-Projekt gibt es nun mehrere Dutzend verschiedener Domänen zur Beschreibung von Systemen. Ein Problem, was sich daraus ergibt, ist die Anzahl der fundamentalen Aktoren, die theoretisch für jede Domäne neu definiert werden müssen. Die Definition erfolgt dabei in der Sprache Java. Dadurch ist eine Wiederverwendung auf verschiedenen Computersystemen garantiert. Zu den vielen verschiedenen Domänen kommen noch die verschiedenen Typen von Datensignalen, die übertragen bzw. von Aktoren bearbeitet werden können. Um diese Probleme in den Griff zu bekommen, wurden die Begriffe des Domänen-Polymorphismus und des Typen-Polymorphismus eingeführt und bei der Programmierung von fundamentalen Aktoren umgesetzt. Mit diesen Begriffen ist gemeint, dass die Aktoren so definiert sein müssen, dass eine Vielzahl an Domänen und eine Vielzahl an Typen mit derselben Java-Beschreibung auskommen. Dieses wird dadurch erreicht, dass Aktoren von einer speziellen Java-Klasse abgeleitet werden, welche die Grundeigenschaften des Domänen-Polymorphismus und des Typen-Polymorphismus besitzt.

Bei den so erstellten ausführbaren Modellen handelt es sich um Simulationen. Für eine Realisierung des Modells auf einem konkreten System muss erst eine Generierung von Programmcode für die Systemplattformen durchgeführt werden. Wie bereits in den vorhergehenden Abschnitten erläu-

tert, gibt es zwei Arten von Zielplattformen: Prozessoren und konfigurierbare Bausteine. Beide Arten von Zielplattformen sollen durch das Ptolemy-Projekt abgedeckt werden. Im Bereich der Prozessoren ist dies auch möglich. Sollte auf dem Prozessor eine virtuelle Maschine von Java laufen, kann das Java-Programm direkt übernommen werden. Trotzdem gibt es hier Bestrebungen, ein neues Java-Programm zu erzeugen. Dies liegt daran, dass durch den Typen- und Domänen-Polymorphismus die Java-Programme, die direkt zur Simulation der Modelle benutzt werden, sehr umfangreich und damit auch sehr langsam sind. Da die Programme von Ptolemy hauptsächlich in eingebetteten Systemen zur Anwendung kommen, sollten die Zielprogramme möglichst klein und schnell sein. Daher besteht der Schwerpunkt der Codegenerierung in Ptolemy in der Erzeugung von Java-Programmen, die den obigen Anforderungen genügen. Konvertierungstools können diese Java-Programme auch in andere Sprachen, wie z.B. in C, übersetzen.

Die Codegenerierung erfolgt mit Hilfe eines abstrakten Syntaxbaumes. Dieser wird erzeugt, indem die Quellen für jeden Aktor analysiert werden und die in den Aktoren vorkommenden abstrakten Typen an die konkreten Typen gebunden werden [Tsa00]. In einem nachfolgenden Schritt werden dann noch zusätzliche domänenabhängige Informationen dem Baum hinzugefügt. Mittels eines Back-End-Compilers wird dann der abstrakte Syntaxbaum in die Zielsprache, wie z.B. Java, überführt. Theoretisch ist es auch möglich, Back-End-Compiler für andere Zielsprachen zu entwickeln. Da es jedoch auch Compiler gibt, die direkt eine Konvertierung von Java in andere Sprachen vornehmen können, wird hier die Benutzung solcher Compiler empfohlen. Solange die Zielplattform ein Prozessor ist, decken die bei Ptolemy mitgelieferten Werkzeuge den kompletten Entwurfszyklus ab.

Anders sieht es jedoch bei konfigurierbaren Bausteinen aus. Hier ist eine Codegenerierung zunächst nur auf die SDF-Domäne beschränkt. In [Wil98] wird ein Ansatz für die Umsetzung von SDF-Modellen nach VHDL beschrieben. Dieser Ansatz beruht auf der bereits vorgestellten Sprache JHDL. In den hierbei erstellten VHDL-Programmen findet vorrangig eine Abbildung der Algorithmen des Java-Programms auf VHDL-Sprachelemente statt. Die Realisierung für eine konkrete FPGA-Plattform muss nachträglich noch von Hand durchgeführt werden.

Zusammenfassend kann gesagt werden, dass das Projekt Ptolemy eine sehr gute Möglichkeit darstellt, heterogene Systeme zu entwickeln, zu beschreiben und zu simulieren. Mit seinem Domänen-Ansatz sowie der Typen- und Domänen-Polymorphie wird eine einfache und schnelle Entwicklung von komplexen Bibliotheken ermöglicht. Leider beschränkt sich die Funktionalität von Ptolemy auf die Modellierung, den Entwurf und die Simulation. Bei der Verwendung von Modellen für konkrete Plattformen sowie die Unterstüt-

zung von verteilten Systemen befindet sich das Projekt noch in der Entwicklungsphase. Gerade im Hinblick auf die konfigurierbaren Bausteine ist zurzeit nicht zu erkennen, wie die Vielzahl an verschiedenen FPGA-Plattformen beherrscht werden kann und die Einbindung dieser in ein heterogenes System vonstatten gehen soll.

3.7 Hardware-Software-Codesign

Neben den bereits vorgestellten Methoden zur Entwicklung und Modellierung von heterogenen Systemen werden seit Jahren Ansätze verfolgt, die das Problem auf die Beschreibung und Modellierung von Software zurückführen [Bal97, GLD+03, HBK96, VCA94]. Für die Softwareentwicklung gibt es bereits eine große, seit Jahrzehnten erprobte Anzahl an Funktionen. Werden diese genutzt, kann die Entwicklungszeit von heterogenen Systemen verkürzt werden. Ausgangspunkt der Betrachtung ist eine Systembeschreibung in einer gängigen Software-Programmiersprache wie C/C++, die mit zusätzlichen Systembeschreibungselementen ausgestattet ist (Abbildung 3.10). Durch Analyse des Codes ist dann ein Programm in der Lage zu erkennen, welche Teile des Algorithmus auf welchen Teilen des heterogenen Systems ausgeführt werden müssen, damit das Gesamtsystem den vorgegebenen Randbedingungen genügt. Dazu muss das Programm nicht nur in der Lage sein, den Algorithmus der Software zu analysieren und Parallelisierbarkeiten automatisch zu erkennen, es muss auch in der Lage sein abzuschätzen, inwiefern die durch eine entsprechende Partitionierung gewählte Zerlegung die physikalischen Verbindungen zwischen den verschiedenen Plattformen belastet.

Um eine Partitionierung zu finden, müssen folgende Schritte durchgeführt werden. Als erstes muss der Algorithmus in Teilalgorithmen für die Zielkomponenten zerlegt werden. Dabei werden zwischen den einzelnen Zielkomponenten logische Verbindungen auftreten, die in einem zweiten Schritt auf entsprechende physikalische Verbindungen abgebildet werden müssen. In einem dritten Schritt muss überprüft werden, inwiefern die so erzeugte Partitionierung den gegebenen Randbedingungen genügt. Sollten die Randbedingungen nicht erfüllt sein, muss wieder mit dem ersten Schritt angefangen werden und eine neue Zerlegung bzw. Verbindungszuordnung gewählt werden. Auf diese Weise werden alle möglichen Partitionierungen des Systems getestet, um eine Lösung zu finden. Verfahren der Künstlichen Intelligenz erlauben es, die Randbedingungen bereits bei der Zerlegung des Algorithmus bzw. der Verbindungszuordnung einzubeziehen. Das hat den Vorteil, dass nicht alle möglichen Zerlegungen und Zuordnungen getestet werden müssen.

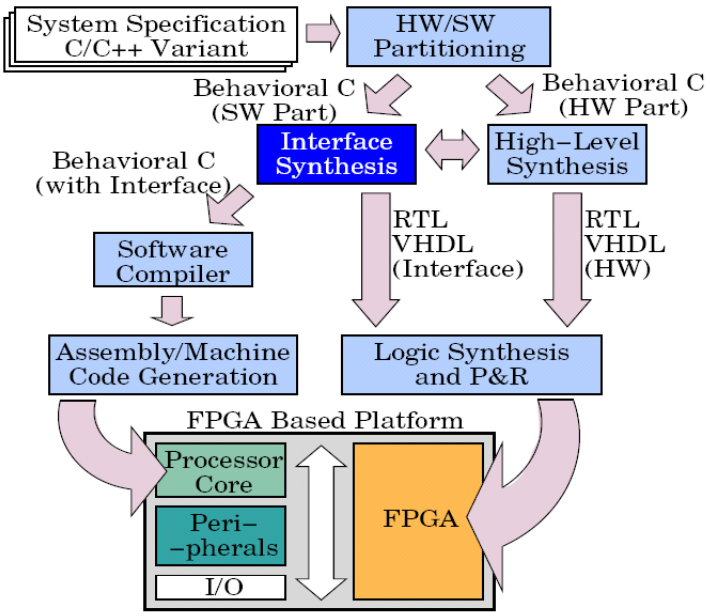

Abbildung 3.10: Hardware-Software-Codesign aus [GLD+03]

Allerdings ist es schwierig, die Erfüllung der Randbedingungen automatisch zu untersuchen [Bal97, VCA94].

Als Ergebnis der Partitionierung ergeben sich dann mehrere unabhängige Beschreibungen für die verschiedenen Zielplattformen. Diese Beschreibungen können dann mit den entsprechenden Werkzeugen für die Zielplattform, wie C- oder VHDL-Compiler, in ein ausführbares Programm für die Plattform überführt werden. Neben der Erzeugung von ausführbaren Programmen für die Zielplattform unterstützen diese Projekte den heterogenen Systementwurf meist mit einer zusätzlichen Hardware-Software-Cosimulation.

Die hier beschriebene Arbeitsweise stellt das gewünschte Ergebnis eines Hardware-Software-Codesigns dar. Leider ist das Problem der automatischen Partitionierung so komplex, dass sie derzeit für eine Produktentwicklung noch nicht eingesetzt werden kann.

3.8 Diskussion der Entwurfsmethoden

Die in den vorhergehenden Abschnitten vorgestellten Methoden des Entwurfs von Programmen für FPGA wurden bereits unter verschiedenen Gesichtspunkten betrachtet. Da diese jedoch stark von der Methode selbst abhängig

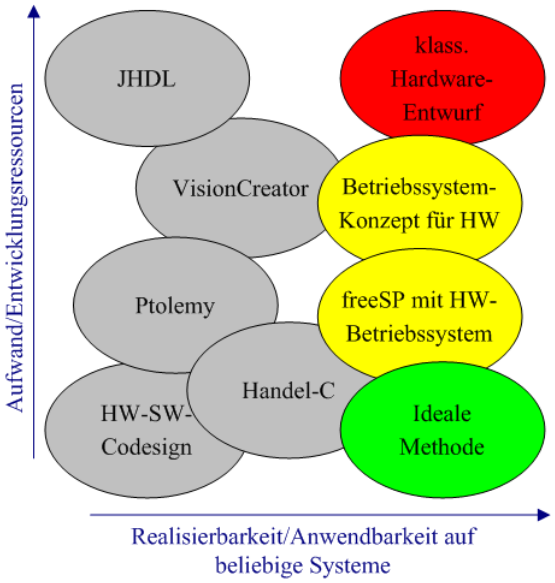

Abbildung 3.11: Bewertung verschiedener Entwurfsmethoden

sind, wird für eine vergleichende Betrachtung die Anzahl der Kriterien auf zwei reduziert. Damit soll eine stark abstrahierte Bewertung der Methoden ermöglicht werden. Ein Kriterium soll der Entwicklungsaufwand sein. Damit ist zum einen die Zeit gemeint, die benötigt wird, um von der Idee eines Systems zu einem fertigen Produkt zu kommen, zum anderen aber auch die Zeiten für die Einarbeitung in die Methoden. Darunter fällt zum Beispiel die Einarbeitung in die verschiedenen Programmiersprachen.

Das zweite Kriterium soll angeben, inwieweit die jeweilige Methode den Entwurf von beliebigen Algorithmen auf beliebige konfigurierbare Hardware unterstützt.

Für die in den vorhergehenden Abschnitten aufgeführte Auswahl an Methoden ergibt sich dann das in Abbildung 3.11 gezeigte Diagramm. Bei der Verwendung der Sprache JHDL für einen Hardwareentwurf ist mit einer langen Entwicklungszeit zu rechnen. Dies liegt daran, dass die Sprache JHDL zwar auf Java aufbaut, jedoch die Klassen zur Realisierung von bestimmten Hardwareelementen erst erlernt werden müssen. Außerdem befindet sich dieses Projekt zurzeit noch in der Entwicklungsphase, sodass nur eine eingeschränkte Anzahl an Hardwarekonstrukten unterstützt werden. Ein weiterer Nachteil ist die fehlende Einbettung in einen Entwurf für ein heterogenes System.

Etwas besser ist die Software VisionCreator positioniert. Aufgrund ihrer

graphischen Oberfläche und der Unterstützung von Signalverarbeitungsmodulen benötigt der Hardwareentwurf nicht viel Zeit. Dieses gilt jedoch nur, solange eine entsprechende FPGA-Karte benutzt wird. Soll das System auf einer noch nicht unterstützten Karte laufen, muss hier eine Anbindung der Karte an das VisionCreator-System durchgeführt werden. Leider fehlt hier ebenfalls die Einbettung in ein heterogenes System sowie die Unterstützung einer Simulation.

Das Projekt Ptolemy verkürzt aufgrund der sehr umfangreichen Aktoren sowie der Unterstützung von Simulationen und Modellen auf verschiedenen Ebenen die Modellierung von Systemen immens. Leider fehlt hier die Möglichkeit der Realisierung auf verschiedenen Zielplattformen bzw. muss umständlich mit JHDL durchgeführt werden. Dadurch ist die Anwendbarkeit gerade im Hinblick auf konfigurierbare Bausteine stark eingeschränkt.

Einer noch stärkeren Einschränkung unterliegen die Hardware-Software-Codesign-Ansätze. Projekte, in denen diese Ansätze umgesetzt werden können, besitzen eine sehr kurze Entwicklungszeit, da sowohl auf umfangreiche Software-Bibliotheken zurückgegriffen werden kann als auch der Prozess der Partitionierung automatisch abläuft. Solange jedoch die automatische Partitionierung unter den gegebenen Randbedingungen nicht allgemein funktioniert, kommen diese Ansätze nicht für Produktentwicklungen in Frage.

Im Gegensatz dazu können mit Handel-C Produktentwicklungen durchgeführt werden. Leider ist diese Methode weniger durch die Algorithmik als durch die möglichen Zielplattformen in der Anwendbarkeit beschränkt. Außerdem gibt es keine Möglichkeit, Handel-C in einen Gesamtentwurf für ein heterogenes System zu integrieren.

Auf der rechten Seite der Abbildung 3.11 befindet sich oben der klassische Entwurf mit einer Hardwarebeschreibungssprache und den typischen Hardwarewerkzeugen. Dieser beinhaltet ebenfalls keine Möglichkeit eines integrierten Entwurfs von heterogenen Systemen. Dafür existieren jedoch für jegliche Art von Plattform entsprechende Werkzeuge, um ein Programm zu erstellen. Dieser Vorteil ist jedoch mit einem erhöhten Einarbeitungsaufwand verbunden.

Ganz unten rechts befindet sich die ideale Methode. Mit ihr soll es mit minimalen Aufwand möglich sein, für jegliche Art von heterogenen Systemen einen Entwurf durchzuführen. Diese Methode gibt es noch nicht und es ist auch nicht absehbar, welche der verschiedenen hier diskutierten Methoden am ehesten dazu in der Lage sein wird.

Auf der rechten Seite der Abbildung 3.11 gibt es noch zwei Methoden, die bisher nicht vorgestellt wurden. Da ist zum einen die Methode unter Benutzung eines Betriebssystemkonzepts und zum anderen die in [Wei] beschriebene Methode freeSP. Beide Methoden werden in den nachfolgenden

Kapiteln erläutert. Dazu werden im nächsten Kapitel allgemeine Probleme aufgezeigt, die bei Entwürfen von heterogenen Systemen auftreten. Um diese Probleme zu lösen, wird eine Entwurfsmethode auf Basis eines Betriebssystems vorgestellt.

Kapitel 4

Heterogene Systeme

In diesem Kapitel werden die Probleme bei der Realisierung von Entwurfsmethoden für heterogene Systeme diskutiert. Des Weiteren wird daraus eine Entwurfsmethode abgeleitet, die mit Hilfe eines Betriebssystemkonzepts in der Lage ist, diese Probleme zu lösen.

Das Ziel eines Entwurfs für heterogene Systeme ist, auf Basis einer Systemspezifikation die Programme für alle programmierbaren Elemente des Systems zu generieren. Diese Elemente können entweder die Programme für Prozessoren sein oder Programme für konfigurierbare Hardware.

Die Erzeugung dieser Programme sollte weitestgehend automatisch erfolgen. Dies garantiert eine möglichst kurze Entwicklungszeit sowie eine hohe Zuverlässigkeit. Zurzeit existiert jedoch noch keine Methode, die dazu in der Lage ist. Dies liegt vor allem an den teilweise sehr komplexen Randbedingungen, denen ein solches System unterliegt. Zu diesen Randbedingungen gehört zum Beispiel, dass der Systemalgorithmus auf das komplette System untergebracht werden kann und natürlich auch die Teilalgorithmen auf die entsprechenden Zielplattformen. Schon allein dieses für alle Systeme geltende Kriterium zu erfüllen, ist hinreichend schwierig. Gerade auf einer konfigurierbaren Hardware ist es relativ schwer anzugeben, wie groß der endgültige Entwurf ist. Meistens sind hier nur grobe Schätzungen oder Hochrechnungen möglich. Ähnlich sieht es mit dem Datendurchsatz aus. Hier spielt es eine entscheidende Rolle, ob die Einzelsysteme genügend Datendurchsatz zulassen, um den Systemdatendurchsatz zu garantieren. Aber auch bei der Abschätzung des Datendurchsatzes eines Einzelsystems werden schnell die Grenzen des Berechenbaren erreicht. Immer dann, wenn mehrere Instanzen versuchen, eine Ressource zu teilen, z.B. einen Speicher oder die Kommunikationspfade, hängt der Datendurchsatz auch von dem zeitlichen Verhalten eines Datenstroms ab. Andere Kriterien können z.B. Stromverbrauch, Wärmeabgabe oder Latenzzeiten sein.

Bei der Realisierung einer entsprechenden Methode für heterogene Systeme treten drei wesentliche Probleme auf. Zum einen muss festgelegt werden, wie der Systemalgorithmus beschrieben wird. Aufgrund des heterogenen Systems muss die Beschreibung so gewählt sein, dass sie unabhängig von der Zielplattform ist. Das zweite Problem besteht darin, eine Partitionierung des Systems zu finden. Durch die Partitionierung wird festgelegt, welcher Teil des Systemalgorithmus auf welcher Zielplattform laufen soll. Hier spielen natürlich auch die Randbedingungen des Entwurfs eine sehr wichtige Rolle. Der dritte wichtige Punkt ist die Darstellung der Verbindungen zwischen den Plattformen. Aufgrund der Vielzahl der möglichen physikalischen Verbindungen zwischen den Systemkomponenten muss eine Beschreibung gefunden werden, die eine Abstraktion von der konkreten Realisierung darstellt. Alle drei Probleme werden im Weiteren genauer betrachtet.

4.1 Systembeschreibung

Die Systembeschreibung legt das Verhalten eines Systems fest. Zu dem Verhalten gehört zum einen das algorithmische Verhalten, das heißt das Verhalten der Ausgabewerte in Abhängigkeit von den Eingabewerten. Zum anderen gehören auch das zeitliche Verhalten sowie andere Randbedingung des Systems dazu. Alle zusammen in einer Beschreibung zu vereinen und weiterzuverarbeiten, stellt ein hohes Maß an Komplexität dar. Daher wird im Weiteren nur die algorithmische Systembeschreibung betrachtet. Erst beim Partitionierungsprozess werden die anderen Anforderungen an das System eine wichtige Rolle spielen.

Die algorithmische Systembeschreibung kann zum einen auf bereits bestehenden Beschreibungssprachen, wie z.B. C oder Java, aufbauen. Es ist aber auch möglich, eine eigene Sprache zu definieren. Der Vorteil einer bereits bestehenden Sprache liegt darin, dass hier die Spezifikation der Sprache entfällt sowie der Einarbeitungsaufwand geringer ist. Außerdem kann auf bereits vorhandene, teilweise recht umfassende Bibliotheken an Funktionen zurückgegriffen werden. Dies verkürzt die Entwicklungszeit und erhöht die Zuverlässigkeit des Gesamtsystems. Die Erzeugung von Programmen für verschiedene Prozessorzielplattformen stellt kein größeres Problem dar. Wird die Sprache C benutzt, so gibt es für fast alle Zielprozessoren einen Compiler, der ein entsprechendes Zielprogramm erzeugen kann. Sollte die Zielplattform keinen Compiler für diese Sprache unterstützen, so gibt es genügend Compiler, die zwischen der Systemsprache und einer umsetzbaren Sprache übersetzen können, z.B. Java2C [Au99]. Die Erzeugung von Programmen für Prozessoren ist daher unproblematisch. Schwieriger sieht es jedoch mit der

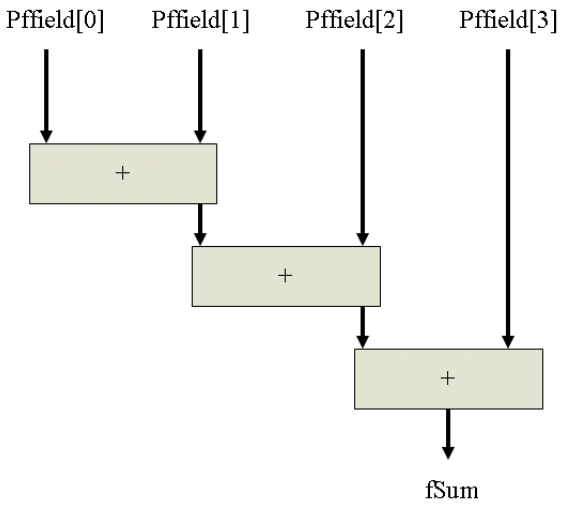

Abbildung 4.1: Hardwarestruktur zur Addition von vier Elementen ohne Optimierung

konfigurierbaren Hardware aus. Wären Compiler verfügbar, die von einer Softwarebeschreibungssprache in eine Hardwarebeschreibungssprache übersetzen könnten, würde die Systembeschreibung kein Problem darstellen.

Bei der Realisierung eines entsprechenden Compilers treten jedoch zwei wesentliche Hindernisse auf. Zum einen sind dies die Randbedingungen des Systems, die bei der Programmierung der konfigurierbaren Hardware einen größeren Einfluss haben als für die Prozessoren. Zum anderen fließt in einen optimalen Entwurf eine Menge an implizitem Wissen ein. Dies soll an einem einfachen Beispiel demonstriert werden. Sei der Algorithmus dadurch gegeben, dass die Summe über vier Elemente berechnet werden soll. In C würde das folgendermaßen aussehen:

```
float pffield[4];
unsigned int uii;
float fsum=0;

for (uii=0;uii<4;++uii) fsum+=pffield[uii];
```

Aus der Analyse der Ablaufreihenfolge würde sich die in Abbildung 4.1 gegebene Hardwarestruktur ergeben. Diese lässt sich nun hinsichtlich verschiedener Kriterien optimieren. Soll z.B. die Signallaufzeit möglichst kurz sein, kann die in Abbildung 4.2 gezeigt Baumstruktur gewählt werden. Eine Bedingung nach minimalem Platzverbrauch ergibt dagegen die in Abbildung

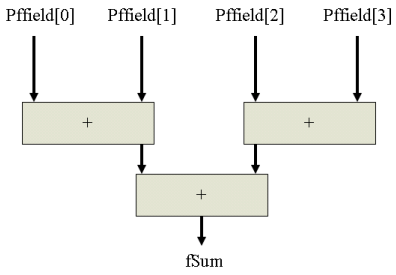

Abbildung 4.2: Hardwarestruktur optimiert nach minimalen Signallaufzeiten

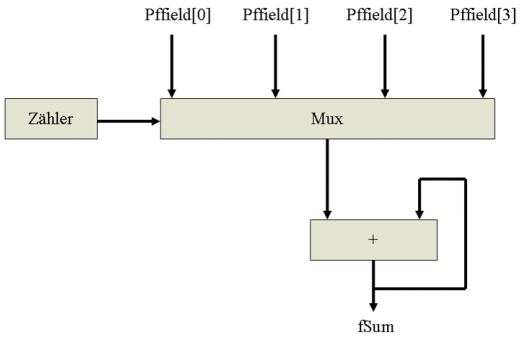

Abbildung 4.3: Hardwarestruktur optimiert nach minimalem Platzverbrauch

4.3 gezeigt Struktur. Die gezeigten Modifikationen sind durch Einbeziehung impliziten Wissens über die Operation, in diesem Fall die Addition, möglich. Die Umformung nach Abbildung 4.2 ist nur deshalb möglich, da die Addition assoziativ ist. Soll an Stelle der Addition eine andere, frei-definierte Funktion ausgeführt werden, würde diese Umformung nur möglich sein, wenn die Assoziativität von dem Compiler bewiesen würde. Dies bedeutet jedoch, dass ein entsprechender Compiler den Leistungsumfang eines Theorembeweisers [BBS+98] beinhalten müsste.

Einen möglichen Ausweg stellt die Erweiterung der Sprache durch Elemente dar, welche die Struktur der Hardware genauer beschreiben. Ein solches Vorgehen wurde bereits mit Handel-C in Abschnitt 3.3 aufgezeigt. Dies hat jedoch den Nachteil, dass nicht mehr auf bereits vorhandene Bibliotheken zurückgegriffen werden kann. Außerdem müssen alle Bestandteile des Algorithmus mit entsprechenden Zusatzinformationen ausgestattet werden, da an dieser Stelle die Partitionierung noch nicht feststeht.

Eine dritte Möglichkeit ist die deklarative Beschreibung. Im Gegensatz zu den normalen imperativen Sprachen, bei denen mit Hilfe der Sprache beschrieben wird, wie etwas zu lösen ist, kommt es bei der deklarativen Beschreibung nur auf das Ziel einer Berechnung an. Bei der Umsetzung dieser Beschreibung in ein Programm muss dann der Compiler den Lösungsweg mit entsprechendem Expertenwissen ermitteln. Durch diesen Ansatz findet eine Trennung zwischen einer aktuellen Implementierung und der abstrakten Beschreibung statt. Die Repräsentation einer deklarativen Sprache ist mit einem gerichteten Graphen möglich. Hierbei stellen die Knoten die Operationen dar. Die Kanten drücken die Verbindungen bzw. die Ergebnisübermittlungen aus. Während Kanten nicht weiter untergliedert werden können, kann ein Knoten durch einen anderen Signalflussgraphen näher beschrieben werden. Die Anzahl der äußeren Kanten muss dabei erhalten blei-

ben. Diese Art der Beschreibung wird bereits in vielen Projekten angewendet [CZNJ03, Gro01, HLL+03, Wei] und soll auch hier zur Beschreibung des Algorithmus benutzt werden.

Die konkrete Realisierung der Kanten spielt auf dieser Ebene der Beschreibung noch keine Rolle. Es ist sogar möglich, dass die Kanten innerhalb einer Zielplattform durch verschiedene Realisierungen repräsentiert werden. Beispiele für Kantenbeschreibungen in einer Prozessorrealisierung wären Interprozesskommunikationsmittel, wenn die Knoten als eigenständige Prozesse auftreten. Alternativ wäre es auch möglich, die Kanten als einfache Datenpuffer zu interpretieren, über welche die Operationen der Knoten als Funktionen ausgeführt werden. Bei einer FPGA-Implementierung stellen die Kanten eine Menge von Signalen dar, die zur Übertragung der Informationen genutzt werden. Die Menge der Signale und deren Protokoll der Übertragung muss wohl-definiert sein. Beispiele für Implementierungen von Kanten für die verschiedenen Plattformen werden in Kapitel 6 vorgestellt.

4.2 Partitionierung

Ist der Systemalgorithmus durch eine allgemeine Beschreibungssprache spezifiziert, muss als nächstes eine Partitionierung des Systems gefunden werden. Durch die Partitionierung wird der Systemalgorithmus in Teilalgorithmen zerlegt, die auf den verschiedenen programmierbaren Elementen des Zielsystems ausgeführt werden sollen [Tei97]. Bei der Zerlegung spielen die Randbedingungen, unter denen der Systemalgorithmus umgesetzt werden soll, eine große Rolle. Zu diesen Randbedingungen gehören z.B. Datendurchsatz, Latenzzeiten und Verbrauch an Platz und Energie. Eine mit jedem System verbundene Bedingung ist dabei, dass die Programme für die einzelnen Systemkomponenten so dimensioniert sind, dass sie auf diesen auch ausgeführt werden können. Gerade für die konfigurierbare Hardware stellt dies ein entscheidendes Kriterium dar. Hier werden mit heutiger Hardware schnell die Grenzen des Realisierbaren erreicht.

Bei der Durchführung einer Partitionierung gibt es zwei Extremfälle. Der erste besteht darin, den kompletten Systemalgorithmus auf einem Prozessorelement ausführen zu lassen, z.B. auf einem PC. In den seltensten Fällen wird es zu Platzproblemen bei der Umsetzung des Algorithmus kommen. Diese Lösung sichert jedoch häufig nicht den erforderlichen Datendurchsatz. Aufgrund der eingeschränkten Parallelität fällt der Datendurchsatz bei Prozessoren deutlich geringer aus als bei konfigurierbaren Hardwareelementen. Ein Vorteil einer solchen Systemlösung liegt jedoch in dem einfachen Systemaufbau und einer kurzen Entwicklungszeit. Aufgrund dieser Eigenschaften

wird solch eine Lösung auch gern als Simulationsbasis genutzt.

Der andere Extremfall besteht darin, den kompletten Algorithmus in einer konfigurierbaren Hardware ablaufen zu lassen. Solch ein System ist optimal an die Systemeigenschaften angepasst. Der Nachteil liegt jedoch darin, dass zum einen der Systemalgorithmus größer ist, als dass er in gängige konfigurierbare Hardware passen würde. Zum anderen würde die Entwicklungszeit gegenüber einem Prozessor oder heterogenen System immens steigen. Deshalb werden Lösungen von komplexen Systemen vorzugsweise als heterogene Systeme umgesetzt.

Ein weiterer wichtiger Faktor, der bei der Partitionierung eine große Rolle spielt, ist die Art der Verbindung der verschiedenen Systemkomponenten. Die Verbindungen, die z.B. durch PCI-Busse, USB, Ethernet oder RS-232 zustande kommen, besitzen verschiedene Eigenschaften bzgl. Datendurchsatz, Latenzzeit, Zuverlässigkeit und Komplexität. Erschwerend kommt noch hinzu, dass es zwischen zwei Systemkomponenten mehr als eine Verbindung geben kann, und zwar sowohl von demselben Typ als auch von unterschiedlichen Typen. Dies erhöht natürlich die Anzahl der Freiheitsgrade bei der Systempartitionierung und erschwert damit auch die Findung einer optimalen Lösung.

Eine Partitionierung kann nun automatisch oder manuell erfolgen. Die automatische Generierung von Partitionierungen werden im Bereich des Hardware-Software-Codesigns untersucht (vgl. Abschnitt 3.7). Aufgrund der Komplexität der automatischen Untersuchung der Randbedingungen soll die Partitionierung hier manuell erfolgen. Die drei Schritte der Generierung einer Partitionierung (Zerlegung des Systemalgorithmus, Abbildung der logischen Verbindungen, Überprüfung der Randbedingungen) bleiben dabei erhalten. Es besteht jedoch die Möglichkeit, bereits frühzeitig implizites Wissen und Erfahrung in den Partitionierungsprozess einfließen zu lassen, um die Anzahl der Partitionierungszyklen minimal zu halten. Die Überprüfung, ob eine Partitionierung den Randbedingungen genügt, erfolgt durch vollständige Implementierung des Gesamtsystems. An diesem kann dann durch Messung die Einhaltung der Randbedingungen überprüft werden. Sollten diese nicht erfüllt sein, muss eine neue Partitionierung gewählt und der Zyklus noch einmal durchlaufen werden. Damit dies in einem realen Entwicklungsprozess praktikabel ist, muss nach der Partitionierung die Erstellung der Implementierung der Zielprogramme möglichst automatisch erfolgen.

Wird ein gerichteter Graph als Grundlage der Systembeschreibung benutzt, stellt sich die Partitionierung als Abbildung der Knoten auf die verschiedenen Zielplattformen dar. Sollte ein Knoten hierarchisch aufgebaut sein, besteht die Möglichkeit, die verschiedenen Teilknoten auf verschiedene Plattformen zu verteilen. Die Zuordnung der Knoten zu bestimmten Plattfor-

men bestimmt die Zuordnung für die Kanten. Hier ergeben sich zwei Fälle. Zum einen können beide Knoten einer Kante auf derselben Zielplattform liegen. In diesem Fall wird auch die Kante komplett auf dieser Plattform erzeugt werden müssen. Im zweiten Fall liegen die beiden Knoten einer Kante auf verschiedenen Plattformen. Diese Kante muss durch spezielle Kommunikationsmittel repräsentiert werden. Dies geschieht durch das Hinzufügen von zwei besonderen Knoten auf jeder Plattform, so genannte Kommunikationsknoten. Für die Kommunikation muss dann eine der physikalischen Verbindungen zwischen den Plattformen genutzt werden. Damit es mehreren Kanten erlaubt ist, plattformübergreifend zu wirken, soll es möglich sein, über eine physikalische Verbindung mehrere logische Verbindungen aufzubauen. Um diese dann auf beiden Seiten der Kommunikation unterscheiden zu können, müssen sie mit einer Identifizierungsnummer (ID) versehen werden (vgl. Abbildung 4.5). Ein weiteres Attribut legt fest, welche physikalische Verbindung genutzt werden soll, falls im System mehrere davon möglich sind. Die Kombination aus Attributen und ID muss für eine Kante im System eindeutig sein. Damit ist es möglich, jede logische Verbindung einer Kante zuzuordnen.

4.3 Verbindungsbeschreibung

Für die automatische Erzeugung einer Systemlösung aus einer manuellen Partitionierung müssen die im Partitionierungsschritt auftretenden physikalischen Verbindungen zwischen den Plattformen einheitlich angesprochen werden. Nur so ist mit minimalem Aufwand ein Wechsel zwischen den verschiedenen Verbindungen möglich. Die physikalischen Verbindungen zwischen den Plattformen müssen bestimmte Bedingungen erfüllen. So muss es möglich sein, über eine physikalische Verbindung beliebig viele logische Verbindungen aufzubauen. Dies ist meistens durch die Einführung eines Protokolls möglich, welches z.B. das zusätzliche Attribut ID in den Datenstrom mit überträgt. Eine weitere Bedingung ist, dass die logischen Verbindungen als unidirektionale Verbindungen zwischen zwei Plattformen ausgeführt sind. Da die Datenübertragung auf logischer Ebene auch synchron geschehen soll, muss zumindest die physikalische Verbindung bidirektional sein. Andernfalls besäße der Empfänger nicht die Möglichkeit, den Sender über mögliche Datenverluste zu informieren.

Die unidirektionale Ausführung der logischen Verbindungen ermöglicht die Unterteilung der beiden Endpunkte eindeutig nach Sender und Empfänger. Die physikalischen Verbindungen lassen sich in verschiedene Gruppen einteilen. Zum einen unterscheiden sich die Verbindungen in der Art

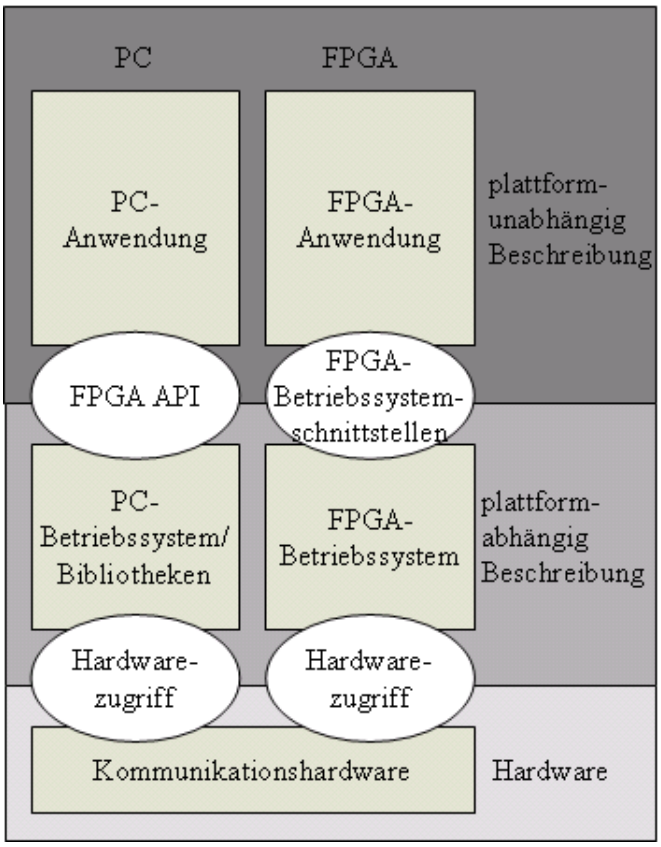

Abbildung 4.4: Abstraktion der Verbindungsbeschreibung mit Hilfe von Betriebssystemen für das Beispiel einer PC-FPGA-Verbindung

des Zugriffs auf das Übertragungsmedium. So gibt es Verbindungen, bei denen das Medium zwischen verschiedenen Instanzen aufgeteilt wird. Typische Vertreter dafür sind das Ethernet oder Wireless-LAN. Bei diesen Verbindungen tritt das Problem auf, dass durch die Aufteilung des Mediums geringere Bandbreiten und teilweise auch ein nicht-vorhersagbares zeitliches Verhalten auftreten kann. Dies führt dazu, dass die Bestimmung des Gesamtdurchsatzes des Systems unberechenbar wird. Diese Probleme treten bei Verbindungen mit ungeteiltem Medium, so genannten Peer-to-Peer-Verbindungen, wie z.B. RS-232, PCI-X, nicht auf.

Bei jedem Verbindungsaufbau ist es wichtig zu wissen, wer eine Kommunikation initialisiert. So gibt es Verbindungen, bei denen jeder Teilnehmer eine entsprechende Kommunikation veranlassen kann. Hier ist das Ethernet wieder ein typischer Vertreter. Andererseits gibt es auch Verbindungen, bei denen nur bestimmte Instanzen, so genannte Master, eine Kommunikation

initialisieren können. Ein Beispiel dafür ist die USB-Kommunikation. Die Instanzen, die dies nicht können, werden als Slave bezeichnet. Bei diesen Verbindungen gibt es nun noch die Unterscheidung, ob es nur einen oder beliebig viele Master auf dem Medium geben kann.

Sollte der Sender einer logischen Verbindung auch Master sein, stellt dies kein weiteres Problem dar. Wenn jedoch der Sender ein Slave auf dem Medium ist, muss der Master den Zustand des Senders ständig abfragen, was die Latenzzeit der Übertragung erhöht, oder aber es muss einen zusätzlichen Kommunikationsweg geben, der den Master über den Zustand des Senders informiert. Hierfür kann z.B. wie beim PCI-Protokoll eine Interruptleitung dienen.

Um die Verbindungen zwischen den Plattformen mit ihren unterschiedlichen Eigenschaften einheitlich ansprechen zu können, helfen die Benutzung eines Betriebssystems und die Definition von standardisierten Schnittstellen weiter. Sowohl für den PC als auch für die konfigurierbare Hardware werden Schnittstellen definiert, die einen einheitlichen Zugang zu den Transportwegen gestatten (Abbildung 4.4).

Für Prozessoren können die Schnittstellen in Form einer API[1] festgelegt werden. Durch die Einführung von Schnittstellen können die Programmbeschreibungen oder -modelle in plattformabhängige und plattformunabhängige Anteile zerlegt werden.

Eine Verbindung über standardisierte Schnittstellen ist nicht nur - wie aus Abbildung 4.4 hervorgeht - zwischen PC und FPGA, sondern auch zwischen PC und PC sowie FPGA und FPGA möglich. Bei der Kommunikation zwischen PC und PC bietet es sich jedoch an, auf bereits bestehende Kommunikationsschnittstellen, wie z.B. der Socketkommunikation, zurückzugreifen.

4.4 Simulation

Die Simulation spielt bei der Entwicklung von heterogenen Systemen eine wichtige Rolle. Eine Simulation ist dabei ein ausführbares Modell des Systems. Aufgrund der Vernachlässigung bestimmter Eigenschaften des Systems sind Simulationen gegenüber realen Systemen einfacher in der Ausführungskomplexität, billiger in den Ausführungskosten und kürzer in der Ausführungszeit.

Mit Hilfe einer Simulation werden zwei Ziele verfolgt. Zum einen kann überprüft werden, ob die Bedingungen an das System erfüllt werden. Dazu gehört in erster Linie zu überprüfen, ob das System auf der algorithmischen

[1] Application Programming Interface

Ebene das gewünschte Verhalten aufweist, d.h. ob die Reaktion des Systems auf Stimuli mit den Anforderungen übereinstimmt. Es ist jedoch auch möglich, das zeitliche Verhalten oder den Ressourcenverbrauch zu überprüfen. Die zweite wichtige Aufgabe einer Simulation besteht in der Unterstützung der Fehlersuche (*Debugging*). Mit Hilfe einer Simulation können innere Zustände bzw. die innere Dynamik des Systems untersucht werden, was in einem realen System nicht möglich bzw. mit erhöhtem Aufwand verbunden wäre.

Für ein heterogenes System ergeben sich verschiedene Möglichkeiten der Simulation. An erster Stelle steht dabei die Simulation des Gesamtsystems auf algorithmischer Ebene. Hier liegt der Schwerpunkt in der Untersuchung des Gesamtalgorithmus hinsichtlich der algorithmischen Systembedingungen. Vernachlässigt werden alle Aspekte der Partitionierung, des Datentransports, des Zeitverhaltens und des Ressourcenverbrauchs. Durch Verfeinerung der Simulation, d.h. auch durch Einbeziehung der Partitionierung und somit des Datentransportes zwischen den verschiedenen Teilkomponenten des Gesamtsystems, können immer mehr Randbedingungen des realen Systems in die Simulation miteinbezogen werden. Je stärker die Verfeinerung der Simulation ist, desto umfangreicher und komplexer wird sie jedoch. Dies bezieht sich sowohl auf die Durchführung der Simulation als auch auf die Auswertung der Simulationsergebnisse. Daher sollte die Anzahl der Simulationszyklen bei jeder Verfeinerung kleiner werden.

Neben der Simulation des Gesamtsystems auf verschiedenen Ebenen besteht die Möglichkeit, auch Teilkomponenten des System zu simulieren [JB03, Mat01]. Aufgrund der Standardisierung der Transportwege zwischen den verschiedenen Subsystemen können Simulationsmodelle für die Verbindungen definiert werden. Diese Simulationsmodule sind unabhängig von einem konkreten Projekt und können daher wiederverwendet werden. Sie hängen jedoch von der Zielplattform ab. So werden entsprechende Modelle für FPGA in VHDL und für Prozessoren in Form von C/C++ Funktionen benötigt.

Durch die Benutzung von Betriebssystemschnittstellen besteht weiterhin die Möglichkeit, eine Simulation mit einem stark vereinfachten Modell des Betriebssystems durchzuführen. Durch die standardisierten Schnittstellen wird außerdem garantiert, dass die Modelle des Betriebssystems auch wiederverwendet werden können.

4.5 Entwurf mit Hilfe von Betriebssystemen

Die hier benutzte Methode des Entwurfs für heterogene Systeme mit Hilfe von Betriebssystemen besteht aus den in Abbildung 4.5 gezeigten Schritten. Diese sollen im Weiteren näher erläutert werden.

Algorithmusentwicklung

An erster Stelle des Systementwurfs stehen die Algorithmusentwicklung und -beschreibung. Dazu wird, gerade wenn es um signalverarbeitende Systeme geht, in erster Linie eine Hochsprache wie C/C++ verwendet. Dies ermöglicht es, den Algorithmus einfach zu entwickeln, zu debuggen und zu testen. Die Entwicklungszyklen von der Änderung des Algorithmus bis zum Testen sind relativ kurz, da sie nur durch die Kompilation der Hochsprache bestimmt werden. Diese Zeit liegt im Bereich von wenigen Sekunden. Durch die Benutzung einer weit verbreiteten Sprache sind außerdem umfangreiche Bibliotheken verfügbar, sodass auch dadurch die Entwicklungszeit minimiert wird. Ein weiterer Vorteil einer Hochsprache ist, dass hier eine leichte Visualisierung in Form von Diagrammen und Bildern von Zwischenergebnissen möglich ist. Der Nachteil liegt in einer langen Ausführungszeit, d.h. auch einer langen Simulationszeit. Im Bereich der Signalverarbeitung kann die Simulationszeit um mehrere Größenordnungen länger dauern als bei der Ausführung durch das Gesamtsystem.

Signalflussgraph

Nachdem der Algorithmus in einer Hochsprache formuliert ist, muss im nachfolgenden Schritt eine Transformation in eine Systembeschreibungssprache durchgeführt werden. Wie bereits beschrieben, soll dies hier durch einen Signalflussgraphen geschehen. Der Graph besteht aus Knoten und gerichteten Kanten. Um diese in der Hochsprachenbeschreibung zu identifizieren, müssen Datenpfade extrahiert und komplexe Operationen ermittelt werden. Die komplexen Operationen werden dann zu Knoten und die zwischen den Operationen vermittelnden Datenpfade zu gerichteten Kanten. Bei der Wahl der Operationen kann der Entwickler selbst festlegen, wie detailliert die Zerlegung in Suboperationen sein soll. Eine sehr detaillierte und damit auf elementare Operationen zurückgehende Beschreibung hat den Vorteil, dass diese Operationen auf allen möglichen Zielplattformen realisierbar sind. Der Nachteil ist jedoch, dass durch die detaillierte Beschreibung im Signalflussgraphen die Freiheiten bei der Umsetzung einer komplexen Operation auf konkrete Zielplattformen eingeschränkt werden. Im Sinne einer optimalen Umsetzung des Algorithmus auf das Zielsystem sollten daher alle Operationen nur soweit zerlegt werden, bis es für sie entsprechende Realisierungen auf den verschiedenen Zielplattformen gibt.

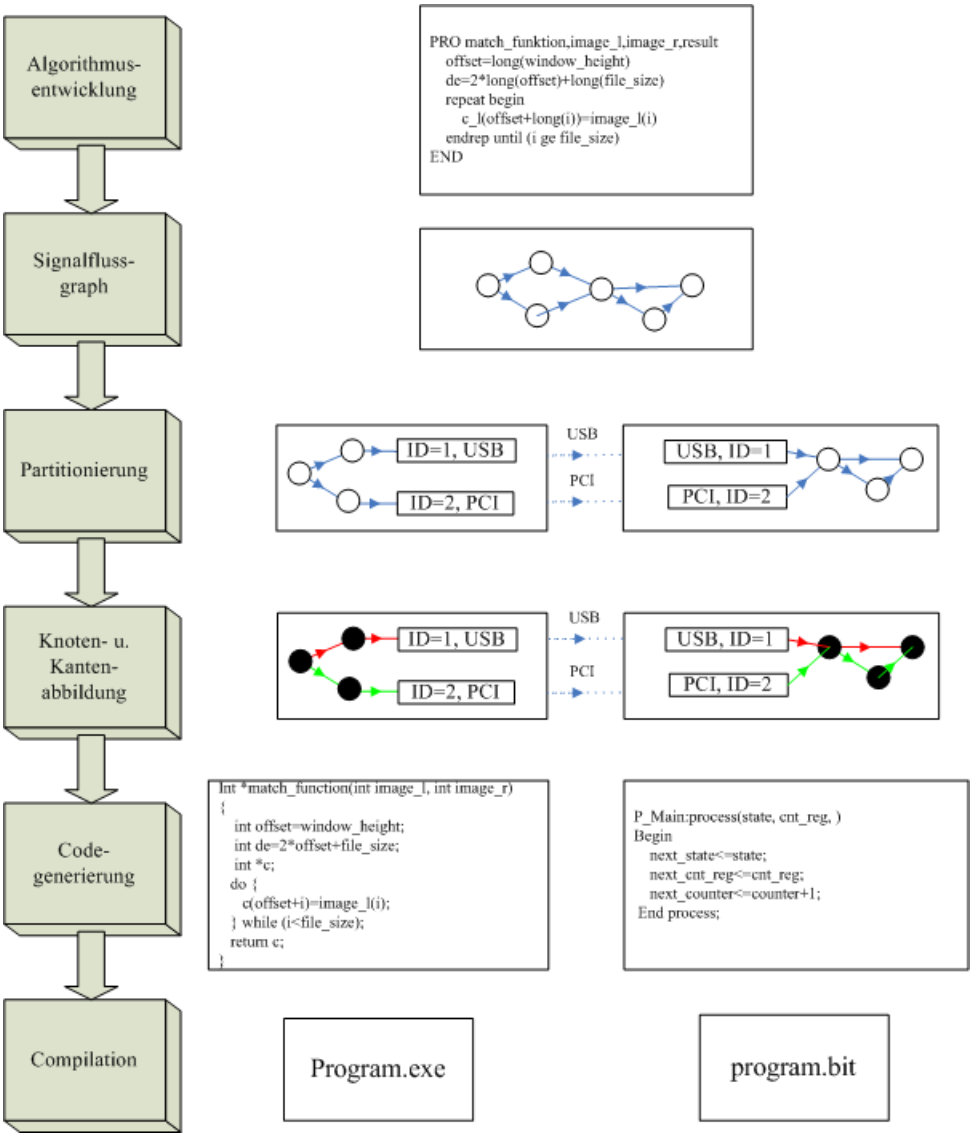

Abbildung 4.5: Entwurfsmethode für heterogene Systeme

Partitionierung

Der Signalflussgraph bildet die Grundlage des nachfolgenden manuellen Partitionierungsprozesses. Auf einen Graphen bezogen bedeutet die Partitionierung eine Abbildung der Knoten des Algorithmus auf die verschiedenen Zielplattformen des Gesamtsystems. Als mögliche Zielplattformen kommen hierbei nicht nur die verschiedenen Hardwaretypen des Zielsystems in Frage, d.h. Prozessor oder FPGA, sondern auch die bereits im vorhergehenden Abschnitt 4.4 erläuterten Simulationen der Teilsysteme.

Neben der Festlegung der Abbildung der Knoten auf die Plattformen muss definiert werden, wie die zwischen den Plattformen vermittelnden Kanten auf die vorhandenen Transportwege abgebildet werden. Dabei müssen sowohl die Art des Transportweges als auch eine system-eindeutige Identifizierung der logischen Verbindung über den Transportweg festgelegt werden. Die system-eindeutige Identifizierung kann dabei mit Hilfe eines inkrementellen Zählers umgesetzt werden. Die Art des Transportweges muss dagegen vom Entwickler auf Basis der vorhandenen Verbindungen zwischen den zwei Plattformen sowie des zu erwartenden Soll-Datendurchsatzes gewählt werden.

Am Ende des Partitionierungsprozesses ist das Gesamtsystem in voneinander getrennte Teilsysteme zerlegt. Diese können nun unabhängig voneinander in ein ausführbares Programm überführt werden.

Knoten- und Kantenabbildung

Nachdem für jede Plattform ein entsprechender Teilalgorithmus festgelegt wurde, muss als nächstes vom Entwickler angegeben werden, in welcher Art die Knoten und Kanten interpretiert und umgesetzt werden sollen. Dabei können alle Kanten auf einer Plattform vom gleichen Typ, aber auch von unterschiedlichen Typen sein. Die Wahl der Kantenrealisierung richtet sich dabei auch nach den gewünschten Datendurchsätzen, den gewünschten Latenzzeiten sowie dem gewünschten Ressourcenverbrauch. Außerdem müssen die Kantenrealisierungen so gewählt sein, dass diese mit der Knotenrealisierung korrespondieren. Für einen Knoten wird im einfachsten Fall nur eine Realisierung für diese Plattform existieren. Dadurch wird dann auch die Art der Kantenrealisierung festgelegt. Sollten mehrere Realisierungen für einen Knoten existieren, bedeutet das zusätzliche Freiheitsgrade für den Entwurf eines Systems.

Codegenerierung

Die in den vorhergehenden Abschnitten beschriebenen Schritte der Signalflussgraphbestimmung, der Partitionierung und der Knoten- und Kantenabbildung müssen von einem Entwickler manuell durchgeführt werden. Die nachfolgenden Schritte der Codegenerierung und Kompilation können jedoch automatisiert werden. Bei der Codegenerierung wird, ausgehend von den vorhergehenden Festlegungen der Teilalgorithmen und der Knoten- und Kantenrealisierungen, der Code für die entsprechende Zielplattform generiert. Diese Generation kann z.B. durch freeSp [Wei] auf Basis einer XML-Beschreibung erfolgen.

Kompilation

Das bei der Codegenerierung entstandene Programm kann mit einem Compiler in eine ausführbare Datei übersetzt werden. Für die zu verwendenden Prozessoren müssen dafür Bibliotheken bereitgestellt werden, welche die Realisierung der Knoten für den jeweiligen Prozessor beinhalten.

Für konfigurierbare Hardware wurde im Rahmen dieser Arbeit ein Strukturcompiler entwickelt, der ebenfalls Realisierungen der Knoten in Form von Bibliotheken zur Verfügung stellt.

Bei der Bewertung diese Methode, stellt sich die Frage, welche Schritte unternommen werden müssen, um sowohl neue Operatoren als auch neue Zielplattformen in die Methode zu integrieren.

Um einen neuen Operator, d.h. einen neuen Knoten, zu integrieren, müssen die Bibliotheken um eine oder mehrere Beschreibungen des Operators erweitert werden. Die Beschreibungen sind im Allgemeinen abhängig von den Zielplattformen und von den Kantendefinitionen. An dieser Stelle ermöglicht es die Benutzung eines Betriebssystems, die Anzahl der Beschreibungen zu verringern. Durch die Benutzung von Betriebssystemschnittstellen erfolgt die Beschreibung der Operatoren unabhängig von den Zielplattformen. Aufgrund der schlechten Umwandlung von Beschreibungen für Prozessoren in Beschreibungen für konfigurierbare Hardware müssen für einen neuen Operator mindesten zwei Beschreibungen erstellt werden. Aufgrund der vielseitigen Unterstützung wird als Beschreibungssprache für die Prozessoren die Sprache C/C++ gewählt. Als Beschreibungssprache für die konfigurierbare Hardware soll die Sprache VHDL dienen. Da die Sprache VHDL nicht die Voraussetzungen erfüllt, um ein Betriebssystemkonzept umzusetzen, wurde

im Rahmen dieser Arbeit ein entsprechendes Werkzeug entwickelt. Dieses wird im Kapitel 5 und 7 vorgestellt.

Die Beschreibungen eines Operators sind nicht nur von den Zielplattformen, sondern auch von den Kantendefinitionen abhängig. Um die Anzahl an Beschreibungen für einen Operator einzuschränken, muss somit auch die Anzahl an Kantendefinitionen eingeschränkt werden. Wie in Kapitel 6 gezeigt wird, werden im Weiteren drei Arten von Kantendefinition benutzt. Diese haben sich bisher als ausreichend erwiesen.

Um eine neue Plattform in diese Methode zu integrieren, muss für sie ein Betriebssystem entwickelt werden. In einem Betriebssystem muss für jede Betriebssystemschnittstelle eine Realisierung auf dieser Plattform existieren. Die Betriebssystemschnittstellen trennen damit plattformunabhängige Beschreibungen von den plattformabhängigen Beschreibungen (vgl. Abbildung 4.4). Das Betriebssystemkonzept stellt damit einen integralen Bestandteil der vorgestellten Methode des Entwurfs für heterogene System dar.

Kapitel 5

Betriebssystemkonzept

In diesem Kapitel werden die theoretischen Grundlagen eines Betriebssystemkonzepts für Hardwarebeschreibungssprachen wie VHDL hergeleitet. Dazu werden am Anfang die Ideen, die Voraussetzungen sowie die Vor- und Nachteile eines Betriebssystems beschrieben. In Anlehnung daran wird erläutert, inwiefern die Hardwarebeschreibungssprachen die notwendigen Voraussetzungen für die Umsetzung eines Betriebssystemkonzepts nicht erfüllen. Zur Lösung des Problems wird ein neuer Ansatz vorgestellt. Die Beschreibung erfolgt noch ohne den Strukturcompiler, der in Kapitel 7 beschrieben wird.

5.1 Softwarebetriebssysteme

Der Begriff des Betriebssystems ist aus der heutigen Softwareentwicklung nicht mehr wegzudenken. Wann immer eine Software entwickelt werden soll, wird dabei eine der vielen heute verfügbaren Betriebssystemlösungen benutzt. Es gibt nur noch wenige Anwendungsgebiete, in denen kein Betriebssystem benutzt wird. Aus Anwendersicht lassen sich Softwarebetriebssysteme in zwei Gruppen einteilen. Zur einen Gruppen gehören Softwarelösungen, in denen zwar ein Betriebssystem genutzt wird, dieses jedoch nicht Bestandteil des Programms ist. Das Programm selber wird vom Betriebssystem geladen und kann dann die Funktionen des Betriebssystems nutzen. Typische Vertreter für solche Betriebssysteme sind Linux und Windows. Zur anderen Gruppe zählen solche Betriebssysteme, in denen das Betriebssystem selbst Bestandteil des Programms wird. Sie werden vor allem bei eingebetteten Lösungen verwendet (z.B. RTEMS [On-98], eCos [Red03]).

Ein Betriebssystem ist dabei ebenfalls eine Software, welche die verschiedenen Betriebsmittel eines Rechners verwaltet und die Ausführung eines Programms ermöglicht. Zu den Betriebsmitteln eines Rechners gehören dabei die

verschiedenen Hardwarekomponenten, die mit einem Rechner verbunden sein können, wie z.B. Netzwerk oder Drucker, aber auch der Speicher des Rechners und der Prozessor. Eine weitere Aufgabe eines Betriebssystems besteht darin, die verschiedenen Betriebsmittel des Systems, die sich auf verschiedenen Plattformen ändern können, durch die Betriebssystemschnittstellen zu kapseln.

Für das Anwendungsprogramm stellt sich das Betriebssystem als eine Menge von Schnittstellen dar, über die bestimmte Funktionalitäten abgerufen werden können. Das Betriebssystem muss für jede Betriebssystemschnittstelle ein entsprechendes Modul zur Verfügung stellen. Für ein Betriebssystem spielen somit zwei Komponenten eine wichtige Rolle: die Betriebssystemschnittstellen und die Betriebssystemmodule. Dadurch, dass die Betriebssystemmodule für verschiedene Anwendungen benutzt werden können, findet eine Wiederverwendung von Code statt. Diese Wiederverwendung reduziert die Entwicklungszeit der Software. Auf der anderen Seite wird auch die Zuverlässigkeit erhöht, da die Betriebssystemmodule in einer Vielzahl von Anwendungen verwendet werden.

Ein weiterer Vorteil eines Betriebssystems liegt in der plattformunabhängigen Beschreibung der Anwendung. Verschiedene Beschreibungsarten sind dabei möglich. Eine besteht darin, das bereits kompilierte Programm als Beschreibung zu benutzen. Dies setzt jedoch voraus, dass die Plattformen, auf denen die Anwendung laufen soll, denselben Bytecode besitzen. Für ein Hardwarebetriebssystem ist dies eine zu starke Einschränkung. Deshalb soll diese Art der Beschreibung nicht weiter betrachtet werden. Eine weitere Möglichkeit der Beschreibung ist durch den Programmcode der Anwendung gegeben. Bevor Programmcode auf einem Prozessor ausgeführt werden kann, muss er mit einem Compiler übersetzt werden. Dadurch ist die Beschreibung der Anwendung unabhängig vom Prozessorbefehlssatz. Für konfigurierbare Hardware ist dies mit Hilfe von Hardwarebeschreibungssprachen und den Synthesecompilern ebenfalls möglich.

Zwischen der Anwendungsbeschreibung und der Betriebssystembeschreibung muss eine definierte Trennung bestehen. Sie wird über Betriebssystemschnittstellen realisiert. Sie trennen das Gesamtprogramm in eine plattformunabhängige (die Anwendung) und einen plattformabhängige Beschreibung (das Betriebssystem). Durch Standardisierung der Betriebssystemschnittstelle kann eine Anwendung auf eine Vielzahl von Plattformen realisiert werden.

Neben der Standardisierung der Betriebssystemschnittstellen vereinfacht auch die Nutzung standardisierter Bibliotheken den Entwurf. Des Weiteren ist natürlich auch die Programmiersprache selber standardisiert, um zu garantieren, dass verschiedene Compiler funktional identische Programme erzeugen.

Die Nutzung eines Betriebssystems ist jedoch mit gewissen Nachteilen verbunden. Die große Verbreitung von Betriebssystemen beweist aber, dass diese Nachteile nur noch in den seltensten Fällen die Erfüllung der Systembedingungen verhindern. Ein Nachteil liegt in der komplexen Struktur der Betriebssystemschnittstellen und der Betriebssystemmodule. Dies ist notwendig, um einen möglichst hohen Wiederverwendungsgrad zu erreichen. Die Komplexität erhöht jedoch den Ressourcenverbrauch des Gesamtssystems. Die hohe Leistungsfähigkeit und Integrationsdichte heutiger Systeme sind jedoch in der Lage, diese Nachteile zu kompensieren.

Nach [Tan92] kann ein Betriebssystems als virtuelle Maschine oder als Ressourcenmanager betrachtet werden. Aus beiden Betrachtungsweisen leiten sich bestimmte Bedingungen an die Programmiersprache des Betriebssystems ab. Wie noch gezeigt wird, erfüllt die Sprache VHDL diese Bedingungen nicht.

Bei der Betrachtung des Betriebssystems als virtuelle Maschine stehen seine Schnittstellen im Vordergrund. Damit sie die Beschreibung des Systems in einen plattformunabhängigen und einen plattformenabhängigen Anteil trennen können, muss das Konzept der Datenkapselung (*information hiding*) umsetzbar sein. Durch dieses Konzept werden die plattformabhängigen Implementierungsdetails des Betriebssystems vor der Anwendung verborgen.

Wird ein Betriebssystem als Ressourcenmanager betrachtet, steht die Verwaltung der Ressourcen im Vordergrund, d.h. die Kontrolle darüber, welche Instanz welche Ressourcen benutzt, oder die Bewilligung von Ressourcenanfragen und Vermittlung bei Ressourcenkonflikten. Damit ein Betriebssystem als Ressourcenmanager arbeiten kann, muss es in der Lage sein, gleichzeitig Kontrolle über mehrere gleichartige oder verschiedene Betriebssystemschnittstellen zu haben. In Softwarebetriebssystemen ist dies kein Problem, da Programme immer sequentiell ausgeführt werden und ein Programm unter Nutzung von globalen Variablen in der Lage ist, Informationen über Schnittstellen global und zwischen verschiedenen Aufrufen verfügbar zu machen.

5.2 Hardwarebetriebssystem

Auf FPGA-Plattformen gibt es zwei Arten von Komponenten, die sich ändern können und somit vom Betriebssystem erfasst werden müssen. Eine Komponente ist der FPGA-Baustein selber, die andere zusätzliche Hardware, die mit dem FPGA verbunden ist. Die Kapselung der verschiedenen Arten von FPGA-Bausteinen wird durch die Hardwarebeschreibungssprache und die verschiedenen FPGA-Werkzeuge ermöglicht. Da es jedoch auch möglich sein soll, interne Strukturen des FPGA durch externe Hardwarekomponen-

ten zu ersetzen, sollen die internen FPGA-Komponenten ebenfalls durch die Betriebssystemschnittstellen gekapselt werden. So können zum Beispiel je nach Wunsch und Größe für eine benötigte FIFO-Operation entweder der Block-RAM des FPGA oder ein extern vorhandenes SDRAM benutzt werden. Auch ein Hardwarebetriebssystem kann als virtuelle Maschine oder als Ressourcenmanager betrachtet werden. Beide Sichtweisen werden im Folgenden erläutert. Als Hardwarebeschreibungssprache wird hierbei die Sprache VHDL betrachtet. Für andere Hardwarebeschreibungssprachen, wie Verilog, ergeben sich aber die gleichen Probleme.

5.2.1 Hardwarebetriebssystem als virtuelle Maschine

Damit ein Betriebssystem als virtuelle Maschine betrachtet werden kann, müssen alle möglichen plattformabhängigen Beschreibungen des Betriebssystems verborgen sein, d.h. unabhängig von den für die Anwendungsbeschreibung sichtbaren Betriebssystemschnittstellen. Voraussetzung ist die Datenkapselung. Die Beschreibungssprache VHDL bietet hierfür kein Konzept, weil sie vom Standpunkt einer elektrischen Schnittstelle definiert wurde. Alle Signale eines VHDL-Modells, die mit externen Komponenten des FPGA-Bausteins verbunden sind, müssen Bestandteil der Schnittstelle des Modells sein.

Dies soll anhand eines Speichermoduls veranschaulicht werden. Für dieses Speichermodul soll es drei Umsetzungen für verschiedene Plattformen geben: als Block-RAM-Lösung, als SRAM-Lösung und als SDRAM-Lösung.

Das Anwendungsmodell benötigt für den Zugriff auf einen abstrakten Speicher die Signale `data`, `address` und `command`. Diese Signale stellen damit die Betriebssystemschnittstelle für einen abstrakten Speicher dar. Wie die folgende Tabelle zeigt, ist jedoch nur die Schnittstelle der Block-RAM-Lösung identisch mit der Betriebssystemschnittstelle des abstrakten Speichers.

	Block-RAM	SRAM	SDRAM

Signale für das Anwendungsmodell	`data`
	`address`
	`command`
		`data`
		`address`
		`command`
			`data`
			`address`
			`command`

zusätzliche Signale zum externen Hardwarezugriff		`extern_data`
		`extern_address`
		`extern_we`
			`extern_data`
			`extern_address`
			`extern_we`
			`extern_ras`
			`extern_cas`
			...

Aufgrund der unterschiedlichen Anbindungen der externen Speicher würden

die Schnittstellen für jede Art von Speicher anders aussehen. Damit kann die Art des physikalischen Speichers nicht durch eine Betriebssystemschnittstelle gekapselt werden.

Das hier aufgezeigt Problem tritt bei allen Betriebssystemschnittstellen auf, bei denen Plattformen existieren können, auf denen das entsprechende Betriebssystemmodul Zugriff auf externe Hardware benötigt. Ein Betriebssystemkonzept ist mit dieser Eigenschaft der Hardwarebeschreibungssprache nicht umsetzbar.

5.2.2 Hardwarebetriebssystem als Ressourcenmanager

Als Ressourcenverwalter muss ein Betriebssystem in der Lage sein, die Nutzung der Ressourcen durch Instanzen zu kontrollieren und zu überwachen, Ressourcenanfragen zu bewilligen bzw. abzulehnen und bei Ressourcenkonflikten zu vermitteln. Angewandt auf ein Hardwarebetriebssystem bedeutet dies hauptsächlich, den verschiedenen anfragenden Instanzen der Anwendung Ressourcen durch zeitliches oder räumliches Multiplexen zur Verfügung zu stellen. Zeitliches Multiplexen kommt vor allem bei Kommunikationsressourcen vor, wo über Zuweisung von Zeitfenstern jeder Instanz eine entsprechende Ressource zur Verfügung gestellt wird. Räumliches Multiplexen spielt vor allem bei den Ressourcen eine wichtige Rolle, die in großer Anzahl in dem System vorliegen, wie z.B. Speicherblöcke.

In Softwarebetriebssystemen wird dieses Problem durch die Benutzung von globalen Variablen gelöst, die von verschiedenen Betriebssystemmodulen aus sichtbar sind. In Hardwarebetriebssystemen auf Basis von VHDL ist dies nicht möglich. Es gibt keine Möglichkeit, Informationen ohne Signalzuweisungen über die Schnittstellen anderen Modulen zur Verfügung zu stellen. Mit der Verknüpfung Komponente-Architektur kann jedes Betriebssystemmodul nur eine Betriebssystemschnittstelle bedienen. Um trotzdem ein Hardwarebetriebssystem als Ressourcenmanager nutzen zu können, muss es eine Möglichkeit geben, mehrere, auch untereinander verschiedene Betriebssystemschnittstellen mit einem Betriebssystemmodul zu verbinden.

5.2.3 Lösungsansatz

Um die in den vorhergehenden Abschnitten 5.2.1 und 5.2.2 beschriebenen Probleme zu lösen, wird hier ein Lösungsansatz in Form eines Präcompilers vorgestellt. Der Präcompiler verbindet mit Hilfe verschiedener Transformationsschritte auf der Ebene der Hardwarebeschreibungssprache ein plattformunabhängiges Anwendungsprogramm mit Betriebssystemmodulen. Das Ergebnis der Transformation ist eine Beschreibung, die mit den typischen

FPGA-Werkzeugen weiterverarbeitet werden kann. Auf diese Weise können die von den Hardware-Herstellern mitgelieferten Synthesewerkzeuge benutzt werden, die optimal an die Hardware angepasst sind. Der Lösungsansatz wird dabei unabhängig von einer konkreten Hardwarebeschreibungssprache angegeben. Eine Realisierung des Präcompilers für die Sprache VHDL wird in Kapitel 7 vorgestellt.

Ausgangspunkt der Transformationen des Präcompilers ist die plattformunabhängige Beschreibung einer Anwendung. Die Plattformunabhängigkeit wird durch die Benutzung von Betriebssystemschnittstellen erreicht. Der Zugriff auf die Betriebssystemmodule, d.h. die Einbindung der Betriebssystemschnittstellen, soll sich an der Einbindung von Submodulen orientieren. Weiterhin wird eine Beschreibung des Betriebssystems benötigt. Diese Beschreibung soll in der gleichen Hardwarebeschreibungssprache vorliegen, wie die Beschreibung der Anwendung. Als Letztes muss der Entwickler noch festlegen, welche Betriebssystemschnittstelle mit welchem Betriebssystemmodul verknüpft werden sollen. Die Art der Festlegung ist an dieser Stelle noch beliebig. Im Strukturcompiler für VHDL wird dies mit Hilfe verschiedener Konfigurationsdateien geschehen. Innerhalb des Präcompilers finden dann folgende Transformationsschritte statt:

- Umleitung aller Signale der Betriebssystemschnittstellen des Anwendungsprogramms zur Schnittstelle des Toplevelmoduls der Anwendung

- Umleitung aller Signale der Betriebssystemschnittstellen der Betriebssystemmodule zur Schnittstelle des Toplevelmoduls der Anwendung

- Erzeugung eines neuen Toplevelmoduls, welches das modifizierte Anwendungsprogramm und alle benötigten modifizierten Betriebssystemmodule einbindet und Bündelung der umgeleiteten Signale der Betriebssystemschnittstellen und Zuweisung zu Betriebssystemmodulen

Diese einzelnen Schritte werden im Folgenden anhand eines einfachen abstrakten Beispiels erläutert. Die Beschreibung des Anwendungsprogramms sei durch die in Abbildung 5.1 dargestellte Struktur gegeben. Das Beispiel besteht aus einem Toplevelmodul (oberes Rechteck), das zwei weitere Module (untere Rechtecke) einbindet. Die Module bestehen aus einer Schnittstelle und einem Ausführungsteil. Die Schnittstellenanteile, bilden den oberen Rand der Rechtecke. In diesem Fall bestehen die Schnittstellen für die Submodule nur aus den Signalanteilen, die für die Kommunikation zwischen dem Toplevelmodul der Anwendung und den Submodulen benötigt werden (grüner Schnittstellenanteil). Die Schnittstelle des Toplevelmoduls der Anwendung

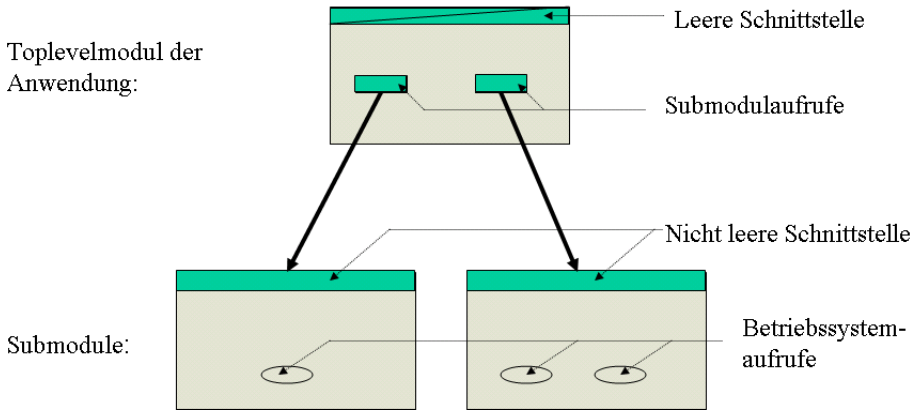

Abbildung 5.1: Struktur einer plattformunabhängigen Beschreibung einer Anwendung, bestehend aus einem Toplevel- und zwei Submodulen mit Betriebssystemaufrufen

ist dagegen leer. Im Ausführungsteil sind nur die Einbindungen von Submodulen (Rechtecke) und von Betriebssystemschnittstellen (Ellipsen) dargestellt. Andere Anweisungen werden durch die Transformationsschritte nicht verändert und wurden daher in der Darstellung weggelassen.

Eine plattformunabhängige Beschreibung eines Anwendungsprogramms ist dadurch gekennzeichnet, dass das Toplevelmodul eine leere Schnittstelle besitzt. Andernfalls würde die Plattformunabhängigkeit verletzt werden. Um keine neuen Sprachelemente in die Hardwarebeschreibungssprache einführen zu müssen, soll die Einbindung von Betriebssystemschnittstellen mit der gleichen Syntax erfolgen wie die Einbindung von Submodulen. Aufgrund dieser Festlegungen erfüllt die Beschreibung der Anwendung das Konzept der Datenkapselung. Zugriffe auf die externe Hardware werden erst bei den folgenden Transformationen aufgelöst.

Das Ziel der ersten Transformation besteht darin, auf alle Bestandteile der Betriebssystemschnittstellen von außen, d.h. über das Anwendungstoplevelmodul, zugreifen zu können. Dazu müssen alle Betriebssystemschnittstellen durch Signalzuweisungen ersetzt und diese in die Toplevelschnittstelle der Anwendung umgeleitet werden. Dieses ist in Abbildung 5.2 dargestellt. In dem Beispiel wurden die drei Betriebssystemschnittstellen in den beiden Submodulen durch Signalzuweisungen (Sechsecke) ersetzt. Dazu müssen den Schnittstellen der einzelnen Submodule und des Toplevelmoduls der Anwendung neue Signale hinzugefügt werden (blauer Anteil an den Schnittstellen). Die Typen der neuen Signale richten sich nach den Typen der Signale der Betriebssystemschnittstelle. Die neuen Signale werden mit den Signal-

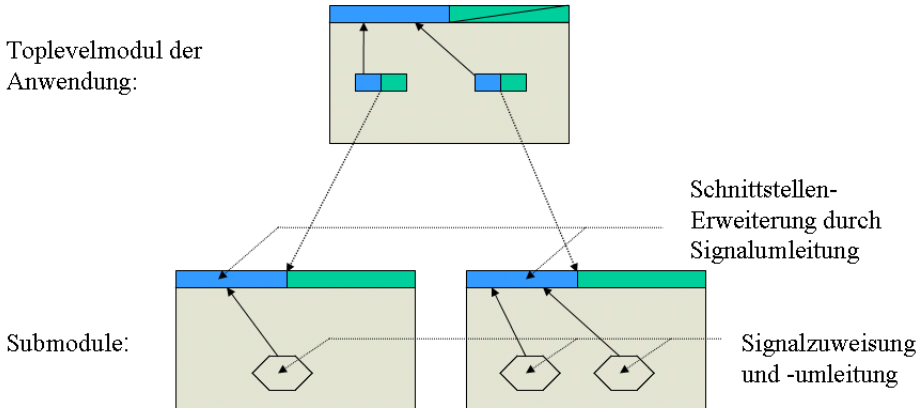

Abbildung 5.2: Ersetzung der Betriebssystemschnittstellen durch Signalzuweisungen und Umleitung der Signale zur Schnittstelle des Toplevelmoduls der Anwendung

ausdrücken verbunden, die in der Ursprungsbeschreibung mit den Signalen der Betriebssystemschnittstelle verknüpft waren. Aufgrund der strukturellen Beschreibung können sich diese Signalumleitungen durch eine Vielzahl von Modulen ziehen.

Im zweiten Transformationsschritt wird die im ersten Schritt ausgeführte Transformation auf die Betriebssystemmodule angewendet. Welche Betriebssystemmodule benötigt werden, wurde vom Entwickler durch die Abbildung der Betriebssystemschnittstellen zu Betriebssystemmodulen festgelegt. Der Schritt der Transformation der Betriebssystemmodule ist notwendig, da auch Betriebssystemmodule Betriebssystemschnittstellen enthalten dürfen. Damit wird ein hierarchischer Aufbau des Betriebssystems ermöglicht.

Für das gegebene Beispiel ist der zweite Transformationsschritt in Abbildung 5.3 dargestellt. Auf der rechten Seite befinden sich die bereits im ersten Schritt transformierten Module der Anwendung. Auf der linken Seite sind die transformierten Betriebssystemmodule dargestellt. Sowohl die Betriebssystemmodule als auch die Anwendung können Submodule enthalten. Im Gegensatz zu der Schnittstelle des Toplevelmoduls der Anwendung sind jedoch die Schnittstellen der obersten Betriebssystemmodule vor der Transformation nicht leer. In diesen Schnittstellen können zwei Anteile auftreten, die durch die Transformation nicht weiter verändert werden. Das ist zum einen der Anteil, der zur Verbindung mit den Betriebssystemschnittstellen benötigt wird (schwarzer Anteil der Schnittstelle der Betriebssystemmodule). Dieser Anteil darf nicht leer sein. Zum zweiten kann ein Schnittstellenanteil auftreten, der zum Zugriff auf die externe Hardware benötigt wird (gelber

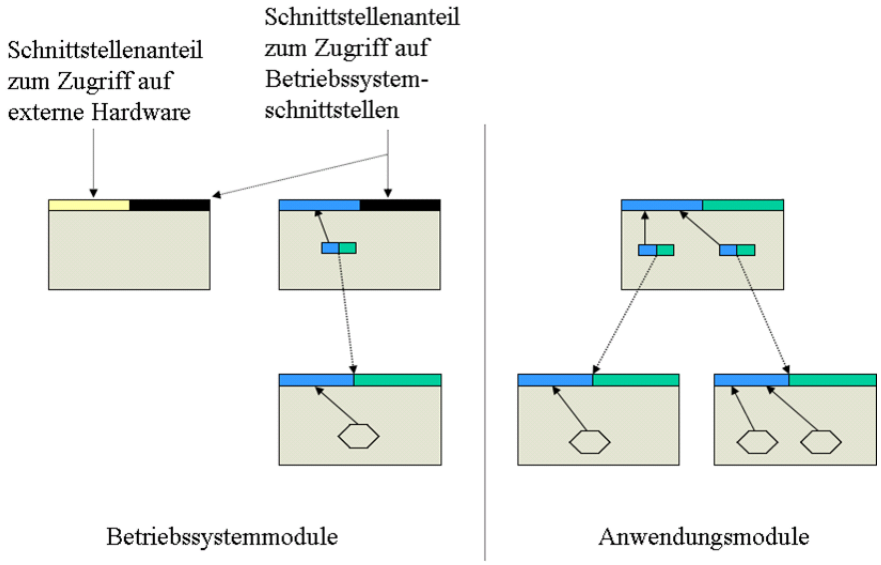

Abbildung 5.3: Ersetzung der Betriebssystemschnittstellen durch Signalzuweisungen und Umleitung der Signale zur Schnittstelle der Betriebssystemmodule

Schnittstellenanteil im linken Betriebssystemmodul).

Ein Betriebssystemmodul kann nun ebenfalls Betriebssystemschnittstellen enthalten (in Abbildung 5.3 rechte Betriebssystemmodul). Die daraus resultierenden neuen Betriebssystemmodule müssen dann ebenfalls der Umformung, d.h. der Signalumleitung, unterzogen werden. Dies wird solange fortgesetzt, bis alle Betriebssystemmodule, die benötigt werden, umgeformt sind.

In der Schnittstelle eines transformierten obersten Betriebssystemmoduls können somit drei Anteile auftreten:

- Anteile, die durch die Umleitung der eigenen Betriebssystemschnittstellen enstehen (blau),
- Anteile zur Verbindung mit den Betriebssystemschnittstellen von dem Anwendungsmodulen oder anderen Betriebssystemmodulen (schwarz),
- Anteile zum Zugriff auf die externe Hardware (gelb).

Nachdem alle Anwendungsmodule und alle Betriebssystemmodule transformiert wurden, muss als nächstes ein neues Toplevelmodul erstellt werden, das alle transformierten Module einbindet. Als Schnittstelle besitzt dieses

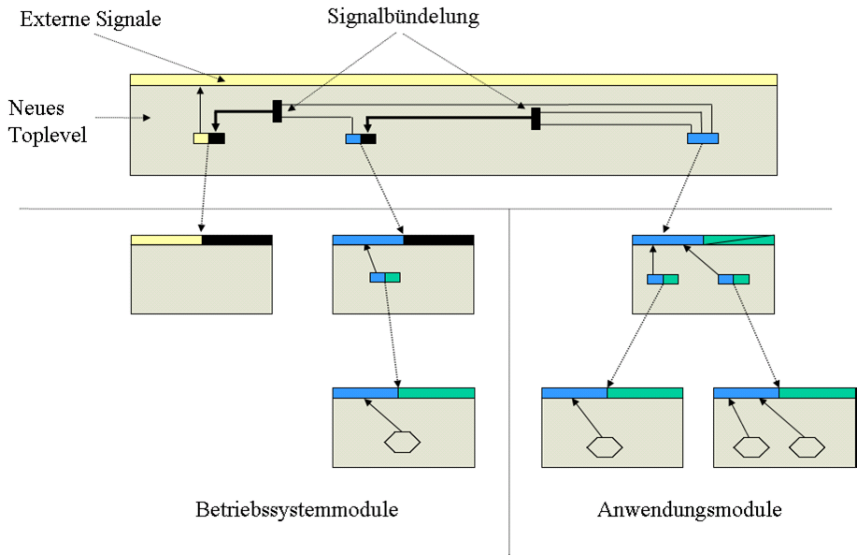

Abbildung 5.4: Hinzufügen eines neuen Toplevelmoduls, das die transformierten Betriebssystemmodule und Anwendungsmodule einbindet sowie die Signale der Betriebssystemschnittstellen bündelt

neue Toplevelmodul alle Bestandteile an Zugriffen auf externe Hardware der einzelnen Betriebssystemmodule. In dem Beispiel der Abbildung 5.4 gibt es nur ein Betriebssystemmodul (das linke), das solche Schnittstellenanteile besitzt. Innerhalb des Ausführungsteils des neuen Toplevelmoduls kommen nur Submodulaufrufe und Signalverknüpfungen vor. Als Submodule müssen das Anwendungstoplevel sowie alle obersten Betriebssystemmodule, die vorher transformiert wurden, eingebunden werden.

Um die Bedingung des Ressourcenmanagments zu erfüllen, kann es nun vorkommen, dass mehr als eine Betriebssystemschnittstelle mit einem Betriebssystemmodul verbunden ist. Um diese Fälle zu behandeln, muss eine Signalbündelung durchgeführt werden. Eine Signalbündelung bedeutet, dass die Signale der verschiedenen Aufrufe an Betriebssystemschnittstellen zu einem Bündel an Signalen verknüpft werden. Im einfachsten Fall sind die Bündel Felder der Signale der Betriebssystemschnittstelle. Die Größe des Feldes wird durch die Anzahl der Betriebssystemschnittstellen bestimmt, die mit dem Betriebssystemmodul verbunden werden sollen. Das so erzeugt Bündel kann dann dem entsprechenden Betriebssystemmodul übergeben werden.

In dem Beispiel der Abbildung 5.4 gibt es nur ein Anwendungstoplevelmodul. Die vorgestellten Transformationsschritte ermöglichen es jedoch auch, eine beliebige Anzahl an unabhängigen Anwendungstoplevelmodulen zu un-

terstützen. Dieses entspricht der Unterstützung von mehreren unabhängigen Programmen bei Softwarebetriebssystemen. Die Verknüpfung der Anwendungsprogramme erfolgt dann in dem neuen Toplevelmodul durch die Bündelung der Betriebssystemschnittstellen.

Die beschriebenen Transformationsschritte ermöglichen es, auf Basis einer Hardwarebeschreibungssprache eine Anwendung mit Hilfe eines Betriebssystems zu entwickeln. Die Anwendung wird dabei so beschrieben, dass sie sowohl den Aspekt der Datenkapselung als auch die Möglichkeit des Ressourcenmanagements unterstützt.

Bevor im Kapitel 7 eine Realisierung des Präcompilers für die Sprache VHDL vorgestellt wird, werden im nachfolgenden Kapitel 6 verschiedene Arten von Kantenrealisierungen definiert. Diese Definitionen werden für das Verständnis der Beispiele für den Strukturcompiler benötigt.

Kapitel 6

Kommunikation

In diesem Kapitel werden die unterschiedlichen Arten von Kantenrealisierungen auf dem FPGA und dem PC vorgestellt. Diese werden zum Verständnis des Beispiels im nachfolgenden Kapitel benötigt. Des Weiteren wird diskutiert, wie diese Protokolle mit unterschiedlichen physikalischen Verbindungen zwischen zwei Plattformen realisiert werden können und welche Einschränkungen und Probleme es dabei geben kann.

Zur Realisierung der Kanten in den Algorithmen wurden drei Kommunikationsarten definiert:

- Channel-Kommunikation: asynchroner Transfer
- SChannel-Kommunikation: synchroner Einzelworttransfer
- Pipe-Kommunikation: synchroner Blocktransfer

Alle diese Kommunikationsarten dienen zur Übertragung von Daten. Die Daten besitzen auf dem FPGA den Datentyp `std_logic_vector` mit einer beliebigen Breite. Auf einem Prozessor werden sie auf Datentypen mit einer entsprechenden Bytelänge abgebildet. Die interne Struktur der Daten spielt für das Protokoll keine Rolle. Es findet somit keine Typisierung der Daten statt. Die Schnittstellen für die Module und das Betriebssystem sollen so entworfen werden, dass sie aus diesen Kommunikationsarten zusammengesetzt sind. Das garantiert eine leichte Verständlichkeit der Schnittstelle sowie eine einfache Verknüpfung der Module untereinander.

Die Synchronität der SChannel- und Pipe-Kommunikation bezieht sich auf den Datenstrom des Senders und des Empfängers. Im Falle eines synchronen Transfers dürfen keine Datenworte des Datenstromes verloren gehen bzw. neue hinzukommen. Bei einem asynchronen Transfer dürfen ebenfalls auf Empfängerseite keine neuen Datenworte hinzukommen. Durch die Art

der Übertragung ist es jedoch erlaubt, dass Daten des Senders verloren gehen können.

Aufgrund ihrer unterschiedlichen Eigenschaften besitzen die Arten von Kantenrealisierungen verschiedene Einsatzgebiete. Mit Hilfe der Channel-Kommunikation können Parameter von Algorithmen übermittelt werden, wie z.B. Filterparameter oder Threshold-Werte. Diese Art von Kommunikation wird dann benutzt, wenn eine Änderung der Werte relativ selten vorkommt. Mit Hilfe der SChannel-Kommunikation können Daten übermittelt werden, die zum einen nicht verloren gehen dürfen und zum anderen untereinander nicht in einem logischen Zusammenhang auftreten, wie z.B. die Daten einer RS-232-Schnittstelle. Sollten die Daten noch zusätzlich in einem logischen Zusammenhang auftreten, z.B. als Zeile eines Bildes oder als komplettes Bild, so bietet die Pipe-Kommunikation die Möglichkeit, die Daten in so genannten Frames einzupacken. Dies garantiert zusätzlich einen möglichst hohen Datendurchsatz durch Ausnutzung spezieller physikalischer Kommunikationsarten, wie z.B. DMA[1].

6.1 Protokolle auf dem FPGA

Wie bereits in Abbildung 4.5 dargestellt, müssen zur Verbindung des FPGA mit anderen Plattformen, wie Prozessoren oder anderen FPGA, allen Kommunikationskanälen spezielle zusätzliche Attribute in Form von Generics übergeben werden. Dazu zählt die Breite der Daten sowie eine ID, die zur Identifikation der Gegenstelle dient. Bei der Benutzung der Kanäle im FPGA zwischen verschiedenen Modulen entfallen diese beiden Generics.

Channel-Kommunikation

Die Channel-Kommunikation stellt die einfachste der Kommunikationsarten dar. Es gibt zwischen Sender und Empfänger nur ein Signal, welches das Datenwort überträgt. Aufgrund der fehlenden zusätzlichen Signale fehlt jede Synchronität zwischen Sender und Empfänger. Der Empfänger bestimmt selber, wie schnell er das Signal abtastet und auswertet. Ein Beispielcode für die Darstellung der Verbindung zwischen Sender und Empfänger ist gegeben durch:

```
-- Daten Breite ist 18 bit
```

[1] Direct Memory Access

```
    signal channel_data: std_logic_vector(17 downto 0);

    component sender
        port (
            -- Channel Kommunikation
            data: out std_logic_vector(17 downto 0);

            -- weitere Signale
        );
    end component;

    component receiver
        port (
            -- Channel Kommunikation
            data: in std_logic_vector(17 downto 0);

            -- weitere Signale
        );
    end component;

begin
    I_sender: sender
        port map (
            -- channel output
            data => channel_data

            -- weitere Signalzuweisungen
        );

    I_receiver: receiver
        port map (
            -- channel input
            data => channel_data

            -- weitere Signalzuweisungen
        );
end;
```

SChannel-Kommunikation

Die nächste Stufe der Kommunikation stellen SChannels dar. Hier findet eine synchrone Datenübertragung statt. Da die Zeitdomänen von Sender und Empfänger unterschiedlich sein können, muss hier eine Anpassung der Daten

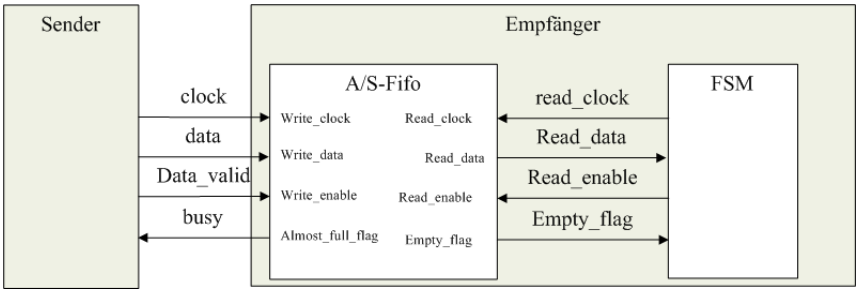

Abbildung 6.1: Struktur eines SChannel-Empfängers

an die Zeitdomäne des Empfängers stattfinden. Die Synchronität bedeutet aber unter anderem auch, dass vom Protokoll garantiert wird, dass kein Datenwort des Senders bei der Übertragung verloren gehen kann.

Alle Signale der SChannel-Schnittstelle beziehen sich auf den Takt, der durch das Signal `clock` vom Sender vorgegeben wird. Das Signal `data_valid` zeigt an, dass mit der nächsten steigenden Taktflanke ein gültiges Wort in `data` anliegt. Mit Hilfe des Signals `busy` ist der Empfänger in der Lage, den Transfer von Daten zu unterbrechen. Um die Zustandsmaschinen auf Empfänger- und Senderseite möglichst einfach und als Mooreautomat zu realisieren, darf der Sender bei aktivem `busy` Signal noch ein Datenwort schicken. Die Vorteile und Nachteile eines Mooreautomaten wurden bereits in Abschnitt 2.2.6 diskutiert.

Um das Kommunikationsprotokoll in VHDL umzusetzen, ist es angebracht, zwischen Sender und Empfänger eine asynchrone FIFO-Operation einzubauen. Damit kann eine Synchronität zwischen den verschiedenen Taktdomänen hergestellt werden. Andererseits ermöglicht eine FIFO-Operation auch die Umsetzung des Verhaltens des `busy` Signals mit Hilfe des Signals `almost_full_flag`[2]. In Abbildung 6.1 ist eine entsprechende strukturelle Beschreibung der Verknüpfung der Elemente angegeben. Anhang A beschreibt die Umsetzung der Struktur in VHDL. Für eine Anwendung ergibt sich folgende Verknüpfung zwischen Sender und Empfänger:

```
type valid_type is (valid, invalid);

-- Daten Breite ist 18 bit
signal schannel_clock: std_logic;
signal schannel_data: std_logic_vector(17 downto 0);
```

[2]Flag ist aktiv, wenn FIFO-Puffer voll ist oder nur noch ein Datenwort gespeichert werden kann

```
    signal schannel_data_valid: valid_type
    signal schannel_busy: valid_type;

    component sender
        port (
            -- SChannel Kommunikation
            schannel_clock: out std_logic;
            schannel_data: out std_logic_vector(17 downto 0);
            schannel_data_valid: out valid_type;
            schannel_busy: in valid_type

            -- weitere Signale
        );
    end component;

    component receiver
        port (
            -- SChannel Kommunikation
            schannel_clock: in std_logic;
            schannel_data: in std_logic_vector(17 downto 0);
            schannel_data_valid: in valid_type;
            schannel_busy: out valid_type

            -- weitere Signale
        );
    end component;

begin
    I_sender: sender
        port map (
            schannel_clock      =>  schannel_clock,
            schannel_data       =>  schannel_data,
            schannel_data_valid =>  schannel_data_valid,
            schannel_busy       =>  schannel_busy

            -- weitere Signalzuweisungen
        );

    I_receiver: receiver
        port map (
            schannel_clock      =>  schannel_clock,
            schannel_data       =>  schannel_data,
            schannel_data_valid =>  schannel_data_valid,
```

```
            schannel_busy       =>  schannel_busy

            -- weitere Signalzuweisungen
        );
end;
```

Pipe-Kommunikation

Die Pipe-Kommunikation baut auf der SChannel-Kommunikation auf. Zusätzlich werden die Datenworte zu so genannten Frames zusammengefasst. Frames sind Folgen von Daten, die außerdem noch eine 32-bit-Frame-Number besitzen. Die Anzahl der Daten pro Frame kann dabei in jedem Frame, selbst über denselben Kommunikationskanal, unterschiedlich sein. Zusammengefasst werden die Frames mit Hilfe der Signale `request` und `confirm`. Nur wenn beide Signal `valid` sind, ist es erlaubt, Daten über die integrierte SChannel-Kommunikation zu senden. Das Signal `frame_number` darf nur dann vom Sender verändert werden, wenn beide Signale `request` und `confirm` `invalid` sind, d.h. die Kommunikation sich im Grundzustand befindet. Durch Setzen von `request` auf `valid` gibt der Sender zu verstehen, dass er ein Frame senden möchte. Wenn der Empfänger in der Lage ist, ein neues Frame zu empfangen, setzt er als Antwort `confirm` auf `valid`. Sobald der Sender diese Änderung sieht, kann er Daten nach dem SChannel-Protokoll über die integrierte SChannel-Signale verschicken. Wenn das Frame zu Ende ist, muss der Sender zuerst das Signal `request` auf `invalid` setzen. Danach muss der Sender warten, bis der Empfänger als Bestätigung der kompletten Abarbeitung des Frames `confirm` ebenfalls auf `invalid` gesetzt hat. Dann befinden sich Sender und Empfänger wieder im Grundzustand, sodass nach obigem Schema ein neues Frame gesendet werden kann. Da sich Sender und Empfänger in unterschiedlichen Taktdomänen aufhalten können, müssen die Signale `request` und `confirm` vor einer Auswertung durch die Gegenstelle unter Ausnutzung eines Registers an die entsprechende neue Taktdomäne angepasst werden. Ein Beispiel der Verknüpfung von Sender und Empfänger ist gegeben durch:

```
type valid_type is (valid, invalid);
type Pipe_Framenumber_Type is std_logic_vector(31 downto 0);

-- Daten Breite ist 18 bit
signal pipe_clock       :   std_logic;
signal pipe_request     :   valid_type
signal pipe_confirm     :   valid_type
signal pipe_frame_number:   Pipe_Framenumber_Type
```

```
    signal pipe_data        :   std_logic_vector(17 downto 0);
    signal pipe_data_valid  :   valid_type
    signal pipe_busy        :   valid_type;

    component sender
        port (
            -- Pipe Kommunikation
            pipe_clock          : out std_logic;
            pipe_request        : out valid_type;
            pipe_confirm        : in  valid_type;
            pipe_frame_number   : out Pipe_Framenumber_Type;
            pipe_data           : out std_logic_vector(17 downto 0);
            pipe_data_valid     : out valid_type;
            pipe_busy           : in  valid_type

            -- weitere Signale
        );
    end component;

    component receiver
        port (
            -- Pipe communication
            pipe_clock          : in  std_logic;
            pipe_request        : in  valid_type;
            pipe_confirm        : out valid_type;
            pipe_frame_number   : in  Pipe_Framenumber_Type;
            pipe_data           : in  std_logic_vector(17 downto 0);
            pipe_data_valid     : in  valid_type;
            pipe_busy           : out valid_type

            -- weitere Signale
        );
    end component;

begin
    I_sender: sender
        port map (
            pipe_clock          =>  pipe_clock,
            pipe_request        =>  pipe_request,
            pipe_confirm        =>  pipe_confirm,
            pipe_frame_number   =>  pipe_frame_number,
            pipe_data           =>  pipe_data,
            pipe_data_valid     =>  pipe_data_valid,
```

```
            pipe_busy           =>  pipe_busy

            -- weitere Signalzuweisungen
        );

    I_receiver: receiver
        port map (
            pipe_clock          =>  pipe_clock,
            pipe_request        =>  pipe_request,
            pipe_confirm        =>  pipe_confirm,
            pipe_frame_number   =>  pipe_frame_number,
            pipe_data           =>  pipe_data,
            pipe_data_valid     =>  pipe_data_valid,
            pipe_busy           =>  pipe_busy

            -- weitere Signalzuweisungen
        );
end;
```

6.2 Protokolle auf dem Prozessor

Für alle drei hier vorgestellten Kommunikationsarten muss es auch eine entsprechende Realisierung für einen Prozessor geben. Da der Schwerpunkt dieser Arbeit auf konfigurierbarer Hardware liegt, soll hier nur kurz theoretisch vorgestellt werden, wie eine entsprechende Realisierung der Protokolle aussehen könnte. Dabei spielt natürlich auch eine Rolle, wie Operatoren auf dem Prozessor dargestellt werden, d.h. ob sie beispielsweise in einem eigenen Prozess oder Thread existieren.

Zur Kommunikation des PC mit dem FPGA wurde eine API[3] definiert (vgl. Abbildung 4.4). Diese API wurde für die drei hier vorgestellten Kommunikationsarten für das Betriebssystem Windows als DLL implementiert. Die Beschreibung der Funktionen der API befindet sich in [Kru05].

Channel-Kommunikation

Das einfachste Modell für eine Realisierung dieser Kommunikation auf einem Prozessor besteht aus einem Speicherbereich, auf den der Sender schreibend und der Empfänger lesend zugreift. Da beide Zugriffe asynchron verlaufen, besteht nicht die Notwendigkeit, zusätzliche Mechanismen zur Synchronisation vorzusehen. Ein Problem stellt natürlich die Breite des Channels dar.

[3] Application Programming Interface

Es tritt auf, wenn Lese- und Schreibzugriff nicht atomar erfolgen und Sender und Empfänger in unterschiedlichen Prozessen existieren. Die Unterbrechung der Schreiboperation kann dann in einem solchen Moment erfolgen, in dem ein Teil des Speicherbereichs bereits den neuen Wert, der restliche Teil aber den alten Wert enthält. Wenn der Empfänger in diesem Moment lesend auf den Speicher zugreift, liest er einen inkonsistenten Wert. Damit dies nicht passiert, müssen bei einer solchen Datenbreite trotz der Asynchronität Mittel der Interprozesskommunikation benutzt werden, z.B. Semaphoren, die ein solches Verhalten verhindern.

SChannel-Kommunikation

Ähnlich wie bei der Channel-Kommunikation besteht der Kern dieses Protokolls auf einem Prozessor aus einem gemeinsamen Speicherbereich, auf den sowohl der Sender als auch der Empfänger zugreifen können. Die Größe dieses Speicherbereichs richtet sich nach der Breite des SChannels. Damit Sender und Empfänger synchron arbeiten, müssen hier zusätzliche Mechanismen benutzt werden. Zur Umsetzung der Synchronität werden zwei Ereignisobjekte benötigt, eines für den Sender und eines für den Empfänger. Wenn der Sender ein neues Datenwort zu übertragen hat, kopiert er dieses in den gemeinsamen Speicher und informiert den Empfänger durch Auslösung des Empfänger-Ereignisses, dass ein neues Wort bereit steht. Danach wartet der Sender auf das Sender-Ereignis, das ausdrückt, dass der Empfänger das Wort abgearbeitet hat und bereit ist, neue Daten zu übernehmen. Der Empfänger auf der anderen Seite, muss dagegen nur auf das Empfänger-Ereignis warten, dann das Datenwort im gemeinsamen Speicher abarbeiten und den Sender durch Auslösung des Sender-Ereignisses über die erfolgreiche Abarbeitung informieren.

Pipe-Kommunikation

Die Pipe-Kommunikation erfolgt ähnlich der SChannel-Kommunikation. Anstatt jedoch die Worte einzeln zu transferieren, wird ein Block an Worten übertragen. Damit reduziert sich die Übertragungszeit und erhöht sich der Datendurchsatz. Die Größe des Blockes und damit des gemeinsamen Speichers richtet sich nach der Größe der Frames. Wenn diese vorher nicht bekannt ist, muss die Möglichkeit bestehen, die Frames in Blöcke zu zerlegen und diese hintereinander zu transferieren. Der Transfer eines Blockes erfolgt mit dem gleichen Mechanismus wie bei der SChannel-Kommunikation. Zusätzlich muss jedoch noch die Frame-Number sowie der Status des Frames übergeben werden. Zum Status gehört z.B., ob das Frame mit dem aktuel-

len Block komplett übertragen wurde. Auf diese Weise können auch große Frames mit hinreichender Geschwindigkeit übertragen werden.

6.3 Realisierung der Protokolle

Die in dem vorhergehenden Abschnitt vorgestellten Realisierungen der Kanten müssen auch für die Kommunikation zwischen den Plattformen umgesetzt werden. Diese Kommunikation muss auf die vorhandenen physikalischen Verbindungen aufbauen. Die verschiedenen Arten der Verbindungen wurden bereits in Abschnitt 4.3 diskutiert. Hier soll nun dargestellt werden, welche Möglichkeiten es gibt, die definierten Kommunikationsarten auf die physikalischen Verbindungen abzubilden.

Channel-Kommunikation

Die einfachste Art, die Channel-Kommunikation zwischen zwei Plattformen eines heterogenen Systems zu realisieren, besteht darin, als physikalische Verbindung ein entsprechend breites Datensignal zwischen den Komponenten zu benutzen. Dies kann zum Beispiel bei der Verbindung von zwei FPGA benutzt werden. Der Nachteil ist, dass über die physikalische Verbindung jeweils nur eine logische Verbindung aufgebaut werden kann. Außerdem ist die Breite der logischen Channel-Verbindung durch die Breite der physikalischen Verbindung begrenzt. Daher wird diese Lösung nur in den seltensten Fällen zum Einsatz kommen.

Wird jedoch ein zusätzliches Protokoll über die physikalische Verbindung aufgebaut, um eine beliebige Breite und eine beliebige Anzahl an Channel-Verbindungen zu unterstützen, gibt es prinzipiell drei Übertragungsarten:

- der Sender gibt die Abtastung vor, mit welcher der aktuelle Channel-Wert übertragen wird
- der Empfänger gibt die Abtastung vor, mit welcher der Channel-Wert übertragen wird
- jede Änderung des Channel-Werts beim Sender wird unmittelbar übertragen

Jede Übertragungsart hat Vor- und Nachteile. Bei der zuerst genannten Übertragung liegt der Vorteil darin, dass eine Übertragung von Daten bei der physikalischen Verbindung nur in eine Richtung verläuft. Damit ist der Datendurchsatz relativ gering. Dem gegenüber steht jedoch der Nachteil, dass nicht der Empfänger entscheiden kann, wann die Abtastung des Channel-Wertes

erfolgt. Dies ist aber bei der zweiten Übertragungsart möglich. Hier legt der Empfänger die Abtastung fest. Dafür wird jedoch eine größere Bandbreite auf dem physikalischen Medium benötigt, weil der Empfänger die Anfrage zur Abtastung noch zusätzlich übertragen muss. Die dritte Art der Übertragung hat den Vorteil, dass jede Änderung des Channel-Wertes zum Empfänger übertragen wird. Der Nachteil liegt jedoch darin, dass, wenn die Anzahl der Änderungen im Vergleich zur Bandbreite der physikalischen Verbindung sehr groß ist, Änderungen verloren gehen können bzw. die physikalische Verbindung blockiert wird.

Die zweite und dritte Art der Kommunikation wird bei den Beispielen in Abschnitt 8.2 für die PCI, USB und Ethernetverbindung verwendet.

SChannel-Kommunikation

Bei der Realisierung der SChannel-Kommunikation über eine physikalische Verbindung ist vor allem die Synchronität der Datenübertragung wichtig. So muss garantiert werden, dass kein Datenwort der logischen Verbindungen verloren gehen kann. Des Weiteren muss aber auch garantiert werden, dass bei Blockade einer logischen Verbindung durch den Empfänger nicht auch alle anderen logischen Verbindungen blockiert sind. Nur so ist es möglich, die logischen Verbindungen als unabhängig voneinander zu betrachten.

Aus diesen Gründen ist es notwendig, vor der Übertragung von SChannel-Daten zu überprüfen, ob der entsprechende SChannel beim Empfänger nicht blockiert ist. Weil diese Übertragung zusätzlichen Kommunikationsaufwand bedeutet, ist es hier angebracht, für jeden SChannel auf Empfängerseite einen Puffer zur Verfügung zu stellen. Durch Auslesen des Füllzustandes des Puffers durch den Sender, ist dieser in der Lage, mehrere SChannel-Werte als Block zu übertragen, ohne dass Worte verloren gehen. Die Größe des Puffers richtet sich nach der Zielplattform. Auf einem Prozessor ist es möglich, ohne größere Schwierigkeiten für jede SChannel-Verbindung einen mehrere Kilo- oder Megabyte umfassenden Puffer anzulegen. Auf einem FPGA muss man dagegen mit Block-RAM oder Registern auskommen, sodass hier die Größe der Puffer eher im Bereich von einige 100 Bytes liegen wird.

Eine wichtige Rolle spielt auch, wer einen Datentransfer einleiten darf. Wenn es für die physikalische Verbindung nur einen Master gibt, wie z.B. für eine PCI-Kommunikation, so muss bei einem FPGA-zu-PC-SChannel-Transfer der Master, d.h. der PC, über den Transfer informiert werden. Beim PCI-Bus wird dafür ein Interrupt benutzt. Wenn dieser gesetzt ist, muss der Master die Daten beim Sender, d.h. dem FPGA, abholen und in Puffern auf dem PC ablegen. Dies erfolgt solange, bis der Interrupt durch den FPGA zurückgenommen wird. Eine Voraussetzung für diese Art des Transfers ist,

dass die Einzelzugriffe auf dem Bus eine sehr kurze Latenzzeit haben. Nur so sind hinreichend hohe Transferraten für die SChannel-Kommunikation zu bewerkstelligen.

Eine andere Art von Transfer liegt z.B. beim Ethernet vor. Hier kann jede Instanz Datenpakete verschicken. Damit entfällt das Informieren eines Masters und der SChannel-Sender kann die Daten direkt an den SChannel-Empfänger verschicken. Aufgrund der blockorientierten Übertragung (Ethernetframes sind bis zu 1518 Bytes groß) ist es bei diesen Transfers umso wichtiger, die Übertragungen in Paketen zu bündeln. Nur so kann die Nettoübertragungsrate näherungsweise die Bruttoübertragungsrate erreichen.

Pipe-Kommunikation

Prinzipiell gelten für die Umsetzung der Pipe-Kommunikation auf physikalische Verbindungen dieselben Betrachtungen wie bei der SChannel-Kommunikation. Um jedoch größere Datenraten zu erzielen, müssen die Daten in Blöcken übertragen werden.

Bei der PCI-Kommunikation können dafür DMA-Transfers benutzt werden. Dies setzt jedoch voraus, dass die Daten auf Senderseite zwischengespeichert werden. Die Größe des dafür benötigten Puffers darf nicht zu klein gewählt sein, da ein DMA-Transfer mit einem erhöhten Verwaltungsaufwand verbunden ist. Daher ist es angebracht, bei PCI-Verbindungen den FPGA mit zusätzlichem externen Speicher zur Aufnahme der Blöcke der Pipe-Frames auszustatten.

Ein weiterer Unterschied zur SChannel-Kommunikation besteht darin, dass hier zusätzliche Informationen zum Frame einer Pipe übertragen werden müssen. Zu diesen Informationen gehört, z.B. die Frame-Number, aber auch, ob ein Frame komplett übertragen wurde oder noch Teile des Frames zu übertragen sind.

Bei einer Umsetzung der Pipe-Kommunikation auf eine physikalische Verbindung, über die jeder Teilnehmer Daten übertragen darf, wird dieser zusätzliche Speicher nicht benötigt. Die Daten der Pipe, inklusive des Status des Frames, können direkt in das Frame der Übertragung, z.B. Ethernetframe, transferiert werden.

Die hier vorgestellten Kommunikationsarten werden für das Verständnis der Beispiele für den Strukturcompiler benötigt. Dieser wird im nachfolgenden Kapitel als eine Realisierung des Präcompilers für die Sprache VHDL vorgestellt.

Kapitel 7

Strukturcompiler

In diesem Kapitel wird die Realisierung des in Kapitel 5 vorgestellten Präcompilers für die Sprache VHDL vorgestellt. Aufgrund der Funktionsweise wird dieses Werkzeug auch Strukturcompiler genannt. Die Arbeitsweise des Strukturcompilers wird anhand des einfachen Beispiels eines Multiplexers demonstriert.

7.1 Beispielprojekt

Der Strukturcompiler ist ein Programm, das die in Kapitel 5 gezeigten Transformationsschritte für die Sprache VHDL umsetzt. Diese Schritte sind notwendig, um ein Betriebssystemkonzept für die Sprache VHDL zu ermöglichen.

Zur Einführung in die Funktionsweise soll ein kleines Beispielprojekt dienen, anhand dessen die Eigenschaften des Strukturcompilers demonstriert werden. Das Beispiel eines 2-Kanal-Multiplexers wurde wegen seiner Einfachheit sowie seiner Verwendung in zahlreichen VHDL-Lehrbüchern [Ash98, SR03] gewählt. Die Breite der Datenkanäle beträgt 8 bit.

Der Signalflussgraph dieses Beispiels ist in Abbildung 7.1 gezeigt. Die Knoten für den Datenkanal 0 bzw. 1 sowie die Kanalauswahl sind Eingabeknoten. Der Multiplexer führt dann die Operation des Multiplexens aus. Das Ergebnis dieser Operation wird dem Ausgabeknoten Multiplexkanal übergeben.

Der in Abbildung 7.1 gegebene Signalflussgraph soll auf zwei Systemen realisiert werden. Die Zerlegung des Graphen ist dabei für beide Systeme gleich (Abbildung 7.2). Die erste zu erzeugende Beschreibung des Systems soll eine Simulation der Operationen eines FPGA sein. Nachdem an der Simulation die Funktionsweise des Multiplexers überprüft wurde, soll eine zweite

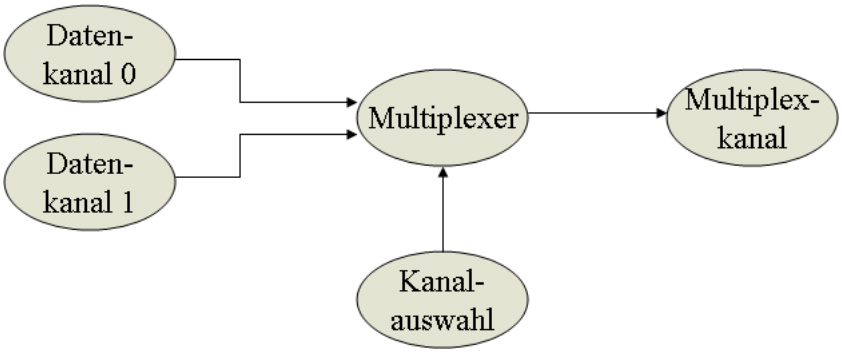

Abbildung 7.1: Signalflussgraph des Multiplexers

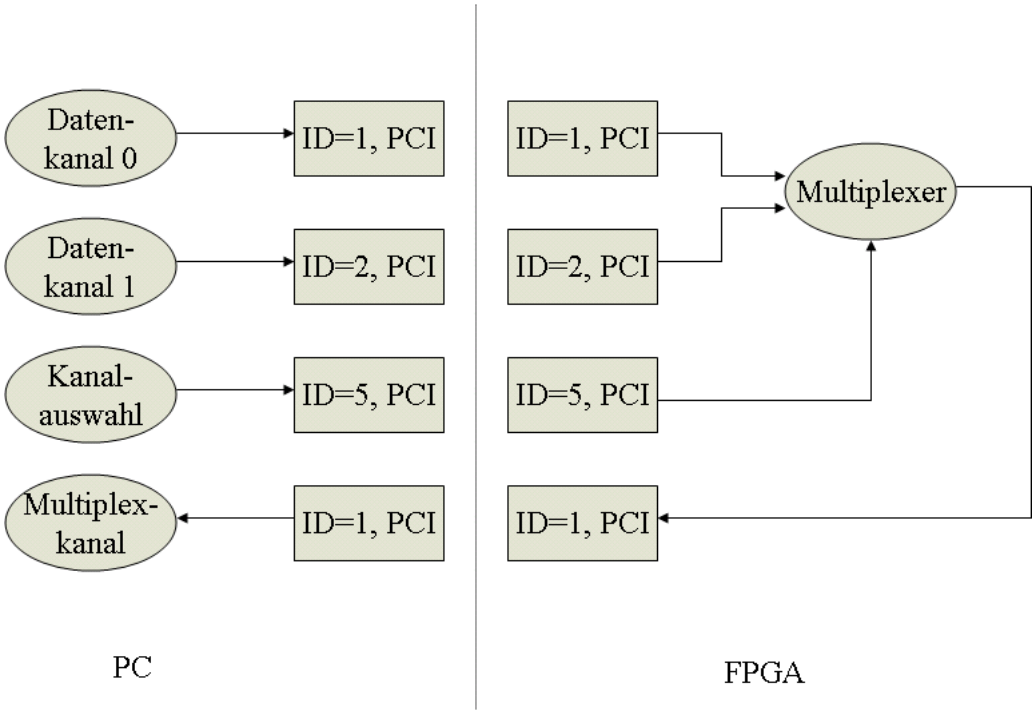

Abbildung 7.2: Partitionierung des Multiplexers

Beschreibung für eine FPGA-Karte erzeugt werden. Die FPGA-Karte wird im Abschnitt 8.1 vorgestellt. Die Ein- bzw. Ausgabe der Werte der verschiedenen Ein- bzw. Ausgabeknoten seitens des PC erfolgt über eine graphische Benutzeroberfläche.

Ausgehend von dem partitionierten Signalflussgraphen wird das VHDL-Programm entwickelt, das die Betriebssystemfunktionalität benutzt. Der Name dieses Programms ist `Mux.vhdl`. In dieser Arbeit werden alle VHDL-Quellen, die Betriebssystemschnittstellen benutzen, d.h. mit dem Strukturcompiler übersetzt werden müssen, mit der Endung vhdl versehen. Die Ergebnisdateien, die durch den Strukturcompiler erzeugt werden, besitzen die Endung vhd.

Wie bereits in Abschnitt 5.2.3 vorgestellt, besitzt ein VHDL-Anwendungsprogramm eine leere Toplevelschnittstelle:

```
1   entity Mux is
2   end;
```

Der Name dieser Anwendung ist `Mux`. Dieser Name wird später für die Konfiguration des Strukturcompilers benötigt. Eine passende Architektur zu dieser Schnittstelle bindet ein Standardpaket aus der IEEE-Bibliothek ein. Hinzu kommt noch ein Paket (Zeile 6), das die Definition der Kommunikationsarten zur Verfügung stellt.

```
4   library IEEE;
5   use IEEE.std_logic_1164.all;
6   use work.basic_package.all;
```

Innerhalb des Deklarationsblockes des Anwendungsprogramms treten die verschiedenen Komponenten zur Kommunikation mit dem PC auf. Für das Multiplexerbeispiel wurde die Channel-Kommunikation gewählt. Dafür werden zwei Betriebssystemschnittstellen benötigt, eine für die Eingabe vom PC zum FPGA (channel_input) und eine für die Ausgabe vom FPGA zum PC (channel_output). Wie bereits erwähnt, besteht die Channel-Kommunikation aus nur einem Signal `data`, ohne zusätzliche Synchronisationssignale. Dieses Signal kann aber verschiedene Breiten (Generic `WIDTH`) besitzen. Außerdem besitzen die Schnittstellen noch das Generic `ID` zur Zuordnung der logischen Kanäle zwischen den verschiedenen Komponenten des Systems, d.h. dem PC und dem FPGA. Bei der Komponente channel_input tritt zusätzlich ein Generic `INIT` auf. Dieses gibt den Initialisierungswert für das Datensignal an. Solange keine Änderung von der Gegenstelle gefordert wird, bleibt dieser Wert erhalten.

```
8   architecture Mux_arch of Mux is
9
```

```
10      component channel_input
11          generic (
12              WIDTH: CHANNEL_WIDTH_TYPE;
13              ID: CHANNEL_ID_TYPE;
14              INIT: integer
15          );
16          port (
17              data : out std_logic_vector(WIDTH-1 downto 0)
18          );
19      end component;
20
21      component channel_output
22          generic (
23              WIDTH: CHANNEL_WIDTH_TYPE;
24              ID: CHANNEL_ID_TYPE
25          );
26          port (
27              data : in std_logic_vector(WIDTH-1 downto 0)
28          );
29      end component;
```

Neben den Komponenten müssen auch die Signale für die Verbindung der Komponenten mit der Multiplexerfunktion deklariert werden. Diese Signale entsprechen den Kanten der in Abbildung 7.2 gewählten Partitionierung.

```
31      signal input1: std_logic_vector(7 downto 0);
32      signal input2: std_logic_vector(7 downto 0);
33      signal output: std_logic_vector(7 downto 0);
34      signal sel: std_logic_vector(0 downto 0);
```

Im Ausführungsteil der Architektur werden als erstes alle vier Verbindungen zum PC instanziiert, d.h. die zwei Dateneingabekanäle der Breite 8 bit, den Datenausgabekanal der Breite 8 bit sowie den Auswahlkanal mit der Breite 1 bit.

```
36  begin
37      I_input1: channel_input
38          generic map (
39              WIDTH       => 8,
40              ID          => 1,
41              INIT        => 0
42          )
43          port map (
44              data        =>  input1
45          );
```

```
46
47      I_input2: channel_input
48          generic map (
49              WIDTH       => 8,
50              ID          => 2,
51              INIT        => 0
52          )
53          port map (
54              data        =>  input2
55          );
56
57      I_output: channel_output
58          generic map (
59              WIDTH       => 8,
60              ID          => 1
61          )
62          port map (
63              data        =>  output
64          );
65
66      I_sel: channel_input
67          generic map (
68              WIDTH       => 1,
69              ID          => 5,
70              INIT        => 0
71          )
72          port map (
73              data        =>  sel
74          );
```

Je nach Art des Auswahlsignals muss das Ausgangssignal mit dem Dateneingabekanal 1 oder 2 verbunden werden. Theoretisch müsste diese Funktionalität in ein eigenes Submodul ausgelagert werden. Dieses Submodul könnte dann in anderen Anwendungen wiederverwendet werden. Aufgrund der Einfachheit der Funktionalität sowie der Reduzierung der Komplexität dieses Beispiels wurde aber auf ein zusätzliches Modul verzichtet.

```
76      output<=input1 when sel="0" else input2;
77
78  end;
```

Anhang B enthält das gesamte Programm. Dieses Beispielprogramm soll nun für verschiedene Plattformen übersetzt werden. Zur Ansteuerung des Strukturcompilers wird dafür neben dem Anwendungsprogramm noch eine

so genannte Projektkonfigurationsdatei benötigt. Diese legt die Quelldateien, Ausgabedateien sowie die verschiedenen Realisierungen fest. Die Realisierungen unterscheiden sich in der unterschiedlichen Art der Anbindung der Betriebssystemmodule zu den Betriebssystemschnittstellen. Der Name der Datei ist *project.cfg*.

Der erste Teil der Festlegung in der Projektkonfigurationsdatei betrifft alle Realisierungen des Beispiels. Dazu muss zum einen definiert werden, welche Anwendungsdateien überhaupt zum Projekt gehören (Zeile 1). In diesem Beispiel gehört nur die Datei *Mux.vhdl* dazu. Innerhalb dieser Dateienliste kann eine beliebige Anzahl an Modulen vorkommen, die sich auch untereinander aufrufen. Damit der Strukturcompiler weiß, welches das Anwendungstoplevelmodul ist, wird dieses über das Schlüsselwort `TOPLEVEL NAME` festgelegt (Zeile 2). Hier können auch mehrere Module angegeben werden. In solch einem Fall werden mehrere Anwendungen in dem Gesamtprogramm verbunden.

```
1   SOURCE FILES      =   Mux.vhdl;
2   TOPLEVEL NAME     =   Mux;
```

Für das durch die Transformation erzeugte neue Toplevelmodul (vgl. Abbildung 5.4) müssen die Namen der Schnittstelle und der Architektur definiert werden (Zeile 3 und 4). Der Name für die Schnittstelle wird als Einstiegspunkt für die Simulation oder die Synthese benötigt.

```
4   MAIN ENTITY NAME  = main_entity;
5   MAIN ARCHITECTURE NAME = main_architecture;
```

Nach den allgemeinen Einstellungen folgen die Definitionen für die einzelnen Realisierungen. Als erstes folgt die Definition für die FPGA-Karte namens ois3000.

```
7   ois3000
8   {
9           OUTPUT FILE         = ois3000/Mux.vhd;
10          UCF FILE            = ois3000/Mux.ucf;
11
12          channel_input       = ois_pci_channel_input;
13          channel_output      = ois_pci_channel_output;
14          ois_card            = ois3000_card;
15  };
```

Der Name ist dabei beliebig und sollte die Zielplattform wiedergeben. Innerhalb der Definition muss als erstes der Name für die VHDL-Ausgabedatei angegeben werden. In diese Datei werden alle Module ausgegeben, die für die

Erstellung der Simulation oder Synthese benötigt werden. Sollte die Realisierung eine Beschreibung für eine Hardware sein, so muss zusätzlich noch der Name einer UCF-Datei angegeben werden. Diese enthält zusätzliche Bedingung für die Synthese, wie z.B. die Zuordnung der externen Signale des FPGA zu den Pins oder Zeitbedingungen an die Signallaufzeiten. Für eine Simulation wird diese Datei nicht benötigt.

Nach diesen allgemeinen Einstellungen folgt eine Festlegung der Zuordnung der Betriebssystemschnittstellen zu den Betriebssystemmodulen. Für das Multiplexerbeispiel müssen hier die zwei Schnittstellen der Channel-Eingabe und Channel-Ausgabe zugewiesen werden. Die Betriebssystemmodule lauten `ois_pci_channel_input` bzw. `ois_pci_channel_output`. Diese verweisen dabei auf Einträge in eine Datenbank, wodurch der Strukturcompiler weiß, dass die Channel-Ein- bzw. Channel-Ausgabeschnittstellen für eine OIS-FPGA-Karte über die PCI-Schnittstelle realisiert werden soll. Da es außerdem verschiedene OIS-Karten gibt, muss angegeben werden, welche Art von OIS-Karte vorliegt (Zeile 14). In diesem Fall soll die Beschreibung für die OIS3000-Karte erfolgen.

Eine zweite Realisierung soll eine Simulation des Programms durchführen. Ihre Definition ist:

```
17  sim
18  {
19      OUTPUT FILE         = sim/Mux.vhd;
20
21      channel_input       = sim_channel_input;
22      channel_output      = sim_channel_output;
23  };
```

Genauso wie bei der vorhergehenden Realisierung wird hier die Ausgabedatei festgelegt. Da es sich jedoch um eine Simulation handelt, darf das Schlüsselwort für die UCF-Datei nicht erscheinen. Danach folgen Zuordnungen der Betriebssystemschnittstellen zu Betriebssystemsimulationsmodule für die beiden Channel-Kommunikationen.

Nachdem das Anwendungsprogramm und die Realisierungen definiert wurden, besteht der nächste Schritt darin, das Beispielprojekt zu simulieren. Dazu muss als erstes der Strukturcompiler aufgerufen werden, um die VHD-Datei, in der das Anwendungsprogramm mit den Betriebssystemmodulen verknüpft ist, zu erstellen. Der Aufruf erfolgt durch (siehe auch 7.2.1):

```
prjx_v11.exe -m sim -os_dir w:\Version1\FPGA_OS
```

Neben der Angabe der Realisierung (Schalter `-m`), die mit dem Namen in der Projektkonfigurationsdatei übereinstimmen muss, wird mit dem Schalter

`-os_dir` das Verzeichnis der Betriebssystemmodule festlegt. Die Ausgabe des Strukturcompilers dazu ist:

```
Parsing file: Mux.vhdl
Parsing file: ../FPGA_OS/interface/channel/channel_input_if.vhdl
Parsing file: ../FPGA_OS/package/basic_package.vhdl
Parsing file: ../FPGA_OS/interface/channel/channel_output_if.vhdl
Parsing file: ../FPGA_OS/arch/channel_input/sim_channel_input.vhdl
Parsing file: ../FPGA_OS/package/basic_arch_package.vhdl
Parsing file: ../FPGA_OS/package/command_file_handling.vhdl
Parsing file: ../FPGA_OS/package/std_arch_package.vhdl
Parsing file: ../FPGA_OS/interface/control_interfaces/clock_if.vhdl
Parsing file: ../FPGA_OS/arch/control_interfaces/sim_clock.vhdl
Parsing file: ../FPGA_OS/arch/channel_output/sim_channel_output.vhdl
```

Die Ausgabe gibt an, welche Dateien vom Strukturcompiler geparst und somit in die Ergebnis-VHDL-Datei übernommen wurden. Die Ergebnisdatei des Strukturcompilers wird, wie in der Projektkonfigurationsdatei angegeben, im Verzeichnis `sim` erzeugt. Diese Datei `Mux.vhd` kann mit einem Simulationsprogramm simuliert werden. Dafür wird jedoch noch eine Beschreibung des Verhaltens der Channel-Schnittstellen benötigt. Diese Beschreibung erfolgt, wie bei allen Kommunikationsschnittstellen, über Kommandodateien. Für die Channel-Eingabe-Schnittstelle lautet diese Datei `channel_input_command.txt` und besitzt den Inhalt:

```
NOP 3
WR   0001 12
NOP 2
WR   0001 34
NOP 4
WR   0001 56
NOP 2
WR   0002 FE
NOP 3
WR   0002 DC
NOP 2
WR   0005 1
NOP 3
WR   0002 BA
NOP 2
WR   0002 98
NOP 2
WR   0002 76
NOP 2
```

Diese Datei besteht aus einer Folge von Anweisungen, wie sich die Schnittstelle verhalten soll. Einerseits gibt es den Befehl `NOP`, der eine Verzögerung bewirkt, andererseits den Befehl `WR`, der den Eingabewert des Channels mit der entsprechenden ID (erster Parameter) neu setzt. Das gegebene Beispiel bewirkt, dass zuerst die Werte 0x12, 0x34 und 0x56 auf dem Datenkanal 1 ausgegeben werden, dann die Werte 0xFE und 0xDC auf dem Datenkanal 2. Da die ganze Zeit der Multiplexer auf den ersten Datenkanal geschaltet ist, sollte die Ausgabe mit den Eingabewerten des Datenkanals 1 übereinstimmen. Dann wird der Multiplexer auf den zweiten Kanal umgeschaltet und die Werte 0xBA, 0x98 und 0x76 ausgegeben. Diese sollten jetzt auf dem Ausgabekanal erscheinen.

Ähnlich wie für die Channel-Eingabe-Schnittstelle gibt es auch für die Channel-Ausgabe-Schnittstelle eine Kommandodatei. In dieser wird definiert, wann und welche Channels ausgelesen werden sollen. Die entsprechende Datei lautet `channel_output_command.txt`. Das Ergebnis wird in die Datei `channel_output_result.log` ausgegeben. In dem hier betrachteten Fall besteht die Kommandodatei nur aus Leseoperation vom Ausgabekanal (`RD 0001`). Das Ergebnis lautet damit:

```
RD       0001 value = 0x00
RD       0001 value = 0x00
RD       0001 value = 0x00
RD       0001 value = 0x12
RD       0001 value = 0x12
RD       0001 value = 0x34
RD       0001 value = 0x34
RD       0001 value = 0x34
RD       0001 value = 0x34
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0x56
RD       0001 value = 0xDC
RD       0001 value = 0xDC
RD       0001 value = 0xDC
RD       0001 value = 0xDC
RD       0001 value = 0xBA
RD       0001 value = 0xBA
RD       0001 value = 0x98
RD       0001 value = 0x98
```

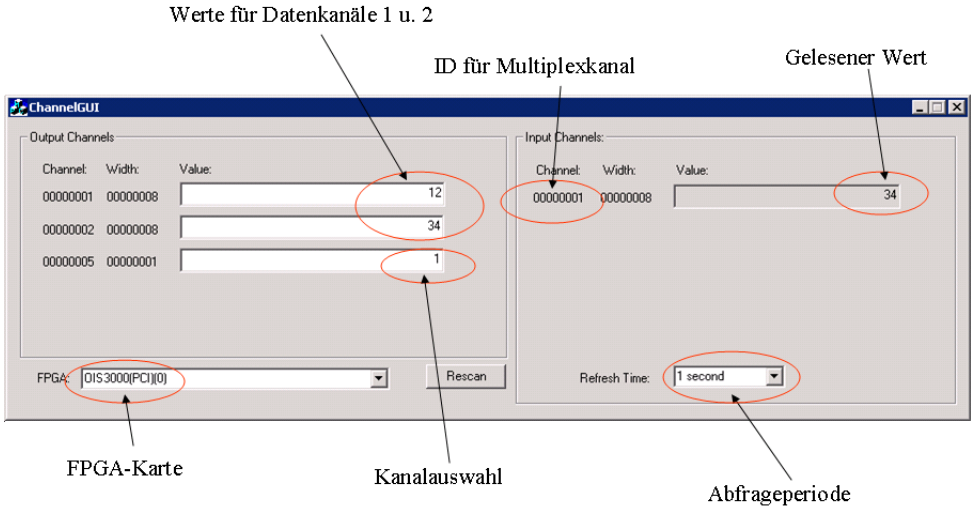

Abbildung 7.3: GUI für die Channel-Kommunikation des Multiplexerbeispiels

```
RD        0001 value = 0x98
RD        0001 value = 0x76
```

Die Ergebnisdatei zeigt das gewünschte Verhalten des Multiplexers. Deshalb soll als nächstes eine Realisierung für eine existierende FPGA-Karte erzeugt werden. Durch Aufruf des Kommandos:

```
prjx_v11.exe -m ois3000 -os_dir w:\Version1\FPGA_OS
```

werden in dem Unterverzeichnis `ois3000` zwei Dateien, die VHDL-Datei und die UCF-Datei erstellt. Mit Hilfe dieser und der Kenntnis des FPGA-Typen, in diesem Fall ein Xilinx VirtexII-3000, kann eine Synthese durchgeführt werden, die eine BIT-Datei zur Programmierung des FPGA erzeugt. Um das Beispiel des Multiplexers auch unter realen Systembedingungen zu testen, wird für den PC noch Software benötigt, die den PC-Teil der Partitionierung in Abbildung 7.2 ausführt. Da dies ein immer wiederkehrendes Problem bei der Benutzung der FPGA-Karten ist, wurden hierfür verschiedene GUI[1] entwickelt, mit denen die verschiedenen Kommunikationskanäle stimuliert bzw. visualisiert werden können. Für das Multiplexerbeispiel ist die GUI für die Channel-Kommunikation in Abbildung 7.3 dargestellt. Durch Veränderung der Werte der Datenkanäle und der Kanalauswahl ist eine manuelle Verifikation der Funktionsweise des Multiplexerbeispiels möglich.

[1] Graphical User Interface

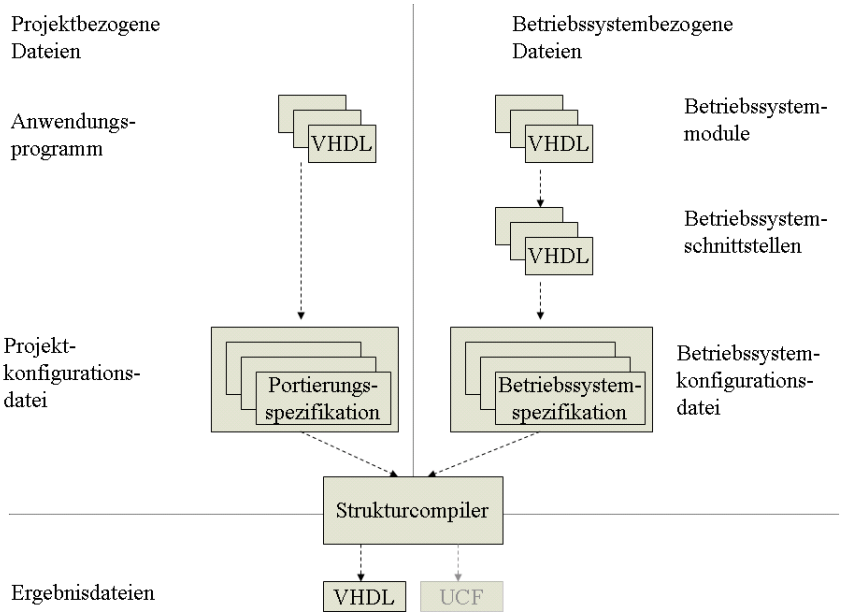

Abbildung 7.4: Komponenten des Strukturcompilers

7.2 Konfiguration des Strukturcompilers

Die Arbeit des Strukturcompilers wird über verschiedene Dateien konfiguriert. Die Dateien, die aus der Sicht des Anwenders wichtig sind, wurden bereits im vorhergehenden Abschnitt eingeführt. Dazu zählen das Anwendungsprogramm in VHDL und die Projektkonfigurationsdatei.

Darüber hinaus gibt es noch weitere Dateien, die für die Definition des Betriebssystems notwendig sind. Drei Arten von Dateien werden benötigt: eine Betriebssystemkonfigurationsdatei, die Schnittstellendateien und die Moduledateien. Die Schnittstellendateien definieren die Betriebssystemschnittstellen, die in den Anwendungsprogrammen benutzt werden können. Zu jeder Schnittstelle kann eine beliebige Anzahl an Realisierungen existieren. Diese Realisierungen sind durch die Moduldateien festgelegt. Das Zusammenspiel wird in der Betriebssystemkonfigurationsdatei beschrieben. Das Ergebnis des Strukturcompilers ist eine VHDL-Datei und eine UCF-Datei, wenn es sich um eine Synthese handelt. Alle Komponenten des Strukturcompilers sind in Abbildung 7.4 dargestellt.

7.2.1 Kommandozeile

Der Strukturcompiler kann mit verschiedenen Kommandos aufgerufen werden. Der allgemeine Aufruf lautet:

```
prjx_v11.exe [-c project.cfg] [-m ois3000] [-d] [-t] [-h]
             [-os_dir w:\Version1\FPGA_OS]
```

-c	Dateiname	Legt den Namen der Projektkonfigurationsdatei fest. Grundeinstellung ist *project.cfg*.
-m	Realisierungsname	Legt die Realisierung fest. Der Name muss in der Projektkonfigurationsdatei vorkommen.
-os_dir	Verzeichnis	Legt das Verzeichnis fest, in dem die Betriebssystemkonfigurationsdatei *main.cfg* gefunden wird.
-d		Schaltet lexikalische und syntaktische Debuginformation ein.
-t		Ausgabe der Namen aller geparsten VHDL-Dateien in Makefile-kompatibler Datei.
-h		Ausgabe von Informationen zu den verschiedenen Schaltern.

7.2.2 Projektkonfigurationsdatei

Die Projektkonfigurationsdatei beinhaltet die relevanten Informationen für die Definition der Realisierungen des Anwendungsprogramms. Ein Beispiel wurde bereits in Abschnitt 7.1 gegeben. Zu den Informationen gehört die Angabe der Dateien, die das Anwendungsprogramm beschreiben (`SOURCES FILES`) und die Namen der Anwendungstoplevelmodule (`TOPLEVEL NAME`). Weiterhin muss definiert werden, wie die Namen der Schnittstelle (`MAIN ENTITY NAME`) und der Architektur (`MAIN ARCHITECTURE NAME`) für das neu erzeugte Toplevelmodul sind. Nach diesen Angaben, die immer vorhanden sein müssen, kann eine Liste von Bibliotheksmodulen durch das Schlüsselwort `MODULE` angegeben werden. Die Bedeutung von Bibliotheksmodulen wird im Abschnitt 7.5 erläutert. Diese Angaben sind für alle Realisierungen gleich.

Nach diesen allgemeinen Informationen folgen die so genannten Realisie-

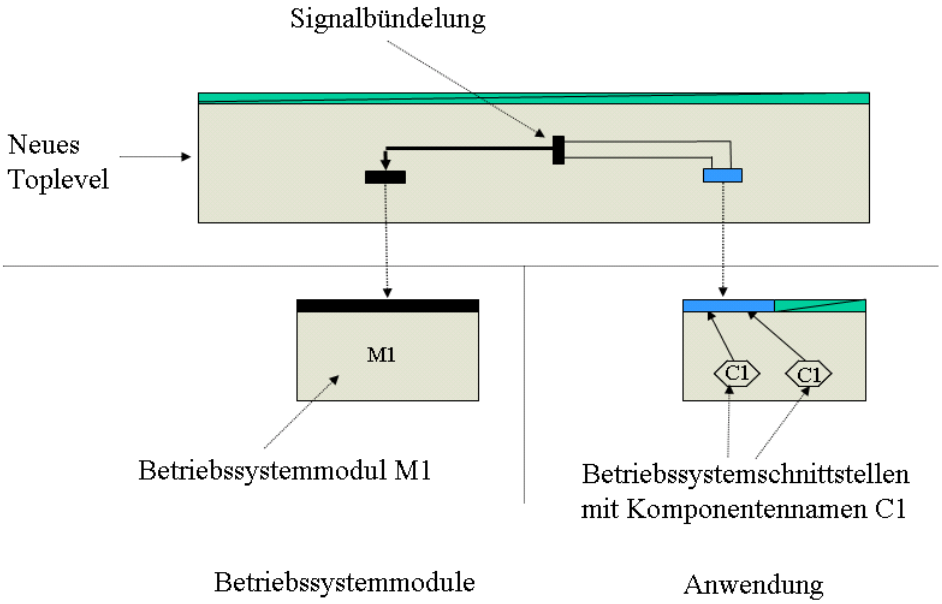

Abbildung 7.5: Verbindung zweier Betriebssystemschnittstellen C1 zu einem Betriebssystemmodul M1 durch die Zuweisung C1=M1

rungsspezifikationen. Jede Realisierungsspezifikation ist durch einen Namen sowie verschiedene Realisierungsanweisungen gegeben. Zu den Realisierungsanweisungen, die immer angegeben werden müssen, gehört die Angabe der Ausgabedatei (`OUTPUT FILE`). Soll es sich um eine Syntheserealisierung handeln, muss zusätzlich noch der Name der UCF-Datei angegeben werden (`UCF FILE`). Im Falle einer Simulation darf dieses Schlüsselwort nicht angegeben sein.

Die Angabe der Zuordnung der Betriebssystemschnittstellen zu Betriebssystemmodulen erfolgt über die Zuweisung

```
comp_inst_state_name = os_modul_name.
```

Auf der linken Seite müssen die Komponentennamen der Betriebssystemmodule stehen. Auf der rechten Seite steht der Name eines Betriebssystemmoduls, das mit dieser Schnittstelle verbunden werden soll. Sollten mehrere Komponenten mit demselben Namen in einer Anwendung vorkommen, werden alle diese Betriebssystemschnittstellen mit demselben Betriebssystemmodul verbunden (Abbildung 7.5).

Auf der anderen Seite kann es vorkommen, dass in der Anwendung zwei Betriebssystemschnittstellen mit identischen Schnittstellendefinition aber unterschiedlichen Namen auftreten. Wenn bei beiden Komponenten die obige

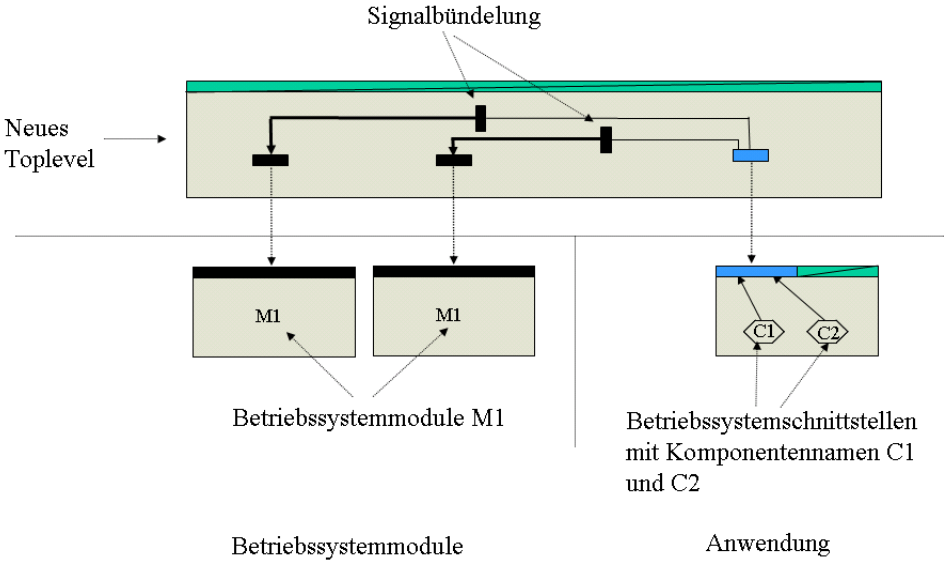

Abbildung 7.6: Verbindung zweier Betriebssystemschnittstellen C1 und C2 zu zwei Betriebssystemmodulen M1 und M2 durch die Zuweisungen C1=M1 und C2=M1

Zuweisung denselben Namen des Betriebssystemmoduls hat, wird das entsprechende Betriebssystemmodul zweimal instanziiert und jede Instanz wird mit genau einer der beiden Betriebssystemschnittstellen verbunden (Abbildung 7.6).

Bei der Zuweisung der Betriebssystemschnittstellen zu den Betriebssystemmodulen ist es möglich, Listen von Genericzuweisungen anzugeben. Ein Beispiel ist:

```
C1=M1 { Generic1:="command_simulation.txt"; Val1:=17;};
```

Durch diese Zuweisung können aus der Konfigurationsdatei heraus die externen Generics in einem Betriebssystemmodul beeinflusst werden. Externe Generics werden im Abschnitt 7.4 erläutert.

7.2.3 Betriebssystemkonfigurationsdatei

Die Betriebssystemkonfigurationsdatei definiert das Betriebssystem. Die Datei legt sowohl die Definition der Betriebssystemschnittstellen als auch der entsprechenden Betriebssystemmodule fest. Der voreingestellte Name der Datei ist `main.cfg`. Neben der Zuordnung und Definition von Schnittstellen und

Modulen definiert diese Datei auch die Bibliotheksmodule. Ihre Konfiguration wird in Abschnitt 7.5 erläutert.

Zur Definition der Betriebssystemschnittstellen müssen Blöcke der Art

```
CLOCK_IF {
  FILE = ${INTERFACE_DIR}/control_interfaces/clock_if.vhdl,
         ${PACKAGE_DIR}/basic_package.vhdl;
};
```

definiert werden. Der Name der Schnittstelle wird zu ihrer Definition und zur Verknüpfung mit den Betriebssystemmodulen benötigt. Von Anwenderseite spielt der Name keine Rolle. In diesem Block müssen alle Dateien vorkommen, die zur Definition der Schnittstelle benötigt werden. Dazu gehört eine Datei, welche eine Schnittstellendefinition in Form eines Entity-Blocks enthält sowie, falls erforderlich, verschiedene Dateien mit Paketdefinition. Der Name der Entity muss mit dem Namen des Blockes übereinstimmen.

Zu dieser Betriebssystemschnittstelle können nun verschiedene Realisierungen definiert werden, z.B. zur Simulation:

```
CLOCK_IF {
    ARCH = sim_clock
    {
        FILE = ${ARCH_DIR}/control_interfaces/sim_clock.vhdl,
               ${PACKAGE_DIR}/std_arch_package.vhdl,
               ${PACKAGE_DIR}/basic_arch_package.vhdl;
    };
};
```

bzw. für die OIS-FPGA-Karte:

```
CLOCK_IF {
    ARCH = ois_clock
    {
        FILE = ${ARCH_DIR}/control_interfaces/ois_clock.vhdl,
               ${PACKAGE_DIR}/basic_arch_package.vhdl,
               ${PACKAGE_DIR}/std_arch_package.vhdl;
        ois_card        =   ois1000_card;
        BUFG            =   FIX;
        DCM             =   FIX;
    };
};
```

Aufgrund des Blocknames `CLOCK_IF` wird die Realisierung als Betriebssystemmodul zu der oben definierten Betriebssystemschnittstelle erkannt. Zu

einer Betriebssystemschnittstelle können beliebig viele Betriebssystemmodule existieren. Umgekehrt besitzt jedoch jedes Betriebssystemmodul nur eine entsprechende Betriebssystemschnittstelle. Der Name des Betriebssystemmoduls wird durch den Namen nach dem Schlüsselwort `ARCH` festgelegt. Diese Namen werden auf Anwendungsseite zur Zuordnung der Betriebssystemschnittstellen zu Betriebssystemmodulen benutzt. Ähnlich wie bei der Schnittstellendefinition wird für die Definition der Betriebssystemmodule eine Liste von Dateien benötigt, welche die VHDL-Beschreibung für das Modul enthält. Diese werden durch das Schlüsselwort `FILE` festgelegt. Dateien, die bereits bei der Schnittstellendefinition benötigt wurden, müssen dabei nicht noch einmal angegeben werden. Innerhalb der Dateien zur Beschreibung des Betriebssystemmoduls können nun drei Arten von Submodulaufrufen existieren. Zwei wurden bereits eingeführt und zwar Aufrufe von Betriebssystemschnittstellen und Aufrufe von normalen VHDL-Submodulen. Aufrufe von Betriebssystemschnittstellen sind dadurch gekennzeichnet, dass bei der Definition des Moduls mit Hilfe der Zuweisung `os_comp=os_modul` die Schnittstelle an ein Modul gebunden wird.

Daneben kann es aber auch Submodulaufrufe geben, deren Implementierung dem Strukturcompiler nicht bekannt ist. Typische Beispiele dafür sind Aufrufe von I/O-Buffern, wie z.B. BUFG. Diese Submodule werden bei der Synthese durch das Synthesewerkzeug automatisch an entsprechende Hardwarestrukturen des FPGA gebunden. Um bei diesen Submodulaufrufen keine Fehlermeldung vom Strukturcompiler zu bekommen, muss die Zuweisung `comp_name = FIX` angegeben werden. Damit wird eine Untersuchung dieser Komponente durch den Strukturcompiler unterdrückt.

Mit Hilfe der Schlüsselwörter `NET MAP` und `CONSTRAINTS` kann Einfluss auf die UCF-Datei genommen werden. Ein Beispiel für deren Anwendung ist:

```
OISCARD_IF {
    ARCH = ois1000_card
    {
        ...
        CONSTRAINT = "NET #$extern_sdram1_clk0$#
              OFFSET = OUT 8 ns BEFORE $extern_clock$";
        CONSTRAINT = "NET #$extern_sdram1_clk1$#
              OFFSET = OUT 8 ns BEFORE $extern_clock$";

        NET MAP
        {
            extern_sdram1_clk0 => "Y31";
            extern_sdram1_clk1 => "T29";
```

```
            extern_sdram1_a    =>("G32","F32","J31","H31",
                                  "L27","M27","AD28","AE28",
                                  "L29","AE27","AE29","AF29",
                                  "V30","V27");
        };
    };
};
```

Die externen Signale `extern_sdram1_clk0` und `extern_sdram1_clk1` werden mit Bedingungen an das zeitliche Verhalten versehen sowie an die Pads Y31 bzw. T29 gebunden. Der Adressvektor `extern_sdram1_a` wird ebenfalls an bestimmte Pads als Vektor gebunden.

7.3 Betriebssystemschnittstellen

Bei der Betrachtung des Strukturcompilers spielt die Definition der Betriebssystemschnittstellen eine wichtige Rolle. An dieser Stelle soll nun diskutiert werden, wie diese definiert sind bzw. welche Einschränkungen sich aus der Benutzung des Strukturcompilers ergeben.

Wie bereits in Abschnitt 5.2.3 erläutert, soll sich der Aufruf einer Betriebssystemschnittstelle an dem Aufruf eines Submoduls orientieren. Im Falle von VHDL handelt es sich damit um Komponentenaufrufe. Aus dieser Darstellung eines Betriebssystemmoduls ergeben sich unmittelbar die Eigenschaften bzw. die Zusammensetzung einer Betriebssystemschnittstelle. Zu allererst besitzt jede Komponente bzw. deren Instanziierung ein Label und einen Namen. Wie bereits erläutert, wird der Komponentenname zur Identifizierung von Betriebssystemschnittstellen genutzt, die mit demselben Betriebssystemmodul verbunden sind. Die Komponente selbst besteht aus einer Genericliste, einer Signalliste sowie eine Liste von Paketen, die für die Typen der Generic- bzw. Signalliste benötigt werden.

Damit der Strukturcompiler weiß, ob eine Betriebssystemschnittstelle in einer Anwendung die richtige Zusammensetzung der verschiedenen Bestandteile einer Schnittstelle besitzt, verwaltet der Strukturcompiler eine Liste aller Betriebssystemschnittstellen. Sollte eine Betriebssystemschnittstelle, d.h. die Komponentendeklaration und -instanziierung, nicht mit der Schnittstellendeklaration übereinstimmen, wird eine entsprechende Fehlermeldung ausgegeben.

Aus der Funktionsweise des Strukturcompilers ergeben sich gewisse Einschränkungen an die Definition einer Betriebssystemschnittstelle. Eine erste Einschränkung ergibt sich aus der Umleitung der Signale einer Betriebssystemkomponente. Da eine Signalumleitung nur für gerichtete Signale möglich

ist, können Signale mit dem Attribut *inout* nicht behandelt werden. Daher dürfen Signale in der Schnittstelle nur vom Typ *in* oder *out* sein. Diese Einschränkung kann jedoch umgangen werden, indem anstelle des *inout*-Signals drei neue Signale benutzt werden: jeweils ein *in*- und *out*-Signal mit dem gleichen Typ wie das ursprüngliche *inout*-Signal sowie ein zweiwertiges Signal, das festlegt, welche der beiden vorhergehenden Signale gerade aktiv ist.

Weitere Einschränkungen ergeben sich durch die Signalbündelung. Eine allgemeine Schnittstelle ist gegeben durch:

```
entity example is
    generic (
        generic1: generic_type1;
        ...
        genericN: generic_typeN
    );
    port (
        signal1: signal_type1(generic1, ... , genericN);
        ...
        signalM: signal_typeM(generic1, ... , genericN)
    );
end entity;
```

Zur Vereinfachung der Schnittstelle sollen die Typen der Genericwerte(generic_type1 , ... , generic_typeN) nicht von anderen Genericwerten abhängig sein. Um nun die Signale einer Bündelung zu unterziehen, müssen alle Signale denselben Typen besitzen. Durch die allgemeine Definition der Schnittstellen ist dies ein Problem. So können die Typen von Genericwerten abhängen, die für die Signale unterschiedliche Werte besitzen. Um das zu demonstrieren, sei ein Schnittstellensignal mit zwei Genericwerten `N` und `M` definiert, das vom Typ `std_logic_vector(N to M)` ist. Das erste Signal sei vom Typ `std_logic_vector(1 to 7)` mit `N=1` und `M=7`. Das zweite Signal sei vom Typ `std_logic_vector(0 to 2)` mit `N=0` und `M=2`. Ähnlich wie in der Softwareentwicklung, muss zur Bündelung ein Typ definiert werden, der alle Typen, die gebündelt werden sollen, umfasst. Auf das Beispiel bezogen ist der Typ durch `std_logic_vector(0 to 7)` gegeben mit 0, dem Minimum aller Genericwerte `N`, und 7, dem Maximum aller Genericwerte `M`. Der Typ für das gebündelte Signal ist dann durch `field_type` mit der Definition

```
TYPE sub_field_type is std_logic_vector(0 to 7);
TYPE field_type is array(0 to 1) of sub_field_type;
```

gegeben. Einem gebündelten Signal können dann die zwei vorigen Signale zugeordnet werden, wobei bestimmte Anteile des gebündelten Signals nicht verbunden sind. Ein mehrdimensionaler Fall könnte ähnlich gehandhabt werden. Da ein mehrdimensionaler Fall jedoch bisher nicht notwendig war, sollen

in den Betriebssystemschnittstellen nur Signaltypen mit maximal einem Genericwert vorkommen. Damit hängen die Signale entweder von keinem oder von einem Genericwert ab. Erstere werden im Weiteren als einfache Signale, die zweiten als komplexe Signale bezeichnet. Für die einfachen Signale stellt sich die Bündelung als einfache Zuweisung der verschiedenen Instanzen dar. Analog dazu werden die Generics in einfache und komplexe Generics eingeteilt, je nachdem, ob das Generic in einem Signaltyp vorkommt oder nicht.

Eine allgemeine Betriebssystemschnittstelle ist damit definiert als:

```
entity example is
    generic (
        generic1: generic_type1;
        ...
        genericN: generic_typeN
    );
    port (
        -- einfache Signale
        signal_S1: signal_simple_type1;
        ...
        signal_SM: signal_simple_typeM;
        -- komplexe Signale
        signal_C1: signal_complex_type1(genericI);
        ...
        signal CK: signal_complex_typeK(genericJ)
    );
end entity;
```

7.4 Betriebssystemmodule

Die Betriebssystemmodule stellen die Realisierungen der Betriebssystemschnittstellen dar. Dabei können eine beliebige Anzahl an Betriebssystemschnittstellen mit einem Betriebssystemmodul verbunden sein. Damit dies unabhängig von der realen Anzahl der Betriebssystemschnittstellen beschrieben werden kann, müssen die Signale der Betriebssystemschnittstellen in dem neu gebildeten Toplevelmodul gebündelt werden.

Die Theorie des Betriebssystemkonzepts geht davon aus, dass auch unterschiedliche Betriebssystemschnittstellen mit einem Betriebssystemmodul verbunden sein können. Die Umsetzung dieser Eigenschaft stellt sich jedoch als sehr schwierig heraus. Zum einen treten Probleme bei der Bündelung auf, sie ist nicht mehr homogen. Die verschiedenen Schnittstellen verlangen unterschiedliche Signalzuweisungen der gebündelten Signale. Des Weiteren ist

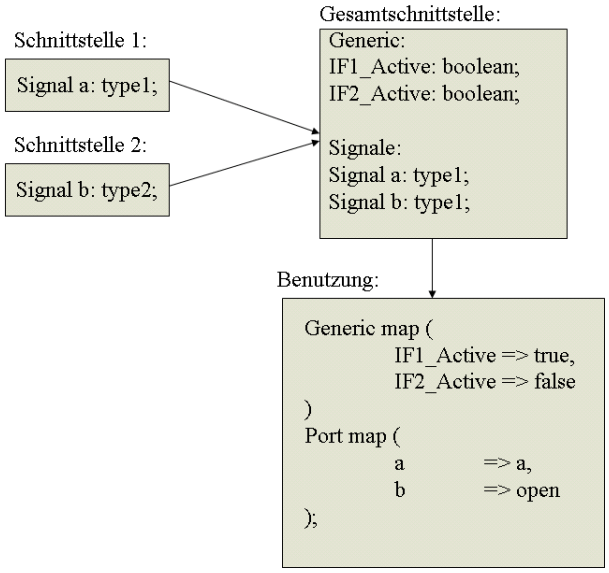

Abbildung 7.7: Verknüpfung zweier verschiedener Schnittstellendefinitionen zu einer komplexen Schnittstelle

nicht unmittelbar klar, wie die Zuordnung der Signale der Betriebssystemschnittstellen zu den gebündelten Signalen ist. Da es auch vorkommen kann, dass in zwei unterschiedlichen Betriebssystemschnittstellen Signale mit gleichen Namen auftreten, kann es bei der Bündelung zu Konflikten kommen. Daher werden im Strukturcompiler nur Betriebssystemmodule zugelassen, die nur eine Art von Betriebssystemschnittstelle unterstützen.

Dies stellt eine sehr starke Einschränkung des Konzepts dar, die jedoch umgangen werden kann. Dazu wird aus den verschiedenen Einzelschnittstellen eine komplexe, alle Signale umfassende Gesamtschnittstelle abgeleitet. Dabei muss dafür gesorgt werden, dass keine Signale denselben Namen haben. Durch Hinzufügen von Genericwerten vom Typ `boolean` für die komplexe Schnittstelle können bestimmte Signalgruppen in der Schnittstelle freigeschaltet werden (Abbildung 7.7).

In der Schnittstelle der Beschreibung eines Betriebssystemmoduls können fünf verschiedene Anteile auftreten:

- Genericwert INSTANCES: legt die Anzahl der Betriebssystemschnittstellen fest
- Generics aus der Definition der Betriebssystemschnittstelle: stellen Indizies dar, die über Konstantentabellen auf die Genericlisten der Betriebssystemschnittstellen zugreifen können

- Generics, die nicht in der Definition der Betriebssystemschnittstelle vorkommen (externe Generics): ermöglichen eine zusätzliche Beeinflussung des Verhaltens durch Konfigurationsdateien
- Signale aus der Definition der Betriebssystemschnittstelle: sind die gebündelten Signale der Betriebssystemschnittstellen
- Signale, die nicht in der Definition der Betriebssystemschnittstelle vorkommen (externe Signale): werden als externe Signale zur Schnittstelle des Toplevelmoduls durchgeleitet

Generic INSTANCES

Die Anzahl der Betriebssystemschnittstellen, die mit dem Modul verbunden sind, wird über das Generic `INSTANCES` vom Typ `integer` festgelegt. Dieser Wert wird automatisch vom Strukturcompiler gesetzt.

Betriebssystemgenerics

Neben den Signalen müssen auch die Generics der Betriebssystemschnittstelle dem Betriebssystemmodul übergeben werden. Die Generics unterliegen damit ebenfalls einer Bündelung. Da diese jedoch statisch ist, muss die Bündelung nicht explizit ausgeführt werden. Normalerweise sollte es möglich sein, die gebündelten Generics, d.h. die Felder aus Genericwerten, den Betriebssystemmodulen zu übergeben. Dafür müssten nur die entsprechenden Generictypen als Felder ausgeführt sein. Für Simulationen funktioniert dieser Ansatz auch. Bei der Synthese gibt es jedoch Schwierigkeiten, da die heutigen Synthesecompiler komplexe Generictypen nicht auswerten können. Aus diesem Grund werden die Felder der Genericwerte nicht direkt an die Module übergeben, sondern in speziellen Paketen als Konstantenfelder definiert. Dem Betriebssystemmodul wird der Index vom Typ `integer` innerhalb dieses Konstantenfeldes übergeben. Mittels dieses Index ist das Betriebssystemmodul in der Lage, auf das Genericfeld und damit auf die Werte der einzelnen Betriebssystemschnittstellen zuzugreifen. Die Erzeugung der Konstantenfelder erfolgt automatisch durch den Strukturcompiler.

Das gesamte Verhalten der Genericbehandlung soll an einem einfachen Beispiel, angelehnt an die Kommunikationsschnittstellen, betrachtet werden. Die Betriebssystemschnittstelle sei definiert durch:

```
entity beispiel is
    generic (
        ID    : integer;
```

```
        WIDTH : integer
    );
    port (
        clock : in std_logic;
        data  : out std_logic_vector(WIDTH-1 downto 0)
    );
end;
```

Diese Betriebssystemschnittstelle wird in der Anwendung zweimal benutzt, einmal mit der Zuordnung ID=1, WIDTH=8 und einmal mit der Zuordnung ID=2, WIDTH=16. Bei dem ersten Genericwert (ID) handelt es sich um einen einfachen Generic, bei dem zweiten um einen komplexen Generic. Durch spezielle Kommandos im Paket erzeugt der Strukturcompiler aus diesen Schnittstellen folgende Typen und Konstantentabellen:

```
package BeispielPackage is

    -- Einträge für einfachen Integertypen
    -- Schlüsselwort:
    --     constant X_S_array_integer:X_S_array_type_integer;
    -- Anzahl Einträge in Tabelle
    CONSTANT X_S_nr_INTEGER: integer:=1;

    -- Anzahl maximaler Schnittstellen pro Modul
    CONSTANT X_S_max_instances_INTEGER: integer:=2;

    -- Typen für Konstantenfeld
    TYPE X_S_type_INTEGER is array(0 to X_S_max_instances_INTEGER-1)
         of INTEGER;
    TYPE X_S_array_type_INTEGER is array(0 to X_S_nr_INTEGER-1)
         of X_S_type_INTEGER;
    -- Konstantenfeld
    CONSTANT X_S_array_INTEGER: X_S_array_type_INTEGER:=
         ((1,2),(1,2));

    -- Einträge für komplexen Integertypen
    -- Schluesselwort:
    --     constant X_C_array_integer:X_C_array_type_integer;

    -- Anzahl Einträge in Tabelle
    CONSTANT X_C_max_value_INTEGER:integer:=1;

    -- Anzahl maximaler Schnittstellen pro Modul
    CONSTANT X_C_max_instances_INTEGER: integer:=2;
```

```
    -- Grösster Wert für Breite
    CONSTANT X_C_max_value_INTEGER: integer:=16;

    -- Typen für Konstantenfeld
    TYPE X_C_type_INTEGER is array(0 to X_C_max_instances_INTEGER-1)
         of integer;
    TYPE X_C_array_type_INTEGER is array(0 to X_C_nr_INTEGER-1) of
                            X_C_type_INTEGER;
    CONSTANT X_C_array_INTEGER: X_C_array_type_INTEGER:=
         ((8,16),(8,16));
    TYPE std_logic_vector_array IS ARRAY (natural RANGE <>) OF
            std_logic_vector(X_C_MAX_VALUE_INTEGER-1 DOWNTO 0);
end package;
```

Wie aus dem Beispiel ersichtlich, sind die Konstantenfelder nicht als einfache Felder der Genericwerte ausgeführt. Vielmehr umfassen die Konstantenfelder immer mehrere Felder von Genericwerten. Dies liegt daran, dass die einfachen und komplexen Generictypen nicht nur in einer Betriebssystemschnittstelle vorkommen können, sondern in verschiedenen. Dadurch kann es auch verschiedene Betriebssystemmodule geben, die auf diese Felder von Generics zurückgreifen müssen. Durch die gewählte zusätzliche Dimension der Konstantenfelder wird garantiert, dass nur ein Konstantenfeld für alle diese Betriebssystemmodule benutzt werden muss.

Betriebssystemsignale

Die Betriebssystemsignale sind die gebündelten Signale der Betriebssystemschnittstellen. Sie unterscheiden sich von den externen Signalen dadurch, dass deren Signalname in der Definition der entsprechenden Betriebssystemschnittstelle vorkommt (vgl. Abbildung 5.4 schwarzer Anteil). Die Typen der Betriebssystemsignale leiten sich aus den Signaltypen der Betriebssystemschnittstelle ab. Die Einschränkungen dieser Signale wurden bereits in Abschnitt 7.3 beschrieben.

Zusätzlich spielen die Konstantenfelder bei der Bündelung von komplexen Signalen eine wichtige Rolle. Da hier Signale gebündelt werden sollen, die unterschiedlich lang sind, muss vor einer Bündelung ein Signaltyp gefunden werden, der groß genug ist, um alle verschiedenen Signale aufzunehmen. Die Signallängen der Betriebssystemschnittstellen werden durch die Generics festgelegt. Durch Bestimmung des maximalen Wertes innerhalb eines Konstantenfeldes kann ein Typ abgeleitet werden, der groß genug ist, um die Signale der Betriebssystemschnittstellen aufzunehmen. Diese Berechnung wird ebenfalls durch den Strukturcompiler durchgeführt.

Externe Signale

Mit Hilfe der externen Signale kann ein Betriebssystemmodul auf die externe Hardware zugreifen. Dieser Anteil der Schnittstelle wird im neugebildeten Toplevelmodul zu der Schnittstelle des Hardwaremodells weitergeleitet (vgl. Abbildung 5.4 gelber Anteil). Innerhalb der Schnittstelle eines Betriebssystemmoduls sind extern Signale dadurch definiert, dass deren Signalname nicht in der Definition der entsprechenden Betriebssystemschnittstelle vorkommt. Im Gegensatz zu den Betriebssystemsignalen unterliegen diese Signale keinen besonderen Einschränkungen. Insbesondere können die Attribute der Signale auch den Wert *inout* besitzen.

Externe Generics

Wie die Signale, können die Generics auch unterteilt werden. Wenn die Genericnamen in der Definition der Betriebssystemschnittstelle vorkommt, so handelt es sich um Betriebssystemgenerics, die bereits diskutiert wurden. Andernfalls handelt es sich um externe Generics. Diese Generics werden zur zusätzlichen Beeinflussung des Verhaltens der Betriebssystemmodule benutzt. In einem Simulationsmodul, z.B. den Kommunikationsmodulen, kann mit Hilfe eines solchen Generics der Name für die Kommandodatei festgelegt werden. Der Wert des Generics wird durch die Projektkonfigurationsdatei oder die Betriebssystemkonfigurationsdatei festgelegt. Der Strukturcompiler ersetzt dann automatisch den Default-Wert des Generics durch den in den Konfigurationsdateien gegebenen Wert.

7.5 Bibliotheksmodule

Der Entwicklungsprozess wird nicht nur durch die Benutzung von Betriebssystemschnittstellen vereinfacht, sondern auch durch die Benutzung von Bibliotheken. Dieses soll bei einem Hardwarebetriebssystem ebenfalls möglich sein. Die Sprache VHDL unterstützt das Konzept von Bibliotheken. Dabei können verschiedene Module und Pakete in eigenen Bibliotheken (Libraries) abgespeichert werden. Auf diese Bibliotheken, z.B. der IEEE-Bibliothek, kann von jedem Anwendungsprogramm zugegriffen werden. Solange es sich um Pakete handelt bzw. um Module, die keine Betriebssystemschnittstellen benutzen, wird dieses Konzept auch von den bisher vorgestellten Eigenschaften des Strukturcompilers unterstützt. Sollen die Module jedoch auch Betriebssystemschnittstellen aufrufen dürfen, wird dies durch die Definition der Sprache VHDL nicht abgedeckt. Damit Betriebssystemschnittstellen auch in

Bibliotheksmodulen angewendet werden können, müssen diese dem Strukturcompiler als Quellen bekannt sein. Die Quellen treten jedoch nicht als Bestandteil des Anwendungsprogramms auf. Die Bibliotheksmodule werden mit Hilfe der Betriebssystemkonfigurationsdatei verwaltet und sind daher von jedem Anwendungsprogramm benutzbar.

In der Betriebssystemkonfigurationsdatei werden Bibliotheksmodule über das Schlüsselwort `MODULE` festgelegt:

```
MODULE delay {
 FILE=${MODULE_DIR}/Operators/Delay/delay.vhdl,
      ${MODULE_DIR}/Operators/Delay/delay_read_handling.vhdl,
      ${MODULE_DIR}/Operators/Delay/delay_write_handling.vhdl,
      ${PACKAGE_DIR}/basic_package.vhdl;
};

MODULE cameragrabber {
 FILE=${MODULE_DIR}/Operators/CameraGrabber/CameraGrabber.vhdl,
      ${PACKAGE_DIR}/basic_package.vhdl;
 MODULE = delay;
};
```

Innerhalb der Bibliotheksmoduldefinition können nun verschiedene bereits bekannte Schlüsselworte auftreten. Dazu gehört zum Beispiel die Angabe der zusätzlichen Dateien, welche die Quellen des Moduls bzw. deren Pakete enthalten. Weiterhin ist es möglich, durch das Schlüsselwort `MODULE` andere Bibliotheksmodule einzubinden, wodurch ein hierarchischer Aufbau von Bibliotheksmodulen entsteht.

Im Anwendungsprogramm wird ein Bibliotheksmodul über eine normale Komponentendeklaration und -instanziierung eingebunden. Mit dem Schlüsselwort `MODULE` in der Projektkonfigurationsdatei wird dem Strukturcompiler eine Referenz auf das Bibliotheksmodul übergeben. Ein Beispiel ist:

```
SOURCE FILES                =   ThresholdCalc1.vhdl;
TOPLEVEL NAME               =   ThresholdCalc1;

MODULE                      =   threshold;
MODULE                      =   delay;
MODULE                      =   splitter;
MODULE                      =   sfifo_xc2v;

MAIN ENTITY NAME            =   main_entity;
MAIN ARCHITECTURE NAME      =   main_architecture;

ois1000
```

```
{
        UCF FILE            =   ois1000/ThresholdCalc.ucf;
        OUTPUT FILE         =   ois1000/ThresholdCalc.vhd;

        pipe_input          =   ois_pci_pipe_input;
        pipe_output         =   ois_pci_pipe_output;
        channel_input       =   ois_pci_channel_input;
        memory              =   ois_sdram_memory;
        reset               =   ois_reset;
        ois_card            =   ois1000_card;
};
```

Hier werden vier verschiedene Module eingebunden. Die in den Bibliotheksmodulen auftretenden Betriebssystemschnittstellen müssen in der Projektkonfigurationsdatei mit angegeben werden. In dem obigen Beispiel treten die Betriebssystemschnittstellen `memory` und `reset` im Bibliotheksmodul `delay` auf und müssen abhängig von der Realisierung mit Betriebssystemmodulen verknüpft werden.

7.6 Transformation des Beispielprojekts

Nachdem die verschiedenen Konfigurationsmöglichkeiten des Strukturcompilers erläutert wurden, sollen in diesem Abschnitt anhand des Beispiels des Multiplexers aus Abschnitt 7.1 für die Simulation die verschiedenen Transformationsschritte beschrieben werden.

Die erste Aufgabe des Strukturcompilers besteht darin, die Kommandozeilenargumente zu analysieren und in Abhängigkeit davon die Konfigurationsdateien zu lesen. Die Analyse der Konfigurationsdateien erfolgt dabei, genauso wie die Analyse der VHDL-Dateien, durch einen Parser, der mit Hilfe der Grammatikwerkzeuge lex/yacc [ASU99] erstellt wurde.

Danach werden alle VHDL-Quellen der aktuellen Anwendung eingelesen und in Form eines Grammatikbaums abgespeichert. Die zugrunde liegende Grammatik entspricht der Definition von VHDL-87 [Det97]. Zu jedem Modul, d.h. jeder Schnittstelle und der dazugehörigen Architektur, wird ein eigener Grammatikbaum der Definition des Moduls in einer Liste abgespeichert. Über diese Grammatikbäume finden nun die verschiedenen Transformationsschritte statt.

Transformation des Anwendungsmoduls

Den Ausgangspunkt bilden die Toplevelmodule der Anwendung, die in der Projektkonfigurationsdatei angegeben sind. Alle Komponenteninstanziierungen, die in diesen Modulen auftreten, werden daraufhin überprüft, ob sie Teil des Anwendungsprogramms sind, d.h. nur ein Submodul instanziieren, eine Betriebssystemschnittstelle darstellen oder als FIX-Komponente definiert sind. Im ersten Fall wird die Transformation auf das entsprechende Submodul fortgesetzt.

Im zweiten Fall wird eine entsprechende Instanziierung dieser Komponente durch eine Menge von Signalzuweisungen ersetzt. Dabei werden entsprechend der Betriebssystemschnittstelle diesem und allen übergeordneten Modulen neue Signale hinzugefügt. Gesetzt werden diese Signale durch die Signale, die ursprünglich der Betriebssysteminstanz zugeführt wurden. Genericanteile der Betriebssystemschnittstelle werden gespeichert, um bei der Erzeugung des Toplevelmoduls eingesetzt zu werden. Da Generics auch komplexe Ausdrücke sein können, die aus anderen Generics oder Konstanten zusammengesetzt sind, müssen diese bei Verlassen eines Moduls durch die entsprechende Repräsentation des Generics oder der Konstanten ersetzt werden. Damit wird gewährleistet, dass die gespeicherten Generics für die Betriebssystemschnittstellen auch im noch zu erstellenden Toplevelmodul vollständig definiert sind. Die entsprechende Komponentendeklaration wird gelöscht. Angewendet auf das Muliplexer-Beispiel ergibt sich für das Mux-Modul:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.basic_package.ALL;

ENTITY Mux IS
   PORT (
      -- Signals for channel_input
      X_data_b: IN  std_logic_vector((8)-1 DOWNTO 0);

      -- Signals for channel_input
      X_data_c: IN  std_logic_vector((8)-1 DOWNTO 0);

      -- Signals for channel_output
      X_data_d: OUT  std_logic_vector((8)-1 DOWNTO 0);

      -- Signals for channel_input
      X_data_e: IN  std_logic_vector((1)-1 DOWNTO 0)
   );
END Mux;
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.basic_package.ALL;

ARCHITECTURE Mux_arch OF Mux IS
   SIGNAL input1: std_logic_vector(7 DOWNTO 0);
   SIGNAL input2: std_logic_vector(7 DOWNTO 0);
   SIGNAL output: std_logic_vector(7 DOWNTO 0);
   SIGNAL sel: std_logic_vector(0 DOWNTO 0);
BEGIN
   input1<=X_data_b;
   input2<=X_data_c;
   X_data_d<=output;
   sel<=X_data_e;
   output<=input1 WHEN sel="0" ELSE input2;
END Mux_arch;
```

Der dritte Fall beim Auftreten einer Komponente in einem Modul ist, dass diese Komponente in der Konfigurationsdatei als FIX definiert wurde. Das bedeutet, dass weder eine Umstrukturierung noch eine rekursive Fortsetzung der Transformation des entsprechenden Submoduls stattfindet.

Transformation der Betriebssystemmodule

Nachdem alle Anwendungsmodule transformiert wurden, steht unter Zuhilfenahme der Konfigurationsdateien fest, welche Betriebssystemmodule benötigt werden. Diese werden dann ebenfalls nach obigem Schema transformiert. Aufgrund des hierarchischen Aufbaus des Betriebssystems kann ein Betriebssystemmodul auch von anderen Betriebssystemmodulen abhängen. Sollten diese neuen Betriebssystemmodule noch nicht transformiert worden sein, werden diese ebenfalls transformiert und zu der Menge der benötigten Betriebssystemmodule hinzugefügt. Dies wird solange fortgesetzt, bis alle benötigten Betriebssystemmodule transformiert wurden. Für das Beispiel des Multiplexers werden drei Betriebssystemmodule benötigt:

- `sim_clock` zur Erzeugung der simulierten Clock- und Resetsignale,
- `sim_channel_input` zur Stimulation der Channel-Eingabe-Schnittstellen und
- `sim_channel_output` zur Analyse der Channel-Ausgabe-Schnittstelle.

Da das Modul `sim_clock` keine weitere Betriebssystemschnittstelle benutzt, bleibt es unter der Transformation unverändert. Die beiden anderen Module benutzen dagegen eine Betriebssystemschnittstelle. Unter der Transformation ergibt sich daher für diese:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.basic_arch_package.ALL;
USE WORK.std_arch_package.ALL;
USE WORK.basic_package.ALL;

ENTITY sim_channel_input IS
   GENERIC (
      -- interface generics
      INSTANCES: integer:=1;
      ID: integer:=0;
      INIT: integer:=0;
      WIDTH: integer:=0;
      --extern generics normal
      COMMAND_FILENAME: string :="channel_input_command.txt";
      RESULT_FILENAME: string :="channel_input_result.log"
   );
   PORT (
      -- interface complex signals
      data: OUT channel_data_array(0 TO instances-1);

      -- Signals for clock
      X_clock_b: IN  std_logic;
      X_reset_b: IN  reset_type
   );
END sim_channel_input;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE std.textio.ALL;
USE IEEE.std_logic_textio.ALL;
USE WORK.basic_package.ALL;
USE WORK.basic_arch_package.ALL;
USE WORK.command_file_handling.ALL;

ARCHITECTURE sim_channel_input_arch OF sim_channel_input IS
   SIGNAL clock: std_logic;
```

```
   SIGNAL reset: reset_type;
   CONSTANT INIT_ID: X_S_type_integer :=
        X_S_array_integer(INIT);
   CONSTANT CHANNEL_ID: X_S_type_CHANNEL_ID_TYPE :=
        X_S_array_CHANNEL_ID_TYPE(ID);
   CONSTANT WIDTH_ID: X_C_type_CHANNEL_WIDTH_TYPE :=
        X_C_array_CHANNEL_WIDTH_TYPE(WIDTH);
   ...
BEGIN
   ...
   P_P1:PROCESS
      ...
   BEGIN
      ...
   END PROCESS;
   reset<=X_reset_b;
   clock<=X_clock_b;
END sim_channel_input_arch;
```

In diesem Beispielcode wurden die verhaltensrelevanten Programmzeilen weggelassen. Diese bewirken, dass die Stimulanzwerte für die verschiedenen Channels aus der Datei `channel_input_command.txt` gelesen und dem Signal `data` zugewiesen werden. Ein Beispiel für den Aufbau dieser Datei wurde bereits in Abschnitt 7.1 angegeben. Ein ähnliches Ergebnis ergibt sich für das Modul `sim_channel_output`.

Generierung des neuen Toplevelmoduls

Der nächste Schritt des Strukturcompilers besteht darin, ein neues Toplevelmodul zu erzeugen. Dieses instanziiert alle Anwendungstoplevelmodule sowie alle benötigten Betriebssystemmodule. Die Verbindung untereinander findet durch die Signalbündelung statt. Für das Beispiel ergibt sich das Toplevelmodul zu:

```
-- Main Entity:

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY main_entity IS
END main_entity;

-- Main architecture:
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.basic_package.ALL;
USE WORK.basic_arch_package.ALL;
USE WORK.std_arch_package.ALL;

ARCHITECTURE main_architecture OF main_entity IS
   ... -- Deklaration der Komponenten
   SIGNAL X_data_f: channel_data_array(0 TO (3)-1);
   SIGNAL X_data_b: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_g: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_c: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_h: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_e: std_logic_vector((1)-1 DOWNTO 0);
   SIGNAL X_data_k: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_i: channel_data_array(0 TO (1)-1);
   SIGNAL X_data_d: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_data_j: std_logic_vector((8)-1 DOWNTO 0);
   SIGNAL X_clock_c: std_logic_array(0 TO (2)-1);
   SIGNAL X_reset_c: reset_array(0 TO (2)-1);
   SIGNAL X_clock_b: std_logic;
   SIGNAL X_reset_b: reset_type;
   SIGNAL X_clock_d: std_logic;
   SIGNAL X_reset_d: reset_type;
BEGIN
   X_sim_channel_input_b: sim_channel_input
      GENERIC MAP (ID=> 0, INSTANCES=> 3, INIT=> 0, WIDTH=> 0)
      PORT MAP(data=> X_data_f, X_clock_b=> X_clock_b,
               X_reset_b=> X_reset_b);
   X_data_g<=X_data_f(0);
   X_data_b((8)-1 DOWNTO 0)<=X_data_g((8)-1 DOWNTO 0);
   X_data_h<=X_data_f(1);
   X_data_c((8)-1 DOWNTO 0)<=X_data_h((8)-1 DOWNTO 0);
   X_data_k<=X_data_f(2);
   X_data_e((1)-1 DOWNTO 0)<=X_data_k((1)-1 DOWNTO 0);
   X_sim_channel_output_b: sim_channel_output
      GENERIC MAP (ID=> 1, INSTANCES=> 1, WIDTH=> 1)
      PORT MAP(data=> X_data_i, X_clock_b=> X_clock_d,
               X_reset_b=> X_reset_d);
   X_data_i(0)<=X_data_j;
   X_data_j((8)-1 DOWNTO 0)<=X_data_d((8)-1 DOWNTO 0);
   X_sim_clock_b: sim_clock
      GENERIC MAP (FREQUENCY=> 1, INSTANCES=> 2)
```

```
      PORT MAP(clock=> X_clock_c, reset=> X_reset_c);
   X_clock_b<=X_clock_c(0);
   X_clock_d<=X_clock_c(1);
   X_reset_b<=X_reset_c(0);
   X_reset_d<=X_reset_c(1);
   X_Mux_b: Mux
      PORT MAP(X_data_b=> X_data_b, X_data_c=> X_data_c,
                X_data_d=> X_data_d,
         X_data_e=> X_data_e
      );
END main_architecture;
```

Paketerzeugung

Der letzte Schritt besteht darin, das Paket zu erzeugen, das für die Signalbündelung und die Genericübergabe benötigt wird:

```
1   LIBRARY IEEE;
2   USE IEEE.std_logic_1164.ALL;
3
4   PACKAGE std_arch_package IS
5
6     CONSTANT X_S_nr_INTEGER: integer:=2;
7     CONSTANT X_S_max_instances_INTEGER: integer:=3;
8     TYPE X_S_type_INTEGER is array
9          (0 to X_S_max_instances_INTEGER-1) of INTEGER;
10    TYPE X_S_array_type_INTEGER is array(0 to X_S_nr_INTEGER-1) of
11         X_S_type_INTEGER;
12    CONSTANT X_S_array_INTEGER: X_S_array_type_INTEGER:=((0,0,0),
13    (100000000,100000000,100000000));
14    ...
15  END std_arch_package;
16
17  LIBRARY IEEE;
18  USE IEEE.std_logic_1164.ALL;
19  USE work.basic_package.ALL;
20  USE work.std_arch_package.ALL;
21
22  PACKAGE basic_arch_package IS
23    CONSTANT X_S_nr_CHANNEL_ID_TYPE: integer:=2;
24    CONSTANT X_S_max_instances_CHANNEL_ID_TYPE: integer:=3;
25    TYPE X_S_type_CHANNEL_ID_TYPE is array
26         (0 to X_S_max_instances_CHANNEL_ID_TYPE-1)
27         of CHANNEL_ID_TYPE;
```

```
28     TYPE X_S_array_type_CHANNEL_ID_TYPE is array
29          (0 to X_S_nr_CHANNEL_ID_TYPE-1)
30          of X_S_type_CHANNEL_ID_TYPE;
31     CONSTANT X_S_array_CHANNEL_ID_TYPE:
32          X_S_array_type_CHANNEL_ID_TYPE:=((1,2,5),(1,1,1));
33     CONSTANT X_C_nr_CHANNEL_WIDTH_TYPE: integer:=2;
34     CONSTANT X_C_max_instances_CHANNEL_WIDTH_TYPE: integer:=3;
35     CONSTANT X_C_max_value_CHANNEL_WIDTH_TYPE: integer:=8;
36     TYPE X_C_type_CHANNEL_WIDTH_TYPE is array
37          (0 to X_C_max_instances_CHANNEL_WIDTH_TYPE-1)
38          of CHANNEL_WIDTH_TYPE;
39     TYPE X_C_array_type_CHANNEL_WIDTH_TYPE is array
40          (0 to X_C_nr_CHANNEL_WIDTH_TYPE-1)
41          of X_C_type_CHANNEL_WIDTH_TYPE;
42     CONSTANT X_C_array_CHANNEL_WIDTH_TYPE:
43           X_C_array_type_CHANNEL_WIDTH_TYPE:=((8,8,1),(8,8,8));
44     ...
45     TYPE channel_data_array IS ARRAY (natural RANGE <>) OF
46      std_logic_vector(X_C_MAX_VALUE_CHANNEL_WIDTH_TYPE-1 DOWNTO 0);
47     ...
48   END basic_arch_package;
49
50   PACKAGE BODY basic_arch_package IS
51      ...
52   END basic_arch_package;
```

In diesem Paket wurden zwei Konstantentabellen definiert, für die Frequenzen und für die ID. Die Frequenzen besitzen in dem Beispiel alle den Wert 100 MHz (Zeile 13). In Zeile 32 stehen die verschiedenen Werte für die ID, die den Modulen `sim_channel_input` und `sim_channel_output` übergegeben werden. Das erste Subfeld (1,2,5) wird dabei für das Modul `sim_channel_input`, das zweite für das Modul `sim_channel_output` benutzt. In Zeile 43 sind in derselben Reihenfolge die Werte für die Breite des Channel-Kanals definiert. Mit Hilfe des maximalen Wertes in diesem Feld (Zeile 35) kann ein Typ definiert werden, der die gebündelten Signale aufnimmt (Zeile 45).

Am Ende gibt der Strukturcompiler alle benötigten Pakete sowie alle Module, einschließlich des neuen Toplevelmoduls in eine Ausgabedatei aus. Sollte eine Syntheserealisierung vorliegen, was durch das Schlüsselwort `UCF` erkannt wird, generiert der Strukturcompiler außerdem noch eine UCF-Datei, die zusätzliche Bedingungen an die externen Signale enthält.

7.7 Einschränkungen

Der entwickelte Strukturcompiler unterliegt bezüglich der in Kapitel 5 vorgestellten theoretischen Betrachtungen gewissen Einschränkungen. Diese kommen durch die Komplexität und die zeitliche Begrenzung der Entwicklung zustande.

Eine Einschränkung gibt es in der unterstützten VHDL-Version. Da zur Generierung des Compilers das Programm *bison* benutzt wird, muss die verwendete Grammatik aus der Menge der LALR(1) Grammatiken sein. Die Sprache VHDL in ihren '87 und '93 Versionen ist nicht als LALR(1) konform spezifiziert worden. Daher wurde im Programm eine LALR(1)-Grammatik [Det97] benutzt, die aus der '87 Version abgeleitet wurde, jedoch eine Übermenge der VHDL-Sprache beschreibt. Daher können fehlerhafte VHDL-Programme als syntaktisch korrekt vom Strukturcompiler anerkannt werden. Die verbleibenden Fehler treten jedoch nicht in den strukturrelevanten Teilen des Programms auf, sodass hier der Strukturcompiler korrekt arbeiten kann. Noch verbleibende Fehler werden durch den Synthese- oder Simulations-Compiler erkannt.

Eine weitere Einschränkung besteht darin, dass die Betriebssystemmodule nur eine Art von Schnittstellen instanziieren können. Mit dem in Abschnitt 7.4 erläuterten Ansatz kann diese Einschränkung umgangen werden.

Eine im Hinblick auf die Funktionalität große Einschränkung ergibt sich aus der Aufgabe des Strukturcompilers. Wie im vorhergehenden Abschnitt gezeigt, besteht eine Aufgabe darin, Komponenteninstanzen durch Signalzuweisungen zu ersetzen. Komponenteninstanzen können nun an verschiedenen Stellen im Programm auftreten: innerhalb der obersten Ebene einer Architektur, innerhalb eines Blockes oder innerhalb eines Generate-Abschnittes.

Das Auftreten innerhalb der obersten Ebene einer Architektur oder aber innerhalb eines Blockes wird durch den Lösungsansatz erfasst und ist im Programm auch umgesetzt.

Beim Auftreten von Komponenteninstanzen innerhalb von Generate-Abschnitten kommt es bei dem hier vorgestellten Lösungsansatz zu einem Problem. Es ist nicht ohne weiteres erkennbar, wie viele Instanzen generiert werden. Die Anzahl der Instanzen ist aber notwendig, um die Schnittstelle der entsprechenden Architektur anzupassen. Daher dürfen in einem Generate-Abschnitt oder in einer Architektur, die von einem Generate-Abschnitt direkt oder indirekt instanziiert wird, keine Betriebssystemschnittstellen benutzt werden.

Ein weiteres Problem tritt bei der Konfiguration einer Anwendung auf. Die Verbindung der Betriebssystemschnittstellen zu Betriebssystemmodulen erfolgt über den Namen der Betriebssystemschnittstelle. Diese Art der Defi-

nition stellt sich aber als sehr problematisch dar. Da diese Zuordnung global für alle Komponenten im VHDL-Programm benutzt wird, muss die Wahl der Komponentennamen sehr sorgfältig erfolgen, um nicht unerwünschte Signalbündelungen zu erzeugen. Durch die Wahl der Namen wird aber auch gleichzeitig die Bündelung der Signale festgelegt. Durch die Konfigurationsdatei wird nur noch bestimmt, an welches Betriebssystemmodul die gebündelten Signale zu übergeben sind. Besser wäre es, wenn die Zuordnung nicht nur anhand des Komponentennames, sondern auch anhand des Labels und der Position in der Anwendung erfolgen würde. Dadurch könnte in der Projektkonfigurationsdatei explizit festgelegt werden, dass bestimmte Betriebssystemschnittstellen mit demselben oder aber mit unterschiedlichen Betriebssystemmodule verbunden werden sollen. Ein weiterer Vorteil dieser Herangehensweise wäre, dass eine einheitliche Benennung der Betriebssystemschnittstellen durchgeführt werden könnte, wodurch die Verständlichkeit des Programmcodes besser wäre.

Kapitel 8

Anwendung des Hardwarebetriebssystems

Nachdem in den letzten Kapiteln die Theorie eines Hardwarebetriebssystems und das Werkzeug, der Strukturcompiler, beschrieben wurden, werden in diesem Kapitel einzelne Schwerpunkte bei der Realisierung eines Hardwarebetriebssystem betrachtet. Dazu wird auf verschiedene wichtige Aspekte der Umsetzung eingegangen, wie z.B. die verwendeten FPGA-Plattformen, die Umsetzung der Kommunikation mit einem PC, die Einbindung eines Prozessors im FPGA sowie die Anbindung der freeSp-Software. Außerdem wird auch ein Projekt vorgestellt, in dem das Hardwarebetriebssystem ein integraler Bestandteil ist.

8.1 FPGA-Karten

Für die Erprobung und Anwendung des Hardwarebetriebssystemkonzepts standen zwei Typen von FPGA-Karten zur Verfügung: eine RC1000-PP FPGA-PCI-Karte der Firma Celoxica und mehrere am DLR entwickelte FPGA-Prototyp-Karten mit der Bezeichnung OIS.

RC1000-PP-Karte

Bei der RC1000-PP FPGA-Karte der Firma Celoxica [Celb] handelt es sich um eine PCI-Karte, die mit verschiedenen Xilinx-VirtexE-FPGA ausgestattet ist. Zur Anwendung kam eine Karte mit einem VirtexE-2000. Die Karte besitzt folgende Merkmale:

- Xilinx-VirtexE-FPGA

- programmierbare Taktgenerierung für den FPGA
- vier 512K x 32 SRAM Bänke, sowohl vom FPGA als auch vom PCI-Bus ansprechbar

Die Programmierung des FPGA erfolgte über einen PC. Zur Ansprechung der verschiedenen Funktionen des FPGA und des Speichers sowie der Konfiguration der FPGA-Karte wurden vom Hersteller ein Treiber und eine Bibliothek mitgeliefert.

OIS-Karten

Im Rahmen des Projekts OIS [DRK+03] wurde eine FPGA-PCI Karte für CompactPC entworfen, die als Framegrabber für Kameras im visuellen und im infraroten Bereich arbeitet. Von dieser Karte wurden zwei Varianten produziert: OIS1000 mit einem Virtex2-1000 und OIS3000 mit einem Virtex2-3000. Nach Abschluss des Projekts wurde, ausgehend von diesen FPGA-Karten, eine allgemeine FPGA-Prototyp-Karte entwickelt, die ebenfalls mit Virtex2-FPGA ausgestattet ist (OIS4000X mit Virtex2-4000 und OIS6000x mit Virtex2-6000). Diese FPGA-Karte besitzt jedoch eine PCI-Schnittstelle, sodass sie auch in normalen PC eingesetzt werden kann. Des Weiteren wurden der Karte zu Testzwecken verschiedene zusätzliche Hardwarekomponenten hinzugefügt. Folgende Komponenten befinden sich auf den verschiedenen OIS-Karten (Abbildung 8.1):

- Xilinx-Virtex2-FPGA (1000, 3000, 4000, 6000)
- zwei CameraLink-Schnittstellen [PA00]
- PLX-PCI-Controller mit 128 K x 32 Bit FIFO als Kommunikationspuffer
- Ethernet PHY[1]
- 256 MByte SDRAM
- zwei 16 MByte SDRAM[1]
- 4 MByte Synchrones SRAM[1]
- 16 LED für Status- und Debuganzeigen
- Erweiterungskarte mit USB-Controller und 14 x RS-232[1]

[1] nur OIS4000X, OIS6000X

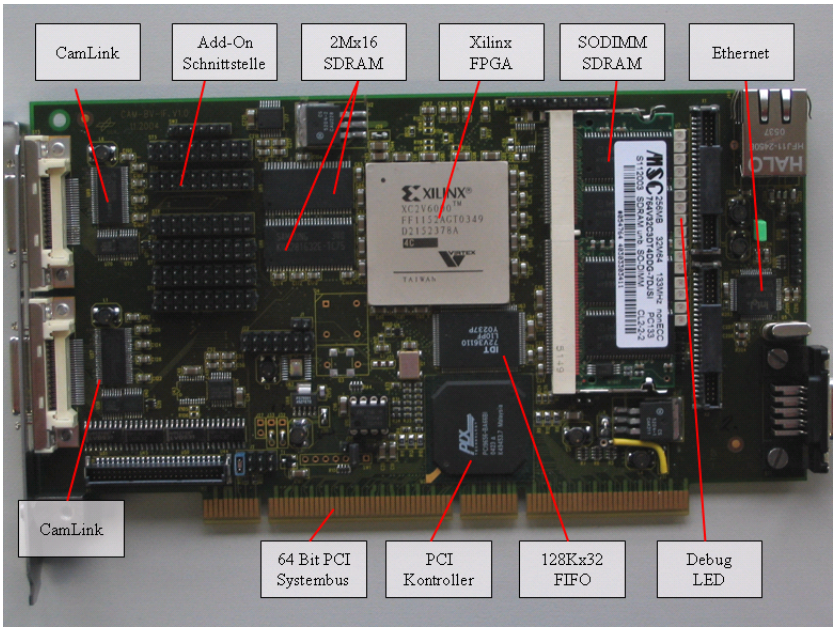

Abbildung 8.1: OIS6000x FPGA-Karte

8.2 PC-FPGA-Kommunikation

Einer der ersten Schritte, der bei der Umsetzung des Betriebssystemkonzepts durchgeführt wurde, war die Entwicklung der Kommunikationsmodule zwischen PC und FPGA. Dabei sollten die Kommunikationsarten, die in Kapitel 6 eingeführt wurden, implementiert werden. Ziel war, die Kommunikationsmodule für einen PC und einen FPGA so zu gestalten, dass einerseits ein hoher Datendurchsatz andererseits auch ein hohes Maß an Wiederbenutzbarkeit der Module für andere Kommunikationsarten garantiert wird.

Um die Wiederbenutzbarkeit von Modulen zu garantieren, wurde die in Abbildung 8.3 gezeigte vereinfachte Struktur gewählt. Als Basis diente der in Abbildung 8.2 dargestellte Teilgraph. Die Datenkommunikation setzt sich aus Modulen zusammen, die unabhängig von der physikalischen Verbindung sind (Channel-, SChannel-, Pipemodule, inklusive AMBA-Bus auf FPGA-Seite und die Funktionen der Ressourcenverwaltung auf PC-Seite), und Modulen, die von der physikalischen Verbindung abhängen (AMBA-Bridge auf FPGA-Seite und Funktionen zum Zugriff auf den AMBA-Bus über die physikalische Verbindung). Die letzteren müssen für jede physikalische Verbindung entwickelt werden. Als Verbindung zwischen den Kommunikationsmodulen und der Bridge dient der AMBA-Bus, weil dieser die Möglichkeit bietet, auch

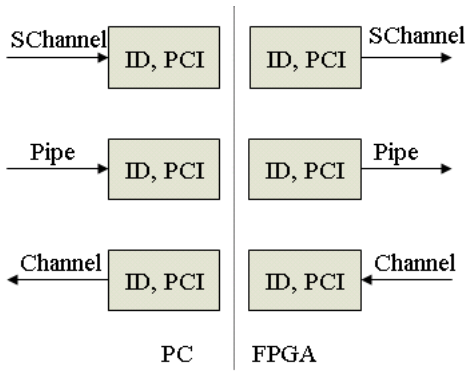

Abbildung 8.2: Ausschnitt eines Signalflussgraphen mit PC-FPGA-Datenkommunikation

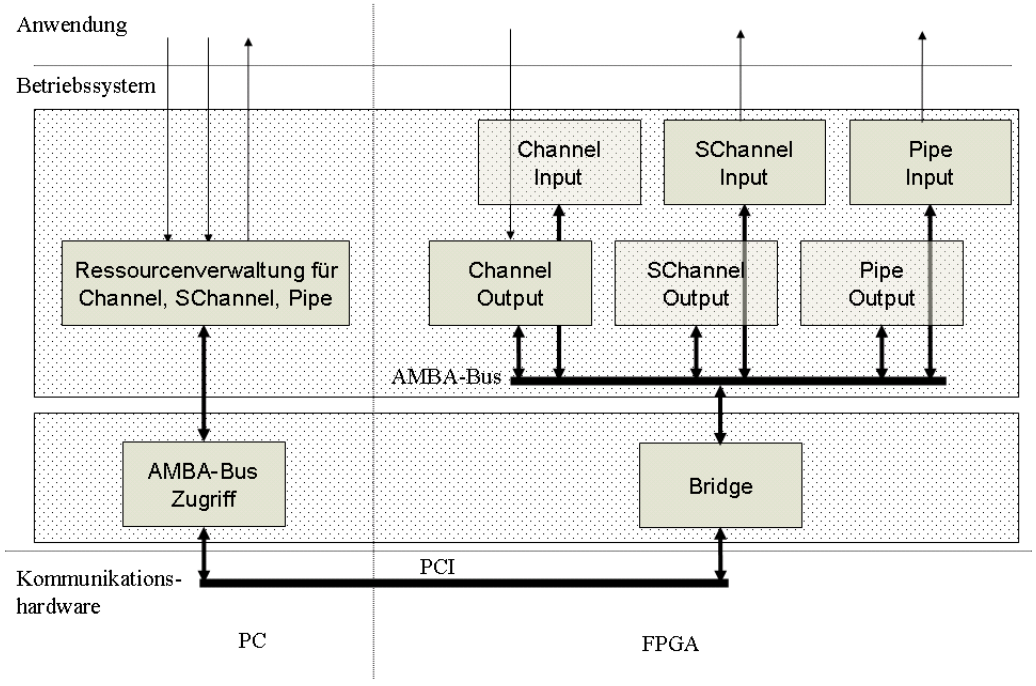

Abbildung 8.3: Struktur der PC-FPGA-Datenkommunikation zur Realisierung des Signalflussgraphen aus Abbildung 8.2

den LEON-Prozessor anzuschließen.

Bei dem AMBA-Bus handelt es sich um einen 32-bit-Multi-Master-Bus [ARM99], der von ARM entwickelt wurde. Um über die physikalische Verbindung Lese- und Schreiboperationen auf dem AMBA-Bus durchführen zu können, muss ein Protokoll eingeführt werden. Auf dem PC findet die Umsetzung des Protokolls in den Funktionen des AMBA-Zugriffes statt, auf dem FPGA mit Hilfe der AMBA-Bridge. Die AMBA-Bridge muss daher ein Master auf dem AMBA-Bus sein. Bei den anderen Modulen am AMBA-Bus handelt es sich um Slaves. Bei einer solchen Struktur ist der PC immer der Initiator von Operationen auf dem AMBA-Bus und damit auch Initiator von Operationen in den Kommunikationsmodulen.

Für die OIS- und Celoxica-Karten wurde dieses System für PCI, USB und Ethernet umgesetzt. Die Kommunikation über USB wurde im Rahmen einer Diplomarbeit [Odd06] entwickelt. Für die Kommunikation über Ethernet wurde auf IP-Cores aus dem Projekt GRLIB [GHC05, Iso04] zurückgegriffen. Die PCI-Kommunikation wurde dagegen komplett neu entworfen.

Bei der Umsetzung der Bridges für die Kommunikation spielen auch die Eigenschaften der physikalischen Verbindungen eine wichtige Rolle. Die Auswirkungen der z.B. hohen Latenzzeiten von USB und Ethernet wurden bereits in Abschnitt 6.3 erläutert. Dort wurde auch eine Lösung präsentiert, die in den Realisierungen der Kommunikation umgesetzt wurde.

8.3 FPGA-Prozessoren

Neben externen Prozessoren können auch Prozessoren im FPGA selbst untergebracht werden, und zwar entweder fest verdrahtet oder unter Nutzung der FPGA-Ressourcen. Bei den Prozessoren kann es sich um sehr einfache 8-bit-Prozessoren, z.B. PicoBlaze [Cha03], oder etwas komplexere 16-bit-Prozessoren oder 32-bit-Prozessoren wie den LEON-Prozessor [Gai] handeln.

Bei der hier verwendeten Methode des Entwurfs heterogener Systeme gibt es zwei Ansätze für die Einbindung der Prozessoren. Zum einen besteht die Möglichkeit, die Prozessoren als eigenständige Plattform aufzufassen. In diesem Fall können auf dem Prozessor eine beliebige Anzahl an Knoten des Signalflussgraphen untergebracht werden. Die Handhabung und Erzeugung des Programms für diese neue Plattform erfolgt genauso wie für einen externen Prozessor. Dies bedeutet aber auch, dass auf dem Prozessor ein Betriebssystem laufen muss. Erst durch dieses kann die Kommunikation des FPGA-Prozessors mit den restlichen FPGA vereinheitlicht werden. Durch die Benutzung eines Betriebssystems auf dem FPGA-Prozessor benötigt der Prozessor jedoch unabhängig von dem umgesetzten Algorithmus viele Res-

sourcen. Dieses drückt sich z.B. in einem hohen Speicherverbrauch aus. Daher wird diese Art der Umsetzung nur für komplexe Prozessoren, wie den LEON-Prozessor, sinnvoll sein.

Die andere Möglichkeit besteht darin, den Prozessor als Bestandteil einer Knotenrealisierung zu benutzen. In diesem Fall ist der Prozessor fest mit einem bestimmten Programm verbunden. Damit entfällt auch die Notwendigkeit, ein Betriebssystem für die Software zu benutzen. Die Prozessoren treten damit nicht als eigenständige Plattform auf und sind für den Anwender nicht sichtbar. Solange die Prozessoren keinen Zugriff auf spezielle Komponenten des FPGA, wie z.B. Multiplizierer, haben, können sie auch innerhalb von Bibliotheksmodulen realisiert sein und müssen somit nicht Bestandteil der Betriebssystemmodule sein.

8.3.1 LEON-Prozessor

Der in verschiedenen Versionen verfügbare LEON-Prozessor ist ein synthetisierbares VHDL-Modell eines 32-bit-Prozessors, der kompatibel zur SPARC V8 Architektur [SPA92] ist. Die Kommunikation mit Peripheriegeräten erfolgt über einen AMBA-Bus. Der LEON2-Prozessor [Gai] wurde unter der Leitung der ESA[2] u.a. für Satellitenanwendungen entwickelt. Eine Weiterentwicklung ist der LEON3-Prozessor der Firma Gaisler Research. Dieser besitzt gegenüber dem LEON2 eine tiefere Pipelinestruktur sowie Unterstützung von Multiprozessorsystemen. Der LEON3-Prozessor wird eingebettet in einer Sammlung von IP-Cores geliefert, die den Zugang zu verschiedenen externen Komponenten von FPGA-Karten ermöglichen, wie z.B. RS-232, Ethernet, PCI. Diese Sammlung wird unter der Bezeichnung GRLIB [Gai04a] geführt.

Im Rahmen des Betriebssystemkonzepts wurde der Prozessor als eigenständige Plattform definiert. Damit war es aber auch nötig, dem Prozessor Zugang zu den verschiedenen Kommunikationsarten auf dem FPGA zu ermöglich. Wie im Abschnitt 8.2 gezeigt, existieren für die Kommunikation des FPGA mit externen Geräten bereits entsprechende Module. Da bereits dort Wert darauf gelegt wurde, dass die Anbindung mit Hilfe des AMBA-Busses geschieht, beschränkt sich die Einbindung des LEON-Prozessors auf den Austausch der Bridge in Abbildung 8.3 durch den LEON-Prozessor (Abbildung 8.4). Am AMBA-Bus fehlen außerdem noch zusätzliche Module, die z.B. den RAM oder ROM zur Verfügung stellen.

Der RAM des Prozessors ist dabei unabhängig von der gewählten Anwendung und kann damit fest mit dem AMBA-Bus verbunden werden. Die

[2]European Space Agency

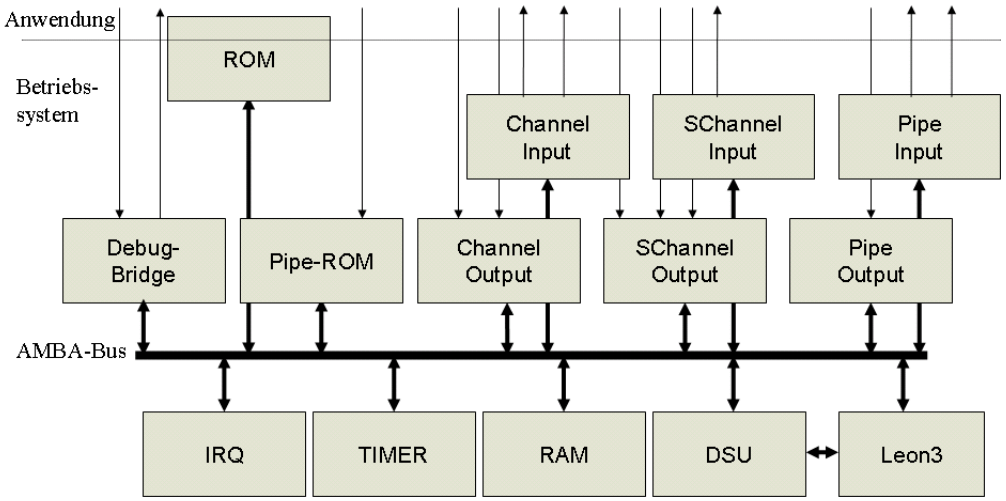

Abbildung 8.4: Betriebssystemanbindung des LEON-Prozessor

Umsetzung des RAM auf die vorhandenen Speicher einer konkreten Zielplattform ist dann wieder Aufgabe des Betriebssystems. Der ROM ist dagegen nicht unabhängig von der Anwendung, die auf dem Prozessor laufen soll. Im ROM ist das entsprechende Programm abgespeichert. Es muss daher die Möglichkeit bestehen, von Seiten der Anwendung das Programm vorzugeben. Dies kann zum einen über das Modul ROM erfolgen. In diesem Fall ist das Programm des Prozessors statisch im VHDL-Code der Anwendung festgelegt und wird über eine Betriebssystemschnittstelle dem AMBA-Modul ROM übergeben.

Alternativ besteht aber auch die Möglichkeit, durch Einbindung des Moduls Pipe-ROM einen ROM mit dem AMBA-Bus zu verbinden, dessen Inhalt während der Laufzeit des FPGA verändert werden kann. Das Programm wird dabei als Frame über die Pipe-Schnittstelle mit der Breite 32 bit dem Modul übermittelt. Bei Übertragung eines neuen Frames wird der Prozessor in den Resetzustand geschaltet und die Daten des Frames ab Adresse 0 in den ROM-Speicher übertragen. Nachdem das Frame in den Speicher kopiert wurde, kann der Prozessor den Resetzustand verlassen und durch Booten von Adresse 0 das neue Programm ausführen. Aufgrund der Möglichkeit der Umprogrammierung muss das Pipe-ROM-Modul Zugriff auf den veränderbaren Speicher haben. Dies geschieht ähnlich wie bei dem RAM-Modul mit einer Betriebssystemschnittstelle, die Speicher zur Verfügung stellt und bei einer konkreten Realisierung auf vorhandenen RAM der Plattform zurückgreift.

Neben den Modulen, die Speicher für den Prozessor zur Verfügung stellen,

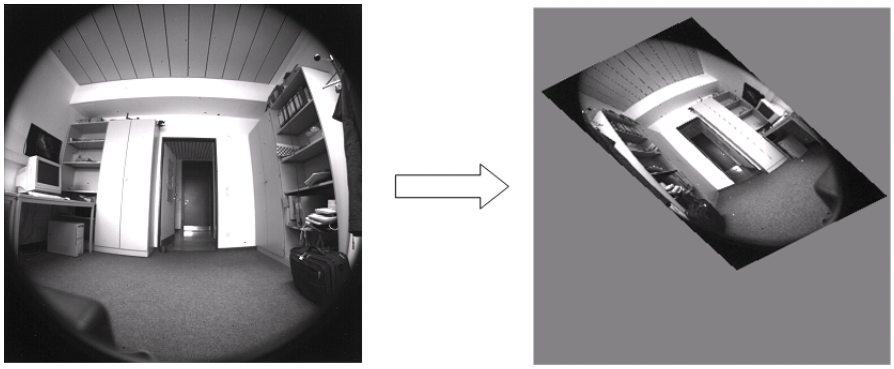

Abbildung 8.5: Beispiel für eine affine Transformation eines Kamerabildes

gibt es auch noch Module, die für verschiedene Aufgaben des Prozessors notwendig sind. Dazu zählen das Modul TIMER, das verschiedene Zeitzähler zur Verfügung stellt, und das Modul IRQ, das zur Verwaltung von Interrupts benötigt wird.

Die Module DSU[3] und Debug-Bridge werden für das Debuggen des Prozessors benötigt. Mit Hilfe des DSU-Moduls kann über den AMBA-Bus der Zustand des Prozessors ausgelesen und verändert werden. Zu den Zuständen gehören z.B. die Registerinhalte des Prozessors. Für den Zugriff auf das DSU-Modul wird ein weiteres Modul benötigt, das Lese- und Schreiboperationen auf dem AMBA-Bus durchführen kann. Verwendet wird dazu das Modul Debug-Bridge. Auf dieses Modul kann von der Anwendung über SChannel-Schnittstellen zugegriffen werden. Durch Verbindung der SChannel-Schnittstellen der Debug-Bridge in der FPGA-Anwendung mit SChannel-Schnittstellen, die mit dem PC verbunden sind, besteht die Möglichkeit, Standardwerkzeuge des LEON-Prozessors zum Debuggen zu benutzen [Gai04b].

8.3.2 Softwaretransformation zwischen PC und LEON

Durch die Benutzung der definierten Schnittstellen zur Kommunikation ist es sehr einfach, Software auf verschiedenen Prozessorplattformen zu realisieren. Dies soll anhand einer affinen Transformation demonstriert werden (Abbildung 8.5). Die Bilddaten werden von einer Kamera geliefert. Die Ausgabe des transformierten Bildes soll auf einem PC erfolgen. Der Signalflussgraph für dieses Beispiel ist in Abbildung 8.6 dargestellt. Die Parameter der affinen Transformation, die im Generator-Knoten erzeugt werden, sind dabei zeitlich nicht konstant.

[3]Debug Support Unit

In einer ersten Realisierung (Abbildung 8.7) findet auf dem FPGA nur die affine Transformation statt. Die Parameter der Transformation werden auf dem PC in einem C-Programm berechnet und über die PCI-Schnittstelle dem Transformationsmodul auf dem FPGA übermittelt.

In einer anderen Realisierung sollen die Parameter der affinen Transformation nicht auf dem PC, sondern innerhalb eines LEON-Prozessors im FPGA berechnet werden (Abbildung 8.8). An dieser Stelle kommen nun die Vorteile des Betriebssystemkonzepts besonders zur Geltung. Zur Erzeugung der Beschreibung wird kein neuer Code benötigt. Sowohl für den FPGA als auch für den Prozessor liegen die Module bereits vor, sodass diese ohne zusätzliche Anpassung wiederverwendet werden können. Auf FPGA-Seite muss die Projektkonfigurationsdatei nur derart angepasst werden, dass die Parameter des Generators nicht über die PCI-Schnittstelle geliefert werden, sondern über die Schnittstelle zum LEON-Prozessor. Des Weiteren ist eine Entscheidung darüber erforderlich, wie das Programm für den LEON-Prozessor in den ROM gelangen soll. Hier soll die Programmierung über eine Pipe-Schnittstelle vom PC aus durchgeführt werden. Dazu muss für die FPGA-Anwendung ein Bibliotheksmodul hinzugefügt werden, das die Pipe-Schnittstelle des PC (Pipe mit ID=2 in Abbildung 8.8) mit der Pipe-ROM Schnittstelle verbindet.

Die beiden Realisierungen unterscheiden sich hinsichtlich ihres Ressourcenverbrauches auf dem FPGA und dem PC sowie der Beanspruchung der physikalischen Verbindungen. Für eine FPGA-Karte vom Typ OIS3000 ergaben sich folgende Werte für die Ressourcen auf dem FPGA:

Ressource	Generator im PC	Generator im LEON
Anzahl Slices	5756	11027
Anzahl Block-RAM	29	94
Anzahl Multiplizierer	3	4

Weiterhin gibt es auch Unterschiede im zeitlichen Verhalten beider Realisierungen. Dieses Verhalten wird bestimmt durch die Taktfrequenzen der Prozessoren (PC: 2 GHz, LEON: 33 MHz) und die Auslastung der Betriebssysteme (PC: Windows XP, LEON: Single-Task-Betriebssystem BCC). Eine Messung ergab, dass die Ausführungsgeschwindigkeit der LEON-Realisierung um 20% höher war als die PC-Realisierung.

8.4 Speicherschnittstelle

Neben den Kommunikationsmodulen mussten für die FPGA-Karten auch Module für den Zugriff auf die verschiedenen Speicherarten entwickelt wer-

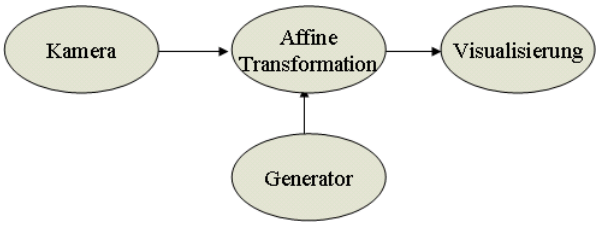

Abbildung 8.6: Signalflussgraph für eine affine Transformation

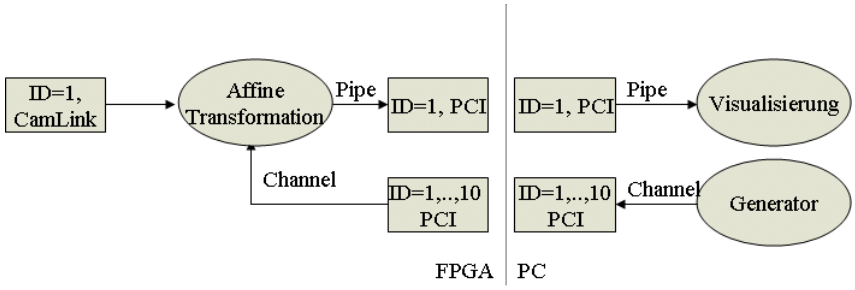

Abbildung 8.7: Realisierung mit dem Generator im PC

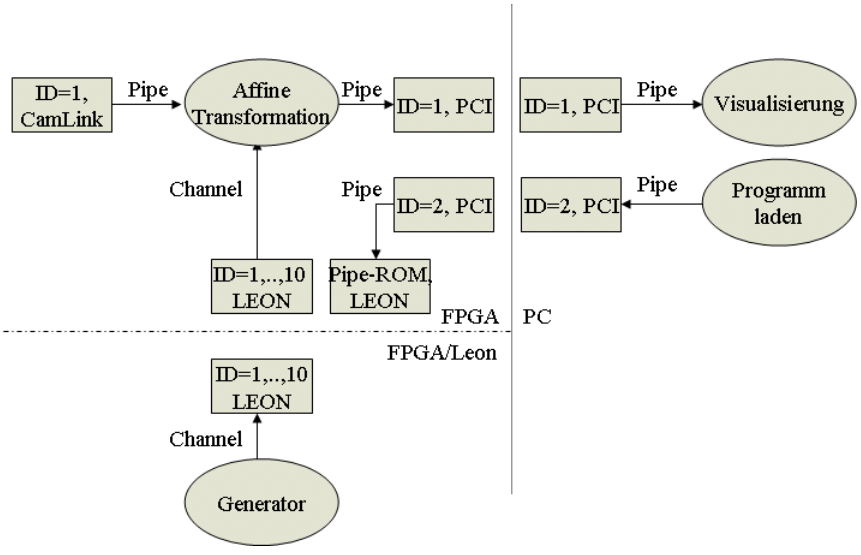

Abbildung 8.8: Realisierung mit dem Generator im LEON-Prozessor

den. Wie in Abschnitt 8.1 aufgeführt, besitzen die verwendeten Karten verschiedene Speichermodule.

Der erste wichtige Schritt, der bei der Entwicklung von Speichermodulen durchgeführt werden muss, liegt in der Definition einer allgemeinen Speicherschnittstelle. Auf der einen Seite muss diese Speicherschnittstelle so allgemein sein, dass alle möglichen Speicher (z.B. Distributed RAM, Block-RAM, SRAM, SDRAM) optimal unterstützt werden, d.h. auch hinsichtlich ihres Datendurchsatzes. Auf der anderen Seite muss die Schnittstelle aber auch so definiert sein, dass sie für den Anwender noch gut beherrschbar ist.

Prinzipiell gibt es zwei Arten von Anwendungen, die auf Speicherschnittstellen zurückgreifen müssen. Zum einen sind dies Anwendungen, bei denen die Adressen des Speicherzugriffes zufällig sind. Es gibt aber auch Anwendungen, gerade im Bereich der Signalverarbeitung, bei denen die Adressierung vorhersagbar ist. Typisches Beispiel dafür ist ein Ringpuffer zum Zwischenspeichern von Bildern.

Für den ersten Fall wurde in [Rät05] eine entsprechende Schnittstelle entworfen. Deshalb wird hier eine Schnittstelle vorgestellt, die an die Anforderungen der Signalverarbeitung angepasst ist. Der Aufbau eines Speichermoduls entspricht dem einer UMA[4]-Architektur (Abbildung 8.9). Jede Instanz besitzt einen Cache, der unter Ausnutzung der Lokalität von Speicherzugriffen die Latenzzeit verringert. Die Zugriffe der Caches auf den Speicher werden über einen Arbiter kontrolliert. Dabei können verschiedene Arbitrierungsverfahren, wie z.B. Round-Robin, zur Anwendung kommen. Der Zugriff auf den physikalischen Speicher wird durch den Speichercontroller realisiert. Diese Architektur wird jedoch nur dann verwendet, wenn nicht Speicher aus den internen Ressourcen des FPGA, d.h. Distributed RAM oder Block-RAM, benutzt werden sollen. Andernfalls wird jeder Instanz der Speicher direkt zugeordnet.

Es gibt jedoch mehrere grundlegende Unterschiede zwischen den hier benötigen Speichermodulen und bekannten UMA-Systemen. So besitzt jede Instanz seinen eigenen Adressbereich im physikalischen Speicher. Damit entfallen alle Kohärenz- und Konsistenzbedingungen zwischen den einzelnen Instanzen. Außerdem wird die Breite der Speicherschnittstelle nicht durch den physikalischen Speicher festgelegt. Vielmehr wird von Anwenderseite vorgegeben, wie breit ein Datenwort ist, das gespeichert werden soll. Die Adressen beziehen sich dann auf diese Breite. Aufgrund dieser Eigenschaften ist es nicht möglich, bekannte Verfahren und Protokolle für UMA-System, wie z.B. das MSI- oder Dragon-Protokoll [CPSG99], für das Cachesystem zu benutzen.

Die aus diesen Bedingungen abgeleitete Schnittstelle für den Speicherzu-

[4]Unified Memory Access

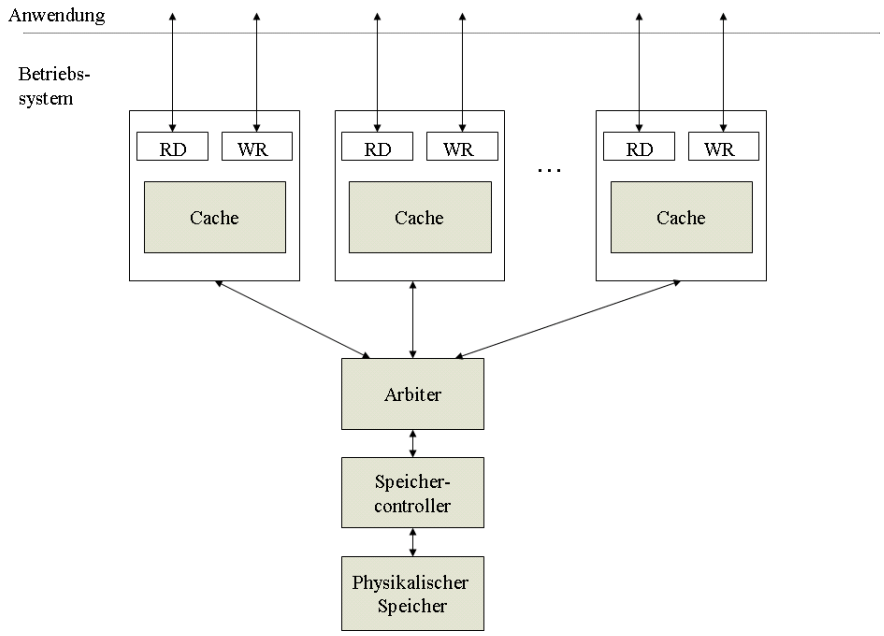

Abbildung 8.9: Allgemeiner Aufbau eines UMA-Speichermodules zum Zugriff auf externen Speicher

griff lautet:

```
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.basic_package.all;

entity MEMORY_IF is
  generic (
    DATA_WIDTH      : MEMORY_DATA_WIDTH_TYPE;
    RD_PRIORITY     : MEMORY_PRIORITY_TYPE;
    WR_PRIORITY     : MEMORY_PRIORITY_TYPE;
    BUFFER_SIZE     : MEMORY_BUFFER_SIZE_TYPE;
    ADDRESS_WIDTH   : MEMORY_ADDRESS_WIDTH_TYPE
  );
  port (
    clock           : in std_logic;

    -- write interface
    wr_address      : in std_logic_vector(ADDRESS_WIDTH-1 downto 0);
    wr_data         : in std_logic_vector(DATA_WIDTH-1 downto 0);
    wr_valid        : in valid_type;
```

```
    wr_busy         : out valid_type;
    wr_in_operation : out valid_type;

    -- read interface
    rd_address      : in std_logic_vector(ADDRESS_WIDTH-1 downto 0);
    rd_valid        : in valid_type;
    rd_busy         : out valid_type;
    rd_in_operation : out valid_type;

    rd_data         : out std_logic_vector(DATA_WIDTH-1 downto 0);
    rd_ack          : out valid_type;
    rd_output_busy  : in  valid_type
  );
end;
```

Mit den Generics werden die gewünschte Datenbreite (`DATA_WIDTH`), die Adressweite (`ADDRESS_WIDTH`), die Anzahl der maximal zu speichernden Datenworte (`BUFFER_SIZE`) und die Prioritäten für den Zugriff auf den physikalischen Speicher festgelegt. Mit Hilfe der Prioritäten entscheidet der Arbitrierer auf Basis von priorisierten Round-Robin, welche Instanz als nächstes Zugriff auf den physikalischen Speicher bekommt.

Bei den Signalen gibt es nun eine Unterscheidung nach Schnittstellenanteilen für lesende und für schreibende Speicherzugriffe. Eine Synchronität zwischen beiden Schnittstellen existiert nicht. Die Abarbeitung der Anforderungen wird über die Prioritäten geregelt.

Die Signale der Schreibschnittstelle folgen dem Protokoll der SChannel-Datenkommunikation. Hinzu kommt jedoch noch das Signal `wr_address`, mit dem die Adresse übermittelt wird, sowie das Flag `wr_in_operation`, mit dem die Anwendung ermitteln kann, ob alle Schreiboperationen ausgeführt wurden oder noch welche zur Ausführung vorliegen.

In der Leseschnittstelle sind zwei SChannel-Datenkommunikationen integriert. Über die erste SChannel-Datenkommunikation (Signale `rd_address`, `rd_valid`, `rd_busy`) werden die Adressen vorgegeben, von denen gelesen werden soll. Über die zweite werden die Ergebnisse der Leseoperationen ausgegeben (Signale `rd_data`, `rd_ack`, `rd_output_busy`). Auch hier gibt es wieder ein Flag (`rd_in_operation`), das anzeigt, ob noch Anforderungen für Leseoperationen vorliegen.

Wie aus der Schnittstellendefinition ersichtlich, können die Speichermodule die Anfragen nach dem Pipeline-Prinzip ausführen, wodurch ein hoher Datendurchsatz erzielt wird. Die große Herausforderung bei der Entwicklung von Speichermodulen liegt in der Ansteuerung des Cachesystems. Dazu muss sowohl ein neues Cacheprotokoll definiert als auch eine Datenkonvertie-

rung realisiert werden. Konvertiert werden die Datenworte von der Breite der Schnittstelle auf die Breite des physikalischen Speichers. Der Cache selbst besteht aus Dualport-RAM, dessen eine Zugriffsschnittstelle in der Taktdomäne der Instanz liegt. Die andere Zugriffsschnittstelle wird vom Arbitrierer und einem Speichercontroller genutzt, um Cachezeilen in oder aus dem physikalischen Speicher zu transferieren. In [Kru06b] ist eine detaillierte Beschreibung der Funktionsweise der Speichermodule und ihrer Anwendung angegeben.

Die Speichermodule wurden für alle Speicherarten auf der OIS-Karte realisiert, d.h. für verschiedene SDRAM und SSRAM[5]. Angewendet wird die Speicherschnittstelle in allen Speichern des LEON-Prozessors (RAM, ROM, Pipe-ROM), in dem Bibliotheksmodul zur Bildzwischenspeicherung (Delay) und in dem Bibliotheksmodul der affinen Transformation, das in Abschnitt 8.3.2 vorgestellt wurde.

8.5 freeSp

Ein Problem, das bei der Umsetzung des Betriebssystemkonzepts mit Hilfe des Strukturcompilers noch ungelöst ist, besteht in der Erzeugung des VHDL-Anwendungsprogramms aus dem Signalflussgraphen. In den vorhergehenden Abschnitten wurden bereits vereinzelt Signalflussgraphen und Partitionierungen von Anwendungen angegeben. Die VHDL-Programme und die Projektkonfigurationsdatei für den Strukturcompiler wurden dabei manuell erzeugt. An dieser Stelle soll nun als weiteres Werkzeug freeSp vorgestellt werden, das in der Lage ist, diesen Schritt zu automatisieren. Die Codegenerierung erfolgt dabei nicht nur für die Programme des FPGA, sondern auch für die Programme von Prozessoren. Die Entwicklung von freeSp steht als Open-Source-Projekt zur Verfügung [Wei].

Mit freeSp lassen sich signalverarbeitende Systeme als Signalflussgraph modellieren. Die Beschreibung des Graphen findet dabei in der Sprache XML statt. Weiterhin wird eine Beschreibung der Partitionierung benötigt. Diese erfolgt ebenfalls in der Sprache XML. Die Erzeugung beider Beschreibungen durch den Anwender entspricht den in Abbildung 4.5 gegebenen Schritten der Signalflussgrapherstellung, der Partitionierung und der Knoten- und Kantenabbildung. Der Schritt der Codegenerierung wird durch freeSp ausgeführt. Ausgehend von den beiden Beschreibungen wird mit Hilfe des Programms PART eine separate XML-Repräsentation jeder Plattform erzeugt (Abbildung 8.10). In Abhängigkeit von der Zielsprache, die in der Partitionierungsbeschreibung angegeben ist, wird für die XML-Repräsentation ein passendes XSL-Stylesheet ausgewählt. Mit den Programmen XSL und

[5]Synchronous Static Random Access Memory

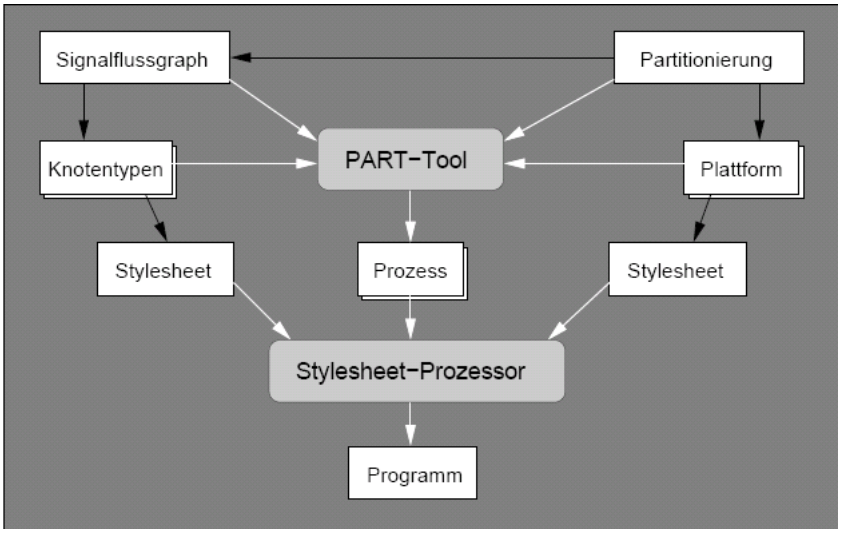

Abbildung 8.10: Datenfluss und Werkzeuge in freeSP aus [WW06]

XSLT, die ursprünglich zur Transformation von XML nach HTML entwickelt wurden [W3C], findet eine Transformation der XML-Repräsentation in die Zielsprache statt. Unterstützt werden zurzeit die Sprachen C und VHDL.

Damit freeSp den Aufruf des Strukturcompilers unterstützt, müssen drei verschiedene Dateien erzeugt werden: die VHDL-Datei, die Projektkonfigurationsdatei und ein Makefile. Zur Erzeugung dieser Dateien müssen entsprechende Vorlagen in Form von Sylesheets angelegt werden. Die Vorlage für das Makefile ist dabei unabhängig von dem Projekt. Für die beiden anderen Dateien müssen in den Vorlagen Regeln eingeführt werden, die in Abhängigkeit von dem Signalflussgraphen und der Partitionierung bestimmte Sprachkonstruktionen erzeugen.

Die Konfiguration von freeSp in Verbindung mit dem Strukturcompiler soll anhand des einfachen Beispiels einer Schwellwertberechnung dargestellt werden. Der Signalflussgraph mit Partitionierung ist in Abbildung 8.11 gegeben. Die XML-Beschreibung des Signalflussgraphen dazu lautet:

```
<signal_graph>
   <nodes>
      <ImageInput name="Bildeingabe"/>
      <ValueInput name="Schwellwerteingabe"/>
      <Threshold  name="Schwellwertberechnung"/>
      <ImageView  name="Visualisierung"/>
   </nodes>
   <edges>
```

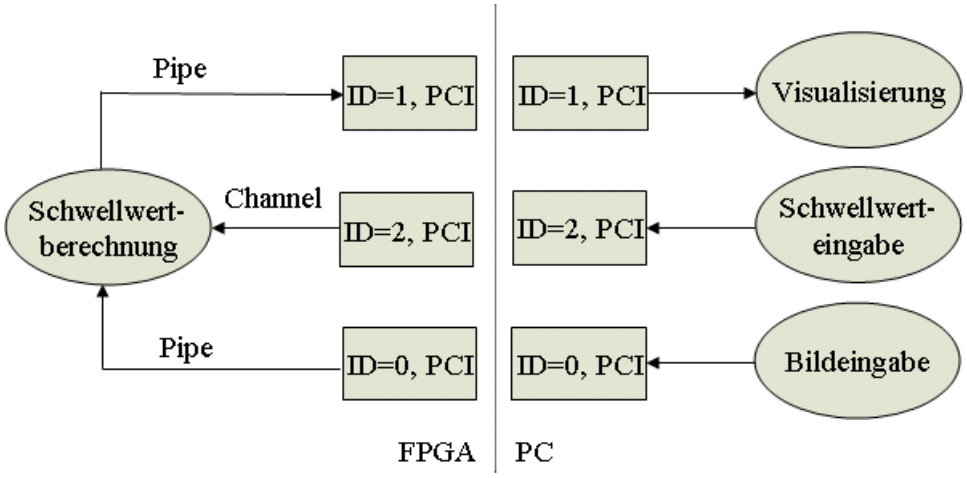

Abbildung 8.11: Signalflussgraph und Partitionierung für Schwellwertberechnung

```
      <connect from="Bildeingabe" to="Schwellwertberechnung"/>
      <connect from="Schwellwerteingabe" to="Schwellwertberechnung"/>
      <connect from="Schwellwertberechnung" to="Visualisierung"/>
   </edges>
</signal_graph>
```

Die Knoten des Algorithmus werden im Block *nodes* definiert. Für jeden Knoten müssen der Typ (*ImageInput*, *ValueInput*, *Threshold*, *ImageView*), als auch ein Knotenname (*name*) festgelegt werden. Die Kanten des Algorithmus werden im Block *edges* beschrieben. Die XML-Beschreibung der Partitionierung lautet:

```
<partition ref="signalgraph.sml">
   <arch name="linux" ref="linux.adml">
      <connect arch-name="fpga" out-port="2"
            out-type="PipeOutput" name="route1"/>
      <connect arch-name="fpga" in-port="1"
            in-type="PipeInput" name="route2"/>
      <connect arch-name="fpga" out-port="3"
            out-type="ChannelOutput" name="route3"/>
      <process name="test"/>
   </arch>
   <arch name="fpga" ref="ois6000x.adml">
      <connect arch-name="linux" in-port="2"
            in-type="PipeInput" name="route1"/>
      <connect arch-name="linux" out-port="1"
```

```
            out-type="PipeOutput" name="route2"/>
      <connect arch-name="linux" in-port="3"
            in-type="ChannelInput" name="route3"/>
      <process name="test"/>
   </arch>
   <map-node name="Schwellwertberechnung" process="fpga/test"/>
   <map-node name="Bildeingabe"           process="linux/test"/>
   <map-node name="Schwellwerteingabe"    process="linux/test"/>
   <map-node name="Visualisierung"        process="linux/test"/>
</partition>
```

Über das Schlüsselwort *arch* werden die verschiedenen Zielplattformen definiert. In diesem Fall werden Programme für eine Linux-Plattform (*linux*) und für eine OIS6000X-FPGA-Karte (*fpga*) erstellt. Mit Hilfe des Parameters *ref* wird auf die entsprechende Architekturbeschreibung verwiesen. Danach wird definiert, wie plattformübergreifende Kanten umgesetzt werden sollen. In diesem Beispiel werden die Kanten durch Channel- und Pipeverbindungen realisiert. Durch das Schlüsselwort *map-node* wird festgelegt, auf welcher Plattform die Operation eines Knoten ausgeführt werden soll. Entsprechend der Zerlegung aus Abbildung 8.11 findet nur die Schwellwertberechnung auf dem FPGA statt.

Diese zwei XML-Beschreibungen genügen, um die Programme für die Linux-Plattform und den FPGA mit Hilfe der freeSp-Werkzeuge automatisch zu erstellen. Dadurch wird die Zeit für den Systementwurf verkürzt. Leider ist die XML-Beschreibung nicht sehr benutzerfreundlich, da sie gerade bei größeren Entwürfen sehr unübersichtlich werden kann. Außerdem besteht keine Möglichkeit, bereits auf Systemebene Hierarchien einzuführen. Deshalb wäre es wünschenswert, wenn zur Erstellung der XML-Beschreibung eine graphische Oberfläche zur Verfügung stehen würde, die eine Modellierung und Simulation eines Signalflussgraphen ermöglicht. Eine solche Oberfläche ist Bestandteil des Ptolemy-Projektes. Zur Nutzung dieser Oberfläche durch freeSp müsste jedoch die XML-Beschreibung von freeSp an die XML-Beschreibung von Ptolemy angepasst werden. Des Weiteren müsste Ptolemy um die Eigenschaft erweitert werden, den Signalflussgraphen auch zu partitionieren.

Neben dem Werkzeug freeSp gibt es eine Vielzahl anderer aktueller Forschungsprojekte, die sich mit dem Entwurf von heterogenen Systemen beschäftigen. Aus dem Bereich der objektorientierten Modellierung auf Basis einer Systembeschreibung mit UML[6] sei zum Beispiel auf das Projekt GeneralStore [RGMG04] verwiesen. Auch hier sollte es möglich sein, durch Integra-

[6]Unified Modeling Language

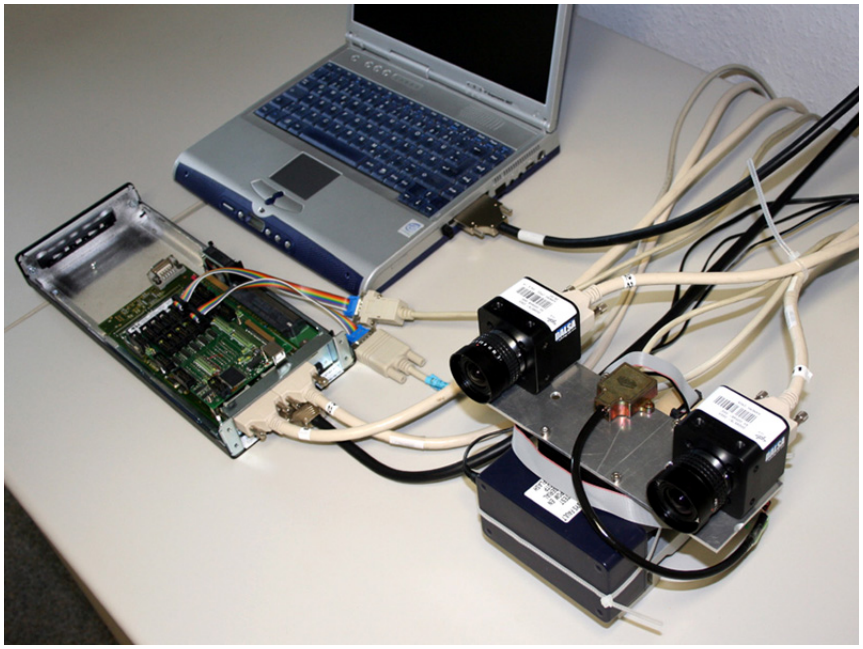

Abbildung 8.12: ModoS Sensorkopf mit OIS-FPGA-Karte

tion des Strukturcompiler die Codegenerierung für konfigurierbare Hardware zu vereinfachen.

8.6 Beispielprojekt: ModoS

An dieser Stelle soll das Projekt ModoS vorgestellt werden, in dem das Hardwarebetriebssystem ein wichtiger Bestandteil ist. Das Ziel dieses Projekts ist die Entwicklung eines universell einsetzbaren Systems zur Generierung von Umweltmodellen [Sup03]. Für diese Aufgabe ist ein Multisensoraufbau notwendig (Abbildung 8.12). Durch Verwendung mehrerer Sensoren können Messfehler einzelner Sensoren kompensiert sowie eine hohe Informationsdichte erreicht werden. Zur Anwendung sollen dabei verschiedenste Sensoren zur Lage- und Positionsbestimmung kommen. Ein Multisensoraufbau verlangt jedoch einen höheren Aufwand, die zeitliche und räumliche Beziehung der Sensorsignale untereinander zu messen. Gerade für die Synchronisation erweist sich die Verwendung eines Standard-PC als nicht ausreichend. Für diese Aufgabe wird daher eine OIS-FPGA-Karte benutzt.

An dem FPGA sind alle Sensoren angeschlossen. Die Aufgabe des FPGA besteht darin, die Sensoren anzusteuern, eine Datenextraktion aus den Sens-

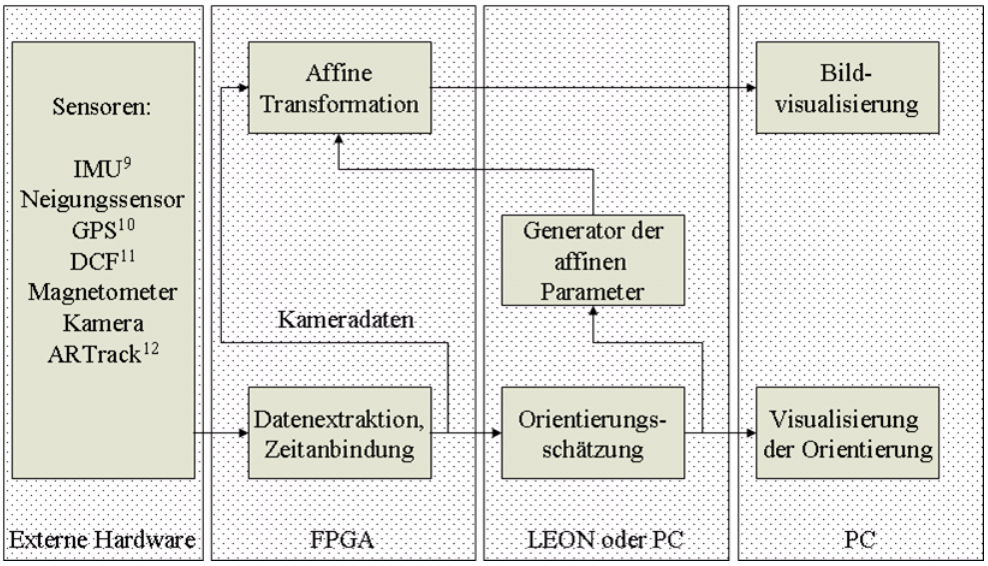

Abbildung 8.13: Blockschaltbild für das Projekt ModoS

ordaten vorzunehmen und diese mit einer lokalen FPGA-Zeit zu versehen (Abbildung 8.13). Durch Anbindung der lokalen FPGA-Zeit an die Weltzeit können im Nachhinein alle Sensordaten mit der Weltzeit synchronisiert werden. Die so bearbeiteten Sensordaten werden dann von einem Orientierungsschätzer, der zurzeit aus einem Kalman-Filter [DZ99, Zar04] besteht, weiterverarbeitet. Die Entwicklung des Orientierungsschätzers ist Teil einer anderen Dissertation. Die Schätzung liefert als Ergebnis die sechs Freiheitsgrade der Bewegung, d.h. die Position und die Richtung des Sensorkopfes bezogen auf Weltkoordinaten in Abhängigkeit von der Weltzeit.

Die so gewonnene Information über die Bewegung des Sensorkopfes kann benutzt werden, um eine Offline-Verarbeitung von anderen Sensordaten, wie z.B. den Kameradaten, durchzuführen. Alternativ besteht aber auch die Möglichkeit, diese Verarbeitung in Echtzeit auszuführen. In Abbildung 8.13 ist dies anhand der Kameradaten demonstriert. Aus der Orientierung des Sensorkopfes können Parameter für eine affine Transformation generiert werden, mit denen die Bilder der Kamera im FPGA transformiert werden. Das Er-

[9]Inertial Mesurement Unit, Sensor zur Messung von Beschleunigungen und Winkelgeschwindigkeiten [BEI05]

[10]Global Positioning System, satellitengestützter Sensor für Weltkoordinaten und Weltzeit [DH04, Par96]

[11]Sensor für Weltzeit über Radiofrequenz [PHB04]

[12]Lokales visuelles Trackingsystem [Adv02]

gebnis zeigt Abbildung 8.14. Auf der rechten Seite ist eine Visualisierung der Orientierung des Sensorkopfes dargestellt. Auf der linken Seite wurde das Ergebnis eines nicht in horizontaler Lage befindlichen Kamerabildes ausgegeben, das durch die affine Transformation in der Lage korrigiert wurde. Die Transformation zur Stabilisierung der Kameradaten ist jedoch nur in erster Näherung affin. Für genauere Transformationen sowie Einbeziehung der inneren Orientierung der Kamera müsste der Operator der affinen Transformation angepasst werden [Bör99].

Das Blockschaltbild der Abbildung 8.13 ist auf einem Virtex2-6000 realisiert worden. Dabei wurden die Operationen der Orientierungsschätzung und der Generation der affinen Parameter auf der PC-Plattform durchgeführt. Hierbei haben sich folgenden Projektkenngrößen ergeben:

- Anzahl Anwendungsmodule: 22
- Anzahl Betriebssystemaufrufe: 20
- Anzahl Bibliotheksmodule: 3
- Anzahl VHDL-Module inkl. Betriebssystem: 86
- Größe der VHDL-Module inkl. Betriebssytem: 29519 Zeilen (1,3 MB)
- Größe der vom Strukturcompiler erzeugten VHDL-Datei: 35000 Zeilen (1,4 MB)
- FPGA-Auslastung bei Virtex2-6000:
  - Slice: 26% (8953 von 33792)
  - Blockram: 15% (22 von 144)
  - IOB: 76% (628 von 824)
  - Multiplizierer: 2% (3 von 144)

Zurzeit finden die Orientierungsschätzung und die Berechnung der affinen Parameter noch auf dem PC statt. Hier ist aber vorgesehen, die Berechnung im LEON-Prozessor des FPGA ablaufen zu lassen. Es wird sich aber erst noch zeigen, ob der LEON-Prozessor genügend Rechenleistung aufbringt, um die teilweise sehr komplexen Berechnungen des Kalman-Filters durchführen zu können.

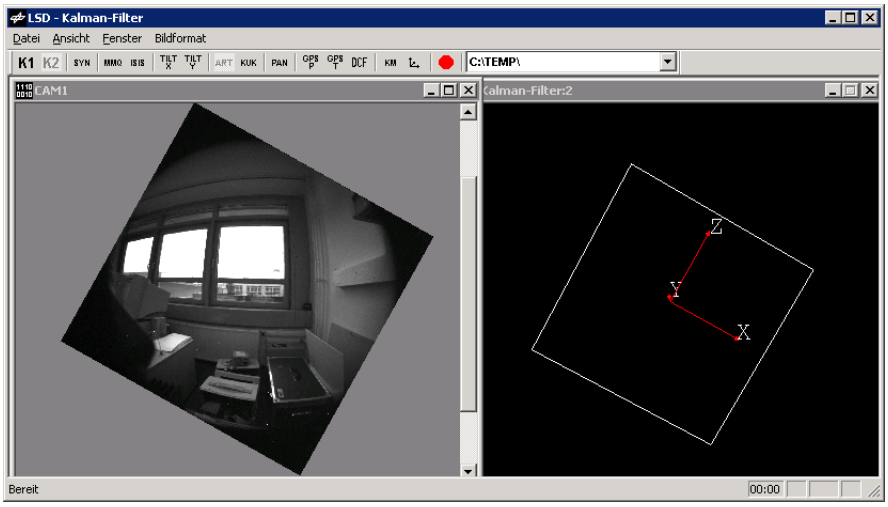

Abbildung 8.14: ModoS-GUI: Links das affin transformierte Kamerabild und rechts die Orientierung der Kamera aus der Orientierungsschätzung

Kapitel 9

Zusammenfassung und Ausblick

Die Wiederverwendbarkeit konfigurierbarer Hardware in Form von FPGA und FPGA-Karten stellt eine neue Herausforderung an den Entwurf dar. Die Nutzung der Vorteile, die durch extrem kurze Entwurfszyklen gegenüber anderen ASIC-Technologien bestehen, erfordert in besonderem Maße wiederverwendbaren Code zur Programmierung. Führende FPGA- und FPGA-Karten-Hersteller folgen diesem Trend und bieten proprietäre Lösungen dafür an.

In dieser Arbeit wurde gezeigt, dass die Benutzung eines Betriebssystems für Hardwarebeschreibungssprachen die Entwicklung von Programmen für konfigurierbare Hardware systematisiert und vereinfacht. Durch die Benutzung eines Betriebssystems und der damit verbundenen Wiederbenutzung von Betriebssystemmodulen reduziert sich die Entwicklungszeit, verbessert sich die Nachnutzbarkeit von Entwürfen, erhöht sich die Zuverlässigkeit von Programmen und vereinfacht sich die Partitionierung von Systemen.

In Kapitel 2 wurden die Grundlagen zum Verständnis der Arbeit beschrieben. Dazu zählt eine Einführung in die konfigurierbare Hardware sowie in die Hardwarebeschreibungssprache VHDL. In Kapitel 3 wurden verschiedene, den Stand der Technik entsprechende Methoden zum Entwurf von konfigurierbarer Hardware vorgestellt. Neben der weit verbreiteten Methode mit Hilfe einer Hardwarebeschreibungssprache wurden primär Methoden diskutiert, die Entwürfe für signalverarbeitende Systeme ermöglichen.

Die Entwicklung des Betriebssystemkonzepts erfolgt eingebettet in ein Gesamtkonzept für die Entwicklung von heterogenen Systemen. Als Basis der Entwurfsmethode für heterogene Systeme wurde in Kapitel 4 eine deklarative Beschreibung in Form von Signalflussgraphen erläutert. Für jeden möglichen Operator in diesem Signalflussgraphen gibt es mindestens zwei Repräsentationen: eine für die Umsetzung in konfigurierbarer Hardware und eine für die Umsetzung auf Prozessoren. Um die Umsetzung für möglichst verschiedene

Hardware-Plattformen zu realisieren, muss die Beschreibung der Operatoren unabhängig von der konkreten konfigurierbaren Hardware geschehen. Dies erfolgt hier durch Nutzung eines Betriebssystemkonzepts für die Hardwarebeschreibungssprache VHDL.

Bei der Umsetzung des Betriebssystemkonzepts zeigt sich, dass die Hardwarebeschreibungssprachen bestimmte Voraussetzungen nicht erfüllen. Zu diesen Voraussetzungen gehören die Möglichkeit der Datenkapselung und die Möglichkeit, mehrere Betriebssystemschnittstellen mit einem Betriebssystemmodul zu verbinden. Zur Lösung der Probleme wurden in Kapitel 5 verschiedene Transformationsschritte vorgestellt. Diese bewirken durch Umformungen der Hardwarebeschreibung eine Anbindung von Betriebssystemmodulen an die Anwendung. Die Transformationen sind dabei noch unabhängig von der Hardwarebeschreibungssprache.

Bevor eine Umsetzung dieser Transformationen für die Sprache VHDL erläutert wird, wurden in Kapitel 6 verschiedene Protokolle definiert, welche die Kanten für die verschiedenen Plattformen darstellen. Drei Arten von Kanten wurden definiert: Channels, SChannels und Pipes. Diese Kantendefinitionen besitzen verschiedene Eigenschaften bzgl. des Datendurchsatzes, der Komplexität und der Synchronisation zwischen Sender und Empfänger. Es wurde auch erläutert, wie diese Protokolle auf die verschiedenen physikalischen Verbindungen zwischen den Plattformen abgebildet werden müssen.

Ein Schwerpunkt dieser Arbeit lag in der Entwicklung des Strukturcompilers. Dieser setzt die vorher beschriebenen Transformationsschritte zur Unterstützung eines Betriebssystems für die Sprache VHDL um. Die Funktionsweise des Programms wurde in Kapitel 7 beschrieben. Der Strukturcompiler verbindet das Anwendungsprogramm mit den Betriebssystemmodulen und erzeugt daraus ein VHDL-Programm, das mit den typischen FPGA-Entwicklungswerkzeugen simuliert oder synthetisiert werden kann.

Zur Analyse und Verifikation des Betriebssystemkonzepts wurde ein Betriebssystem, d.h. eine Sammlung von Betriebssystemmodulen für verschiedenen FPGA-Karten sowie verschiedene Anwendungszwecke, entwickelt. Die Betriebssystemmodule deckten dabei einen Großteil der auf den FPGA-Karten vorhandenen Hardwarekomponenten ab, wie z.B. die verschiedenen Speicher (SRAM, SDRAM, SSRAM) und verschiedene Kommunikationscontroller (PCI, Ethernet, USB). In Kapitel 8 werden verschiedene wichtige Aspekt der Realisierung des Betriebssystems vorgestellt. Dazu zählt die Vorstellung der verwendeten FPGA-Karten, die Beschreibung der Umsetzung der Kommunikation zum PC und die Einbindung von FPGA-Prozessoren und Speichern. Neben den Projekten zur Verifikation der Betriebssystemmodule und des Gesamtkonzepts wurde das hier vorgestellte Konzept bereits in einigen Forschungsprojekten, wie z.B. ModoS, angewandt.

Bei der Umsetzung dieser verschiedenen Projekte stellte sich heraus, dass auf der Anwendungsschicht die Entwicklung und Verifikation von Entwürfen durch das Betriebssystemkonzept stark vereinfacht wird. Gerade im Bereich des Prototypings, wo der Entwurf ständigen Änderungen unterworfen ist, bringt das Betriebssystemkonzept seine Vorteile zur Geltung. Auch die Realisierung von Projekten auf verschiedenen FPGA-Karten ist mit Hilfe des Betriebssystemkonzepts und des Strukturcompilers leicht umsetzbar.

Der Schwerpunkt der bisherigen Betriebssystemmodule und der Beispielprojekte lag in der Anbindung der verschiedenen externen Hardwarekomponenten. Es existieren jedoch bereits einige Beispiele für Operatoren der Signalverarbeitung, die auch die Vorteile des Betriebssystemkonzepts in Verbindung mit der Bildverarbeitung zeigen. Der Fokus zukünftiger Entwicklungen wird daher verstärkt in der Entwicklung weiterer Operatoren der Signalverarbeitung liegen.

Es hat sich gezeigt, dass trotz aller Einschränkungen des Strukturcompilers das Betriebssystemkonzept umsetzbar ist. Teilweise führen die Einschränkungen jedoch zu einer komplexeren Konfiguration. Ein weiterer wichtiger Aspekt in der zukünftigen Entwicklung wird daher in der Verbesserung des Strukturcompilers liegen.

Ein weiterer Schwerpunkt wird in der Anbindung des Strukturcompilers in eine Entwicklungsumgebung liegen, die eine Modellierung eines heterogenen Systems ermöglicht. So hat sich gezeigt, dass mit Hilfe von freeSp eine entsprechende Integration möglich ist. Hier ist jedoch eine Entwicklung in Richtung einer graphischen Oberfläche und einfachen Benutzersteuerung zur Beschreibung der Modellierung, der Paritionierung, der Simulation und der Synthese von heterogenen Gesamtsystemen wünschenswert.

Das hier vorgestellte Konzept eines Betriebssystems für konfigurierbare Hardware ordnet sich mit seinen Eigenschaften in das Forschungsgebiet des Hardware-Software-Codesigns ein und wird in zukünftigen Entwicklungsmethoden für heterogene Systeme ein wichtiger Bestandteil sein.

Literaturverzeichnis

[Act06] Actel: *ProASIC3 Flash Family FPGAs*. 1. 2006

[Adv02] Advanced Realtime Tracking GmbH: *ARTrack1 & DTrack*. 1. 2002

[ARM99] ARM: *AMBA Specification*. 1. 1999

[Ash98] Ashenden, P. J.: *The Designer's Guide to VHDL*. 1. Morgan Kaufman Publishers, 1998

[ASU99] Aho, V. A. ; Sethi, R. ; Ullmann, J. D.: *Compilerbau Teil 1 u. 2*. zweite. München : Oldenburg Verlag, 1999

[Au99] Au, M. T.: *JAVA2C Translator*, Monash University, Victoria, Australia, Diplomarbeit, 1999

[Bal97] Balarin, F.: *Hardware-Software Co-Design of Embedded Systems - The POLIS approach*. 1. Kluwer Academic Publishers, Mai 1997

[BBG+04] Briess, K. ; Bärwald, W. ; Gill, E. ; Kayal, H. ; Montenbruck, O. ; Montenegro, S. ; Halle, W. ; Skrbek, W. ; Studemund, H. ; Terzibaschian, T. ; Venus, H.: Technology Demonstration by the BIRD-Mission. In: *Acta Astronautica* 56 (2005) (2004), Nr. 1-2, S. 57 – 63. – LIDO-Berichtsjahr=2004,;

[BBS+98] Benl, H. ; Berger, U. ; Schwichtenberg, H. ; Seisenberger, M. ; Zuber, W.: Proof Theory at Work: Program Development in the Minlog System. In: Bibel, W. (Hrsg.) ; Schmidt, P. H. (Hrsg.): *Automated Deduction: A Basis for Applications. , Systems and Implementation Techniques* Bd. 2. Dordrecht : Kluwer Academic Publishers, 1998

[BEI05] BEI Technologies, Inc: *MMQ User's Guide*. 1. 2005

[BH98] Bellows, P. ; Hutchings, B.: JHDL - an HDL for Reconfigurable Systems. In: Pocek, K. (Hrsg.) ; Arnold, J. (Hrsg.): *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*. Napa : IEEE Computer Society Press, April 1998, S. 175–184

[Bör99] Börner, A.: *Entwicklung und Test von Onboard-Algorithmen für die Landfernerkundung*. Berlin, Institut für Weltraumsensorik und Planetenerkundung, Diss., 1999

[Car05] Carlsson, K.: *Design and Implementation of an On-Chip Logic Analyzer*, Chalmers University of Technology, Diplomarbeit, 2005

[Cela] Celoxica Ltd.: *RC1000-PP Function Reference Manual*. 1

[Celb] Celoxica Ltd.: *RC1000-PP Hardware Reference Manual*. 1

[Cel02] Celoxica Ltd.: *Handel-C for Hardware Design*. 1. 2002

[Cha03] Chapman, K.: PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices / Xilinx, Inc. 2003. – Forschungsbericht. Application Note 213

[CPSG99] Culler, D. ; Pal Singh, J. ; Gupta, A.: *Parallel Computer Architecture - A Hardware/Software Approach*. 1. Morgan Kaufman Publishers, 1999

[CZNJ03] Czerner, F. ; Zellmann, J. ; Neuhäuser, U. ; Jokhovets, L.: Design Methodology for Signal Processing and Machine Vision Applications. In: *GSPx 2003 Conference Paper*, 2003

[Det97] Dettmer, T.: VHDL'87 yacc Grammatik / Universität Hamburg. 1997. – Forschungsbericht. Beispielcode

[DH04] Dodel, H. ; Häupler, D.: *Satellitennavigation — GALILEO, GPS, GLONASS, Integrierte Verfahren*. 1. Hüthig, 2004

[DRK+03] Dalaff, C. ; Reulke, R. ; Kroen, A. ; Ruhé, M. ; Schischmanow, A. ; Schlotzhauer, G. ; Tuchscherer, W. ; Kahl, T.: A Traffic Object Detection System for Road Traffic Measurement and Management. In: Bailey, D. G. (Hrsg.): *IVCNZ Image + Vision comPuting 03*. Palmerston North, Neuseeland : Institute of Information Sciences and Technology (Massey University), November 2003, S. 78–83

[DV01] Daggu, V. R. ; Venkatesan, M. *Design and Implementation of an Efficient Reconfigurable Architecture for Image Processing Algorithms using Handel-C*. http://www.celoxica.com. 2001

[DW03] Dueffert, U. ; Winkler, F.: Evaluation of the Capabilities of Handel-C considering the Implementation of a Classification Algorithm as Example / Humboldt-Universität. 2003. – Forschungsbericht. Report

[DZ99] Dorobantu, R. ; Zebhauser, B.: Field Evaluation of a Low-Cost Strapdown IMU by means GPS. In: *Ortung und Navigation* 1 (1999), S. 51–65

[Fle04] Fleischmann, H.: *VHDL-Entwurf für ein FPGA-basiertes System zum Echtzeit Cluster-Labelling von Bilddaten*, FH Nordhausen, Diplomarbeit, 2004

[FSW00] Fricker, P. ; Sandau, R. ; Walker, A. S.: Progress in the Development of a High Performance Airborne Digital Sensor. In: *Photogrammetric Record*, 2000, S. 911–927

[Gai] Gaisler Research: *LEON2 Processor User's Manual V1.0.22*. 1

[Gai04a] Gaisler, J.: A Dual-Use Open-Source VHDL IP Library. In: *Proceedings of the MAPLD International Conference*, 2004

[Gai04b] Gaisler Research: *GRMON User's Manual*. 1. 2004

[Gas02] Gasiler, J.: Fault Tolerant Microprocessors for Space applications. In: *International Conference on Dependable Systems and Networks (DSN'02)*, 2002

[GHC05] Gaisler, J. ; Habinc, S. ; Catovic, E.: GRLIB IP Library User's Manual / Gaisler Research. 2005. – Forschungsbericht. Manual

[GLD+03] Gupta, S. ; Luthra, M. ; Dutt, N.l ; Gupta, R. ; Nicolau, A. *Hardware and Interface Synthesis of FPGA Blocks using Parallelizing Code Transformations*. http://mesl.ucsd.edu. 2003

[Gro01] Grout, I. A.: Modeling, Simulation and Synthesis: From Simulink to VHDL generated Hardware / Department of Electronic and Computer Engineering, University of Limerick, Ireland. 2001. – Forschungsbericht. Paper

[HBK96] Hartenstein, R. W. ; Becker, J. ; Kress, R.: Two-Level Hardware/Software Partitioning Using CoDe-X. In: *IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96)*, 1996

[HJP03] Halle, W. ; Jahn, H. ; Pleikies, J.: An FPGA Implementation of Road-Traffic Image Observation. In: Kern, H. (Hrsg.): *Tagungsband des 48. Internationales Wissenschaftliche Kolloquium*. Ilmenau : Technische Universität Ilmenau, 2003, S. 111–112

[HLL+03] Hylands, C. ; Lee, E. ; Liu, J. ; Liu, X. ; Neuendorffer, S. ; Xiong, Y. ; Zhao, Y. ; Zheng, H.: Overview of the Ptolemy Project / University of California, Berkeley, Engineering-Electrical Engineering and Computer Sciences. 2003. – Forschungsbericht. UCB/ERL M03/25

[htt06] http://www.freehdl.seul.org: *The FreeHDL project*. 1. 2006

[IBM99] IBM: *The CoreConnect Bus Architecture*. 1. 1999

[Ins] The Institute of Electrical and Electronics Engineers: *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*. 1

[Ins00] The Institute of Electrical and Electronics Engineers: *IEEE Standard VHDL Language Reference Manual*. 1. 2000

[Ins03] The Institute of Electrical and Electronics Engineers: *IEEE P1076.1.1/D1 Draft Standard for Standard VHDL Analog and Mixed-Signal Extension-Packages for Multiple Energy Domain Support*. 1. 2003

[Iso04] Isomäki, M.: *Processor Debugging Through Ethernet*. Göteburg, Chalmers University of Technology, Diplomarbeit, 2004

[JB03] Jungmann, M. ; Beine, M.: Automatic Code Generation for Safety-Critical Systems. In: *Automotive Electronics* (2003), Nr. 09

[JHD04] http://www.jhdl.org/documentation/starter.html: *JHDL — Getting started*. 1. 2004

[KP03] Kraus, O. ; Padeffke, M.: Synthese von asynchronen "Burst-Mode" Automaten. In: *11. E.I.S.-Workshop*, 2003

[Kru02] Krutz, D.: *Konzeption einer hierarchischen VHDL - Strukturgenerierung zur Realisierung von abstrakten Hardwareschnittstellen.* Berlin, Humboldt-Universität, Diplomarbeit, 2002

[Kru05] Krutz, D.: Project.X Reference Manual / DLR. 2005. – Forschungsbericht. PC-API Beschreibung

[Kru06a] Krutz, D.: Beschreibung der Schnittstellen, Module und Bibliotheksmodule für Project.X / DLR. 2006. – Forschungsbericht. FPGA-API Beschreibung

[Kru06b] Krutz, D.: Beschreibung der Speichermodule für Project.X / DLR. 2006. – Forschungsbericht. FPGA-Entwurfsbeschreibung

[KW03] Krutz, D. ; Winkler, F.: Konzept eines auf VHDL basierenden Betriebssystems für FPGA-Plattformen. In: *11. E.I.S.-Workshop*, 2003

[Liu98] Liu, J.: Continous Time and Mixed-signal Simulation in Ptolemy II / University of California, Berkeley, CA. 1998. – Forschungsbericht. UCB/ERL Memorandum M98/74

[LM87] Lee, E. A. ; Messerschmitt, D.: Synchronous Dataflow. In: *Proceeding IEEE* Bd. 75, 1987, S. 1235–1245

[LNW03] Lee, E. A. ; Neuendorfer, S. ; Wirthlin, M. J.: Actor-oriented Design of Embedded Hardware and Software Systems. In: *Journal of Circuits, Systems and Computers* 12 (2003), Nr. 3, S. 231–260

[LWS94] Lehmann, G. ; Wunder, B. ; Selz, M.: *Schaltungsdesign mit VHDL.* 1. Franzis-Verlag, 1994

[Mat01] Matschnig, M.: *Parallele VHDL Simulation mit einem Standard Hardwaresimulator*, Technische Universität Wien, Diplomarbeit, 2001

[MBG94] Müller, W. ; Börger, E. ; Gläsßer, U.: The Semantics of Behavioral VHDL '93 Descriptions. In: *EURO-DAC '94: Proceedings of the conference on European design automation.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1994, S. 500–505

[Men03] Mentor Graphics: *ModelSim Designer Datasheet.* 1. 2003

[Men05] Mentor Graphics: *FPGA Advantage Bookcase*. 1. 2005

[Odd06] Oddoy, T.: *Anbindung einer FPGA-Plattform an einen PC mittels USB-Schnittstelle unter Benutzung eines VHDL-Betriebssystems*. Berlin, Fachhochschule für Technik und Wirtschaft, Diplomarbeit, 2006

[On-98] On-Line Applications Research Corporation: *RTEMS C User's Guide*. 1. 1998

[OPE02] OPENCORES.ORG: *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. 1. September 2002

[Ope06] http://www.opencores.org: *OPENCORES.ORG - Free Open Source IP and Chip Design*. 1. 2006

[PA00] PULNix America, Inc: *Specification of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers*. 1. : Camera Link commitee, Oktober 2000

[Par96] Parkinson, B. P.: Global Positioning System: Theory and Applications. In: *American Institute of Aeronautics and Astonautics* 1 (1996)

[PHB04] Piester, D. ; Hetzel, P. ; Bauch, A.: Zeit- und Normalfrequenzverbreitung mit DCF77. In: *PTB-Mittelungen* 114 (2004), Nr. 4

[Pro04] http://www.prodesign-europe.com: *ImageIt VISIONCreator2*. 1. 2004

[Pro06] http://www.prodesign-europe.com: *ProDesign*. 1. 2006

[Red03] Red Hat, Inc: *eCos User Guide*. 1. 2003

[RGMG04] Reichmann, C. ; Gebauer, D. ; Müller-Glaser, K.D.: Model Level Coupling of Heterogeneous Embedded Systems. In: *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, 2004

[Rät05] Rätzel, M.: *Entwurf einer allgemeinen Speicherschnittstelle für FIFO-gestützte Videodatenerfassung auf FPGA-Boards*. Berlin, Humboldt-Universität, Diplomarbeit, 2005

[SB01] Stöcklein, T. ; Bäsig, J. *Handel-C an Effective Method for Designing FPGAs (and ASICs).* http://www.celoxica.com. 2001

[Sin94] Sinander, P.: VHDL Modelling Guidelines / European Space Research and Technology centre. 1994. – Forschungsbericht. applicable document for ESA developments involving VHDL modelling

[SKR+01] Scheibe, K. ; Korsitzky, H. ; Reulke, R. ; Scheele, M. ; Solbrig, M.: EYESCAN - A High Resolution Digital Panoramic Camera. In: *RobVis '01: Proceedings of the International Workshop on Robot Vision.* London, UK : Springer-Verlag, 2001, S. 77–83

[SPA92] SPARC International, Inc: *The SPARC Architecture Manual Version 8.* 1. 1992

[SR03] Schwarz, B. ; Reichardt, J.: *VHDL-Synthese.* 1. OLDENBOURG, Dezember 2003

[Sup03] Suppa, M.: ModoS / DLR. 2003. – Forschungsbericht. Projektstatus

[Tan92] Tanenbaum, A. S.: *Modern Operating Systems.* 2. Englewood Cliffs : Prentice Hall, 1992

[Tei97] Teich, J.: *Digitale Hardware/Software-Systeme.* 1. Springer-Verlag, 1997

[Tsa00] Tsay, J.: A Code Generation Framework for Ptolemy II / EECS Department, University of California, Berkeley. 2000 (UCB/ERL M00/25). – Forschungsbericht. UCB/ERL M00/25

[VCA94] Vemuri, R. ; Carter, H. ; Alexander, P.: Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment. In: *First Annual RASSP Conference*, 1994

[W3C] W3C. *XSL Transformations (XSLT).* http://www.w3.org/TR/xslt

[Wei] Weiss, A. *Free Signal Processing: freeSP.* http://www.freeSP.de

[Wil98] Williamson, M. C.: *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*, University of California, Berkeley, Engineering-Electrical Engineering and Computer Sciences, Diss., 1998

[WT85] Walker, R. ; Thomas, D.: Model of Design Representation and Synthesis. In: *22nd Design Automation Conference*, 1985

[WW06] Weiss, A. ; Winkler, F.: Entwurf Global Asynchroner Lokal Synchroner Strukturen auf der Basis einer deklarativen Beschreibung mit XML. In: *9. GMM Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2006

[Xil01] Xilinx, Inc.: *Virtex II Plattform FPGA Handbook*. 1. 2001

[Xil03] Xilinx, Inc.: *VirtexIIPro Platform FPGA: Complete Data Sheet*. 1. 2003

[Xil05a] Xilinx, Inc.: *ChipScope Pro - Software and Cores User Guide*. 1. 2005

[Xil05b] Xilinx, Inc.: *ISE 7 In-Depth Tutorial*. 1. 2005

[Zar04] Zarchan, P. ; Titterton, D.H. (Hrsg.) ; Weston, J.L. (Hrsg.): *Strapdown Inertial Navigation Technology*. Bd. 207. 1. Progress in Astronautics and Aeronautics, 2004

Anhang A

Basismodell für SChannel Zustandsmaschine

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.basic_package.all;

entity receiver is
    port (
        ...

        input_schannel_clock        : in  std_logic;
        input_schannel_data         : in
                        std_logic_vector(17 downto 0);
        input_schannel_data_valid   : in  valid_type;
        input_schannel_busy         : out valid_type

        ...
    );
end entity;

library IEEE;
use IEEE.std_logic_1164.all;
use work.basic_package.all;

architecture receiver_arch of receiver is

    ...
    component reset
        port (
            reset: out reset_type
        );
    end component;
```

```
    component afifo
        generic (
            LOG2DEEP: integer;
            WIDTH: integer
        );
        port (
            reset      : in reset_type;
            write_clock: in std_logic;
            write_enable: in valid_type;
            write_data : in
                    std_logic_vector(WIDTH-1 downto 0);
            read_clock : in std_logic;
            read_enable : in valid_type;
            read_data  : out
                    std_logic_vector(WIDTH-1 downto 0);
            full_flag: out valid_type;
            almost_full_flag: out valid_type;
            almost_empty_flag: out valid_type;
            empty_flag: out valid_type;
            read_count       : out std_logic_vector
                               (LOG2DEEP downto 0);
            write_count      : out std_logic_vector
                               (LOG2DEEP downto 0)
        );
    end component;

    signal afifo_read_enable: valid_type;
    signal afifo_read_data: std_logic_vector(17 downto 0);
    signal afifo_full_flag: valid_type;
    signal afifo_almost_empty_flag: valid_type;
    signal afifo_empty_flag: valid_type;
    signal afifo_read_count: std_logic_vector(2 downto 0);
    signal afifo_write_count: std_logic_vector(2 downto 0);

    signal afifo_preload_reg, next_afifo_preload_reg: boolean;

    signal reset: reset_type;

    ...

begin
    I_reset: reset port map (reset=>reset);
```

```
I_afifo:afifo
    generic map (
        LOG2DEEP => 2, -- minimal size for afifo
        WIDTH => 18
    )
    port map (
        reset               =>  reset,
        write_clock         =>  input_schannel_clock,
        write_enable        =>  input_schannel_data_valid,
        write_data          =>  input_schannel_data,
        read_clock          =>  clock,
        read_enable         =>  afifo_read_enable,
        read_data           =>  afifo_read_data,
        full_flag           =>  afifo_full_flag,
        almost_full_flag    =>  input_schannel_busy,
        almost_empty_flag   =>  afifo_almost_empty_flag,
        empty_flag          =>  afifo_empty_flag,
        read_count          =>  afifo_read_count,
        write_count         =>  afifo_write_count
    );

PS:process( afifo_preload_reg, afifo_read_data,
            afifo_empty_flag, ...)
    variable get_next_fifo_entry: boolean;
begin
    next_afifo_preload_reg<=afifo_preload_reg;

    afifo_read_enable<=invalid;

    get_next_fifo_entry:=false;

    -- now state handling
    ...


    -- in case of completeness of fifo data handling
    -- set get_next_fifo_entry to true to
    -- get next data word if possible
    get_next_fifo_entry:=true;


    -- to check if afifo_read_data has valid data word
    -- check afifo_preload_reg for true
```

```
        if (afifo_preload_reg=true) then
            -- new data entry is afifo_read_data
        end if;


        -- to check if no more entries
        if (afifo_preload_reg=false and
            afifo_empty_flag=valid) then
            -- no more data for operating
        end if;


        ...

        -- handle afifo operating
        if (afifo_preload_reg) then
            if (get_next_fifo_entry) then
                if (afifo_empty_flag=invalid) then
                    afifo_read_enable<=valid;
                else
                    next_afifo_preload_reg<=false;
                end if;
            end if;
        else
            if (afifo_empty_flag=invalid) then
                next_afifo_preload_reg<=true;
                afifo_read_enable<=valid;
            end if;
        end if;
    end process;

    PC:process(clock,reset)
    begin
        if (reset=reset_active) then
            afifo_preload_reg<=false;
            ...
        elsif (clock'event and clock='1') then
            afifo_preload_reg<=next_afifo_preload_reg;
        end if;
    end process;
    ...
end;
```

Anhang B

Multiplexerbeispiel

Mux.vhdl

```
entity Mux is
end;

library IEEE;
use IEEE.std_logic_1164.all;
use work.basic_package.all;

architecture Mux_arch of Mux is

    component channel_input
        generic (
            WIDTH: CHANNEL_WIDTH_TYPE;
            ID: CHANNEL_ID_TYPE;
            INIT: integer
        );
        port (
            data : out std_logic_vector(WIDTH-1 downto 0)
        );
    end component;

    component channel_output
        generic (
            WIDTH: CHANNEL_WIDTH_TYPE;
            ID: CHANNEL_ID_TYPE
        );
        port (
            data : in std_logic_vector(WIDTH-1 downto 0)
        );
    end component;
```

```
    signal input1: std_logic_vector(7 downto 0);
    signal input2: std_logic_vector(7 downto 0);
    signal output: std_logic_vector(7 downto 0);
    signal sel: std_logic_vector(0 downto 0);
begin
    I_input1: channel_input
        generic map (
            WIDTH       => 8,
            ID          => 1,
            INIT        => 0
        )
        port map (
            data        =>  input1
        );

    I_input2: channel_input
        generic map (
            WIDTH       => 8,
            ID          => 2,
            INIT        => 0
        )
        port map (
            data        =>  input2
        );

    I_output: channel_output
        generic map (
            WIDTH       => 8,
            ID          => 1
        )
        port map (
            data        =>  output
        );

    I_sel: channel_input
        generic map (
            WIDTH       => 1,
            ID          => 5,
            INIT        => 0
        )
        port map (
            data        =>  sel
```

```
        );

    -- mux
    output<=input1 when sel="0" else input2;

end;
```

project.cfg

```
SOURCE FILES      =   Mux.vhdl;
TOPLEVEL NAME     =   Mux;

MAIN ENTITY NAME  = main_entity;
MAIN ARCHITECTURE NAME
                  = main_architecture;

sim
{
    OUTPUT FILE         = sim/Mux.vhd;

    channel_input       = sim_channel_input;
    channel_output      = sim_channel_output;
};

ois1000
{
        OUTPUT FILE         = ois1000/Mux.vhd;
        UCF FILE            = ois1000/Mux.ucf;

        channel_input       = ois_pci_channel_input;
        channel_output      = ois_pci_channel_output;
        ois_card            = ois1000_card;
};

ois3000
{
        OUTPUT FILE         = ois3000/Mux.vhd;
        UCF FILE            = ois3000/Mux.ucf;

        channel_input       = ois_pci_channel_input;
        channel_output      = ois_pci_channel_output;
        ois_card            = ois3000_card;
```

```
};
```

Anhang C

Liste der Betriebssystemmodule

In der nachfolgenden Tabelle sind alle Betriebssystemmodule aufgelistet, die zurzeit existieren und allgemein benutzbar sind. Daneben gibt es noch eine Reihe von Betriebssystemschnittstellen und Betriebssystemmodulen, die jedoch speziell für eine Plattform entwickelt wurden und nur von den hier aufgelisteten Modulen genutzt werden. Eine detaillierte Auflistung und Erklärung der Betriebssystemschnittstellen und -module ist in [Kru06a] angegeben.

Schnittstelle	Modul	Plattform
Clock_if		
	sim_clock	Sim
	ois_clock	OISxxxx
	pp1000_clock	PP1000
Reset_if		
	sim_reset	Sim
	ois_reset	OISxxxx
	pp1000_reset	PP1000
Time_if		
	sim_time	Sim
	ois_time	OISxxxx
	pp1000_time	PP1000
Channel_input_if		
	sim_channelinput	Sim
	ois_pci_channel_input	OISxxxx
	ois_usb_channel_input	OISxxxx

ois_eth_channel_input	OISxxxx
pp1000_pci_channel_input	PP1000
sim_leon_channel_input	Sim
leon_channel_input	OISxxxx,PP1000

Channel_output_if

sim_channeloutput	Sim
ois_pci_channel_output	OISxxxx
ois_usb_channel_output	OISxxxx
ois_eth_channel_output	OISxxxx
pp1000_pci_channel_output	PP1000
sim_leon_channel_output	Sim
leon_channel_output	OISxxxx,PP1000
ois_led	OISxxxx
pp1000_led	PP1000

SChannel_input_if

sim_schannel_input	Sim
ois_pci_schannel_input	OISxxxx
ois_usb_schannel_input	OISxxxx
ois_eth_schannel_input	OISxxxx
pp1000_pci_schannel_input	PP1000
sim_leon_schannel_input	Sim
leon_schannel_input	LEON
sim_leon_amba_io_input	Sim
leon_amba_io_input	LEON
sim_leon_amba_debug_input	Sim
leon_amba_debug_input	LEON

SChannel_output_if

sim_schanneloutput	Sim
ois_pci_schannel_output	OISxxxx
ois_usb_schannel_output	OISxxxx
ois_eth_schannel_output	OISxxxx
pp1000_pci_schannel_output	PP1000
sim_leon_schannel_output	Sim
leon_schannel_output	LEON
sim_leon_amba_io_output	Sim
leon_amba_io_output	LEON
sim_leon_amba_debug_output	Sim
leon_amba_debug_output	LEON

Pipe_input_if		
	sim_pipeinput	Sim
	ois_pci_pipe_input	OISxxxx
	ois_usb_pipe_input	OISxxxx
	ois_eth_pipe_input	OISxxxx
	pp1000_pci_pipe_input	PP1000
Pipe_output_if		
	sim_pipeoutput	Sim
	ois_pci_pipe_output	OISxxxx
	ois_usb_pipe_output	OISxxxx
	ois_eth_pipe_output	OISxxxx
	pp1000_pci_pipe_output	PP1000
	sim_leon_amba_pipe_rom	Sim
	leon_amba_pipe_rom	LEON
Memory_if		
	sim_memory	Sim
	sim_sdram1_memory	Sim
	sim_sdram2_memory	Sim
	sim_sdram3_memory	Sim
	sim_ssram1_memory	Sim
	ois_sdram1_memory	OISxxxx
	ois_sdram2_memory	OISxxxx
	ois_sdram3_memory	OISxxxx
	ois_ssram1_memory	OISxxxx
	dram_memory	PP1000,OISxxxx
	pp1000_sram_memory	PP1000
Framegrabber_if		
	sim_framegrabber	Sim
	ois_cameralink_framegrabber	OISxxxx
Rs232_input_if		
	sim_rs232_input	Sim
	ois_rs232_input	OISxxxx
Rs232_output_if		
	sim_rs232_output	Sim
	ois_rs232_output	OISxxxx

Rom_if		
	sim_leon_amba_rom	Sim
	leon_amba_rom	LEON

Anhang D

Liste der Bibliotheksmodule

In der nachfolgenden Tabelle sind alle Bibliotheksmodule aufgelistet, die zurzeit existieren. Eine detaillierte Auflistung und Erklärung der Bibliotheksmodule ist in [Kru06a] angegeben.

Modulname	Beschreibung
addtimestamp	Hinzufügen von Zeitinformation zu Frames in Pipe
affinetransformation	Affine Transformation von Pipes
bgreplacer	Hintergrundersetzer
cameragrabber	Einlesen und Zwischenspeichern von Kameradaten
cameralinksync	Synchronisation von CameraLink-Kameras
delay	Zwischenspeicher für Pipies
demuxer	Demuxer für Pipes
diffthreshold	Differenzen-Schwellwertberechnung für Pipes
framesize	Statusinformationen und Konsistenz-Checker von Frames in Pipes
jpeg_grey_encoder	JPEG-Encoder für 8 bit Grauwertbilder
jpegls_grey_encoder	JPEGLS-Encoder für 8 bit Grauwertbilder
logging	Logging Modul für In-Circuit Verifikation
muxer	Muxer für Pipes
patterngenerator	Erzeugt Patternwerte nach CRC-32 Berechnung
pixelmuxer	Muxer für Pixel von Pipes
repeater	Wiederholelement für Pipes
resize	Größenanpassung von Bilddaten in Pipes
pipe_splitter	Splitter für Pipes
threshold	Schwellwertberechnung für Pipes

Abkürzungen

Abkürzung	Erklärung
AMBA	Advanced Microcontroller Bus Architecture
AMS	Analog and Mixed Signals
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced RISC Machines Ltd.
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input Output System
BV	Bildverarbeitung
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CT	Continous Time
DLL	Dynamic Link Library
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
DMA	Direct Memory Access
DSP	Digital Signal Processor
DSU	Debug Support Unit
EDIF	Electronic Data Interchange Format
EDN	Electonic Data interchange Format Netlist
EEPROM	Electrically Erasable Programmable Read Only Memory
ESA	European Space Agency
FIFO	First in - First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPS	Global Positiong System
GUI	Graphical User Interface
HDL	Hardware Description Language
IC	Integrate Circuit
ID	Identifier

IDL	Interactive Data Language
IEEE	Institute of Electrical and Electronics Engineers
IMU	Inertial Mesurement Unit
IOB	Input/Output Blocks
IP	Intellectual property
IRQ	Interrupt Request
I/O	Input/Output
JHDL	Java Hardware Descrition Language
JTAG	Joint Test Action Group
ModoS	Multisensorielle Modellierung mittels referenzierter optische Sensoren
OIS	Optische Informationssysteme zur Verkehrsszenenanalyse und Verkehrslenkung
PAR	Place-And-Route
PC	Personal Computer
PCI	Peripheral Component Interconnect
PLD	Programmable Logic Device
POSIX	Portable Operating System Interface for UniX
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDF	Synchronous Dataflow
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
SSRAM	Synchronous Static Random Access Memory
STL	Standard Template Library
TTL	Transistor-Transistor-Logik
UCF	User Constraint File
UMA	Unified Memory Access
UML	Unified Modeling Language
UMR	Universal Modul Register
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Danksagung

An dieser Stelle möchte ich allen Personen herzlich danken, die am Zustandekommen dieser Arbeit beigetragen haben. Mein besonderer Dank gilt:

- Frau Prof. Dr. Beate Meffert für ihre Gutachtertätigkeit, das Interesse und die Förderung meiner Arbeit,
- Frank Winkler für sein Hilfe bei allen Fragen zur Beschreibung von konfigurierbarer Hardware,
- Anko Börner und Andreas Eckhard für die Unterstützung bei dem Thema,
- Holger Venus für die Hilfe bei der Inbetriebnahme der verschiedenen FPGA-Karten und den vielen Diskussionen über FPGA und VHDL,
- meinen Promotionsbrüdern Karsten Scheibe, Denis Grießbach und Adrian Schischmanow für die vielen ergiebigen Diskussionen in den Kaffeerunden,
- den Mitarbeiterinnen und Mitarbeitern der Abteilung Optische Informationssysteme am Deutschen Zentrum für Luft- und Raumfahrt für das sehr kollegiale und angenehme Arbeitsumfeld,
- und natürlich meinen lieben Eltern und meiner Schwester.

Selbständigkeitserklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „Ein Betriebssystem für konfigurierbare Hardware" selbstständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Berlin, den 03.07.2006