Beitrag zum modellbasierten Entwurf eingebetteter Systeme

Komponentenbasierte Modellierungsansätze für verteilte rekonfigurierbare Plattformen

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der
Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von
Diplominformatiker Marcus Müller
geboren am 27.03.1980 in Neuhaus/Rwg.

vorgelegt am:	07.01.2013
Tag der wissenschaftlichen Aussprache:	19.11.2013

1. Gutachter:	Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler
2. Gutachter:	Univ.-Prof. Dr.-Ing. habil. Christoph Ament
3. Gutachter:	Prof. Dr.-Ing. Volker Zerbe

urn:nbn:de:gbv:ilm1-2013000699
ISBN: 978-3-938843-80-2

Danksagung

An dieser Stelle möchte ich als Autor meinen Dank aussprechen. Zuerst danke ich Professor Wolfgang Fengler für die Möglichkeit, am Fachgebiet Rechnerarchitektur & Eingebettete Systeme der TU Ilmenau tätig sein und promovieren zu können. Für die großartige Gelegenheit, an der Arbeit im Sonderforschungsbereich 622 beteiligt sein zu dürfen, möchte ich außerdem die Professoren Gerd Jäger und Eberhard Manske würdigen.

Desweiteren bedanke ich mich bei meinen Kollegen Dr. Bernd Däne, Dr. Alexander Pacholik, Dr. Arvid Amthor, Torsten Machleidt, Johannes Klöckner, Irina Gushtchina, Hans-Christian Schwannecke und Stephan Zschäck, sowie bei Prof. Christoph Ament und Prof. Volker Zerbe für die ausgezeichnete Zusammenarbeit im Fachgebiet und im SFB, viele fruchtbare gemeinsame Projekte und Diskussionen, und nicht zuletzt für ihren Beistand und ihre Unterstützung für dieses Vorhaben.

Schließlich seien mit Oliver Brandel, Alexander Krahn, Johannes Both und Folker Schwesinger die Menschen gewürdigt, die ich während ihrer Studienzeit bei Projekt- und Abschlussarbeiten im Rahmen meiner Promotionstätigkeit betreut habe und die ihren Beitrag zu den Ergebnissen dieser Arbeit geleistet haben.

Zuletzt möchte ich meinen persönlichen Dank aussprechen - meiner Familie, die mich im Verlauf der Arbeit vor allem moralisch unterstützt hat, und meinem lieben Kollegen Heiner Kotula, der mir immer zeigen wollte, wie wichtig das Leben jenseits der Wissenschaft ist, und der das Ende meiner Promotion nicht mehr miterleben konnte.

Vorwort

Diese Arbeit entstand an der Technischen Universität Ilmenau vor dem Hintergrund des von 2002 bis 2013 von der Deutschen Forschungsgemeinschaft geförderten Sonderforschungsbereiches 622 „Nanopositionier- und Nanomessmaschinen“ (kurz NPMM), in dessen Rahmen das Fachgebiet Rechnerarchitektur mit dem Teilprojekt C1 „Hochleistungsinformationsverarbeitung mit eingebetteten Systemen“ betraut war. Neben der Erstellung von spezifischen eingebetteten Hardware- und Software-Lösungen, die den Aufgabenstellungen aus dem Bereich der Hochpräzisionsmesstechnik, der Regelungstechnik und der Bilddatenverarbeitung gerecht werden sollten, stand die Erarbeitung von modellbasierten Methoden zum Entwurf und der systematischen Entwicklung von eingebetteten Systemen im Vordergrund. Die Entwicklung solcher Systeme unter den hohen Leistungsanforderungen der Domäne der Nanomesstechnik bedeutet für die Forschung eine Gratwanderung zwischen hochleistungsfähigen Speziallösungen, Strukturierung und Systematisierung, und Methodenintegration und Automatisierung für modellbasierte Prozesse zur Beherrschung der Entwurfskomplexität. Die Einleitung zu dieser Arbeit gibt einen kurzen Überblick über die Herausforderungen und die Geschichte der eingebetteten Systementwicklung im Rahmen des SFB 622 als Motivation zu dieser Arbeit.

In der letzten Projektperiode von 2009 bis 2013, in der ich selbst Teil des SFB 622 war, traten Hardware-Plattformen auf der Basis rekonfigurierbarer Logikbausteine in den Vordergrund. Mit der zunehmenden Leistungsfähigkeit dieser Module und den damit verbundenen System-on-Chip-Technologien wurde die Realisierung höher integrierter Plattformen, applikationsspezifischer Hardware-Module und heterogener Multiprozessorsysteme möglich. Diese Arbeit fasst ausgewählte methodische Beiträge zur modellbasierten Entwicklung eingebetteter Lösungen der letzten Projektperiode zusammen. Die zum Verständnis wichtigen Grundlagen zum Entwurf eingebetteter Systeme und zu rekonfigurierbarer Hardware werden in den ersten Kapiteln zusammengetragen. Um die Konsistenz mit den größtenteils englischsprachigen Quellen und damit den etablierten Begriffen in der Fachwelt zu erhalten, sind die Abbildungen in dieser Arbeit in Englisch beschriftet - Erläuterungen und deutsche Entsprechungen sind dem Text zu entnehmen. Für die methodischen Definitionen wird ein Grundverständnis der UML vorausgesetzt.

Begleitend werden prototypische Studien von Plattformlösungen und Modellsyntheseansätzen vorgestellt. Die Realisierungen letzterer und die zur Evaluierung herangezogenen Modelle und Systeme wurden mit Hilfe von MATLAB/Simulink® erstellt und sind im Rahmen der Arbeiten und Kooperationen im SFB 622 entstanden.

Ilmenau, 7.1.2013

Marcus Müller

Kurzzusammenfassung

Der Entwurf verteilter eingebetteter Systeme ist aufgrund steigender Komplexität der Plattformen und der zu realisierenden Anwendungen zunehmend nur unter Einsatz von modellbasierten Entwicklungsmethoden zu realisieren. In der Mess- und Automatisierungstechnik und besonders in Anwendungsbereichen mit außergewöhnlichen Anforderungen, wie die den Hintergrund dieser Arbeit darstellende Realisierung von Regelungssystemen für die Nano-Positioniertechnik, ist die Untersuchung des Reglerentwurfs im Zusammenhang mit dem zu regelnden Prozess im Rahmen der simulativen Validierung und des virtuellen Prototypings wesentlicher Bestandteil des modellbasierten Entwurfsprozesses und der Bewertung von Systementwurfsentscheidungen. Der zunehmende Einsatz von leistungsfähigen FPGA-basierten eingebetteten Plattformen in diesem Anwendungsgebiet motiviert dazu die Entwicklung von zielplattformspezifischen Modellierungswerkzeugen, -bibliotheken und Code-Syntheseverfahren mit dem Ziel der Realisierung eines geschlossenen Entwicklungsprozesses vom ausführbaren Spezifikationsmodell zur Hardware-Beschreibung für diese Zielsysteme.

Die in dieser Arbeit zusammengetragenen Entwurfsmethoden beschreiben aktuelle Meet-in-the-Middle-Ansätze, die die Systementwurfskomplexität durch erhöhte Abstraktionsniveaus, automatische Transformationen sowie durch wiederverwendbare Komponenten und Architekturen reduzieren. Kern der methodischen Beiträge dieser Arbeit ist die Formulierung einer Entwurfsebene, die die Strukturen für die Modelltransformation zum FPGA-plattformspezifischen Systemmodell festlegt. Sie umfasst die Metamodelle für (Multi-)FPGA-Plattformen und die zugehörige plattformspezifische Modellierungsebene, sowie die zur Generierung ausgewählter plattformspezifischer Strukturen. Als Demonstrationsbeispiele für die Anwendung der erarbeiteten Verfahren dienten im Rahmen einer Design-Space-Exploration erstellte Entwürfe zur modellbasierten multi-FPGA-orientierten Realisierung von Regelungssystemen aus dem Kontext des SFB622. Die Ergebnisse verdeutlichen, wie groß der Einfluss von plattformspezifischen Anteilen auf die Modellkomplexität ist, im Gegensatz zu den von klassischer Wiederverwendung profitierenden Modellanteilen, und zeigt, in welchem Maße der Entwurfsaufwand durch Einsatz von automatisierten Strukturgenerierungsmethoden reduziert werden kann.

Abstract

Managing the development of distributed embedded systems for increasingly complex platforms and applications depends more and more on the use of model-based design methods. In the measurement and automation domain and especially in fields of application with extraordinary requirements, like the nano-positioning technology, that is the background of this work, the exploration of a control design in connection with the controlled process in the context of validation by simulation and virtual prototyping is a central part of the model-based design process and the base of system design decisions. The increasing utilization of high-performance FPGA-based embedded platforms in this application domain motivates the development of corresponding platform-specific modeling tools, libraries and code synthesis methods with the ever-present goal to realize a seamless system design process from an executable specification model to the hardware description for these target systems.

The design methods compiled in this work describe current meet-in-the-middle design approaches, which address the system design complexity by increased levels of abstraction and automated transformations as well as by reusable components and architectures. Center of the methodical contributions of this work is the formulation of a design layer, which defines the structures necessary for model transformations towards an FPGA-platform-specific system model. It comprises the meta-models for (multi-)FPGA platforms and the corresponding platform-specific modeling level, as well as extensions for the automated generation for certain platform-specific structures. As example scenarios to demonstrate the application of the developed approaches serve designs from the design space exploration for the multi-FPGA-based realization of control systems from the context of the SFB 622 research project. The results emphasize the influence of platform-specific design partitions on the model and design complexity in contrast to the partitions profiting from classical design reuse, and show the achievable reduction of design effort by utilizing automated structure generation.

Inhaltsverzeichnis

1 Einleitung

Der Fortschritt in der Halbleitertechnologie, der sich durch zunehmend höhere Integrationsdichten beim Bau von Rechen- und Speichertechnik charakterisieren ließ, wurde von IBM-Mitbegründer Gordon Moore aufgrund seiner Beobachtungen als ungefähre Verdopplung alle 18 bis 24 Monate beziffert und ging als „Moore's Law“ (Moore'sches Gesetz) in die Geschichte ein. Die Implikation einer entsprechenden Verdopplung der Leistung digitaler Rechner, die von vielen als selbsterfüllende Prophezeiung bezeichnet wird, ist zum Teil der Strategie der Chip- und Mikroprozessorhersteller geworden und gilt damit als Triebfeder für die Entwicklung neuer Technologien [ITR11, BC12]. Das zunehmende Potential der Rechentechnik erlaubt nicht nur die performantere Bearbeitung von Aufgaben, sondern auch die Bewältigung anspruchsvollerer Aufgaben in vorgegebenen Zeit- und Ressourcenbudgets oder die Minimierung der Zeit- und Ressourcenanforderungen für bestimmte Lösungen. Diese Aspekte sind vor allem für eingebettete Rechnersysteme von Bedeutung, deren Entwicklung durch die engen Wechselwirkungen zwischen den Anforderungen und den Beschränkungen der Realisierungs- und die Einsatzumgebungen charakterisiert sind. Die Domäne der Mess-, Regelungs- und Automatisierungstechnik allein ist ein weites Feld, in dem es eine Vielzahl von Anwendungsgebieten für eingebette Systeme gibt, und in dem an solche Systeme eine Vielzahl von unterschiedlichen Anforderungen gestellt werden. Ob industrielle Prozesssteuerungen, mobile drahtlose Sensornetzwerke, Sensorfelder für die Biosignalanalyse in der Medizintechnik oder Hochgeschwindikeitsregelungssysteme für Avionik - eingebette Systeme zur Realisierung dieser Aufgaben müssen den teilweise orthogonalen Anforderungen an Echtzeitfähigkeit, Verarbeitungsgeschwindigkeit, Datendurchsatz und nicht zuletzt Größe und Energieaufnahme genügen. Durch die fortschreitende Entwicklung eingebetteter Rechentechnik, wie Mikrocontroller, Signalprozessoren (Digital Signal Processor (DSP)), applikationsspezifischer Prozessoren (Application-specific Processor (ASP)) oder auch vollständig rekonfigurierbarer Hardware-Bausteine wie Field-Programmable Gate Array (FPGA), wird es zunehmend möglich, sehr komplexe Anwendungen hochperformant zu realisieren und auch für Anwendungsgebiete mit zunehmend extremen Anforderungen integrierte und eingebettete Lösungen zu realisieren und zum Einsatz zu bringen. [LZ06,VG02, MC07]

1.1 Nanomesstechnik und Echtzeitdatenverarbeitung

Eine Anwendungsdomäne mit speziellen Anforderungen an die eingebettete Informationsverarbeitung ist die Hochpräzisionslängenmesstechnik, bei der die im Rahmen des Sonderforschungsbereich (SFB) 622 an der TU Ilmenau entwickelten Nanopositionier- und Nanomessmaschinen (NPMM) einen weltweit führende Stellung einnehmen.

Die zwischen 1996 und 2000 entwickelte, zur Marktreife gebrachte und inzwischen weltweit eingesetzte Nanopositionier- und Nanomessmaschine NMM-1 zeichnet sich durch eine Kombination von Merkmalen aus, die sie dem Stand der Technik auf diesem Gebiet auch heute noch ebenbürtig oder überlegen machen. Dies betrifft hauptsächlich das vergleichsweise große Messvolumen von $25*25*5\ mm^3$, die erreichbare Auflösung von 0,1 nm und eine Reproduzierbarkeit von unter 10 nm. Der mechanische und sensorische Aufbau des Positioniersystems besteht aus einem metrologischen Rahmen und einem darin in drei Raumrichtungen beweglich gelagerten Positioniertisch. Die Probe wird unter dem am metrologischen Rahmen fest gelagerten Antastsystem verfahren. Die Position des Tisches im Raum wird dabei durch Laser-Interferometer und eventuelle Verkippungen durch Winkelsensoren bestimmt. Die eigentliche Messinformation wird aus der relativen Position der Probe zum Antastsystem gewonnen. [Hau02,JMHB00,JMH$^+$01,JMHS02,SIO07,Amt10]

Die maschinennahen Prozesse der digitalen Datenerfassung und -verarbeitung, Steuerung und Regelung erfordern den Einsatz von leistungsfähigen eingebetteten Systemen. Diese müssen sich den durch das extreme Anwendungsgebiet der Nanomesstechnik bedingten hohe Anforderungen in Form von immensem Datenaufkommen, hoher Algorithmenkomplexitat, sowie harten und eng bemessene Echtzeitschranken gewachsen zeigen. Im Groben lassen sich zwei wesentliche Verarbeitungsstränge unterscheiden, die je nach Betriebsmodus und eingesetzter Sensorik unterschiedlich dimensioniert und auch unterschiedlich stark verkoppelt sind. Diese Pfade lassen sich allgemein charakterisieren, wie in Abbildung 1.1 illustriert ist, als einen Datenerfassungspfad (*Data acquisition, Preprocessing*) und eine Echtzeitschleife (*Real-time loop*). Im Datenerfassungspfad werden die Daten der Sensorik digital erfasst, korrigiert und vorverarbeitet, um die eigentlichen Messwerte zu synthetisieren. Hierbei spielen nicht nur die Signale des Antastsystems (*Probe system*) und die Signale der Positionssensorik (*Positioning sensors*) eine Rolle, sondern auch die einer Vielzahl von Umweltsensoren (*Environmental sensors*), ohne deren Einbeziehung in die Messwertkorrektur die Präzision dieser Maschine unerreichbar wäre. Die gemessenen und errechneten Werte müssen außerdem an den Bedien-PC zur Lösung der eigentlichen Messaufgabe

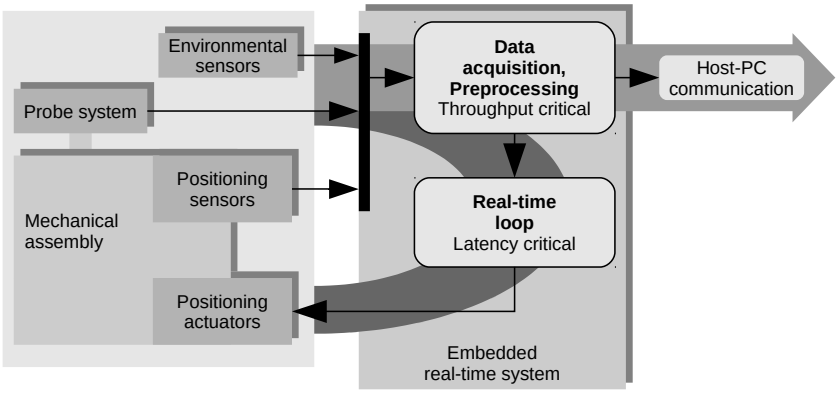


Abbildung 1.1: Schema der prozessnahen Signal- und Datenverarbeitungsstränge der NPMM.

übermittelt werden (*Host-PC communication*). Für die Leistungsfähigkeit dieses Teilsystems ist der erreichbare maximale Datendurchsatz (*Throughput*) des eingebetteten Echtzeitsystems (*Embedded real-time system*) wichtig, denn dieser begrenzt die maximale Rate der Messwertaufnahme des Gesamtsystems. Die Echtzeitschleife dient vor allem der Bewegungszustandsregelung des Positioniersystems. Durch einen Mehrachsenregler werden hier aus den Messwerten und einer vorgegebenen Trajektorie die Steuersignale für die Positionieraktorik (*Positioning actuators*) erzeugt. Ein kritisches Kriterium für die Leistungsfähigkeit der Echtzeitschleife ist die minimale Latenz (*Latency*) zwischen Messwertaufnahme und Stellgrößenausgabe. Die Länge dieses Echtzeitintervalls hat sowohl Einfluss auf die Güte der Bewegungszustandsregelung, als auch auf die Rate der Messwertaufnahme und damit die eigentliche Messgeschwindigkeit (Verfahrgeschwindigkeit). Sei die Abtastrate

$$f = \frac{v}{d} = \frac{1}{l_{RT}} \tag{1.1}$$

mit *Messgeschwindigkeit* v, *Messabstand* d und *Echtzeitintervall (Latenz)* l_{RT}, dann ergibt sich die *Messzeit* t über einen *Messweg* s aus

$$t = \frac{s}{v} = \frac{s * l_{RT}}{d} \rightarrow min. \tag{1.2}$$

Der Messabstand d beeinflusst wesentlich die minimal erreichbare Reglerabweichung für die Posititions- bzw. Bewegungszustandsregelung und hängt ab

von:

$$d = \frac{v}{f} = v * l_{RT} \rightarrow min. \tag{1.3}$$

Für die NMM-1 ist eine Abtastrate von $6,25\ kHz$ angegeben. Beim Scannen mittels Atomic Force Microscopy (AFM) entlang einer Linie über den Messbereich von 25 mm sollen in 10-nm-Schritten Werte aufgenommen werden. Daraus resultieren eine Verfahrgeschwindigkeit von $62,5\ \mu m/s$ und eine Messzeit von 400 Sekunden. Für die gesamte Fläche von $25 * 25\ mm^2$ ergäben sich bereits 31,71 Jahre. Die Minimierung der Messzeit t und des Messabstands d sind entsprechend einer Veränderung der Verfahrgeschwindigkeit entgegengesetzte Zielstellungen. Neben einer vorausschauenden Prüfplanung zur Minimierung des nötigen Scanwegs s und ohne eine Verringerung der Verfahrgeschwindigkeit in Kauf zu nehmen, profitieren beide Zielstellungen von einer *Minimierung des Echtzeitintervalls* l_{RT}. Dieses Forderung zu erfüllen erfordert die leistungsfähigste Implementierung der Signal- und Datenverarbeitung im maschinennahen eingebetteten Echtzeitsystem.

Die Datenverarbeitungseinheit der NMM-1 ist schematisch in Abbildung 1.2 dargestellt. Sie besteht aus einem modularen Aufbau mit Analog-zu-Digial (A/D)- bzw. Digital-zu-Analog (D/A)-Wandlern (*ADC/DAC module*) als Schnittstellen zum physikalischen System und einem DSP-Modul als Rechenkern. Im Rahmen des SFB 622 wurde als Alternative zum Einzelprozessormodul ein Mehrkern-DSP-System (*Multi-DSP assembly*) entwickelt, um der NMM-1 mit Parallelverarbeitungskapazität Raum für Leistungsteigerungen und/oder Funktionserweiterungen zur Verfügung zu stellen [DB04].

Um den formulierten erweiterten Zielstellungen wie einem Messbereich von mindestens $200*200*25\ mm^3$ und den daraus abgeleiteten Forderungen gerecht werden zu können, wurde der regelungstechnische Ansatz der modellbasierten Trajektorienfolgeregelung verfolgt, der das Potential besitzt, den dynamischen Positionsfehler der Maschine signifikant zu reduzieren [AZA08, Amt10]. In einer Machbarkeitsstudie zur Implementierung dieses Ansatzes, der eine deutlich höhere Berechnungskomplexität als die in der NMM-1 eingesetzten Reglerimplementierungen besitzt, wurde die Validität einer parallelen DSP-Lösung unter der Maßgabe der konstanten Echtzeitschranken der NMM-1 nachgewiesen [MFAA09].

Zunehmend höhere Anforderungen an die Prozessqualität trotz steigender physischer Abmessungen der Messaufbauten verlangen nach zunehmend komplexen Regelungslösungen, um den kombinierten Forderungen nach höher Genauigkeit und hoher Dynamik gerecht zu werden. Die zur Minimierung der Mess- und Regelfehler nötige steigende Berechnungskomplexität und

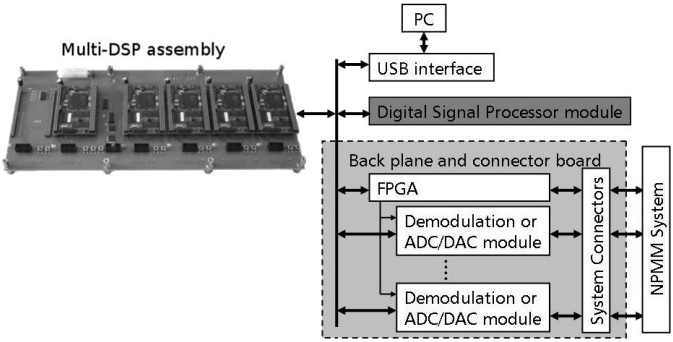


Abbildung 1.2: Schema der Datenverarbeitungseinheit der NMM-1 mit DSP und Multi-DSP-Erweiterung.

die Forderung nach höheren Abtastraten, die ihrerseits positiven Einfluss auf die Bahnfehler haben, sind schwer vereinbare Sachverhalte. Auf die Realisierung des eingebetteten Echtzeitsystems übertragen bedeuten sie, die Berechnungslatenz der Echtzeitschleife auf einer problemangepassten eingebetteten Plattform zu minimieren. Im Rahmen des Transfers der Ergebnisse des SFB 622 in die Entwicklung eines Industrieprototypen der Nachfolgegeneration der NMM-1 ist die Erhöhung der Reglerabtastrate auf bis zu 10 kHz vorgesehen. Zur Realisierung der gesteigerten prozessnahen Funktionalität für die eingebettete Echtzeitverarbeitung stehen heterogene hierarchische Systeme im Vordergrund der Betrachtungen. Die steigende Verfügbarkeit von hochleistungsfähigen Industrie-Standard-Plattformlösungen vereinfacht dabei die Exploration verschiedener Plattformkompositionen und verteilten Implementierungsvarianten, so dass den Strukturen und Berechnungcharakteristika verschiedener Anwendungsteile und Algorithmen entsprochen werden kann (z.B. Filter und Regler auf rekonfigurierbarer Hardware und DSPs, Parametrisierung und Systemsteuerung auf generischen Mikrocontrollern oder Prozessoren u.ä.). Umfassende Fallstudien zu plattformspezifischen Implementierungen haben die Vor- und Nachteile von verteilten Implementierungen auf heterogener Hardware, wie embedded-PCs, DSPs oder FPGAs, gezeigt. Als entscheidender Nachteil bei der Verwendung von hierarchischen Plattformen sind die Datentransferzeiten zwischen den Hierarchie-Ebenen identifiziert worden. Diese wiegen die unter Umständen überragende Prozessorleistung bis zu dem Punkt auf, an dem die akkumulierten Latenzen das verfügbare Echtzeit-Budget überschreiten und sich eine Standardlösung als unrealisierbar erweist [MKG+13].

Die zunehmende Leistungsfähigkeit von FPGAs macht diese Technologie auch für mess- und regelungstechnische Anwendungen interessant, was zur Einführung von FPGA-basierten Datenerfassungsmodulen geführt hat. Diese erlauben es, z.B. Algorithmen zur Messdatenaufbereitung in unmittelbarer Nähe zur Datenerfassung zur Ausführung zu bringen, ohne zusätzliche Latenzen durch Datentransfers ins System einzutragen. Die resultierenden Architekturen für die Echtzeitdatenverarbeitung der NPMM sehen daher die Verlagerung des Großteils der echtzeitkritischen Verarbeitung in rekonfigurierbare Berechnungseinheiten vor. Abbildung 1.3 zeigt das Schema einer modularen hierarchischen Verarbeitungseinheit basierend auf der Infrastruktur PCI eXtension for Instrumentation (PXI)[1]. Ein Verbund mehrerer Echtzeitsysteme realisiert das Messwerterfassungssystem (*Position measurement system*), die Ablaufsteuerung (*Sequence control*) mit Host-PC-Kommunikation, sowie das Regelungssystem (*Control system*). Die Regelung selbst ist auf rekonfigurierbaren I/O-Modulen implementiert. Aufgrund der Komplexität der Regelungsanwendung und der physischen Randbedingungen, wie z.B. verfügbare Chip-Fläche und benötigte I/O-Kanäle, ist hier eine verteilte Implementierung auf mehrere FPGAs vorgenommen worden [ZKA+12, PKMF11b].

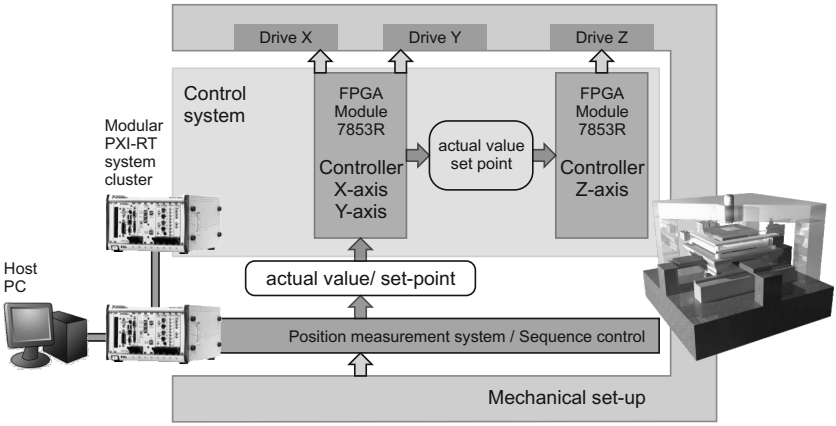


Abbildung 1.3: Schema der Datenverarbeitungseinheit der neuen NPMM-Generation mit modularer hierarchischer Echtzeitverarbeitungseinheit (NI PXI-RT) und FPGA-basierter Implementierung des Regelungssystems (nach [ZKA+12, PKMF11b]).

[1] mit Geräten der Firma National Instruments®

Moderne rekonfigurierbare Hardware-Plattformen bieten die Möglichkeit, den Anforderungen der Mess- und Regelungstechnikdomäne im Allgemeinen und der Nano-Messtechnik im Besonderen gerecht zu werden, indem sie die Implementierung auch komplexer Funktionalitäten in leistungsfähigen prozessnahen Echtzeitsystemen erlauben.

1.2 Modellbasierter Entwurf eingebetteter Systeme und Zielstellungen dieser Arbeit

Der Stand der Technik für eingebettete Hochleistungsplattformen wird unter anderem durch rekonfigurierbare Hardware wie FPGAs charakterisiert. Heutige FPGAs können mehrere 10 Mio. Logik-Gatter enthalten, wodurch genug Logik auf einem Chip verfügbar ist, um sehr komplexe und leistungsfähige Systeme zu realisieren. Zusätzlich arbeiten diese Geräte mit internen Takten, die mit den meisten ASICs vergleichbar sind. Obwohl, oder gerade weil derartigen FPGA-Plattformen nie dagewesene Möglichkeiten zur Entwicklung komplexer eingebetteter Systeme bieten, sind neue Ansätze nötig, um z.B. die Skalierbarkeit der Architekturen und Lösungen, die Entwurfsproduktivität und nicht zuletzt die Benutzbarkeit von Entwurfsmethoden und -werkzeugen sowie Produkten sicherzustellen [Bol02]. Die Entwicklung von komplexen Lösungen für komplexe Aufgabenstellungen erfordert Systematiken, die über reine Implementierung und Test weit hinausgehen. Dem weiteren Aufklaffen der bekannten, durch Moore's Law bedingten *Produktivitätslücke (Productivity gap)* zwischen den komplexen potentiellen Lösungen und dem Aufwand, diese zu erreichen, wird versucht durch strukturierte Entwurfsmethoden und extensive Komponenten- und Architektur-Wiederverwendung und die darauf aufbauenden "System-on-Chip"-Ansätze entgegenzuwirken [ITR11].

Entwurfsmethoden, Entwicklungsverfahren und dazugehörige Werkzeuge werden in dem Bestreben entwickelt, die Komplexität des Entwicklungsprozesses zur Optimierung der Entwicklungszeit und -kosten zu verringern, indem sie einen Entwurf auf immer höheren Abstraktionsniveaus erlauben [VG02] und die zugrundeliegenden Implementierungsschritte zunehmend automatisieren - von der Logik- und RTL-Synthese, über die Verhaltenssynthese (Behavioural oder High Level Synthesis) [JDKR97] bis hin zu implementierungsplattform- und ausführungssematikunabhängigen Beschreibungen [HS07]. Damit tritt die Hardware- und Software-Entwicklung zunehmend hinter die Entwicklung auf *Systemebene* zurück, deren Entwurfsverfahren der Komplexität des sich aufspannenden Design-Raums Rechnung tragen. *Modellbasierte Entwurfsmethoden* erlauben hier die plattformunabhängige Spezifikation, Systembeschrei-

bung und Validierung auf Ebenen, die noch vollständig von Plattformimplementierungen abstrahieren. [HS07, NM10] Gerade für die Erforschung des Entwurfsraumes (*Design space exploration*) ist eine plattformunabhängige Modellierung sowie deren zyklische Verfeinerung und (weitgehend) automatisierte Abbildung auf plattformspezifische Beschreibungen ein zeit- und kostensparendes Mittel, Design-Entscheidungen zu treffen, zu validieren und in Implementierungen zu überführen. Die Vorteile des modellbasierten Entwurfs zeigen sich besonders für die Entwicklung von eingebetteten Realisierungen im Rahmen von komplexen Gesamtsystemen aus eng verknüpften physikalischen und digitalen Teilsystemen - ein Zusammenhang, der unter der Bezeichnung *Cyber-physical Systems* in den Fokus der Betrachtungen zum modellbasierten Entwurf gerückt wurde [Poo10, PSG+12, SKK12, Mar11].

Für die den Hintergrund dieser Arbeit darstellende Anwendungsdomäne der Mess- und Automatisierungstechnik sind modellbasierte Entwicklungsmethoden etabliert, um Reglerentwürfe innerhalb abstrakter Modelle der Regelstrecken und Prozessumgebungen durchführen und funktional validieren zu können. Ziel dieser Arbeit ist es, Beiträge zur modellbasierten Überführung von Spezifikationmodellen in plattformspezifische Systembeschreibungen eingebetteter Implementierungen zu leisten. Aufgrund der Leistungsanforderungen und der physikalischen Randbedingungen der Anwendungsdomäne, die im vorigen Abschnitt beschrieben wurden, beinhalten die möglichen Plattformrealisierungen verteilte FPGA-Anordnungen, was zusätzliche Herausforderungen für den Entwurfsprozess mit sich bringt.

Im Mittelpunkt dieser Arbeit steht die plattformspezifische Modellierung als Phase des Top-Down-Verfeinerungsprozesses im Rahmen des modellbasierten Entwurfs eingebetteter Systeme auf verteilter rekonfigurierbarer Hardware. Wesentlicher Beitrag dieser Arbeit ist die *Definition einer modellbasierten Entwurfsplattformebene*, die die für den Systementwurf wichtigen Charakteristika von Multi-FPGA-Plattformen und komponentenbasierten Logik-Designs abbildet. Da die Entwurfsplattformebene von den darunter liegenden Implementierungsvorgängen abstrahiert, ist es das plattformspezifische Systemmodell, in dessen Rahmen ein Großteil der Design-Space-Exploration während des Systementwurfs stattfindet. In dieser Arbeit spezifizierte Methoden dienen der Abbildung von den funktionsblockorientierten Spezifikationen der Anwendungsdomäne auf komponentenbasierte verteilte Beschreibungen eingebetteter Realisierungen. Die Automatisierung von Modelltransformationen für bestimmte FPGA-plattformspezifische Entwurfsentscheidungen, wie der Verringerung des Ressourcenbedarfs oder die Synthese von Off-Chip-Kommunikationlogik in verteilten Entwürfen, unterstützen die Reduktion der Entwurfskomplexität bei der Umsetzung von Systementwurfsentscheidungen auf dieser Modellierungsebene.

1.3 Inhalt dieser Arbeit

Das zweite Kapitel widmet sich den Methoden zum Entwurf von eingebetteten Systemen. Der Stand der Technik sowie die Möglichkeiten der modellbasierten Ansätze im Bezug auf die Entwicklung von eingebetteten Systeme werden illustriert. Die wesentlichsten Aspekte der Ansätze des plattformbasierten Entwurfs (Platform-based Design) und des modellbasierten Entwurfs (Model-based Design) werden zusammengeführt, um den methodischen Rahmen für die in dieser Arbeit betrachteten Aspekte festzulegen. Im darauf folgenden Grundlagenkapitel wird der Stand der Technik im Bereich der rekonfigurierbaren Rechentechnik für eingebettete Systeme, insbesondere der FPGA-Technologie, erarbeitet. Hierbei liegt ein besonderer Fokus auf den komponentenbasierten Logikentwurfs- und Chip-Design-Ansätzen, die die Grundstrukturen für zunehmend komplexe Entwürfe auf der Basis wiederverwendbarer Design-Elemente beschreiben. Ein Fazit fasst die Rolle, die FPGA-basierte Realisierungen in der Automatisierungstechnik spielen, zusammen.

Basierend auf den Grundlagenerörterungen steht die Definition einer modellbasierten Entwurfsplattformebene im Zentrum des Kapitels 4. Die Abstraktion von Multi-FPGA-Systemen wird mit Hilfe konkreter Implementierungsstudien untermauert. Die strukturellen Zusammenhänge zwischen Spezifikation, plattformspezifischer Modellierung und Implementierungs- und Hardware-Ebene werden mittels Metamodellen definiert. Die ausgewählten Methoden zur Reduktion des plattformspezifischen Entwurfsaufwandes werden in Kapitel 5 unter Einordnung in die modellbasierte Entwurfsplattformebene formuliert.

Die Anwendung der erarbeiteten Methoden in einem projektspezifischen Entwicklungsprozess wird in Kapitel 6 anhand des für die neue Generation der Nano-Positionier- und -Messmaschine (NPMM) entwickelten 3D-Trajektorienfolgeregelungssystems gezeigt. Ausgehend von einem spezifizierten Cyber-Physical-System-Modell bestehend aus dem Mehrachsenreglersystem gekoppelt mit dem Umgebungsmodell in MATLAB/Simulink® geschieht die Verfeinerung in einer geschlossenen Entwurfs- und Validierungskette. Die Optimierungspotentiale, die die jeweilige Anwendung der erarbeiteten Methoden birgt, werden durch prototypische Untersuchungen und Realisierungen bestimmt. Schlussendlich werden im Zuge der Design-Space-Exploration erstellte plattformspezifische Gesamtmodelle präsentiert, die unter gegebenen Randbedingungen realisierbare und optimale Konfigurationen für Multi-FPGA-Implementierungen des Reglersystems darstellen.

Eine Zusammenfassung der Ergebnisse, eine Wertung derselben und ein Ausblick in Richtung weiterführende Arbeiten sowie in den umgebenden Kontext eines geschlossenen Entwurfsprozesses beschließen diese Arbeit.

2 Entwurfsmethodik für eingebettete Systeme

Systementwurf ist der Prozess, der, mehr oder minder automatisiert, aus formulierten Anforderungen eine Systembeschreibung ableitet, von der ausgehend eine Implementierung des Systems vorgenommen werden kann. In welcher Entwurfsdomäne (Software, Hardware etc.) auch immer angesiedelt, beinhalten Systementwurfsprozesse sowohl Bottom-Up- als auch Top-Down-Aktivitäten: die Wiederverwendung und Adaption existierender Komponenten und die sukzessive Verfeinerung von Systembeschreibungen, die die Anforderungen in eine Implementierung umsetzten sollen. Da für eingebettete Rechnersysteme aufgrund vieler technischer und physikalischer Einschränkungen und vieler Wechselwirkungen keine wirkliche Trennung zwischen Hardware, Software und ihrer Umgebung, zwischen Funktionalität und Plattform getroffen werden kann, sind hier besonders integrierte Methoden von Nöten. [HS06]

In diesem Kapitel wird nach einer kurzen Charakterisierung des Begriffes des „eingebetteten Systems“ auf generelle Systementwurfsprozesse eingegangen. Danach werden die spezielleren Verfahren des *Platform-based Design* und des *Model-driven Design* betrachtet, in deren Kontext die Inhalte dieser Arbeit einzuordnen sind. Darauf aufbauend wird der Entwurfsprozess für komponentenorientierten modellbasierten Entwurf verteilter eingebetteter Systeme zusammengetragen und näher erläutert. Ein Fazit umreißt den von namhaften Entwicklungswerkzeug- und Hardware-Herstellern mitbestimmten aktuellen Stand der Technik, der bezüglich der modellbasierten Entwicklung von eingebetteten Lösungen für Mess- und Regelungstechnik eine Rolle spielt.

2.1 Eingebettete Systeme

Einleitend soll an dieser Stelle eine Definition des Begriffes des eingebetteten Systems, oder genauer des eingebetteten Rechensystems, gegeben werden. Bei einem eingebetteten System handelt es sich um einen Rechner als Komponente einer größeren Anlage, auf deren Unterstützung und Betrieb der Zweck, die Gestalt und die Komplexität des eingebetteten Rechners ausgelegt ist. Die Einsatzfähigkeit eines eingebetteten Systems ist damit

auf eine spezielle Anwendung oder Klasse von Anwendungen beschränkt [BC12, VG02]. Wie in Abbildung 2.1 schematisch dargestellt ist, interagiert das eingebettete System (*Embedded System*) über Sensoren und Aktoren (*Sensors, Actuators*) mit dem einbettenden System (*Embedding system*). Oft, wie in der Automatisierungstechnik, müssen eingebettete Systeme als reaktive Systeme entworfen werden, die unter harten Zeitbeschränkungen arbeiten können - diese werden als Echtzeitsysteme bezeichnet, und stellen zusätzliche Anforderungen an den Entwurf und die Leistungsfähigkeit [HNN06]. Zusätzlich kann bei bestimmten eingebetteten Systeme eine Mensch-Maschine-Schnittstelle (Human-Machine-Interface (HMI)) zur Überwachung oder Parametrisierung verfügbar sein. Generell ist die Hauptfunktion des eingebetteten Rechners aber auf Prozesse ausgerichtet, die keine menschliche Interaktion benötigen oder zulassen. Einbettende Systeme größerer Komplexität können es durchaus nötig machen, dass mehrere eingebettete Systeme direkt oder in hierarchischen Strukturen miteinander interagieren, wofür das jeweilige Rechnersystem auch über Schnittstellen zu den entsprechenden Kommunikationskanälen verfügen muss [LZ06].

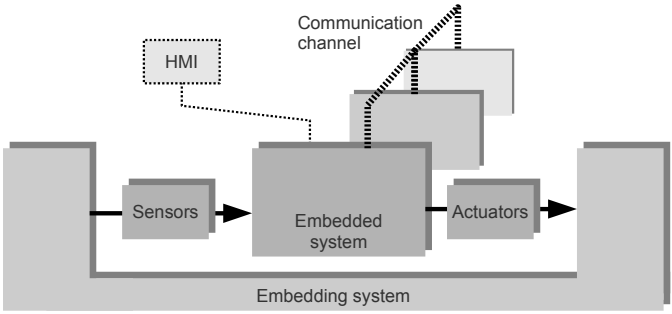


Abbildung 2.1: Eingebettetes System (schematisch).

Je nach Einsatzzweck und Anforderungen kann die eingebettete Realisierung im Spektrum zwischen Software-Applikation, über Implementierungen für Prozessoren mit spezifischem Befehlssatz (Application-specific Instruction Set Processor (ASIP)) bis hin zu komplexen Hardware-Operatoren angesiedelt sein. Die Plattformen können dabei vom universellem embedded-PC mit Echtzeitbetriebssystem, über problemangepasste Mikrokontroller- und DSP-Platinen (System-on-Board (SoB)), kompakte Multi-Chip-Anordnungen (System-in-Package (SiP)), bis hin zu hochintegrierten Ein-Chip-Systemen (System-on-Chip (SoC)) als applikationsspezifische Schaltkreise (Application-

specific Integrated Circuit (ASIC)) variieren [VG02]. Die anwendungsspezifischen Eigenschaften eines eingebetteten Systems erfordern eine Entwurfsmethode, wie sie verallgemeinert in Abbildung 2.2 dargestellt ist. Dabei verlaufen die Entwicklung der Funktion und die der Architektur auf der Basis von Bibliotheken für Funktionsverhalten und Architekturbeschreibungen (*Behavioural + Architecture libraries*) parallel und in Wechselwirkung miteinander. Ein Abbildungsprozess (*Mapping*), der ständigen iterativen Verfeinerungen (*Refinement*) unterliegt, führt zur endgültigen Implementierung (*Implementation*) [LP06, Ash08].

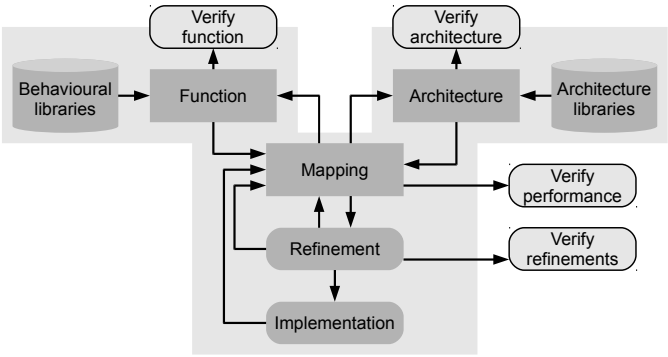


Abbildung 2.2: Entwurfsmethode für eingebettete Systeme (nach [LP06]).

2.2 Systementwurfsprozesse

Um den Entwurfsprozess für eingebettete Systeme zu beschreiben, ließen sich zahlreiche Vorgehensmodelle für Entwurfsprozesse heranziehen. Die verbreitetsten und am meisten verwendeten sind jedoch nach wie vor das V-Modell und eine Vielzahl von Verfeinerungen und Adaptionen desselben[1] [Vig10]. Das „V-Modell für den Systementwurf“ beschreibt die Aktivitäten von der Anforderungsspezifikation über den daraus abgeleiteten Entwurf des Systems und seinen Verfeinerungen bis zur Implementierung als Top-Down-Prozess, gefolgt von Integrations- und den damit verbundenen Test-Schritten, hin zur Installation und Inbetriebnahme des Systems mit finaler Validierung der Funktion gegen die ursprünglichen Anforderungen [Mar11]. Dieses Vorgehen ist in Abbildung 2.3 als dunkelgrauer Verlauf dargestellt.

[1] inklusive des Vorgehensmodells des Bundes zur Entwicklung von IT-Systemen [IAB06]

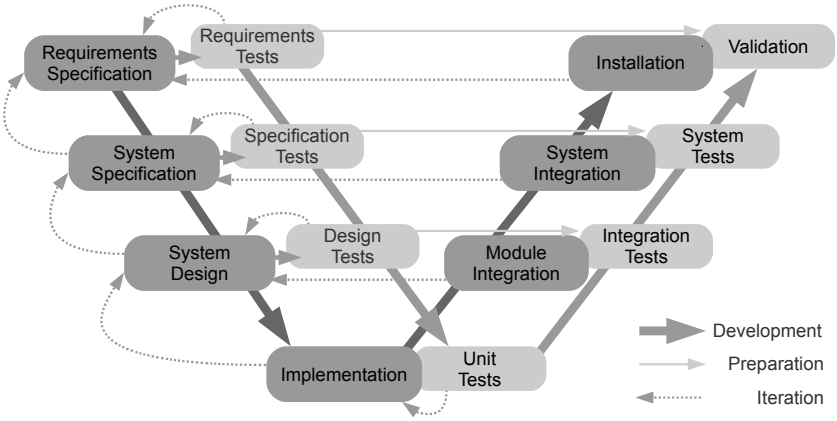


Abbildung 2.3: Vorgehensmodelle für den Systementwurf - V-Modell und W-Modell.

Wesentlicher Aspekt des V-Modells ist die Testpreparation während der Entwurfsphasen für die jeweils korrespondierenden Testphasen. Im Laufe der Zeit seit Definition des V-Modells in den 1970er Jahren wurde die mangelnde Testbarkeit während des Entwurfsprozesses und die fehlende Flexibilität zur Design-Iteration festgestellt. Das von Paul Herzlich 1993 vorgestellte W-Modell definiert die in Abbildung 2.3 zusätzlich dargestellten Test-Schritte zur Anforderungsspezifikation (Requirements Tests), zur Systemspezifikation (Specification Tests) und zum Systementwurf (Design Tests), die die entwurfsbegleitende Validierung der Design-Entscheidungen und Verfeinerungen und damit auch die Design-Iteration der frühen Entwurfsphasen ermöglichen. Dieses Modell deckt damit auch die Möglichkeiten ab, die moderne modellbasierte Entwurfsmethoden bieten, bei denen eine formale Spezifikation bzw. eine ausführbare Spezifikation (*Executable specification*), das Konzept des virtuellen Prototypen (*Virtual prototyping*) in Form von simulierbaren Modellen, die modellbasierte Testgenerierung und das modellbasierte Testen (*Model-based testing*) [PP05] eine zentrale Rolle spielen.

Für eingebettete Systeme ist generell ein holistischer Entwurfsprozess nötig, da die Wechselwirkungen zwischen der zu implementierenden Funktionalität, der ausführenden Hardware-Plattform und dem umgebenden System sehr groß sind. Es ist daher ein Hardware/Software-Codesign-Prozess nötig, um den nichtfunktionalen Anforderungen an das Gesamtsystem durch applikationspezifische Plattformentwürfe gerecht werden zu können. Abbildung 2.4

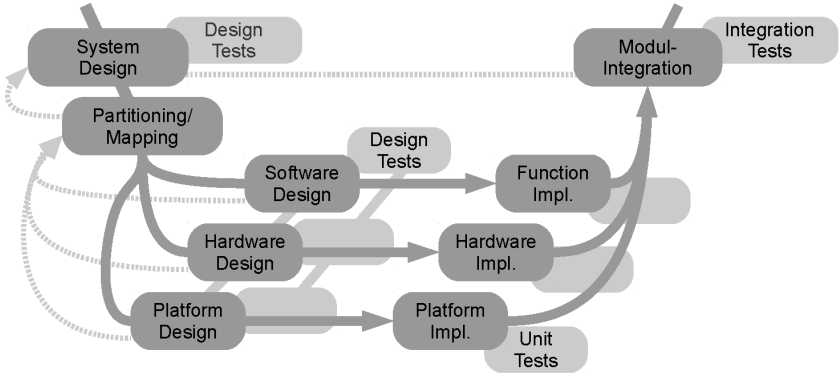


Abbildung 2.4: Hardware/Software-Codesign-Prozess als Teil des V- bzw. W-Modells.

zeigt das Codesign von eingebetteter Plattform, Hardware-Elementen und Software-Funktionen als Verzweigung im unteren Teil des W-Modell-Prozesses. An den Entwurf des Gesamtsystems schließt sich der Prozess der Hardware-Software-Partitionierung und der Abbildung auf die Plattform an, woraufhin die Partitionsentwürfe spezifisch weiter verfeinert werden. Diese Entwürfe und die dazugehörigen Tests laufen jedoch nicht unabhängig voneinander ab. Heterogene Modellierung und Simulation, Co-Simulation und Emulation unterstützen das Codesign. Nach der eigentlichen Funktions- und Hardware-Modul-Implementierung schließen sich die Schritte der Integration und der Integrationstests an.

Aus heutiger Sicht sind gerade die verfeinernden Systementwurfsprozesse von der Wiederverwendung vorgefertigter Lösungen, sowohl im Bereich der Hardware- und Software-Komponenten als auch der Plattformen und Architekturen, gekennzeichnet. Dadurch wird das V- bzw. W-Modell durch eine Vielzahl anderer Life-Cycle-Prozesse erweitert. Im folgenden werden vor allem die Einflüsse des plattformbasierten und des modellbasierten Entwurfs näher betrachtet.

2.3 Plattformbasierter Entwurf

Dem steigenden Druck auf die Elektronikindustrie bezüglich niedriger Entwicklungszeiten, zunehmend komplexen Anwendungs- und System-

entwürfen sowie steigende Einmal-Kosten in der Hardware-Herstellung, besonders bei hochgradig applikationspezifischer Hardware (ASICs), wird mit Entwurfsmethoden begegnet, die sich hauptsächlich auf massive Wiederverwendung von Entwürfen und Komponenten sowie „Correct-the-first-time"-Implementierungen stützen. Zusätzlich sind eine Disaggregation und die Spezialisierung auf Kernkompetenzen in der Industrie zu beobachten, die es zunehmend unmöglich machen, einen durchgängigen applikationspezifischen Top-Down-Entwurf in einem Haus wettbewerbsfähig durchzuführen. [CDSVS02, CBP+05]

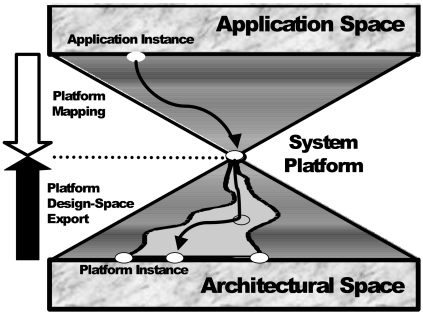


Abbildung 2.5: Plattform-basierter Entwurf [CBP+05, SVCDS04].

Stattdessen wird der plattformbasierte Entwurf als Meet-in-the-Middle-Ansatz verfolgt, bei dem eine oder mehrere *Plattform-Ebenen* als so genannte Artikulations- und Übergabepunkte zwischen den Abstraktionsniveaus und damit den Entwurfsstadien definiert werden [CDSVS02]. Abbildung 2.5 zeigt, wie die Definition einer System-Plattform den anwendungsspezifischen Top-Down-Entwurf von der Bottom-up-Plattformdefinition entkoppelt. Die sukzessiven Verfeinerungen der Spezifikation einer Anwendungsinstanz aus dem Applikationsraum an der Plattformebene treffen auf die Abstraktion potentieller Plattformimplementierungen, die vor dem System-Designer verborgen bleiben. Je nach Mächtigkeit und Abstraktionsgrad der Systemplattform wird ein entsprechender Ausschnitt aus dem Architekturraum zur Bildung konkreter Plattforminstanzen erschlossen, auf die die Anwendung im Rahmen einer Design-Space-Exploration abgebildet werden kann [KNRSV00]. Die Definitionen zum Begriff *Plattform* beziehen sich in der Literatur auf zwei verschiedene Aspekte. Zum einen wird darunter eine Menge von Subsystemen verstanden, die durch Schnittstellen und Zugriffsverfahren eine zugehörige Menge an Funktionalitäten zur Verfügung stellen, die jede unterstützte Applikation nutzen kann, ohne die

Implementierungsdetails dieser Funktionalitäten zu kennen [MM03]. Dies wird durch den Begriff *Ausführungsplattform* näher charakterisiert. Zum anderen wird unter einer Plattform eine Schicht im Entwurfsprozess verstanden, für die die darunterliegenden folgenden Prozessschritte abstrahiert worden sind [CDSVS02]. Dies wird durch den Begriff *Entwurfsplattformebene* genauer beschrieben. Damit bilden die Ideen des plattformbasierten Entwurfs auch die Grundlage für modellbasierte Entwicklungsprozesse.

2.4 Modellbasierter Entwurf

Beim Ansatz des modellbasierten Entwurfs - *Model-based Design* oder *Model-driven Design* sind als Synonyme zu verstehen - beruht der Entwurfsprozess von der Problemanalyse über die Spezifikation bis zur Implementierung der Lösung auf Modellen. Unter *Modell* ist dabei eine formalisierte Beschreibung von Aspekten des zu lösenden Problems oder des zu entwickelnden Systems und seiner Umgebung zu verstehen. Bei der Modellierung spielen die *Dekomposition* - das fortschreitende Zerlegen des Gesamtsystem in leichter zu analysierende Teilsysteme - und die *Abstraktion* - die Herausstellung der wesentlichen Aspekte eines Systems für einen bestimmten Entwurfszweck - eine entscheidende Rolle im Umgang mit der Komplexität der gesamten Entwurfsaufgabe. Komplexe Systeme werden auf verschiedenen Ebenen mit unterschiedlichen Abstraktionsgraden modelliert. Mit fortschreitendem Entwurfsprozess wird das Modell einer Reihe von *Transformationen* unterzogen, die neue Spezifika hinzufügen und den Detailgrad der Modellierung erhöhen. Die zentrale Idee jedes modellbasierten Entwurfsansatzes ist es, Systeme und Komponenten in einer Modellierungsprache zu beschreiben, die den Designer gerade in den Anfangsphasen des Systementwurfs nicht zu früh auf gewisse Ausführungs- oder Interaktionssemantiken festlegt, oder zu spezifischen Implementierungsentscheidungen veranlasst. Ziel des zunehmenden Verfeinerungsprozesses in späteren Entwurfsphasen ist die Abbildung auf eine Ausführungsplattform, das heißt, die finale Transformation in implementierbaren Code oder die Abbildung auf getestete und implementierte Systemkomponenten, deren Integration und Interaktion durch das Modell definiert wird. [MM03, BBCM06, HS07, MG09]

Modellbasierter Entwurf unterstützt die in den frühen Entwurfsphasen wichtige Auslotung des Entwurfsraumes und die Bewertung verschiedener Realisierungsvarianten - *Design Space Exploration* - auf unterschiedlichen Abstraktionsebenen. Dies ist gerade hinsichtlich optimaler Entwurfsentscheidungen für die Entwicklung eingebetteter Hardware/Software-Systeme von Bedeutung. So können zum Beispiel

alternative System- und Plattformarchitekturen mit den richtigen Modellen früh hinsichtlich verschiedener Zielfunktionen (Performance, Ressourcenbedarf, Leistungsaufnahme, ...) evaluiert werden. Eine modellbasierte Methodik definiert nicht notwendigerweise eine feste Abfolge von Entwurfs- und Verfeinerungsschritten. Normalerweise verläuft der Entwurf weder strikt top-down von den Anforderungen zur Implementierung, noch rein bottom-up durch Integration von Bibliothekskomponenten, sondern in einer Iteration von Modellierung, Analyse und Transformation [HS07]. Die Modellierungsinfrastruktur gibt vielmehr die Möglichkeit, mit dem Modell eine zentrale Menge von Artefakten zur Systembeschreibung zu kreieren, auf denen verschiedene Werkzeuge zum Einsatz kommen können. Dieser methodische Raum des modellbasierten Entwurfs ist in [ETZ00] beschrieben worden und wird in Abbildung 2.6 dargestellt.

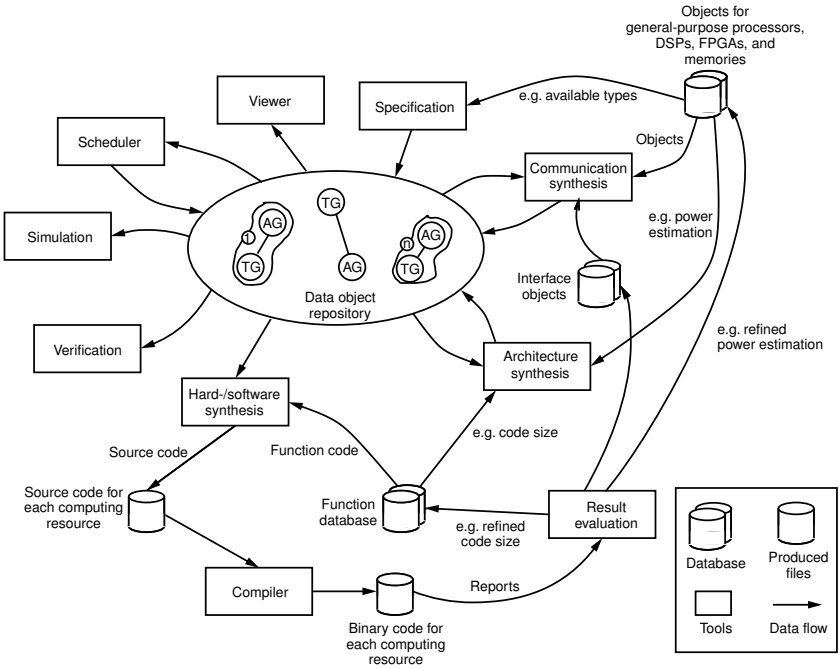


Abbildung 2.6: Methodischer Raum des modellbasierten Entwurfs für verteilte heterogene eingebettete Systeme [ETZ00].

Spezifikationswerkzeuge erlauben die Definition von Anwendungssystemen als Task-Graphen (TG), Systemarchitekturen und Plattform-Strukturen als Architekturgraphen (AG), sowie die jeweiligen Abbildungen und Zuordnungen. Diese bilden als sogenannte *Data Objects* die Modellierungsbasis. Verschiedene Werkzeuge zur Analyse, Validierung und Verifikation arbeiten auf dieser Repräsentation. Außerdem kommen Methoden zum Einsatz, die die Modellrepräsentation verändern können. Dazu zählen die Kommunikations- sowie Architekturoptimierung und -synthese und die implementierungsorientierten Transformationen zur Hardware- und Software-Synthese. Entscheidend für einen reibungslosen Einsatz und eine fortschreitende Automatisierbarkeit modellbasierter Entwurfsmethoden sind die drei in der Abbildung angegebenen Wissensbasen:

- die *Funktionsdatenbank* stellt der Architekturoptimierung nichtfunktionale Eigenschaften über die Funktionskomponenten zur Verfügung und ordnet den Modellelementen während der Synthese/Codegenerierung funktionsgleiche Implementierungskomponenten zu;
- die *Plattform-Objekt-Datenbank* enthält alle möglichen Berechnungsressourcen, die zu einer eingebetteten Plattform kombiniert werden können, sowie ihre Eigenschaften, die ebenfalls als Restriktionen in den Architekturoptimierungsprozess einfließen;
- die *Schnittstellen-Datenbank* beschreibt die Kommunikationsarchitekturen und die entsprechenden Schnittstellenobjekte zum Aufbau verteilter Systeme und zur Generierung der Schnittstellenlogik oder Treiber-Software während der Kommunikationssynthese.

Natürlich kann die Vollständigkeit dieser Datenbanken nicht vorausgesetzt werden, weswegen die Möglichkeit vorgesehen ist, die Wissensbasis aufgrund von Implementierungsergebnissen mit neuen Funktionskomponenten und ihren Performance-Eigenschaften, mit neuen Berechnungselementen und mit Plattformvarianten zu vervollständigen [ETZ00]. Diese Datenbanken stellen die Obermenge allen a-priori-Wissens dar, das bezüglich der Anwendungsdomänen und der Architekturdomänen verfügbar ist. Sie sind eine Ressource im Hintergrund der Modellierungsinfrastruktur, auf die für jedes spezifische Systementwurfsprojekt zurückgegriffen wird. Je umfangreicher und vollständiger diese Datenbanken sind, desto größer ist der Anteil an wiederverwendbarer Funktionalität, an automatisierbaren Optimierungs- und Synthesevorgängen, und umso effektiver kann ein modellbasierter Prozess seine Vorteile ausspielen und umso effizienter gestaltet sich ein spezifisches Systementwurfsprojekt.

Ein zentraler Punkt des modellbasierten Entwurfs ist die *Modelltransformation*. Diese kann entweder „waagerecht“ zwischen verschiedenen Sichten auf

dem gleichen Abstraktionsgrad desselben Systems geschehen, oder „senkrecht“ zwischen verschiedenen Abstraktionsebenen mit zunehmender Granularität bis hin zur Modell-zu-Code-Transformation. Eine Formalisierung der verfeinernden und konkretisierenden Transformation ist in Abbildung 2.7 dargestellt. Eine formalisierte Modellbeschreibung eines Problems aus dem Problemraum wird hier als *conceptual model* M(P) bezeichnet. Aus diesem Modell mit hohem Abstraktionsgrad wird durch Transformation ein Modell einer Lösung *model of solution* M(L) erzeugt.

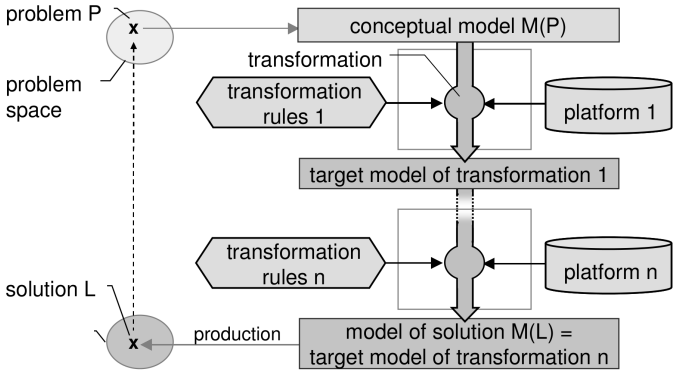


Abbildung 2.7: Modelltransformation im Model-driven Design [MG09].

Voraussetzung für eine Transformation ist die Formalisierung aller die Lösung bestimmenden Faktoren - die Plattform und die Transformationsvorschriften. Wie im Konzept des plattformbasierten Entwurf definiert ist, beschreibt die Plattform alle wiederverwendbaren Artefakte und Teillösungen mit Schnittstellen und Eigenschaften, auf die die Elemente des Quellmodells abgebildet werden können. Dabei muss die Plattform nicht ausschließlich eine Implementierungs- oder gar Hardware-Plattform sein, ebenso sind hierunter die Modellierungsebenen mit verschiedenen Abstraktionsgraden und Ausdrucksmächtigkeiten zu zählen. Die Transformationsregeln enthalten das Expertenwissen über die Abbildungs-, Konfigurations- und Kompositionsregeln der generierten Teillösungen. Manche Transformationen können automatisiert werden, manche erfordern den steuernden Eingriff des Designers. [HS07, MG09]

Als wichtigste Stufen im Verlauf eines modellbasierten Entwurfs und sukzessiven plattformorientierten Verfeinerungs- und Transformationsprozesses können die Folgenden charakterisiert werden:

Computation-independent Model (CIM) - das von der Ausführungssemantik unabhängige Modell
Das CIM wird auch als domänenspezifisches Modell bezeichnet und beruht auf einer Sprache bzw. ist aus Konstrukten aufgebaut, die den Experten des Anwendungsfeldes vertraut sind. Es wird dabei davon ausgegangen, dass Domänenexperten mit den Anforderungen des Anwendungsfeldes vertraut sind, allerdings nicht mit den Mitteln, wie diese Anforderungen in entsprechende Artefakte zu überführen sind. Das CIM soll daher die Lücke zwischen der Spezifikation durch die Experten des Anwendungsfeldes auf der einen Seite und dem Entwurf des Systems zur Erfüllung der Spezifikation auf der anderen Seite schließen.

Platform-independent Model (PIM) - das plattformunabhängige Modell
Das PIM beschreibt ein System in einer von der konkreten Ausführungsplattform unabhängigen Art und Weise. So kann das Gesamtsystemverhalten modelliert werden, ohne bereits auf spezifische implemtierungsorientierte Aspekte, wie HW/SW-Partitionierung oder Abbildung auf eine verteilte Berechnungs- und Kommunikationsplattform Rücksicht nehmen zu müssen.

Platform-specific Model (PSM) - das plattformspezifische Modell
Ein PSM bildet die spezifizierte Gesamtsystemfunktionalität auf die Charakteristika der zugrundeliegenden Ausführungsplattform ab. Es besteht aus implementierbaren Funktionsblöcken und beschreibt ihr konkretes Zugriffsverhalten auf die im PSM mit erfassten Plattformelemente, die als Plattformmodell in den Modellierungsprozess einfließen.

Platform Model - das Modell der Ausführungsplattform
Das Plattformmodell stellt die Menge von technischen Konzepten, Teilen und Diensten, die eine konkrete Ausführungsplattform zur Verfügung stellt, auf Modellebene dar und spezifiziert die benutzung dieser durch die Funktionalitäten des plattformspezifischen Modells. [MM03]

Die Definition von Modellierungsplattformen und Transformationen, sowie die Entwicklung von Implementierungsplattformen und Architekturen fallen in die vom eigentlichen Systementwurfsprojekt unabhängige Infrastruktur-Entwicklungsphase. Hier werden die Artefakte geschaffen, die in den eigentlichen Systementwürfen wiederverwendet werden können. Damit werden möglichst automatisiert Transformationen realisiert - Modell-zu-Modell, um aus abstrakteren detailliertere Modelle zu schaffen, oder Modell-zu-Code, um aus plattformspezifischen Modellen Implementierungen ableiten zu können. [MG09]

2.5 Modellbasierter komponentenorientierter Entwurf für verteilte Plattformen

Die Zusammenführung und Spezifizierung der Ideen aus dem plattformbasierten, komponentenorientierten Entwurf und dem Model-based Design erlauben es, einen Verfahrensraum für den komponentenorientierten Entwurf eingebetteter Applikationen für verteilte Systeme zu umreißen, der in Abbildung 2.8 dargestellt ist [ZAM+10]. Im wesentlichen wird hier ein Top-Down-Systementwicklungsprozess dargestellt, der eine auf Modellebene spezifizierte und validierte Funktionalität in Form einer eingebetteten Applikation auf eine dafür zugeschnittene möglichst optimale Hardware-Struktur implementiert. Zusätzlich sind die Bottom-Up-Prozesse dargestellt, die die für eine modellbasierte Entwicklung nötigen Artefakte und Infrastrukturen bereitstellen. Die Repräsentationen dieser Komponenten, Strukturen und Methoden sind in Form einer modellbasierten Entwurfsplattformebene zusammengefasst.

2.5.1 Der Top-Down-Systementwurf

Der *System-Design*-Zweig des Entwurfsprozessraumes stellt den eigentlichen Vorgang der Schaffung eines eingebetteten Systems zur Erfüllung der spezifizierten Anforderungen dar. Er entspricht im wesentlichen den im W-Modell angegebenen Entwurfs- und Implementierungsaktivitäten. Die Top-Down-Entwicklung zerfällt in zwei durch die Modellierungsplattformebene unterteilte Phasen.

Modellbasierter Systementwurf und Verfeinerung: Von oben nach unten betrachtet, ist die erste Phase der eigentliche modellbasierte Entwurf, der die Erschaffung einer Modellrepräsentation der Struktur und des Verhaltens der eingebetteten Applikation auf der Hardware zum Resultat hat. Diese Entwurfsphase beinhaltet die hauptsächlichen ingenieurtechnischen Entscheidungen während eines problemspezifischen Entwurfs und endet auf der modellbasierten Entwurfsplattformebene. Aus den funktionalen Spezifikationen (*Functional specification*), die auch außerhalb der zum eigentlichen Systementwurf benutzten Umgebung zusammengetragen werden können, entsteht ein Spezifikationsmodell (*Specification model*). Daraus wird im Zuge der Systemmodellierung ein plattformunabhängiges Funktionsmodell (*Platform-independent function model PIM*) erstellt, das die Funktionalität des künftigen Systems beschreibt und für das eine funktionale Validierung durch Simulation durchgeführt werden kann. Ziel der Verfeinerung des Entwurfs auf Modellebene ist die Transformation der Systembeschreibung in ein plattformspezifisches

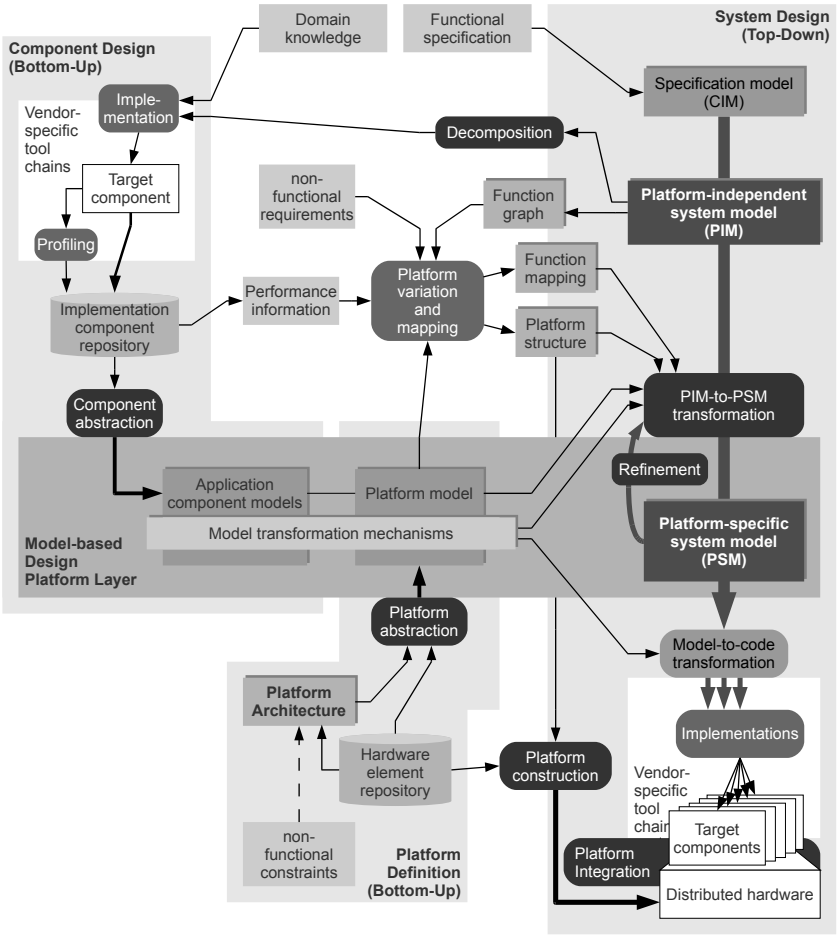


Abbildung 2.8: Übersicht über den modellbasierten Entwurfsprozess für verteilte eingebettete Systeme.

Systemmodell (*Platform-specific system model PSM*), das die Eigenschaft besitzt, durch Modell-zu-Code-Transformationen direkt in eine Form überführt werden zu können, die mittels plattform- oder herstellerspezifischer Werkzeuge auf der Zielplattform implementiert werden kann. Diese Transformation von

einem plattformunabhängigen Modell zu einem plattformspezifischen Modell wird als *PIM-to-PSM Transformation* bezeichnet.

Für verteilte Systeme ist die Zuordnung der zu implementierenden Funktionalität zu Plattformelementen, auf denen sie ausgeführt werden sollen, eine wichtige Entwurfsentscheidung. Für nicht-statische parallele oder verteilte Plattformen kann außerdem die Plattformstruktur den Anforderungen der Anwendung angepasst werden. Der Teilprozess zur Abbildung der Anwendungsstruktur auf eine Plattformstruktur (*Architecture variation and mapping*) leitet für die analysierte Anwendungsstruktur (*Function graph*) eine möglichst optimale Plattform-Struktur ab - eine feste Komposition aus Berechnungsressourcen und Kommmunikationsstrukturen. Diese Informationen werden in Form von Beschreibungen der Plattformstruktur (*Platform structure*) und Verteilungsvorschrift (*Function mapping*) dem Top-Down-Systementwurfsprozess zur Erstellung eines plattformspezifischen Modells zugeführt.

Implementierung und Integration: Unter der Modellierungsplattformebene schließt sich die zweite Top-Down-Entwicklungsphase an, die zunehmend automatisiert wird. Sie besteht erstens aus der auch als Code-Generierung bezeichneten Modell-zu-Code-Transformation, bei der das plattformspezifische Modell in implementierbaren Code (z.B. C, C++ für Mikrocontroller, DSPs oder entsprechende Soft-Prozessoren für SoCs, VHSIC Hardware Description Language (VHDL) für FPGA-Designs und applikationsspezifische Intellectual Property (IP)-Cores) überführt wird. Die anschließende Implementierung übernehmen hardware- und/oder herstellerspezifische Werkzeuge (Compiler, Crosscompiler, Synthese-Tools). Im Falle des Entwicklungprozesses für eine Multi-FPGA-Plattform würde sich hier für jedes Chip-Design der in Abschnitt 3.2.3 und Abbildung 3.8 beschriebene Logik-Synthese- und -Implementierungsprozess einfügen. Parallel zur Implementierung wird gegebenenfalls die Konstruktion der verteilten modularen Hardware-Plattform vorgenommen, mit der die entstandenen Implementierungen integriert werden. Daran anschließend verlaufen die im W-Modell beschriebenen weiteren Integrations- und Validierungsphasen für das zu entwickelnde System ab.

2.5.2 Die Bottom-Up-Definition der modellbasierten Entwurfsplattformebene

Die Entwurfsplattformebene ist hier eine Modellebene, die Modellelemente von Komponenten und Infrastrukturen für den modellbasierten Entwurf von anwendungsspezifischen Systemlösungen zur Verfügung stellt. Sie wird daher als

modellbasierte Entwurfsplattformebene (Model-based Design Platform Layer) bezeichnet. Die Definition dieser Ebene und deren möglicher Erweiterungen im Kontext der jeweiligen Systementwurfsprozesse umfasst sowohl die nötigen Methoden für Modell-zu-Modell- und Modell-zu-Code-Transformationen, als auch abstrakte Repräsentationen der Hardware-Plattform und -infrastruktur sowie spezieller Berechnungskomponenten auf Modellebene. Allerdings existiert dabei das so genannte „Dilemma der ersten Generation“ (the first-generation dilemma). Beim Schaffen einer neuen Entwurfsplattform existieren viele der Komponenten und Modellelemente noch nicht, wurden noch nicht beschafft und eingebunden oder selbst entworfen [QBJ05]. Dies ist ein Problem, das gerade in Forschung und Entwicklung sowie in extremen oder ausgefallenen Anwendungsdomänen auftritt. Diese Komponenten als wiederverwendbare Artefakte zur Schaffung von plattformspezifischen Systemmodellen zur Verfügung zu stellen, ist das Ergebnis von Bottom-Up-Entwürfen. Sie tragen zur Vervollständigung der Entwurfsplattformebene bei, worauf sich die eigentliche Effizienz des Gesamtprozesses des plattformbasierten bzw. modellbasierten Entwurfs begründet.

Komponentenentwurf: Während der Erstellung des plattformspezifischen Modells kann die Notwendigkeit entstehen, gewisse spezifizierte Funktionen durch spezielle Implementierungskomponenten ersetzen zu müssen. Dies kann entweder durch mangelnde Ausdrucksmächtigkeit der modellbasierten Entwurfsplattformebene motiviert sein, sodass zusätzliche Funktionalität als externe Komponente auf Implementierungsebene eingefügt werden soll, oder aber einfach durch das Bestreben, wiederverwendbare Komponenten zu benutzen, um den Entwurfsaufwand niedrig zu halten. Für beide Fälle ist die Erstellung von Komponenten in einem externen Entwicklungsprozess (*Component design*) nötig. Dieser basiert, wie in [LTT11] angegeben, auf Expertenwissen über die Anforderungen der Anwendungsdomäne (*Domain knowledge*), das auch durch *Dekomposition* des plattformunabhängigen Modells und Analyse der spezifizierten Funktionen akquiriert werden kann. Die Komponentenimplementierung (*Implementation*) produziert zweierlei Ergebnisse: erstens eine/mehrere wiederverwendbare plattformspezifische Komponentenimplementierung(en) (*Target component*), und zweitens, durch *Profiling* ermittelte Leistungsinformationen (*Performance information*), die das Verhalten der Funktionskomponente auf bestimmten Ausführungsplattformen charakterisieren, was wiederum für den Optimierungsprozess der Plattformabbildung von Bedeutung ist. Der wesentliche Beitrag der isolierten Komponentenentwicklung zum modellbasierten Systementwurf entsteht durch die Komponentenabstraktion (*Component abstraction*) und die Integration der entsprechenden Komponentenmodelle (*Application component models*) in die Bibliotheken der modellbasierten Entwurfsplattformebene. Diese stehen dann

als Black-Box-Blöcke für die Erstellung des PSM zur Verfügung, während die entsprechenden wiederverwendbaren Komponenten im Zuge der Code-Generierung in den Gesamtentwurf eingefügt werden.

Architekturabstraktion und -modellierung: Die plattformspezifische Modellierung macht es nötig, gewissen Charakteristika der Ziel-Hardware-Plattform gerecht zu werden. Dazu muss neben der Verfügbarkeit von Berechnungselementen auch die anderer physisch verfügbarer Hardware-Ressourcen, wie zum Beispiel I/O- und Speicherschnittstellen, Interprozessorkommunikation oder Intermodulkommunikation in verteilten Systemen in Betracht gezogen werden. Die modellhafte Abstraktion der Zielplattform (*Platform abstraction model*) verkörpert solche Aspekte in Form von verhaltensgleichen Modellkonstrukten in Form des Plattformmodells (*Platform model*), zusammen mit entsprechenden Validierungsverfahren und den Code-Generierungsvorschriften, die die Abbildung auf die tatsächlichen Plattformelemente während der automatischen Implementierung vornehmen. Die Schaffung der Plattformmodellelemente beruht auf einer Bottom-Up-Abstraktion der Plattform-Architektur (*Platform architecture*), dem Wissen um die Eigenschaften der Hardware- und Kommunikationselemente und den Kompositionsregeln für konkrete Plattformrealisierungen.

2.6 Fazit - Modellbasierte Implementierung in der Mess- und Automatisierungstechnik

Der Stand der Technik in der Umsetzung eines geschlossenen plattformbasierten und modellbasierten Entwurfsprozesses für die Anwendungsdomäne der Mess- und Automatisierungstechnik wird von graphischen Entwurfs- und Simulationswerkzeugen wie TheMathworks' MATLAB/Simulink®[2] oder National Instruments' LabVIEW®[3] mit einer steigenden Zahl von Erweiterungen mitbestimmt [Kra04]. Prototyping und Inbetriebnahme von mit diesen Werkzeugen entworfenen Anwendungen wird umfangreich von Plattform-Herstellern wie z.B. dSPACE®[4] oder National Instruments® unterstützt, die sowohl die eingebetteten Hardware-Plattformen als auch die plattformspezifischen Entwicklungswerkzeuge zur Verfügung stellen. Letztere, wie dSPACE RTI® [dSP12a] und LabVIEW RT® [Nat12b] bieten Modellabstraktionen der Ausführungsplattformen und erlauben die nahtlose Integration modellierter Lösungen als Software für die eingebetteten Prozessoren. Diese Fähigkeiten wurden auf Multi-Core-Systeme und auch auf modulare heterogene Systeme

[2] TheMathworks und MATLAB/Simulink: http://www.mathworks.com/products/simulink/
[3] National Instruments und LabVIEW: http://www.ni.com/labview
[4] dSPACE: http://www.dspace.com

mit FPGA-Modulen erweitert (dSPACE RTI FPGA® [dSP12b] und LabVIEW FPGA Module® [Nat12a]), die es erlauben, eine heterogene Plattform auf die spezifischen Anforderungen der eingebetteten Anwendung hin zuzuschneiden. Zusätzlich gibt es auch von den FPGA-Herstellern unternommene Bestrebungen, das Logik-Design in die höhere Abstraktionsebene der plattformspezifischen Modellierung zu integrieren. Zu nennen wären hier die in MATLAB/Simulink® integrierten Tool-Boxen SystemGenerator® [Xil11] von Xilinx®[5] und DSP-Builder® [Alt11] von Altera®[6], die umfangreiche Funktions- und Schnittstellenblockbibliotheken zur Verfügung stellen und die transparente Anbindung zur Hardware Description Language (HDL)-Code-Ebene oder gar direkt zur automatisierten Synthese mit den herstellerspezifischen Tool-Chains realisieren. Die State-of-the-Art-Entwurfswerkzeuge und ihre jeweiligen plattformspezifischen Erweiterungen erlauben es, eine funktionale Spezifikation und, daraus abgeleitet, eine Repräsentation der eingebetteten Lösung zu entwerfen, und dabei auf der selben Entwurfsebene zu verbleiben - der Modellierungsebene. Der eigentliche Implementierungsvorgang ist vollständig automatisiert und transparent für den Ingenieur. Dieser ideale Zustand beschränkt sich allerdings auf den Umfang der vom Hersteller zur Verfügung gestellten modellbasierten Entwurfsplattform - dem Gesamtverbund aus Hardware-Architektur, deren Abstraktion auf Modellebene und die automatischen Transformationsvorschriften - oder erfordert projektspezifische Erweiterungen der Modellierungsplattformebene mittels Bottom-Up-Entwurf von zusätzlichen Komponenten und Modelltransformationen.

Die zunehmende Automatisierung der unteren Implementierungsschritte durch Tools der Plattformhersteller macht die Transformation eines plattformunabhängigen Systemmodells in ein plattformspezifisches Implementierungsmodell (*PIM-to-PSM transformation*) zum entscheidenden implementierungstechnischen Schritt während des modellbasierten Entwurfs eingebetteter Systeme.

[5] Xilinx, Inc.: http://www.xilinx.com
[6] Altera Corporation: http://www.altera.com

3 Rekonfigurierbare Hardware-Plattformen als Teil eingebetteter Systeme

Der Fokus bei der Betrachtung von Hardware-Plattformen zur Realisierung eingebetteter Systeme liegt in dieser Arbeit auf Field-Programmable Gate Arrays (FPGA) - einer rekonfigurierbaren Hardware-Technologie, die ob ihrer Flexibilität, geringen Kosten für kleine Stückzahlen und kurzen Entwicklungszeiten gerade für Entwicklungs- und Prototyping-Projekte das Mittel der Wahl ist, und aufgrund ihrer zunehmenden Leistungsfähigkeit in viele Anwendungsbereiche vordringt, die bisher ASIC-Realisierungen vorbehalten waren.

In diesem Kapitel werden kurz die Grundlagen der FPGA-Technologie zusammengefasst, bevor auf Multi-FPGA-Systeme als spezialisierte Plattformlösungen eingegangen, sowie der Begriff des „System-on-Chip“ (SoC) als Stand der Technik auf dem Gebiet der integrierten Schaltkreise erläutert wird. Es wird weiterhin ein Überblick über den Entwurf digitaler Logik und ihrer Synthese zur FPGA-Konfiguration gegeben, mit dem Fokus auf der komponentenorientierten Entwicklung, die die Grundstrukturen für zunehmend komplexe Entwürfe auf der Basis wiederverwendbarer Design-Elemente beschreibt. Ein Fazit charakterisiert die Rolle FPGA-basierter Realisierungen als Teil eingebetteter Systeme für die Mess- und Automatisierungstechnik.

3.1 Field-Programmable Gate Arrays

Ein FPGA ist ein Logikbaustein, der nach der eigentlichen Silizium-Fabrikation und vor allem auch nach der Integration in das eingebettete Zielsystem, rekonfigurierbar (oder reprogrammierbar) ist - daher „field-programmable“. Die Rekonfigurierbarkeit eines FPGA, also die Möglichkeit, die anwendungsspezifische Schaltkreisrealisierung auf dem Chip auszutauschen, wird dadurch erreicht, dass im Silizium nur eine generische feingranulare Grundstruktur existiert, in der eine große Anzahl von Logikblöcke in einer prorammierbaren Verbindungsmatrix angeordnet sind und von programmierbaren I/O-Blöcken an der Chip-Peripherie umgeben sind. Die Logikblöcke selbst können in

ihrer Funktion konfigurierbare Logikzellen enthalten, aber auch Speicherblöcke oder dedizierte Berechnungsressourcen wie Multiplizierer. [KTR07, Gro08, BFRV92]

3.1.1 Technologieüberblick Field-Programmable Gate Arrays

Die eigentliche Funktionalität des FPGA wird in den Logikzellen umgesetzt. Wie z.B. die einfache Logikzelle (*Configurable logic block (CLB)*) in Abbildung 3.1 enthalten diese Zellen *Look-up Table (LUT)*, *Flip-Flop (FF)* und Multiplexer. Eine LUT dient dabei der Realisierung boolescher Funktionen, welche hier ähnlich einer Wertetabelle abgespeichert werden können. Ergebnisse dieser Berechnungen lassen sich dann in Flip-Flops zwischenspeichern oder weiter kombinatorisch mit LUTs folgender Logikzellen verknüpfen. Die Mächtigkeit der FPGA-Technologie beruht auf der heutzutage realisierbaren Anzahl von Millionen von verknüpfbaren Logikzellen auf einem Chip.

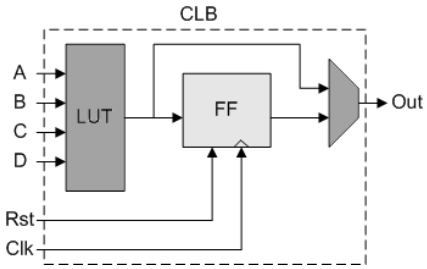


Abbildung 3.1: Struktur eines einfachen konfig. Logikelements [KTR07].

In aktuellen FPGAs wird die konfigurierbare Logik in hierarchischen Strukturen zusammengefasst, die sich allerdings von Hersteller zu Hersteller und zwischen Produktfamilien hinsichtlich ihrer Ausprägung, Anzahl, Verbindungen und Komplexität unterschieden. Anordnungen von LUTs und FFs werden meist als *Slices* bezeichnet und sind in modernen FPGAs deutlich kompizierter als das in Abbildung 3.1 dargestellte Element. Zusäzlich besitzen die meisten FPGAs noch spezielle Hardwareblöcke, welche, anders als die Logikzellen, eine fest vorgegebene Funktion erfüllen. Zu nennen wären hier z.B. Digital Clock Managers (DCMs), interne Block-RAM-Blöcke (BRAM), spezielle DSP-Ressourcen wie Multiplizierer, oder gar fest konfigurierte I/O-Schnittstellen-Controller und Prozessorkerne. Die gewünschte Funktionalität, die in Form eines digitalen Schaltkreises auf dem FPGA realisiert werden soll, wird *Entwurf*

oder *Design* genannt. Es wird von einem Hardware-Design gesprochen, auch wenn die benötigten applikationsspezifischen Gatterstrukturen nie wirklich physisch realisiert werden. Stattdessen werden sie auf die gegebenen granulare Strukturelemente des FPGA abgebildet. Die daraus resultierende Belegung der LUTs, Flip-Flops und Multiplexer in den Logikblöcken sowie der Signalrouten zwischen den Logikblöcken wird als *Konfiguration* bezeichnet. Allein damit lassen sich komplexe Berechnungen und Zustandsmaschinen darstellen - beschränkt wird dies durch die Logikkapazität des FPGA in Relation zum Ressourcenbedarf des synthetisierten Entwurfs. Die Erstellung einer korrekten Chip-Ressourcen-Konfiguration, die die gewünschte Funktionalität unter den gegebenen Beschränkungen realisiert, ist das Ergebnis eines mehrstufigen Synthese- und Implementierungsprozesses. [SS10, CH06, KTR07]

3.1.2 Multi-FPGA-Plattformen

Die Verknüpfung mehrerer FPGAs vergrößert die nutzbare Chip-Fläche künstlich und erlaubt die Implementierung ressourcenintensiver Systementwürfe in rekonfigurierbarer Hardware. Multi-FPGA-Systeme können anhand ihres dem Einsatzzweck angepassten Aufbaus unterschieden werden.

Multi-FPGA-Plattformen für Prototyping-Aufgaben beim ASIC-Entwurf verfügen meist über mehrere auf einer Platine angeordnete FPGAs, die untereinander mit möglichst breiten parallelen Input/Output (I/O)-Verbindungen möglichst direkt verbunden sind. Darauf soll die künftig in Silizium zu fertigende Logik getestet werden, was aufgrund der großen Integrationsdichteunterschiede der Zieltechnologien die Kapazität einzelner FPGAs sehr schnell überschreitet. Kommerzielle Plattformen dieser Art werden von Firmen wie HitechGlobal[1] oder DINIGroup[2] entwickelt. Ein Beispielsystem ist in Abbildung 3.2 (links) dargestellt. Zum Berechnen massiv paralleler und sehr gut skalierbarer Aufgaben im Bereich des High-Performance-Computing kommen Multi-FPGA-Systeme zum Einsatz, die sich durch modulare Aufbauten und hierarchische Kommunikationsstrukturen auszeichnen. Ein Beispielsystem dieser Art ist das Copacobana-System[3], das in Abbildung 3.2 (rechts) gezeigt wird. In der industriellen und medizinischen Mess- und Regelungstechnik sowie in der Forschung werden FPGA-Aufbauten zur Erfassung und Verarbeitung großer Messdatenmengen

[1] HitechGlobal, http://www.hitechglobal.com
[2] DINIGroup - Big FPGA Boards High Performance Comupting, http://www.dinigroup.com
[3] Copacobana, http://www.copacobana.org

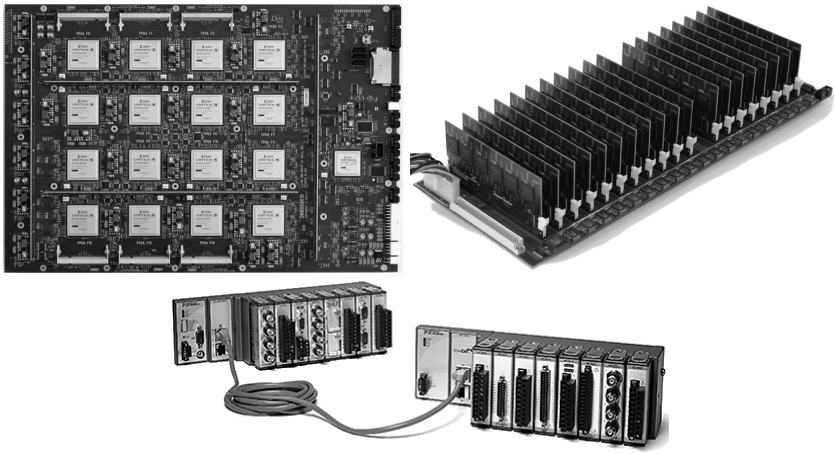


Abbildung 3.2: Multi-FPGA-Systeme: DN9000K10 von DINIGroup (links), Copacobana (rechts), CompactRIO-Verbund von National Instruments (unten).

eingesetzt. Die mögliche Vielzahl von Messstellen und Kanälen erfordert auch hier einen Verbund von multiplen FPGAs, teilweise auch mit loserer Kopplung zwischen physisch weiter verteilten Modulen. Modulare eingebettete Messdatenverarbeitungsplattformen von Herstellern wie National Instruments® oder dSPACE® beinhalten rekonfigurierbare I/O-Module, die den Aufbau von hierarchischen Multi-FPGA-Strukturen erlauben, wie beispielsweise in Abbildung 3.2 (unten) gezeigt ist. Multi-FPGA-Systeme stellen ebenfalls erweiterte Anforderungen an den Entwurfsprozess. Trotz standardisierter Intermodulkommunikation und der Tatsache, dass die einzelnen Chip-Konfigurationen gemäß des Prozesses in Abschnitt 3.2.3 isoliert implementiert werden können, liegt die Herausforderung im übergeordneten Gesamtsystementwurf, der das Verhalten der verteilten Applikation spezifiziert und dessen Umsetzung auf die Multi-FPGA-Implementierung sicherstellen soll.

3.1.3 System-on-Chip-Technologien

Um mit der steigenden Integrationsdichte und dem Leistungspotential der Halbleitertechnologie schritthalten zu können, sind Ansätze zur Produk-

tivitätsteigerung beim Entwurf durch Reduktion der Entwurfskomplexität und des Reimplementierungsaufwandes beim Chip-Entwurf erforderlich. Einer dieser Ansätze ist der des Designs mittels massiver Wiederverwendung vordefinierter Komponenten und Infrastrukturen. Ein Verbund von derartigen Komponenten, die über standardisierte Schnittstellen kommunizieren und auf einem Chip realisiert sind, wird als System-on-Chip (SoC) bezeichnet. Historisch gesehen wird damit auch die zunehmende Fähigkeit dokumentiert, neben Prozessoren und Speicher auch andere Hardware- und Schnittstellenelemente (digital und analog), die in getrennten Baugruppen auf einer Platine angeordnet waren (System-on-Board), mit auf einer Chipfläche unterzubringen [SWM+06]. Der Begriff SoC bezieht sich generell auf Hardware-Entwürfe, schließt also direkt in Silizium realisierte hochintegrierte Schaltkreise und ASICs mit ein. Im Zusammenhang mit der Implementierung auf rekonfigurierbarer Hardware, also FPGA, wird von daher auch von System-on-Reprogrammable-Chip (SoRC) gesprochen. Die Definition von SoRC beinhaltet dabei sowohl die Integration von Teilsystemen in mehreren Chips, als auch die Integration eines gesamten Systems auf einem einzigen FPGA. Die Herausforderungen bezüglich der Integration komplexer Systeme aus wiederverwendbaren Einheiten sind hierbei jedoch dieselben [Xil00].

Durch den komponentenbasierten Ansatz wird die Logik-Design-Problematik in zwei Ebenen geteilt - die Bottom-up-Entwicklung der Komponenten als High-Level-Blöcke für die Systemintegration und die Top-Down-Integration eines anwendungsspezifischen SoC innerhalb einer gewählten Architektur selbst.

On-Chip-Kommunikationsarchitekturen im Überblick

Der Datenaustausch zwischen IP-Cores innerhalb von SoCs geschieht über wohldefinierte Schnittstellen, die die Komponenten und die Kommunikation nach dem GALS-Paradigma entkoppeln und die Datenübertragungen synchronisieren. Eine Taxonomie der On-Chip-Kommunikationsverfahren ist in Abbildung 3.3 dargestellt.

Drei Hauptklassen von Kommunikationsarchitekturen werden hier unterschieden [MSCL06]:

- *Punkt-zu-Punkt-Verbindungen* sind Direktverbindungen zwischen den Datenports von Funktionsblöcken und bilden jeden logischen Kanal auf einen physischen, also eine Gruppe von Signalleitungen auf dem Chip, ab. Sie zeichnen sich vor allem durch Einfachheit und daraus resultierende deterministische Latenz und Performance aus. Hauptnachteile

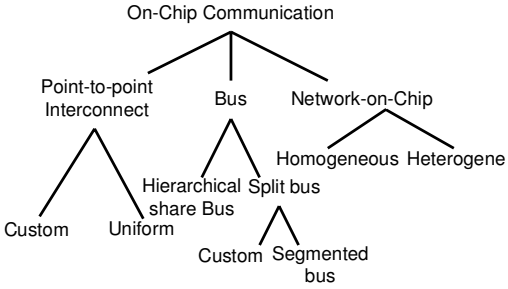


Abbildung 3.3: Taxonomie der On-Chip-Kommunikationsverfahren [MSCL06].

sind die schlechte Skalierbarkeit durch explodierenden Aufwand beim Routing der Signalpfade auf dem Chip, sowie eine geringe Nutzung der bestehenden Kommunikationsressourcen für Anwendungen mit geringem Kommunikationsdurchsatz.

- *Bussysteme* stellen einen einzelnen physischen Kommunikationskanal zur Verfügung der von mehreren logischen Kanälen benutzt wird. Die Schaffung eines Kommunikations-Backbones auf dem Chip reduziert massiv den Ressourcenbedarf für Signalleitungen. Allerdings macht dieses exklusive Kommunikationsmedium eine Arbitrierung der Zugriffe nötig, was die Skalierbarkeit einschränkt, da proportional zur Anzahl der Teilnehmer der Durchsatz geschmälert und die Latenzen erhöht werden. Verschiedene Techniken, wie hierarchische und segmentierte Busse, sowie Burst-Zugriffe und Split-Transactions adressieren diese Nachteile, machen aber das Verhalten des Systems u.U. schwer berechenbar. Eine aktuelle Übersicht über verbreitet eingesetzte on-Chip-Bussysteme und Techniken wird in [MS06] gegeben. Busbasierte SoC sind aktuell die für eingebettete Systeme am meisten verbreitete Architektur.

- *Network-on-Chip-Strukturen* realisieren eine Kommunikationsnetzwerkstruktur aus kurzen Signalpfaden zwischen Switch-Elementen, über die Daten paketorientiert übermittelt werden. Dies soll höhere Übertragungsraten und Datendurchsätze ermöglichen. Zusätzlich ist eine sehr gute Skalierbarkeit und, durch eine Lastbalancierung über die Vielzahl an möglichen Übertragungsrouten, eine hohe Leistungsfähigkeit auch bei hohen Teilnehmeranzahlen gegeben. Die Grundlagen zu NoC wurden z.B. in [BM06, AIS09] zusammengestellt. Gerade für die aufkommenden Multiprozessor-System-on-Chip (MPSoC), die massive Multi-Master-Systeme darstellen, sind NoCs die bevorzugte Architektur.

SoC-Design-Komponenten - IP-Cores

Für wiederverwendbare abgeschlossene Logik-Komponenten, die vorgefertigt und vorverifiziert in einen entstehenden Gesamtentwurf eingebunden werden können wird heute der Begriff *IP-Core* benutzt, wobei IP für „intellectual property“ steht. Synonym werden in der Fachliteratur auch Macro, Module, Block, IP und Core verwendet.Dabei kann es sich um eingebettete Prozessoren, Speicherbaugruppen, Schnittstellenkontroller oder applikationsspezifische Funktionsblöcke handeln. Je nach Ausführung werden Soft-IPs, Firm-IPs oder Hard-IPs unterschieden, die respektive auf HDL-Ebene anpassbar, nur noch parametrisierbar oder gar nur noch als Black-Box integrierbar sind, unterschieden [SWM+06, CH06].

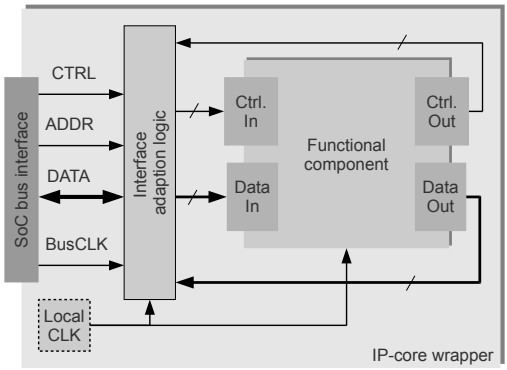


Abbildung 3.4: Schema eines IP-Cores mit on-Chip-Bus-Interface.

IP-Cores, die zu einer bestimmten on-Chip-Kommunikationsinfrastruktur kompatibel sein sollen, müssen die jeweiligen Anforderungen an Interface-Struktur- und verhalten erfüllen. Abbildung 3.4 zeigt schematisch die Struktur eines IP-Cores. Die Daten- und Steuerpfadschnittstelle (Data & Ctrl.) einer zentralen synchronen Funktionskomponente (Funtional component) wird mittels zusätzlicher Anpassungslogik (Interface adaption logic) an den spezifischen Daten- und Adress-Bus der on-Chip-Kommunikation angebunden. Diese lokalen, für die Systemlevelintegration transparenten Anpassungen werden in einem Wrapper zusammegefasst, der das tatsächliche Interface des IP-Cores beschreibt. Diese Maßnahmen zur Erhöhung der Integrierbarkeit erhöhen die Komplexität und auch den Ressourcen-Overhead der einzelnen Komponenten.

Das Streben nach möglichst robusten wiederverwendbaren Design-Elementen und deren nahtloser Integration führte zur Etablierung gewisser Regeln und Standardvorgehensweisen für Design und Verifikation, die unter Empfehlung der großen Chip- und Design-Tool-Hersteller im „Reuse Methodology Manual for System-on-Chip Design“ [KB02] zusammengetragen wurden. Dennoch ist die Formulierung von universellen IP-Core-Schnittstellendefinitionen und IP-Integrationsmethoden Gegenstand aktueller Forschung und Entwicklung.

Plattform-FPGAs

Die Kombination aus dem System-on-Chip und dem FPGA als großflächige rekonfigurierbare Basis zur Implementierung applikationspezifischer Funktionalität führte zum Aufkommen der sogenannten *Plattform-FPGAs*.

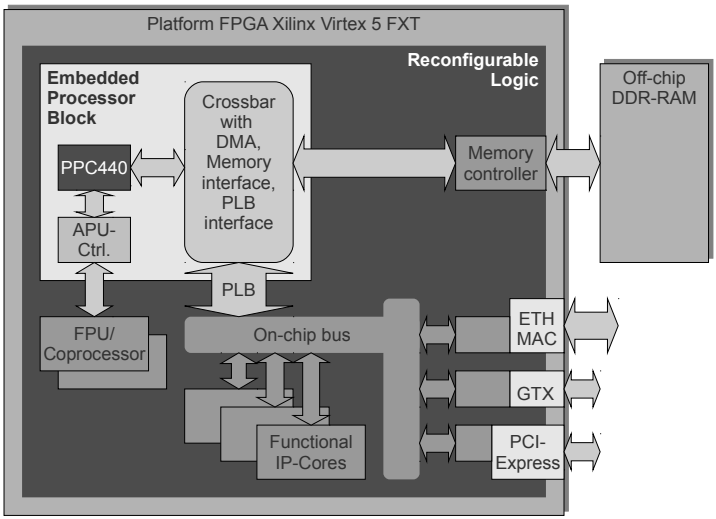


Abbildung 3.5: Plattform-FPGA Xilinx Virtex 5 mit PPC400-Kern.

Diese zeichnen sich durch fest auf dem Chip integrierte Teilsysteme aus, die Prozessorkerne, Speicherkontroller und Speicher, sowie verschiedene Standardschnittstellenkontroller als Hard-Cores enthalten können. Diese Struktur ist in Abbildung 3.5 am Beispiel des FPGA Virtex-5 FXT von Xilinx illustriert. Auf diesem Chip enthält der fest integrierte Embedded Processor

Block einen PowerPC-Kern (PPC440) inklusive Cache und einen Coprozessor-Controller (APU-Ctrl.), über den befehlssatzerweiternde Coprozessoren wie Floating-point-Einheiten als Softcores angefügt werden können. Weiterhin sind ein DMA-Controller, eine Speicherschnittstelle zur Anbindung von on-Chip und off-Chip-Speicher, sowie die Schnittstellen zum Processor Local Bus (PLB)[4] als on-Chip-Kommunikationsinfrastruktur in diesem festen Teil des System-on-Chip vorhanden. Auf dem FPGA selbst existieren noch feste I/O-Interface-Controller für Ethernet (ETH MAC), PCI-Express und serielle Hochgeschwindigkeitsverbindungen (GTX), deren Anbindung optional über frei konfigurierbare Blöcke über den PLB möglich ist. Dieses zentrale System-on-Chip stellt damit schon ein einsatzfähiges Rechensystem dar. Es definiert außerdem die on-Chip-Kommunikationsschnittstellen und kann über diese mit auf der frei konfigurierbaren Chipfläche untergebrachten Modulen - Soft-Cores - erweitert werden. Mittels zusätzlicher Prozessoren, DSPs, Funktions- und Peripherie-Komponenten ist die effiziente Implementierung applikationsspezifischer Architekturen möglich, die den Plattform-FPGA zur zentralen Basis eingebetteter Hard- und Software-Systeme werden lassen [SS10].

3.2 Entwurf und Synthese digitaler Systeme

Der Entwurf logischer Schaltungen ist die Grundlage zur Realisierung von Systemen mittels der FPGA-Technologie. Im folgenden wird zusammengefasst, wie das definierte Verhalten während der Synthese in logische Strukturen überführt und während des FPGA-Implementierungsprozesses auf die granularen Elemente der Chip-Struktur abgebildet wird.

3.2.1 Entwurf digitaler Logik

Die Realisierung Boolescher Funktionen als digitale Schaltungen resultiert in sogenannter *kombinatorischer Logik (combinational logic)*. Sie beschreibt logische Funktionen f, deren Ausgabe Y nur von der jeweiligen Eingabe X abhängt: $Y = f(X)$. Dagegen beschreibt *sequentielle Logik (sequential logic)* Funktionen, deren Ausgaben von einem gespeicherten Zustand (*State*) Z abhängen. Der Zustand selbst ist wiederum eine Funktion δ (Zustandsüberführungsfunktion) der jeweiligen Eingaben. Dieses Konzept wird als *Finite State Machine (endlicher Automat, Zustandsmaschine) (FSM)* bezeichnet. Zwei wesentliche Typen von FSMs werden unterschieden, die als Moore-Automat mit $Y = \mu(Z)$

[4] IBM CoreConnect PLB-Spezifikation: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture - Zugriff 6.11.2012

und $Z = \delta(X, Z))$ bzw. Mealy-Automat mit $Y = \mu(X, Z)$ und $Z = \delta(X, Z)$ bekannt sind [JDKR97, WH03, Gro08, VG02, Ash08].

Von *getakteter Logik* ist die Rede, wenn speichernde Elemente (Flip-Flops bzw. Register) in der Schaltung enthalten sind, die nur zu bestimmten Zeitpunkten durch einen externen globalen Taktgeber (*Clock*) aktiviert werden, um Ergebnisse oder Zustände zu übernehmen. Da in kombinatorischen Schaltungen die zeitliche Verfügbarkeit gültiger Ergebnisse von den einzelnen Signallaufzeiten durch die Schaltung abhängt, werden getaktete Flip-Flop-Stufen zur Synchronisation der Funktionsausführung eingesetzt, weswegen diese Logik auch als *synchrone Logik (synchronous logic)* bezeichnet wird[5] [RC03, Ash08]. Die Leistungsfähigkeit digitaler Schaltungen hängt von der Laufzeit des längsten kombinatorischen Pfades zwischen den Flip-Flop-Stufen ab, da diese die mögliche *Taktfrequenz* f synchroner Logik bestimmt. Aus der Anzahl der Flip-Flop-Stufen ergibt sich die benötigte *Taktanzahl* c zum Durchlaufen der Schaltung. Zur Charakterisierung synchroner Logik werden zwei Begriffe herangezogen: die *Latenz* $l = \frac{c}{f}$, und der *Durchsatz t (Throughput))* als Maß dafür, mit wie vielen Takten Abstand neue Eingangsdaten an die Schaltung angelegt werden, bzw. Ausgangsdaten abgegriffen werden können.

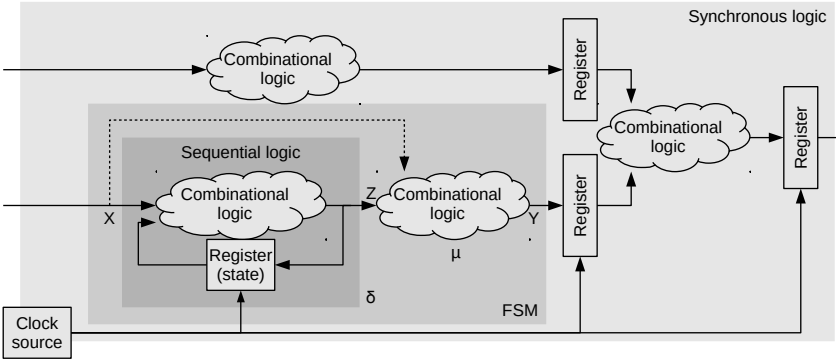


Abbildung 3.6: Synchron getaktete Logik.

Synchrones Design wird generell als vorteilhaft angesehen, da sich damit stabile Schaltungen realisieren lassen, die effizient zu implementieren, zu testen, zu debuggen und zu warten sind, weswegen viele Entwurfsansätze

[5] Als Gegensatz dazu gilt auf dieser Ebene der Begriff der asynchronen Logik, wenn anstelle eines Clock-Signales Handshake-Signale zur Synchronisation der Abarbeitung benutzt werden.

und „Good Design Practices“ darauf verweisen [KB02, CH06]. Die generelle Trennung eines Logikentwurfs in Register zur Ergebnisspeicherung und die dazwischen liegenden Übertragungsfunktionen zur Umwandlungen der Daten ist die grundlegende Betrachtungsweise des synchronen Entwurfsansatzes der Registertransferebene oder *Register-Transfer Level (RTL)* [Rus11].

3.2.2 Synthese

Als *Synthese* wird ein Prozess bezeichnet, der eine Systembeschreibung durch Komponenten und ihre Beziehungen auf einer niedrigeren Abstraktionsebene aus einer Beschreibung des erwarteten Verhaltens auf einer höheren Abstraktionsebene generiert [Mar11].

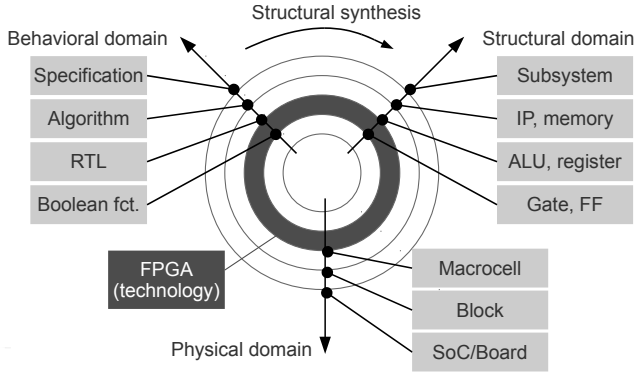


Abbildung 3.7: Gajski-Kuhn-Diagramm (nach [Gro08]).

Der Einstieg in den Prozess der Implementierung einer FPGA-Konfiguration wird generell mit Hardware-Beschreibungssprachen (HDL) wie VHDL oder Verilog durchgeführt. VHDL selbst ist seit 1987 standardisiert und deckt laut Spezifikation den gesamten Entwurfsprozess elektronischer Systeme von der Systemmodellierung in der Spezifikationsphase, über die Modellierung auf (RTL) in der Entwurfsphase bis zur Netzlistenerstellung in der eigentlichen Implementierungsphase ab. Zu bemerken ist hierbei generell, dass nur eine Teilmenge der syntaktisch korrekten HDL-Konstrukte auch in ein für FPGA synthetisierbares Design überführt werden kann [Rus11, CH06]. Die *Logiksynthese (logic synthesis)* ist die automatische Überführung von einem RTL- zu einem Entwurf auf der Gatterebene (*Gate level*) [Rus11]. Die Synthese von

RTL-Beschreibungen aus höheren, zeitunbehafteten Beschreibungen wird als *High-Level-Synthese* bezeichnet [GR94]. Für diese Phase werden heute auch andere Ansätze verfolgt, wie die Beschreibung in C, C++ oder SystemC, oder mit Hilfe von MATLAB oder grafischer Tools wie Simulink - allerdings haben auch diese zunächst eine RTL-Beschreibung als Zwischenrepräsentation vor der Implementierungsphase zum Ziel. Diese Zusammenhänge des Entwurfs und der Synthese elektronischer Systeme sind im sogenannten Y- oder Gajski-Kuhn-Diagramm [GK83,Gro08,GAGS09] in Abbildung 3.7 verdeutlicht. Die unterste Abstraktionsebene (Transistoren, ...) wurde hier weggelassen, da die FPGA-Technologie bereits eine Mindestgranularität aufweist, die sich auf die im Bild angedeuteten Abstraktionsebenen abbilden lässt.

Der wesentlichste Aspekt beim Entwurf digitaler Hardware ist der Übergang zwischen der Verhaltensdomäne (*Behavioral domain*) und der Strukturdomäne (*Structural domain*) bzw. darauf folgend der physischen Domäne (*Physical domain*). Für die FPGA-Entwicklung wird die letzte Phase heutzutage hauptsächlich durch automatisierte Implementierungswerkzeuge übernommen. Die wichtigsten Design-Entscheidungen bei der Struktursynthese (*Structural synthesis*) sind die *Ressourcenallokation (Resource allocation)* - die Bestimmung der Arten und Quantitäten der in der Chip-Architektur benötigten Berechnungsressourcen, und der kombinierte Prozess von *Scheduling und Bindung* - das Zuordnen der auszuführenden Operationen zu Ausführungszeitpunkten (Takten) und das Zuweisen der jeweiligen Operation zu einer entsprechenden Ausführungsressource [GR94, JDKR97, Rus11]. Die verwendeten Entwicklungsbibliotheken, -tools und -plattformen beeinflussen, auf welcher Abstraktionsebene der Domänenübergang realisierbar ist.

3.2.3 Implementierungsprozess für FPGA

In Abbildung 3.8 werden die Entwicklungsphasen der FPGA-Implementierung genauer dargestellt. Während der Entwurfsphase (*Design entry*) wird eine HDL-Beschreibung mittels *Synthese* in eine RTL-Netzliste (*Net list*) umgewandelt. Die HDL-Quellen können, wie in der Abbildung angedeutet, durchaus auch durch andere vorgelagerte Prozesse bereitgestellt werden. Zur Design-Zeit kann auf vorgefertigte Komponenten (IP cores) zurückgegriffen werden, die z.B. als HDL-Quellen in das Design oder als fertige Netzlisten während der Synthese eingebunden werden. Die eigentliche Implementierungsphase (*Design implementation*) beginnt mit dem *Mapping*-Prozess, also der Abbildung der Logik auf chip-spezifische Hardware-Elemente. Die tatsächliche Platzierung und Verdrahtung der Logik auf der Chip-Fläche geschieht während des *Place&Route*-Prozesses. Während der Implementierung

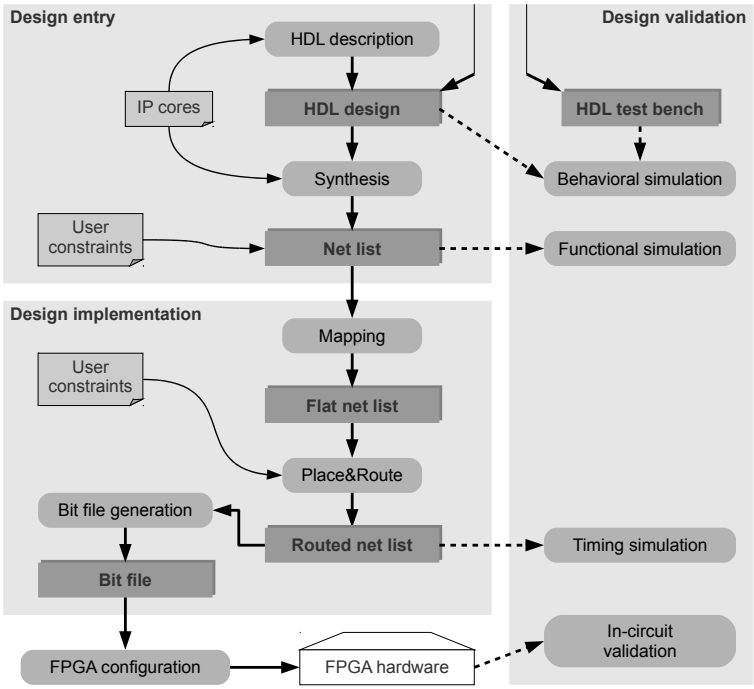


Abbildung 3.8: Der Entwicklungsprozess für FPGA ausgehend von einer HDL-Beschreibung.

werden neben den Beschränkungen durch die Hardware-Charakteristiken (Ressourcenverfügbarkeit, etc.) auch durch den Designer vorgegebene Beschränkungen (*User constraints*) berücksichtigt. Diese betreffen vor allem die Anbindung der inneren Logik an die Chip-Peripherie-Pins, Platzierung von Funktionen auf der Chipfläche, Zugehörigkeit von Logikblöcken zu Clock-Domains und Timing-Beschränkungen für Signale zur Garantie bestimmter Latenzen oder Taktfrequenzen.

Entwurfsbegleitend wird die Validierung (*Design validation*) durchgeführt. Während der Design-Phase kann eine Verhaltenssimulation (*Behavioral simulation*) mit HDL-Simulatoren durchgeführt werden. HDLs besitzen eine über die RTL-synthetisierbaren Konstrukte hinausgehende Mächtigkeit, mit der auch komplexe Testumgebungen (*Test benches*) zu schaffen sind, um Testfälle für das eigentliche synthetisierbare Design als Device-under-Test (DUT)

simulieren zu können. Auf den Netzlisten können funktionale Simulationen (*Functional simulation*) auf Gatterebene durchgeführt werden, während die eigentliche Simulation des Zeitverhaltens der Signale auf dem Chip (*Timing simulation*) erst auf der fertig gerouteten Netzliste (*Routed net list*) möglich ist. Nach der Konfigurierung des FPGAs mit dem generierten Bitfile des implementierten Designs kann auch eine In-Circuit-Validierung (*In-circuit validation*) durchgeführt werden, wenn die dazu nötigen Design-Elemente zur Analyse ausgewählter Signale und Ausgabe der Daten über definierte Schnittstellen in den Entwurf integriert worden sind. [Rus11, CH06, Gro08]

3.3 Komponentenbasierte Logik

Für leistungsfähige Logik spielt das Prinzip der Lokalität eine Rolle. Zusammengehörige Funktionalität in Form von Modulen oder Komponenten zu beschreiben und durch entsprechende Constraints ihre Platzierung auf dem Chip zu definieren, wirkt sich positiv auf Latenzen und Frequenzen, sowohl lokal als auch für den Gesamtentwurf, aus und ermöglicht nicht zuletzt die Wiederverwendbarkeit von Design-Elementen.

3.3.1 Lokalität und Integration

An Logikkomponenten wird die Anforderung gestellt, das interne Timing-Verhalten vom globalen zu entkoppeln, um die Wechselwirkungen zu anderen Komponenten zu minimieren und eine Unabhängigkeit vom applikationsspezifischen Top-Level-Design und von der physischen Platzierung auf der Chip-Fläche zu wahren. Dies geschieht vor allem durch das Vermeiden von Schnittstellen kombinatorischer Logik zu den äußeren Signalpfaden, sodass sich die Laufzeiten der Logikpfade (*logic path delay*) und die Signallaufzeiten durch die Routing-Ressourcen (*routing delay*) nicht akkumulieren und damit die Betriebsfrequenz des gesamten synchronen Entwurfs negativ beeinflussen. Das Mittel zur Isolation von Komponenten ist, wie in Abbildung 3.9 dargestellt, das Einfügen von Registerstufen an Dateneingängen und -ausgängen [KB02, Xil00]. Grundsätzlich ermöglicht dies auch die robuste Integration von Komponenten, die mit nicht synchronen Takten[6] betrieben werden.

Die Betrachtung der Interaktionen von unterschiedlich getakteten Logikkomponenten wirft erneut die Frage der Synchronität auf. Das Prinzip lokaler Synchronität und globaler Asynchronität, kurz „Globally Asynchronous Locally Synchronous (GALS)“-Paradigma, sieht Komponenten vor, die intern synchron

[6] Clock-Signale, die sich in Frequenz und/oder Phase unterscheiden

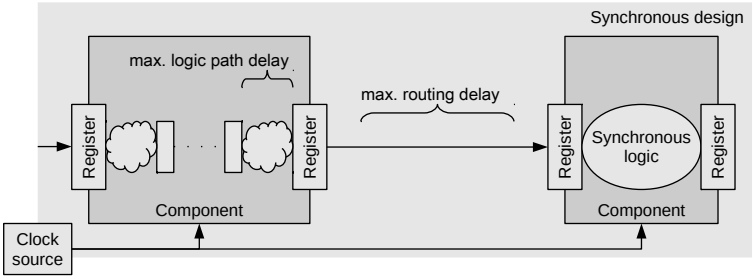


Abbildung 3.9: Durch I/O-Register entkoppelte Komponenten im synchronen Entwurf.

entworfen sind, aber mit anderen Komponenten Daten über asynchrone Mechanismen austauschen. Dies soll die Vorteile abgeschlossenen synchronen Designs mit den Vorteilen asynchroner Systeme, hauptsächlich des taktunabhängigen Routing der Daten über den Chip, kombinieren [RC03, CMSSV99]. Die Integration der lokal synchronen Komponenten erfolgt dabei entweder über Wrapper mit asynchroner Logik, gestaffelte Registerstufen oder, als robustester Ansatz, wie in Abbildung 3.10 dargestellt, über First-In-First-Out (FIFO)-Puffer, die die lokalen Taktdomänen der Komponenten entkoppeln und die Datenübergabe über Status- und Steuersignale synchronisieren [KGGV07].

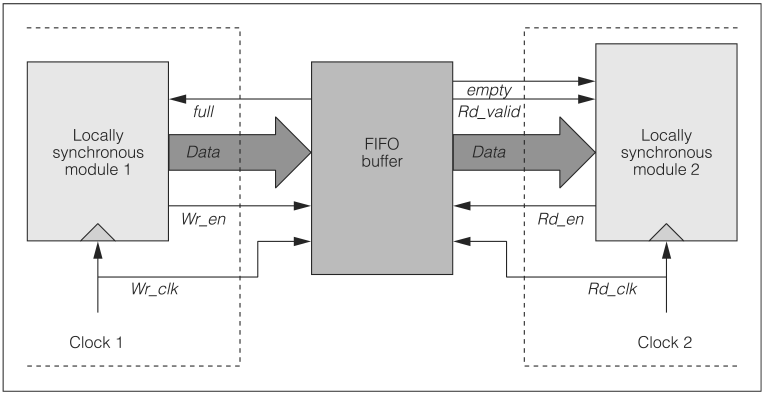


Abbildung 3.10: Asynchrone Kommunikation zweier lokal synchroner Komponenten via FIFO [KGGV07].

3.3.2 Komponentenarchitektur: Datenpfad und Steuerpfad

Ein universelles Architekturmodell für Rechenkomponenten, sowohl für generische Mikroprozessoren als auch für sehr anwendungsspezifische IP-Core-Designs, kann in zwei wesentliche Teile unterteilt werden - den *Datenpfad (Data path)*, in dem die eigentlichen funktionalen Berechnungen ausgeführt werden, und den *Steuerpfad (Control path)*, in dem Steuerentscheidungen über den Datenpfad getroffen und umgesetzt werden. Der Steuerpfad wird üblicherweise in Form einer oder mehrerer paralleler FSM definiert [LAD06, VG02]. Diese kann flach in Form sequentieller Logik oder als Programm für einen Mikrorechenkern realisiert werden. Wie in Abbildung 3.11 dargestellt, ist die Steuerpfad-Hardware durch Statussignale (*Status signals*) und Steuersignale (*Control signals*) mit dem Datenpfad verbunden. Externe Steuersignale (*External control signals*) und die Statussignale des Datenpfades bedingen die internen Zustandsüberführungen, während über Ausgaben von Steuersignalen auf das Verhalten des Datenpfades Einfluss genommen wird. Die Steuersignale sorgen für eine korrekte Selektion, Parametrisierung und Aktivierung der Einheiten des Datenpfades, sowie für die Ausführungssynchronisation zwischen den beteiligten Einheiten [JDKR97].

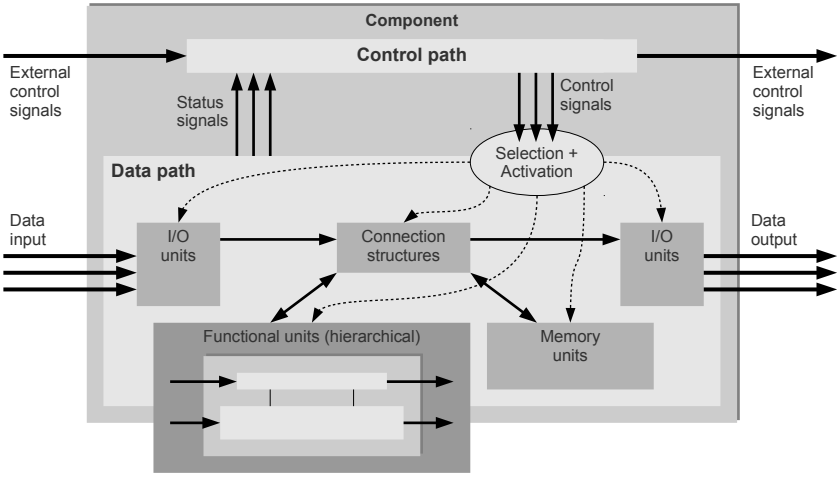


Abbildung 3.11: Komponentenarchitektur mit Datenpfad und Steuerpfad (nach [JDKR97]).

Der Datenpfad besteht aus vier grundsätzlichen Arten von Einheiten:

- *Ein-/Ausgabe-Einheiten (I/O units)*, die die Schnittstellen zur externen Kommunikation realisieren,
- *Speichereinheiten (Memory units)* zum Halten der Variablen während der Ausführung der Funktion, z.B. Register, RAMs, ROMs,
- *Verbindungsstrukturen (Connection structures)* für die Datenübergabe zwischen den Einheiten, z.B. Busse, Multiplexer, Switches, und
- *Funktionseinheiten (Functional units)*, die die eigentliche spezifizierte Funktionalität auf den Daten ausführen, z.B. arithmetische Operatoren, ALUs, Co-Prozessoren.

Diese Architektur lässt sich hierarchisch gliedern, sodass ausgehend vom Top-Level-Design jede Datenpfad-Funktionseinheit selbst diese Architektur aufweist, und das Gesamtsystem über hierarchische Daten- und Steuerpfade mit unterschiedlichen Komplexitäten auf verschiedenen Ebenen verfügen kann.

3.4 Fazit - FPGAs in mess- und automatisierungstechnischen Anwendungen

Die zunehmende Leistungsfähigkeit und der steigende Grad der Integration elektronischer Bauteile, inklusive der der FPGAs, ermöglicht auch technologische Veränderungen in der Automatisierungstechnik. Ausgehend von der klassischen Automatisierungspyramide treten folgende Effekte hervor: Geräte der Feldebene (*Field level*) werden zunehmend „intelligenter“, ihre steigende Integration mit der Sensor-/Aktor-Ebene (*Sensor/actuator level*) bringt „Smart sensors“ und „Smart actuators“ hervor, und vereinheitlicht die Kommunikationshierarchien. Außerdem erlaubt ihre steigende Rechenleistung die Integration von Funktionalität der Steuerungsebene (Control level) zu dezentralisierten Automatisierungssystemen [VHKBW09]. Abbildung 3.12 illustriert die Rolle von SoC- und FPGA-basierten Plattformen in der Feldebene. Die direkte Kopplung der frei konfigurierbaren Logik mit den A/D- und D/A-Schnittstellen zur Sensorik/Aktorik macht die effektive Realisierung der zeitkritischen Datenpfade prozessnaher Signalverarbeitungs- und Regelungsfunktionalität (*Distributed signal processign & control*) in Hardware möglich. Die Verfügbarkeit von Universalrechenkernen als Soft- oder Hard-IP-Cores erlaubt die Integration von höheren Funktionen der Feld- und Steuerungsebene in Software. Spezifisch implementierte bzw. standardisierte Kommunikationsschnittstellen ermöglichen sowohl die Intermodulkommunikation mit Instanzen

auf der Feldebene (*Field level communication*) als auch die Kommunikation mit denen höherer Ebenen (*Control level communication*).

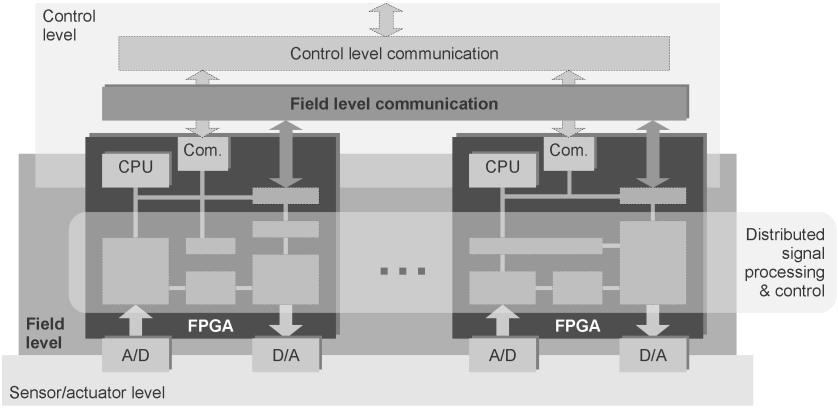


Abbildung 3.12: FPGAs in Automatisierungssystemen.

Der Entwurf verteilter Automatisierungssysteme beinhaltet damit den Entwurf und die verteilte Hardware-Implementierung von Signalverarbeitungs- und Regelungsrealisierungen als Teil eines heterogenen Gesamtsystems. Der Fokus beim Entwurf des Datenpfads der mess- und regelungstechnischen Anwendung liegt auf der Verfeinerung zu FPGA-plattformspezifischen komponentenorientierten Implementierungen der Anwendungsfunktionalität. Dabei verlangen einige Aspekte des Chip-Designs ein Maß an Spezifität, von dem sich nicht leicht auf eine höhere Entwurfsebene abstrahieren lässt. Dies beinhaltet z.B. das Clock-Management, aber auch die Anbindung an spezifische On-Chip-Ressourcen wie Speicher und Peripherie-Schnittstellen. In manchen Fällen mag ein komplettes Chip-Design auf Modellebene möglich sein, als der generelle Fall ist allerdings die Generierung von Teilsystemen und deren Integration in ein Chip-Design auf niedrigerer Entwurfsebene anzusehen [Xil11]. Diese Integration ist umso nahtloser möglich, je besser die Anwendungskomponenten den Struktur- und Verhaltensanforderungen einer verteilten FPGA-Realisierung gerecht werden. Dies setzt die entsprechende Definition einer plattformspezifischen Entwurfsebene zur Repräsentation von Multi-FPGA-Plattformen voraus.

4 Modellbasierte Entwurfsplattform für verteilte eingebettete Systeme auf rekonfigurierbarer Hardware

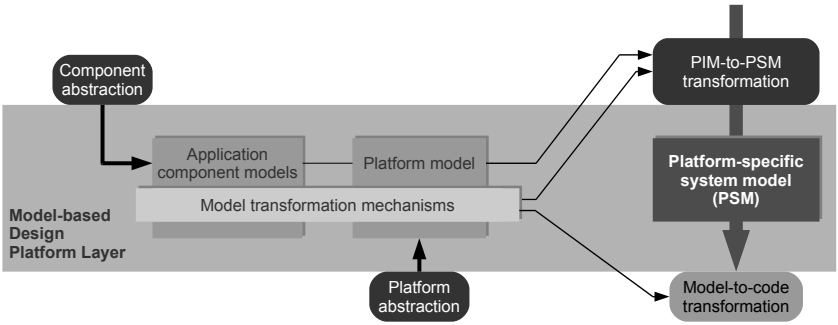


Abbildung 4.1: Entwurfsplattformebene im modellbasierten Systementwurfsprozess.

Wie in Abschnitt 2.5 erarbeitet wurde, ist die Grundlage für ein modellbasiertes Vorgehen eine vorgegebene Entwurfsplattformebene, wie sie in Abbildung 4.1 als *Model-based Design Platform Layer* dargestellt ist. Sie stellt die niedrigste Abstraktionsstufe für die Top-Down-Modellierung dar, und spezifiziert Modellrepräsentationen der Hardware-Plattform (*Platform model*) und von Komponenten ausführbarer Funktionen (*Application component models*), die mit Hilfe der Bottom-up-Abstraktionsprozesse *Platform abstraction* und *Component abstraction* bereitgestellt werden. Die Elemente der Entwurfsplattformebene bilden die Grundlage für die konsistente Erzeugung plattformspezifischer Modelle (*PSM*) der zu entwickelnden Systeme mittels Modelltransformation (*PIM-to-PSM transformation*) aus plattformunabhängigen Modellen und die Code-Generierung (*Model-to-Code transformation*). Inhalt dieses Kapitels ist die Beschreibung der modellbasierten Entwurfsplattformebene für eingebettete Systeme auf Multi-FPGA-Plattformen. Der wesentlichste Aspekt besteht in der Erarbeitung von Metamodellen zur Erfassung der Strukturkonzepte,

erstens von Multi-FPGA-Plattformen und verteilten Entwürfen, und zweitens von plattformspezifischen Systemmodellen. Die Metamodelle werden mit Hilfe von UML-Klassendiagrammen beschrieben. Das Kapitel wird geschlossen mit einer Konkretisierung des modellbasierten Entwurfs- und Testprozesses für Multi-FPGA-Realisierungen.

4.1 Plattform-Modellierung

Da die modellbasierte Methodik Systementwürfe auf Multi-FPGA-Hardware abbilden soll, ist es zunächst nötig, ein entsprechendes Plattformmodell (PM - vgl. Abschnitt 2.4) zu definieren. In diesem Abschnitt wird daher vorerst der Begriff des „System-on-multiple-Chips“ als geschlossen entworfenes bzw. modelliertes, aber verteilt implementiertes System geprägt. Dieses Konzept wird durch Studien von im Rahmen dieser Arbeit entwickelter Multi-FPGA-Hardware und Intermodulkommunikation unterstützt. Darauf aufbauend wird das Metamodell für Multi-FPGA-Systeme und -Entwürfe definiert, das die Grundlage der weiterführenden Erörterungen bildet.

4.1.1 „System-on-multiple-Chips“

Dieser Begriff stellt keineswegs den Rückschritt vom hochintegrierten System-on-Chip zum System-on-Board dar, sondern vielmehr eine Erweiterung des Begriffs des SoC auf Multi-Chip-Systeme, die eine gemeinsame Gesamtfunktionalität realisieren sollen und dementsprechend aus einem Gesamtsystementwurf abgeleitet wurden. Die verteilte Multi-Chip-Realisierung kann dabei aus Gründen der Ressourcenkapazität für komplexe Algorithmen [LSYM08, SQC02] oder große Multi-Prozessor-Anwendungen [KSH07, SLS10] einerseits, oder andererseits aufgrund der Vielzahl der zu verarbeitenden Sensor- und Informationskanäle notwendig sein [ZHCY09, LBT+11].

Abbildung 4.2 skizziert die verallgemeinerte Struktur eines auf einer Multi-FPGA-Hardware implementiertes verteiltes SoC. Jeder FPGA enthält ein Teilsystem, das sich aus Prozessorkernen (CPU), applikationspezifischen Funktionskomponenten (App. core), I/O-Controllern (I/O core) in Form von IP-Komponenten, sowie Speichern (Mem.) zusammensetzen kann. Alle diese Komponenten sind über eine On-Chip-Kommunikationsstruktur entsprechend der in Abschnitt 3.1.3 vorgestellten Taxonomie verknüpft. Die Off-Chip-Verbindung zur Intermodulkommunikation erlaubt den Datenaustausch zwischen den Teilsystemen. Kommunikationskomponenten (Com. core) erlauben die Integration dieser zweiten Kommunikationshierarchie in die lokalen SoCs.

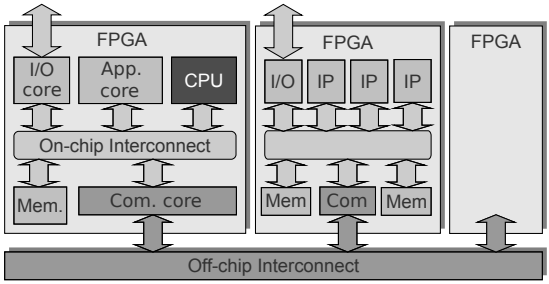


Abbildung 4.2: Verteiltes SoC - „System-on-multiple-Chips“ (SomC).

4.1.2 Konzeptuelle Studien zu Multi-FPGA-Systemen

Die folgenden Realisierungsstudien zeigen eine im Rahmen dieser Arbeit entstandene skalierbare modulare FPGA-Hardware und ein Intermodulkommunikationssystem, das ebenso skalierbar in Form von Schnittstellenkomponenten in verteilte SoCs zu integrieren ist.

Multi-FPGA-Hardware GECKO3STACK

Das Multi-FPGA-System *GECKO3STACK* mit der Interface-Basisplatine *GECKO3 staddle* ist ein Teil des GECKO3-Projektes [BM11]. Das GECKO3-Projekt wird vom Fachbereich MicroLab an der Fachhochschule Bern in der Schweiz betreut und umfasst die Entwicklung einer FPGA-basierten Hardware/Software-Codesign-Plattform, von der Basis-Plattform, über zahlreiche Schnittstellenmodule und System-on-Chip-Designs [ZZ07]. Alle Teile dieses Projektes sind entweder unter der GNU General Public License oder Creative Commons lizensiert und auf der Projekt-Homepage verfügbar [Mic11]. Kern des GECKO3-Projektes ist das *GECKO3main*-FPGA-Modul mit einem Xilinx Spartan-3-4000-FPGA, RAM und Flash-Speichern, sowie USB- und Ethernet-Schnittstellen. Eine wesentliche Anzahl der GPIO-Pins wurde auf zwei Board-zu-Board-Verbinder geroutet und erlaubt damit eine breite digitale Direktverbindung zwischen diesen Modulen und Peripheriemodulen oder untereinander. Damit ist die Möglichkeit gegeben, einen Multi-FPGA-Verbund über einen parallelen Bus als physische Kommunikationsebene zu realisieren.

Der GECKO3STACK als modulare, skalierbare Hardware-Plattform wurde im Rahmen dieser Arbeit an der TU Ilmenau [Bra10] als Experimentalaufbau für die Untersuchung von Verfahren zur Intermodulkommunikation und zum

Entwurf von verteilten SoC-Entwürfen entwickelt [MBKF11]. Im Verlauf dieser Arbeit wird diese Hardware-Plattform die Referenz für modellbasierte Entwürfe und Ziel der experimentellen Untersuchungen darstellen.

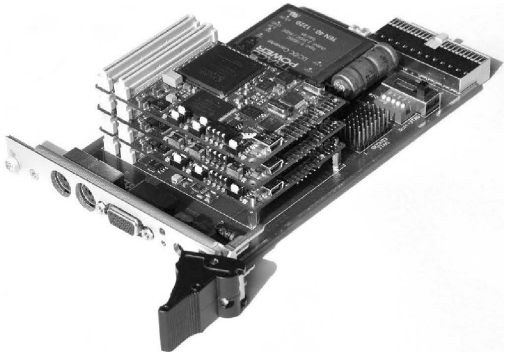


Abbildung 4.3: Foto eines GECKO3STACK-Multi-FPGA-Aufbaus mit drei Modulen.

Intermodulkommunikation via Shared-Memory

Der Ansatz der Shared-Memory-Kommunikation beruht auf der Idee, einen Speicher zu schaffen, auf den von allen Kommunikationsteilnehmern gleichberechtigt zugegriffen werden kann und in dem die auszutauschenden Daten unter global bekannten Adressen abgelegt werden. Die hier vorgestellte Umsetzung dieses Ansatzes zur Kommunikation im Multi-FPGA-System sieht anstelle eines zentralen Speichers einen auf die FPGAs verteilten Speicher, also einen *Distributed Shared-Memory*, unter Nutzung lokaler Block-RAM-Segmente der jeweiligen FPGAs vor. Diese Speicher werden, wie in Abbildung 4.4 gezeigt, unter einem globalen Adressraum zusammengefasst. Dieser erlaubt für die verteilten Funktionskomponenten transparente globale Speicherzugriffe unabhängig von deren Platzierung und unabhängig von der physischen Lage des zugegriffenen Speichersegments nach dem als Non-uniform Memory Access (NUMA) bekannten Prinzip.

Die Etablierung und die Kontrolle dieses virtuellen N-Tor-Speichers (multiport memory) wird durch die Interaktion mehrerer Instanzen einer Schnittstellenkontrollerkomponente erreicht, von denen je eine jedem FPGA zugeordnet ist. Diese wiederverwendbare Design-Komponente wird als SHIVA bezeich-

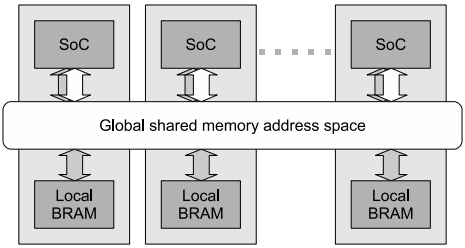


Abbildung 4.4: Global verteilter Shared-Memory auf einer Multi-FPGA-Plattform [MBKF11].

net - eine Abkürzung für *Shared-memory Inferface for Versatile Application* [Kra10].

Intern verwaltet jede SHIVA-Instanz ein lokales BRAM-Segment und steuert alle Zugriffe auf die entfernten Segmente aller anderen verknüpften FPGAs. Da NUMA keine einheitlichen Zugriffslatenzen aufweist, ist SHIVA als *asynchrone Speicherschnittstelle (Asynchronous memory interface)* ausgelegt - siehe Abbildung 4.5 - und verfügt über Synchronisationssignale, die die Belegung der lokalen Schnittstelle (`Busy`) und die Verfügbarkeit ausgelesener Daten (`Data_Valid`) anzeigen. Durch die Interaktionen der SHIVA-Cores sind vier verschiedene NUMA-Zugriffsvarianten realisierbar:

- *Local read/write* - Lese- oder Schreibzugriffen der lokalen Applikation auf das lokale Segment des Distributed-Shared-Memory;
- *Remote read/write* - Anfordern und Durchführen von Lese- oder Schreibzugriffen der lokalen Applikation auf ein entferntes Segment des Distributed-Shared-Memory - die Latenz dieser Zugriffe hängt vom Verhalten der unteren Off-Chip-Kommunikations-lösung ab;

Eine Shared-Memory-Komunikationstransaktion zwischen zwei Applikationspartitionen besteht aus zwei Teilen, einem Schreibzugriff und einem Lesezugriff auf den Speicher. Somit ist vom Empfänger des Datums abhängig, wann die eigentliche Transaktion abgeschlossen ist (comsumer-oriented communication). Der Lesezugriff kann entweder aufgrund globalen Wissen über die Datenverfügbarkeitszeitpunkte initiiert werden oder aufgrund eines Signalisierungsmechanismus. Da ein globales vollständig synchrones Aus-führungsparadigma in einem verteilten SoC nicht vorausgesetzt werden kann, realisiert SHIVA zusätzlich ein Broadcast der jeweils beschriebenen Adressen (`Remote Update Notify`), die von den jeweiligen Partitionen zur

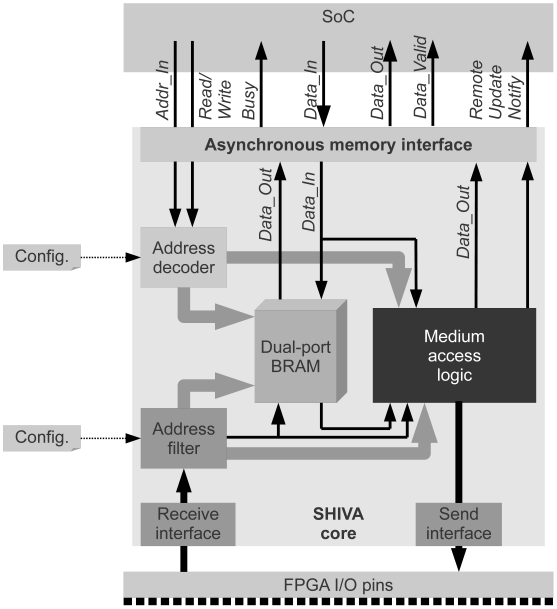


Abbildung 4.5: Übersicht über die Interna und Applikationsschnittstelle des Shared-Memory-Interface-IP-Cores SHIVA.

Rekonstruktion und Resynchronisation des Datenflusses ausgewertet werden können.

Ausgehend von den vier möglichen NUMA-Zugriffen gibt es drei verschiedene Möglichkeiten, wie sich Kommunikationstransaktionen zusammensetzen können:

- *Local write / Remote read* - schnell abgeschlossener Schreibzugriff, der die lokale Schnittstelle der schreibenden Partition schnell wieder verfügbar macht, aber langsames Lesen mit Kommunikationslatenz;
- *Remote write / Remote read* - langsamste Transaktion über ein Speichersegment, das physisch weder auf Sender- noch auf Empfängerseite liegt - ungünstiger Fall, aber möglich bei Systemen mit > 2 Knoten und unsymmetrischem Datenaufkommen.
- *Remote write / Local read* - Lesezugriff auf lokales Speichersegment des Empfängerknotens bereits unabhängig vom Verhalten und Verfügbarkeit

des Kommunikationsmediums - insgesamt schnellste mögliche Kommunikationstransaktion.

Durch entsprechende Strategien bei der Adressierung der an der Off-Chip-Kommunikation beteiligten Daten im Shared-Memory können die Leistungsparameter des Kommunikationssystems zwischen Minimierung des Speicherbedarfs und Minimierung der Transaktionslatenzen während des Systementwurfs beeinflusst werden.

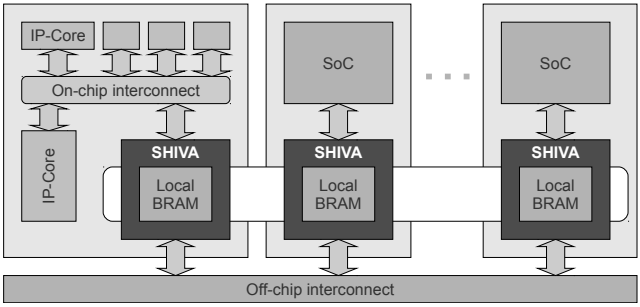


Abbildung 4.6: Durch SHIVA-Interface-Cores realisierter Shared-Memory als Intermodulkommunikationsmedium im verteilten System-on-Chip.

Wie in Abbildung 4.6 zu sehen ist, werden die SHIVA-Cores als Speicherschnittstellen in die jeweiligen lokalen Teile des verteilten System-on-multiple-Chips eingebunden und machen den globalen Adressraum als virtuellen N-Tor-Speicher zugänglich, während von den darunterliegenden Protokollen und der Hardware-Topologie abstrahiert wird. Die Integration des Speichersegments und aller Zugriffslogik macht es außerdem möglich, gut skalierbare Strukturen aus gleichen wiederverwendbaren Komponenten aufzubauen [MBKF11].

Details zur konkreten Implementierung des SHIVA-Cores für die zuvor beschriebene Multi-FPGA-Plattform GECKO3STACK sind [MBKF11, Kra10] zu entnehmen. Die Leistungsfähigkeit paralleler busbasierter Infrastrukturen zur Chip-zu-Chip-Kommunikation ist vor allem für höhere Übertragungsfrequenzen begrenzt, weshalb sie durch serielle Hochgeschwindigkeitsverfahren abgelöst werden. Eine Implementierung von SHIVA zur Abstraktion einer unidirektionalen Ring-Topologie zur Intermodulkommunikation auf der Basis von seriellen Gigabit-Tranceivern [Hei12] ist in [Jes12] beschrieben.

4.1.3 Metamodell für Multi-FPGA-Hardware und SoC-Designs

Die Verallgemeinerung und Formalisierung dieser Aspekte führt zur Definition eines Metamodells, das die strukturellen Beziehungen zwischen Elementen der FPGA- bzw. Multi-FPGA-Hardware und den Elementen der SoC-Entwürfe beschreibt. Dieses Plattform-Metamodell ist in Abbildung 4.7 als UML-Klassendiagramm dargestellt.

Auf der rechten Seite des Diagramms ist die Hierarchie der physischen FPGA-Ressourcen als Metamodell (*Hardware platform metamodel*) dargestellt, wobei Multi-FPGA-Hardware als Komposition aus mehreren FPGAs und mindestens einer Kommunikationsinfrastruktur zu betrachten ist. Auf der linken Seite zeigt das Metamodell für FPGA-Entwürfe (*FPGA design metamodel*) die Struktur eines Top-Level-Entwurfs, der sich aus über die On-Chip-Kommunikation verbundenen Komponenten zusammensetzt und ein auf einem FPGA implementierberes SoC beschreiben kann. Neben den anwendungsspezifischen Komponenten (*ApplicationComponent*), die zur Umsetzung der Funktionalität dienen, sind I/O-Controller (*IOControllerCore*) besondere Komponenten, die die Ansteuerung chip-externer Elemente (off-Chip-RAM, Geräte, ...) über dafür vorgesehene Pin-Gruppen (*IOResource*) übernehmen. Die (1:1)-Relation zwischen I/O-Ressourcen und den I/O-Controller-Komponenten macht letztere zu exklusiven Ressourcen im Chip-Design und auch für die plattformspezifische Modellierung. Zu beachten ist, dass die Schnittstellenkomponente zur Intermodulkommunikation (*OffChipComCore*) als spezieller I/O-Controller angesehen wird, der Zugriff zu genau einer Off-Chip-Kommunikationsinfrastruktur gewährt. Die grauen Assoziationen zwischen Elementen des Design-Metamodells links und des Hardware-Metamodells rechts beschreiben die Zuordnungen, die während der Logiksynthese und FPGA-Implementierung realisiert werden.

Wesentliche Charakteristik der Implementierung für verteilte FPGA-Hardware ist die Tatsache, dass jedem Ziel-FPGA genau eine Top-Level-Beschreibung des lokalen Chip-Designs zugeordnet wird (*targetChip*). Die Struktur, die den Multi-FPGA-Charakteristika Rechnung trägt und damit die Zuordung des Gesamtsystems zur Ziel-Hardware (*targetHardware*) realisieren kann, ist das plattformspezifische Systemmodell (*PlatformSpecificModel*). Auf dieser Ebene kann das Gesamtsystemverhalten definiert werden, bevor schlussendlich die Top-Level-Designs der interagierenden Teilsysteme (*distributedDesigns*) abgeleitet werden können.

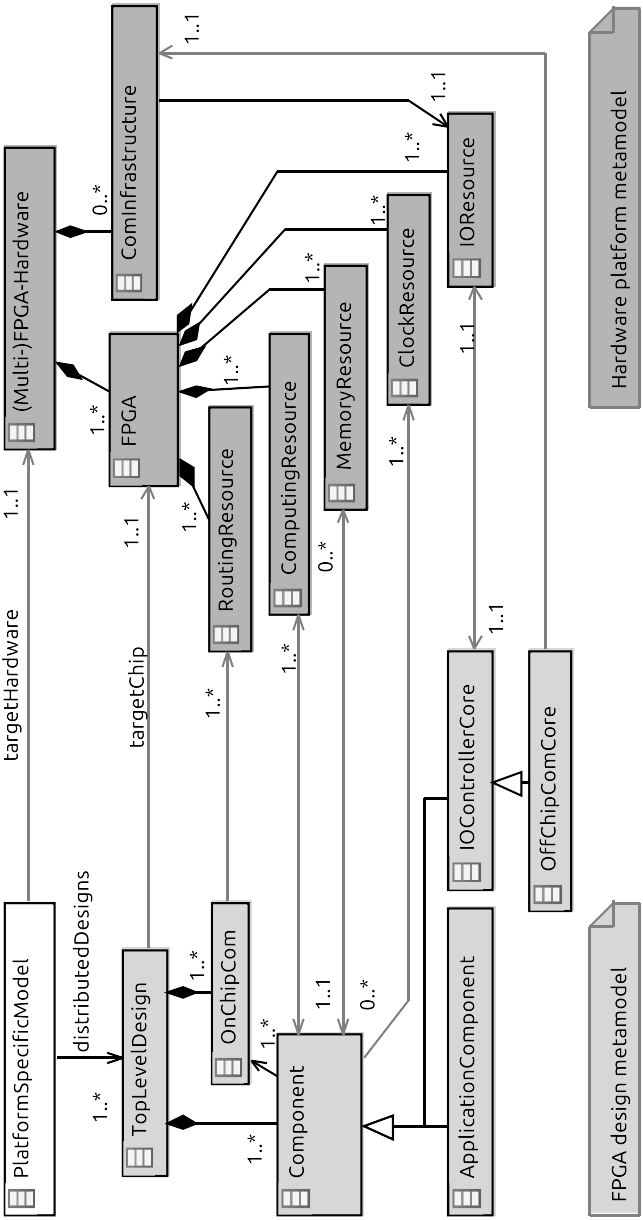


Abbildung 4.7: UML-Diagramm des Metamodells von (Multi-)FPGA-Plattformen mit Ressourcen und Design-Elementen.

4.2 Plattformspezifische Modellierung

Die während der ersten Phasen des Systementwurfsprozesses spezifizierten funktionalen Anforderungen werden im Laufe des modellbasierten Top-Down-Verfeinerungsprozesses auf Komponenten des plattformspezifischen Modells abgebildet. Die Aufgabe der modellbasierten Entwurfsplattformebene ist es, dafür Modellelemente sowie die mit ihnen assoziierten Artefakte und Mechanismen für eine automatisierte Modell-zu-Code-Transformation bereitzustellen. Dabei kommen zwei unterscheidbare Methoden zum Einsatz - die *Instanziierung* von wiederverwendbaren Implementierungskomponenten (IP-Cores) und die *Generierung* von Hardware-Beschreibungen auf der Basis der für Modellelemente definierten Code-Fragmente und Kompositionsregeln.

Wie in Abbildung 4.8 verdeutlicht wird, kann es Unterschiede im Grad der Abstraktion der PSM-Komponenten geben. Je nach Umfang und Ausdruckmächtigkeit der modellbasierten Entwurfsplattform ist es möglich, für gewisse Funktionen bereits auf Modelle komplexer Hardware-Komponenten zurückzugreifen, während andere Anwendungskomponenten hierarchisch verfeinert werden müssen, bevor diese Zuordnung gegeben ist.

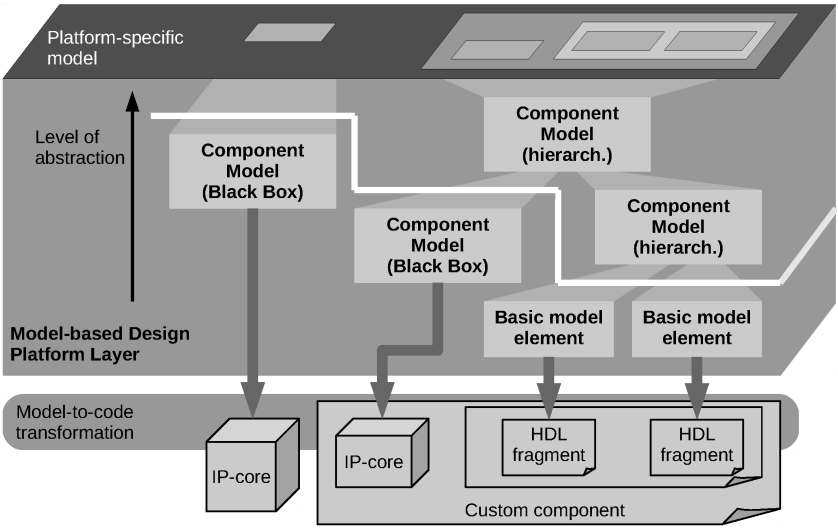


Abbildung 4.8: Plattformspezifische Modellelemente unterschiedlicher Abstraktionsgrade und Granularität.

4.2.1 Modellrepräsentation wiederverwendbarer Implementierungskomponenten

Wiederverwendbare IP-Cores als Funktionskomponenten ebenso wie als Teil der Ausführungsplattform werden auf der Ebene der plattformspezifischen Modellierung als sogenannte *Black-Box*-Modellblöcke repräsentiert. Diese sind in Abbildung 4.8 als *Component model (Black Box)* dargestellt. Entscheidend für die korrekte Integration ist die exakte Abbildung der Komponentenschnittstelle in Struktur und Verhalten. Darauf aufbauend gibt es prinzipiell zwei Möglichkeiten, die vorgefertigte Komponenten in die Gesamtsystemmodellierung und -validierung zu integrieren.

Abstrakte Simulationsmodelle: Eine Möglichkeit besteht darin, Modellelemente zu schaffen, die die Schnittstelle der wiederverwendbaren Komponente exakt abbilden, die Interna allerdings durch sehr abstrakte (plattformunabhängige) Funktionsbeschreibungen realisieren. Diese Abstraktion hat hauptsächlich die Verringerung des Simulationsaufwandes zum Ziel, erlaubt jedoch nur eine *transaktionsorientierte Simulation* der Funktionsausführung. Das Zeitverhalten wird dabei auf Durchschnitts- oder Worst-Case-Annahmen reduziert, oder zugunsten der Simulationszeiten nahezu vollständig abstrahiert. Eine funktional asynchrone Komponenteninteraktion erleichtert die plattformspezifische Modellierung mit solchen Abstraktionen. Während der Modell-zu-Code-Transformation werden die Black-Boxes durch die eigentlichen IP-Komponenten ersetzt.

Cosimulation: Eine andere Möglichkeit beinhaltet die Kopplung zweier Simulationsumgebungen. Dabei wird auf Modellebene ein Block geschaffen, der die Interface-Struktur der wiederverwendbaren Komponente abbildet, aber eine Verbindung zu einem HDL- oder Netzlisten-Simulator erstellt, der dann die *zyklusgenaue Simulation* der tatsächlichen Implementierungskomponente durchführt. Damit kann ein akkurates Timing-Verhalten der Funktionsausführung simuliert werden. Der Zeitaufwand für die zyklusgenaue Simulation ist natürlich entsprechend hoch, allerdings erlaubt diese Art der Modellierung und Validierung unter Umständen den Entwurf komplett synchroner Systeme. Eine andere Variante der Cosimulation ist die Hardware-Cosimulation oder *Hardware-in-the-Loop-Simulation* (HIL), bei der fertige Teile des Designs bereits auf dem Chip ausgeführt werden. Während oder nach der Modell-zu-Code-Transformation müssen anstelle der Cosimulationsblöcke die jeweiligen Berechnungs- oder Schnittstellenkomponenten ins Design eingefügt werden.

4.2.2 Struktur applikationsspezifischer Komponenten

Für Funktionalität, die sich nicht grobgranular mittels Black-Box-Modellkomponenten beschreiben lässt, stellt die modellbasierte Entwurfsplattformebene Elementarblöcke (*Basic model element*) zur Verfügung, die sich in verhaltensgleiche HDL-Konstrukte (*HDL fragment*) überführen lassen. Dieser Zusammenhang ist in Abbildung 4.8 dargestellt. Diese Elemente besitzen einen relativ niedrigen Abstraktionsgrad, erlauben aber eine Modellierung nahe an der RTL-Beschreibung, die für die Definition feingranularer applikationspezifischer Strukturen von Nöten sein kann. Komplexe Funktionalitäten werden dann als hierarchische Komponenten (*Component model (hiearch.)*) modelliert.

Die Struktur der Komponenten des plattformspezifischen Modells wird durch die Forderungen nach Wiederverwendbarkeit und robuster Integration auf Implementierungsebene, sowie nach korrekter Komponenteninteraktion sowohl auf Modell- als auch auf Implementierungsebene bestimmt [LAD04, LAD06]. Wesentlich dafür ist eine Schnittstellendefinition basierend auf den Richtlinien des komponentenbasierten Logikdesigns, wie sie in Abschnitt 3.3 zusammengetragen wurden. Die schematische Skizze in Abbildung 4.9 stellt alle Elemente dar, die bei der Definition einer plattformspezifischen Modellkomponente eine eindeutige Repräsentation benötigen.

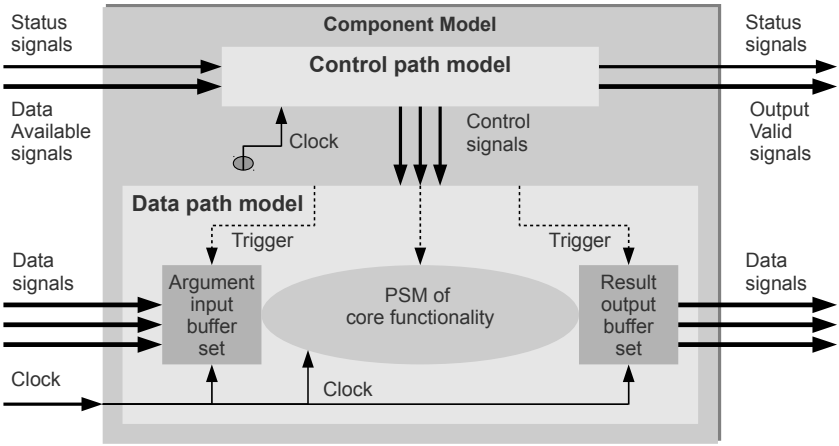


Abbildung 4.9: Plattformspezifisches Modell einer Funktionskomponente.

Die Modellkomponente setzt sich aus Datenpfad (*Data path model*) und Steuerpfad (*Control path model*) zusammen. Wesentlichste Elemente des Datenpfades sind neben dem plattformspezifischen Modell der Kernfunktionalität (*PSM of core functionality*) die Pufferstrukturen für die Argumente und Resultate der Komponente (*Argument input / Result output buffer sets*). Der Steuerpfad verarbeitet zwei Gruppen von Signalen - Statussignale von benachbarten Komponenten oder übergeordneten Steuerpfadstrukturen (*Status signals*) und Datengültigkeitssignale (*Data available signals*). Mithilfe dieser Signale wird die Interaktion der Komponente in der on-Chip-Infrastruktur sichergestellt und das interne Verhalten gesteuert. Auf Modellebene ist dazu von Bedeutung, dass die Argumente zu einem über das *Trigger*-Signal definierten Zeitpunkt in die Eingangspuffer übernommen werden, und dass nach dem Durchlaufen der inneren Logik die Resultate mit einem entsprechend des internen Zeitverhaltens verzögerten Trigger in die Ausgangsregister übernommen werden. Der Interaktionszustand der Komponente und die Validität der Resultat werden wiederum durch Statussignale und Validitätssignale (*Output valid signals*) propagiert.

4.2.3 Metamodell für multi-FPGA-orientierte Modellierung

Aus dem in Abschnitt 4.1.3 erarbeiteten Multi-FPGA-Metamodell und den zuvor diskutierten Zusammenhängen zur Komponentenmodellierung wird nun das Metamodell für die komponentenorientierte multi-FPGA-plattformspezifische Modellierung abgeleitet und als UML-Klassendiagramm in Abbildung 4.10 illustriert.

Das PSM für Multi-FPGA-Entwürfe ist grundsätzlich in Teilmodelle (*PartitionModel*) partitioniert, die ihrerseits, wie durch die Vererbungsbeziehung verdeutlicht wird, die plattformspezifischen Modelle (*PlatformSpecificModel*) der auf den jeweiligen FPGAs zu implementierenden Logik-Designs sind. Diese Partitionsmodelle bestehen aus Modellen interagierender Komponenten (*ComponentModel*) - aus applikationsspezifischen Funktionskomponenten (*FunctionComponentModel*) einerseits, und andererseits aus Plattformkomponenten (*PlatformComponentModel*). Die grauen Assoziationen zwischen den Elementen des Metamodells der plattformspezifischen Modellierung (*PSM metamodel*) und dem der FPGA-Entwurfsebene (*FPGA design metamodel*) beschreiben die Zuordnungen, die währen der Modell-zu-Code-Transformation realisiert werden. Die Plattformkomponentenmodelle stellen abstrakte Repräsentationen von Plattformressourcen und ihren Schnittstellen auf

der Modellebene dar[1]. Die Black-Box-Modelle von I/O-Controllern und Intermodulkommunikationskomponenten (*IOCtrlBlackBox, OffChipComBlackBox*) sind Beispiele solcher Modellelemente - sie sind mit abstrakten Simulationsmodellen des I/O- bzw. Kommunikationsverhaltens verknüpft. Wie durch die Assoziationen zu den Elementen des FPGA-Design-Metamodells (*IOControllerCore, OffChipComCore*) verdeutlicht wird, werden bei der Modell-zu-Code-Transformation bestehende Implementierungskomponenten instanziiert (*instantiatedCore*).

Die spezifizierte Applikationsfunktionalität wird in Form der Funktionskomponenten abgebildet. Entsprechend der Entwurfsentscheidungen während der Verfeinerung des PSM werden Funktionskomponenten unterschiedlich modelliert und auch bei der Zuordnung zu den jeweiligen Implementierungskomponenten (*ApplicationComponent*) unterschiedlich gehandhabt. Black-Box-Repräsentationen bestehender wiederverwendbarer Logikkomponenten (*FunctionalBlackBox*) instantiieren bestehende IP-Cores, über die Assoziation *instatiatedCore* ausgedrückt wird. Andere Funktionsmodelle werden bis auf die niedrigste nötige Abstraktionsebene verfeinert, so dass eine neue applikationspezifische IP-Komponente generiert und synthetisiert werden kann (*CustomComponent*), was durch die Assoziation *generatedCore* verdeutlicht wird. Diese top-down entwickelten Funktionskomponenten entsprechen hierarchischen Kompositionen, deren Kernlogik (*coreLogic*) wiederum aus Funktionskomponenten besteht.

Die (1:1)-Abbildungen zwischen den Metamodellelementen der modellbasierten Entwurfsplattformebene und den entsprechenden implementierenden Logikkomponenten erlaubt in Kombination mit den Code-Generierungsmechanismen moderner Modellierungswerkzeuge eine konsistente Modell-zu-Code-Transformation.

[1] An dieser Stelle ist das Metamodell nur unvollständig, da ausschließlich die im Rahmen dieser Arbeit wesentlichen Plattformabstraktionen hinsichtlich der Intermodulkommunikation betrachtet werden.

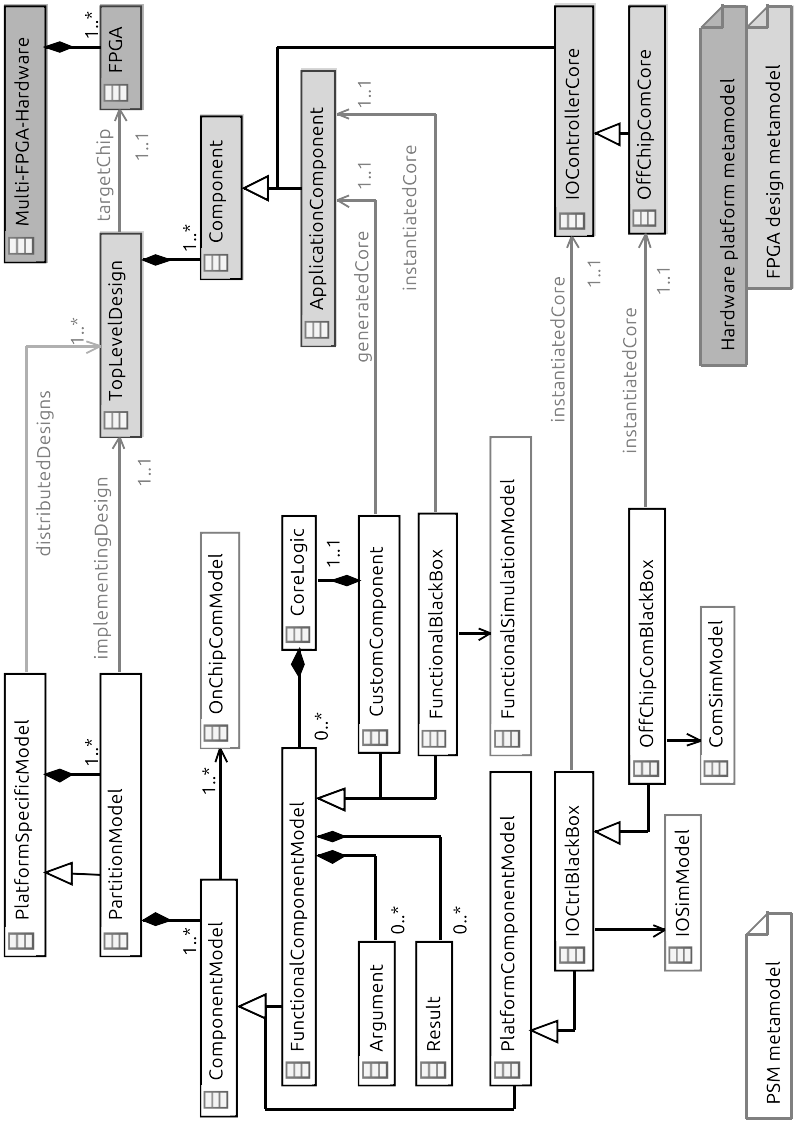


Abbildung 4.10: Metamodel der plattformspezifischen Modellierungsebene (UML).

4.3 Modellbasierter Entwurfs- und Validierungsprozess für Multi-FPGA-Systeme

Der Entwurfs- und Validierungsprozess, der sich für den Multi-FPGA-Systementwurf unter der definierten modellbasierten Entwurfsplattformebene ergibt, ist in Abbildung 4.11 illustriert.

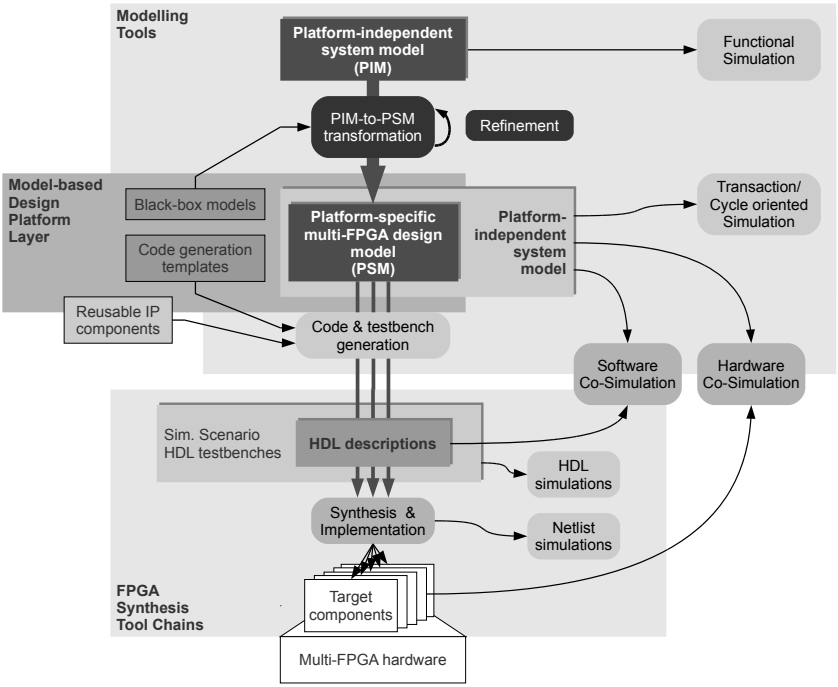


Abbildung 4.11: Modell-zu-Modell- und Modell-zu-Code-Transformationen im Multi-FPGA-Entwurfs- und Validierungsprozess.

Ausgehend von einer als plattformunabhängiges Modell (*PIM*) entworfenen und durch funktionale Simulation (*Functional simulation*) validierten Systemfunktionalität wird durch Modell-zu-Modell-Transformation (*PIM-to-PSM transformation*) das plattform-spezifische Modell (*PSM*) erstellt und schrittweise verfeinert (*Refinement*). Dabei werden Entwurfsentscheidungen und nicht-funktionale Randbedingungen, die im

Gesamtprozess in Abschnitt 2.5 enthalten sind, einbezogen. Die modellbasierte Entwurfsplattformebene (*Model-based design platform layer*) sorgt dafür, dass Modellelemente, Transformationsmechanismen und die Assoziation zu wiederverwendbaren IP-Cores für die Modell-zu-Code-Tranformation zur Verfügung stehen.

Im geschlossenen modellbasierten Entwurf ist es möglich, das plattformspezifische Teil-modell in einem abstrakten Umgebungsmodell (*Platform-independent system model*) zu simulieren und so das Gesamtsystemverhalten vor allem während der Verfeinerungsschritte ständig zu validieren. Aufgrund der möglichen Heterogenität der Abstraktionsgrade des plattformspezifischen Modells, besonders, wenn abstrakte Black-Box-Blöcke eingebunden sind, ist an dieser Stelle eine transaktionsorientierte Simulation (*Transaction-oriented simulation*) vorzunehmen, die vor allem das Schnittstellenverhalten und die Komponenteninteraktion validiert. Natürlich ist auf dieser Ebene bereits eine Verfeinerung bis auf zyklusgenaues Verhalten möglich - die Komplexität der Simulation kann dann aber eine Gesamtsystemvalidierung einschränken. Aus einem plattformspezifischen Modell können dann mittels Codegenerierung die Hardware-Beschreibungen (*HDL descriptions*) für die Teilentwürfe erzeugt werden. Aus dem geschlossenen Gesamtsystemmodell können außerdem die entsprechenden Testszenarien gewonnen werden (*Code & testbench generation*). Mit diesen können dann auf RTL-Ebene taktgenaue Simulationen durchgeführt werden (*HDL simulation*). Ausgehend von der HDL-Beschreibung schließt sich für jede Partition der FPGA-spezifische Synthese- und Implementierungsprozess mit den Simulationen auf Basis der erstellen Netzlisten (*Netlist simulations*) an, der in Abschnitt 3.2.3 beschrieben wurde.

Die Verfügbarkeit des Umgebungsmodells als spezifizierte Testumgebung für das integrierte plattformspezifische Modell des zu entwickelnden Systems erlaubt außerdem die Anwendung zweier weiterer Validierungstechniken. Die Software-Cosimulation oder Software-in-the-Loop (SIL) erlaubt die Kopplung der HDL-Simulation des DUT mit der abstrakten Simulation des Gesamtsystems, während die Hardware-Cosimulation oder Hardware-in-the-Loop (HIL) das vollständig auf der FPGA-Hardware integrierte DUT in die Gesamtsystemsimulation einbindet. Speziell zur Validierung der Integration von Black-Box-Komponenten in das PSM sind diese Simulationsmethoden von Vorteil.

4.4 Fazit

Dieses Kapitel widmete sich der Definition einer Modellierungsplattformebene für den Entwurf eingebetteter Systeme, die auf Multi-FPGA-Plattformen zu implementieren sind. Dazu wurde das Konzept „System-on-multiple-Chips“ in Erweiterung des SoC-Begriffs geprägt, um das Anliegens einer modellbasierten Gesamtentwicklung von verteilten Teilimplementierungen zu verdeutlichen. Als konzeptuelles Beispiel für eine Hardware-Plattform zur Entwicklung und Untersuchung solcher Systeme wurde das im Rahmen dieser Arbeit entwickelte modulare Multi-FPGA-System GECKO3STACK vorgestellt. Als wesentlicher Punkt für die Definition einer Entwurfsplattformebene, die den geschlossenen Entwurf verteilter Systeme erlaubt, wurde die Abstraktion der Intermodulkommunikation charakterisiert. Unabhängig davon, ob on-Chip- oder off-Chip-Kommunikation, der Datenaustausch über speicherabgebildete (memory-mapped) Schnittstellen ist der Stand der Technik. Als konzeptuelles Beispiel einer off-Chip-Kommunikationsinfrastruktur wurde im Rahmen dieser Arbeit das Shared-memory-System SHIVA entwickelt, das jegliche Hardware-Topologie durch einen global transparenten N-Tor-Speicher mit Update-Benachrichtigung abstrahiert.

Aus diesen Betrachtungen wurde ein Metamodell für die Struktur von Multi-FPGA-Plattformen und verteilten komponentenbasierten Logik-Designs abgeleitet. Aufbauend darauf erfolgte die Definition der plattformspezifischen Modellierungsebene, in deren Zentrum die Definition komponentenorientierter plattformspezifischer Systemmodelle steht. Diese Komponentenstruktur definiert die Beschaffenheit und das Verhalten der Schnittstellen, sodass ein konsistentes Systemverhalten auf Modell- wie auch auf Implementierungsebene erreicht werden kann. Das Metamodell für die Modellierungsplattformebene ist ein wesentliches Ergebnis dieses Kapitels. Es beschreibt die Beziehungen zwischen den Teilmodellen eines multi-FPGA-orientierten Gesamtentwurfs, ebenso wie die zwischen den Modellierungs- und den entsprechenden Implementierungskomponenten, die top-down im Zuge des Entwurfs mitentwickelt oder bottom-up als wiederverwendbare Black-Box-Komponenten eingebunden werden.

Das Kapitel wurde geschlossen mit der verfeinerten Darstellung des Entwurfs- und Validierungsprozesses ausgehend von der Modellierungsplattformebene, von der aus ein plattformspezifisches Modell des zu entwickelnden Systems in implementierbare Hardware-Beschreibungen überführt werden kann. Die automatisierte Überführbarkeit eines gut entworfenen plattformspezifischen Komponentenmodells in synthetisierbare Logikbeschreibung ist ein wesentlicher Beitrag zur Reduktion der Fehleranfälligkeit und des Entwurfs- und Im-

plementierungsaufwands durch einen modellbasierten Systementwurfsprozess. Trotz aller Bestrebungen der Automatisierung der unteren Implementierungsschritte, einige Aspekte des FPGA-Entwurfs lassen sich nur bedingt oder unter bestimmten Randbedingungen abstrahieren, sodass die plattformspezifische Modellierung immer noch einen Paradigmenwechsel voraussetzt - von der Modellierung von Funktionen zur Modellierung von Daten- und Steuerfluss durch interagierende on-Chip-Komponenten. Das folgende Kapitel widmet sich daher Konzepten und Methoden, die den Übergang von plattformunabhängigen Funktionsmodell zu einem FPGA-plattformspezifischen Modell unterstützen.

5 Ausgewählte Methoden zur Synthese applikationsspezifischer Modellstrukturen

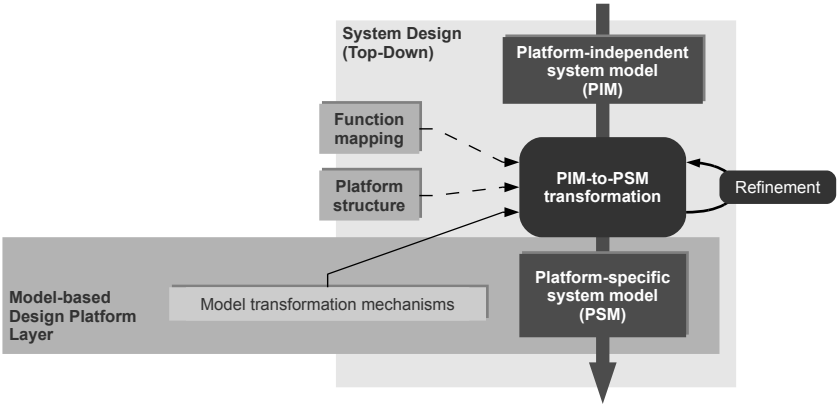


Abbildung 5.1: Vom Funktionsmodell zum Implementierungsstrukturmodell.

Die Definition der Mechanismen (*Model transformation mechanisms*) zur Top-Down-Überführung einer plattformunabhängigen Funktionsbeschreibung (PIM) in ein (multi-)FPGA-orientiertes Systemmodell (PSM) ist der modellbasierten Entwurfsplattform zugeordnet. Die Anwendungen dieser Mechanismen werden, wie Abbildung 5.1 als Ausschnitt des Gesamtprozesses in Abbildung 2.8 hervorgehoben ist, unter dem Begriff *PIM-to-PSM transformation* zusammengefasst. Der Fokus dieses Kapitel liegt nun auf ausgewählten Aspekten dieser Model-zu-Model-Transformation. Erstens ist dies die Definition einer Interaktionssemantik für komponentenorientierte Systementwürfe. Zweitens wird eine Methode zur Schaffung von Komponenten zur Ausführung multipler Funktionen auf einer Berechnungsressource („Ressource-Sharing“) betrachtet, um Systementwurfsentscheidungen bezüglich der Minimierung des On-Chip-Ressourcenbedarfs zu unterstützen. Drittens wird eine Methode zur Generierung applikationsspezifischer Zugriffslogik für die Intermodulkommunikation erarbeitet, die die Synthese von Multi-FPGA-Entwurfsmodellen

erleichtert. Das Kapitel wird geschlossen mit der Einordnung dieser Verfahren in den Modell-zu-Modell-Transformationsprozess, dessen Entwurfsiterationen (*Refinement*) der Schaffung und Bewertung verschiedener plattformspezifischer Lösungen und damit der Design-Space-Exploration auf Modellebene dienen.

5.1 Datenflussorientierte komponentenbasierte Systeme

Für die Anwendungsdomäne der Signalverarbeitung in der Mess- und Automatisierungstechnik spielen datenflussgraphbasierte Modelle als berechnungs- und plattformunabhängige Beschreibungsmittel eine wichtige Rolle und wurden daher auch vielen kommerziellen Tools wie LabVIEW und Simulink zugrundegelegt [GAGS09]. Im folgenden wird daher mit dem Datenflussgraph ein funktionales Beschreibungsmittel eingeführt und ein zugehöriges Paradigma für die Komponenteninteraktion abgeleitet. Diese können sowohl der Transformation von der plattformunabhängigen Beschreibung in das zunehmend verfeinerte plattformspezifische Modell zugrunde gelegt werden, als auch der Ausführung auf der Hardware im Rahmen des GALS-Paradigma. Im Zuge dessen wird die Beziehung der Datenflussdefinition und der Komponentendefinition für die Abbildung von plattformunabhängigen Spezifikationsmodellen auf komponentenorientierte FPGA-plattformspezifische Systemmodelle erarbeitet.

5.1.1 Der Datenflussgraph

Der Datenflussgraph (DFG) ist ein graphisches Beschreibungsmittel, das die Beziehungen von Operationen und Daten in einem Algorithmus über die expliziten Datenabhängigkeiten definiert. Ein DFG kann als bipartiter Graph dargestellt werden, bei dem Operationen und Daten zwei disjunkte Knotenmengen darstellen, während die Kanten die Abhängigkeiten von Operationen und Daten, bzw. umgekehrt, festlegen (siehe Abbildung 5.2).

Wenn zugelassen wird, dass Operationen nicht nur durch elementare, sondern auch durch mehr oder minder komplexe Funktionen beschrieben werden können, die ihrerseits DFG sein können, wird der DFG zu einem hierarchischen Gebilde. Abbildung 5.3 zeigt das Metamodell hierarchischer Datenflussgraphen als UML-Klassendiagramm.

Der DFG ist hier definiert als

$$DFG = (F, D, E)$$

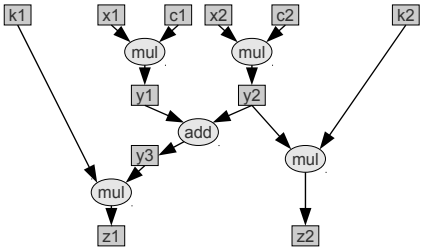


Abbildung 5.2: Beispiel für einen einfachen Datenflussgraph.

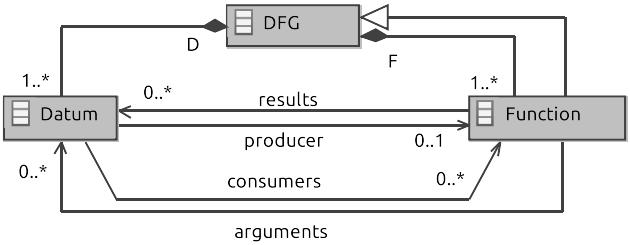


Abbildung 5.3: Metamodell eines hierarchischen Datenflussgraphen (UML).

mit F : Menge der Funktionen (functions), D: Menge der Daten (data). Funktionen produzieren Daten als Resultate und konsumieren Daten als Argumente. E beschreibt die Menge der Kanten (edges) zwischen Funktionen und Daten mit

$$E \subseteq (F \times D) \cup (D \times F), E \neq \emptyset$$

Eine Kante beschreibt, dass ein Datum Argument einer Funktion ist, wenn gilt:

$$d \in arguments(f) \Leftrightarrow (d, f) \in (D \times F) \in E, d \in D, f \in F$$

Entsprechend ist ein Datum Resultat einer Funktion, wenn gilt:

$$d \in results(f) \Leftrightarrow (d, f) \in (F \times D) \in E, d \in D, f \in F$$

Zusätzlich gilt, wie im Metamodell verdeutlicht, dass jede Funktion mehrere verschiedene Daten als Argumente konsumieren, und auch jedes Datum Argument verschiedener Funktionen sein kann. Allerdings gilt weiterhin, dass jede Funktion mehrere Resultate produzieren, aber jedes Resultat nur durch *genau eine* produzierende Funktion erzeugt werden darf.

5.1.2 Das Datenflussparadigma als Berechnungsmodell

Der Datenfluss ist eine Möglichkeit, Anwendungen über die Art und Weise zu beschreiben, in der Daten zwischen Knoten fließen und durch die Knoten atomar transformiert werden. Der Systementwurfsprozess erfordert eine zunehmende Konkretisierung von Eigenschaften, für die die rein funktionalen Beschreibungen eines unbeschränkten DFG unzureichend sind [Mar11]. Eine Ausführungssemantik, auch als Ausführungsparadigma (execution paradigm) oder Berechnungsmodell (model of computation) bzw. für grobgranulare Strukturen Interaktionsmodell (model of interaction) bezeichnet, definierte eine Ordnung zwischen Funktionselementen bezüglich Kausalität und/oder Zeit entsprechend ihrer spezifizierten Anhängigkeiten [GAGS09].

Datenflussorientierte Ausführungsmodelle definieren die Knoten eines Datenflussgraphen als *Akteure*, die ihre Funktionsausführung starten (*feuern*), sobald die benötigten Daten (*Token*) and den Kanten (*Kanäle*) verfügbar sind. Die Ausführung geschieht atomar, konsumiert die Token an den Eingangskanälen und produziert Token in den Ausgangskanälen. Eine komplette Ausführung des gesamten DFG wird als *Iteration* bezeichnet.

Als allgemeinstes datenflusorientiertes Berechnungsmodell wird das Kahn-Process-Network (KPN) genannt [RS11]. Allerdings wird hier eine Kommunikation zwischen den Akteuren über asynchrone FIFOs unbegrenzter Kapazität definiert, was der Grund dafür ist, dass die praktische und effiziente Realisierung von KPNs als schwierig angesehen wird [GAGS09]. Das Modell des synchronen Datenflusses (Synchronous Data Flow (SDF)) erlegt weitere Beschränkungen auf. In einem SDF-Graph sind die Anzahl der Token, die von einem Akteur pro Ausführung konsumiert und produziert wird, festgelegt und konstant. Beim Start der Ausführung eines Akteurs werden die Token aller Eingangskanäle *synchron* konsumiert. Die Konsistenz eines SDF-Graphen kann durch ein Balancegleichungssystem über die Produktions- und Konsumptionsraten aller Akteure festgestellt werden. Diese Eigenschaften erlauben die Konstruktion statischer Schedules und die Ausführung mit beschränktem Speicherbedarf.

Eine zusätzliche Beschränkung der Raten auf 1 definiert den sogenannten homogenen synchronen Datenfluss - Homogeneous Synchronous Data Flow (HSDF) [GAGS09, RS11, Mar11]. Durch seine Eigenschaften eignet sich SDF für die Beschreibung von digitalen Signalverarbeitungsanwendungen mit konstanten Datenraten und den Entwurf eingebetteter Systeme [RS11].

5.1.3 Komponentenmodelle mit Datenfluss-Semantik

Ausgehend von einer plattformunabhängigen Datenflussbeschreibung erfolgt während der Transformation zum plattformspezifischen Systemmodell die Zuordnung bzw. Verfeinerung der Funktionsblöcke zu Modellkomponenten des PSM. Diese Modellkomponenten werden später während der Modell-zu-Code-Transformation auf lokal synchron arbeitende, aber global asynchron kommunizierende Logikkomponenten abgebildet (vgl. GALS-Paradigma in Abschnitt 3.3). Das PSM soll die Brücke zwischen dem Datenflussgraph als Spezifikationsmodell und dem komponentenbasierten Logikentwurf schlagen. Das SDF- bzw. HSDF-Paradigma als Interaktionsmodell soll daher zur Definition der Interaktionen zwischen den Modellkomponenten des PSM herangezogen werden. Dazu müssen Komponentenschnittstellen erarbeitet werden, die in der Lage sind, synchrones Datenflussverhalten auf Modellebene wie auch auf Implementierungsebene zu realisieren. Im Verlauf dieses Kapitels sollen nun die Zuordnungen von Elementen des DFG zu den Elementen des plattformspezifischen Komponentenmodells definiert werden. Diese Zuordnung wird in den folgenden aussagenlogischen Ausdrücken durch den Operator „$\rightarrow$“ beschrieben.

Die innere Struktur der PSM-Komponenten wurde in Abschnitt 4.2.2 erarbeitet. Demnach kann eine Funktionskomponente des PSM definiert werden durch:

$$c_{PSM} = (CoreLogic, ArgumentSet, ResultSet)$$

Sie ist genau dann die Beschreibung der ausführenden Komponente zu einem Funktionsknoten des spezifizierenden DFG, wenn ihre Kernlogik *CoreLogic* die spezifizierte Funktion $f \in F$ berechnet. Desweiteren sind die Argument- und Resultatmengen (*ArgumentSet, ResultSet*) als die Zusammenfassung der Eingangs- bzw. Ausgangspuffer der Komponentenschnittstelle zu betrachten. Diesen Puffern werden die Daten übergeben, die der spezifizierten Funktion im DFG als jeweilige Argumente bzw. Resultate zugeordnet sind (vgl. Abbildung 5.3).

$$ArgumentSet(c_{PSM}(f)) = \{A : \exists! d \in D : A \rightarrow d, d \in arguments(f)\}$$

$$ResultSet(c_{PSM}(f)) = \{R : \exists! d \in D : R \rightarrow d, d \in results(f)\}$$

Darauf aufbauend zeigt Abbildung 5.4 das Metamodell des komponentenorientierten plattformspezifischen Modells (*PSM metamodel*) und die Abbildungsbeziehungen zu den Elementen des plattformunabhängigen Datenflussgraphen-Metamodells (*PIM metamodel*) als UML-Klassendiagramm. Die applikationsspezifischen Komponenten im

plattformspezifischen Modell (*FunctionalComponent*) stellen die Ressourcen dar, die die entprechenden im DFG spezifizierten Funktionen in ihrer Kernlogik (*CoreLogic*) umsetzen. Die Eingangs- und Ausgangsschnittstellen werden mit Hilfe der Argumentmenge (*ArgumentSet*) und Resultatmenge (*ResultSet*) beschrieben, die Argumente und Resultate jeweils mit den entsprechenden Daten im DFG in Beziehung setzen.

Wesentlich bei der Abbildung vom DFG auf ein Komponentenmodell ist die Tatsache, dass sie bezüglich Ressourcenallokation und -bindung eine triviale (1:1)-Beziehung darstellt - jeder Funktionsblock wird zu genau einer Modellkomponente verfeinert, und damit auch, wie in Abbildung 4.10 gezeigt, von genau einer Logikkomponente im Chip-Design repräsentiert. Damit wären die Beziehungen im Datenpfad der resultierenden komponentenbasierten Logik beschrieben. Der Steuerpfad des FPGA-plattformspezifisch modellierten Systems muss so beschaffen sein, dass er die Ausführung und Ergebnispropagation im PSM, wie auch später auf dem FPGA, gemäß des Datenflussparadigmas umsetzt. Dabei interagieren die Steuerpfade innerhalb der Komponenten mit den Strukturen auf Systemebene zwischen den Komponenten.

Aus Sicht der Logikkomponente ist das Starten der Abarbeitung gleichgesetzt mit einem Signal, das die Eingangsdatenregister dazu veranlasst, die anliegenden Werte zu übernehmen. Bezüglich dieses Verhaltens sollen hier zwei Klassen von Komponenten unterschieden werden.

Die erste Klasse, im Metamodell als *SynchTriggeredComponent* bezeichnet, orientiert sich an den am häufigsten anzutreffenden Interfaces wiederverwendbarer, bottom-up zu integrierender IP-Cores. Die Ausführungsvorbedingung (*executionCondition*) ist an die spezifizierte Funktion geknüpft und signalisiert die Verfügbarkeit aller Argumente. Es existiert daher genau ein Signal, das die synchrone Übernahme der Argumente und damit den Start der Ausführung der Komponente veranlassen kann. Diese Konfiguration ist bei getakteten IP-Cores am häufigsten anzutreffen.

Abbildung 5.5 zeigt die Struktur des synchron getriggerten Komponenten-Interfaces und zwei Ansätze, HSDF-orientertes Verhalten im Gesamtsystem zu realisieren. Das zeitgesteuerte (time-triggered) Verfahren (siehe Abbildung 5.5, links) beruht auf einer zentralen Steuerinstanz (*Trigger Controller*). Aufgrund globalen Wissens über die interagierenden Komponenten kann dieser Controller zur Design-Zeit mit einen statischen Schedule des Datenflussgraphen konfiguriert werden. Damit kann die Aktivierung der Komponenten über entsprechende Signale zum Zeitpunkt der Eingangsdatenverfügbarkeit zentral gesteuert werden. Das ereignisgesteuerte (event-driven) Paradigma (siehe Abbildung 5.5, rechts) beruht dagegen auf einem verteilten Steuerfluss. Die Komponenten verfügen über Trigger- und Valid-Signale, deren Flanken

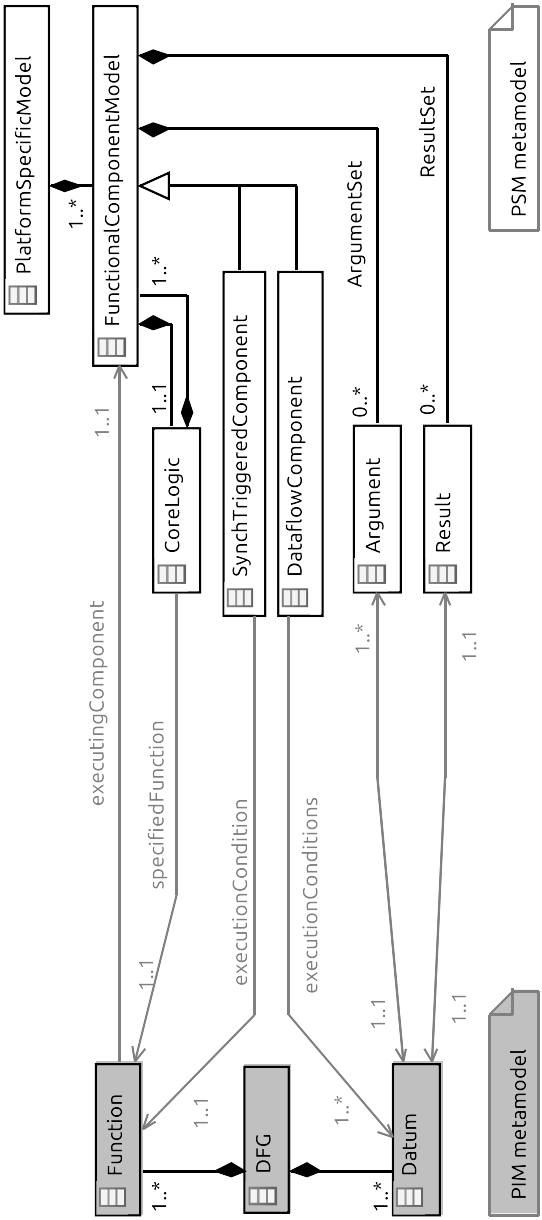


Abbildung 5.4: Metamodell der Abbildung zwischen Datenflussgraph und plattformspezifischem Komponentenmodell.

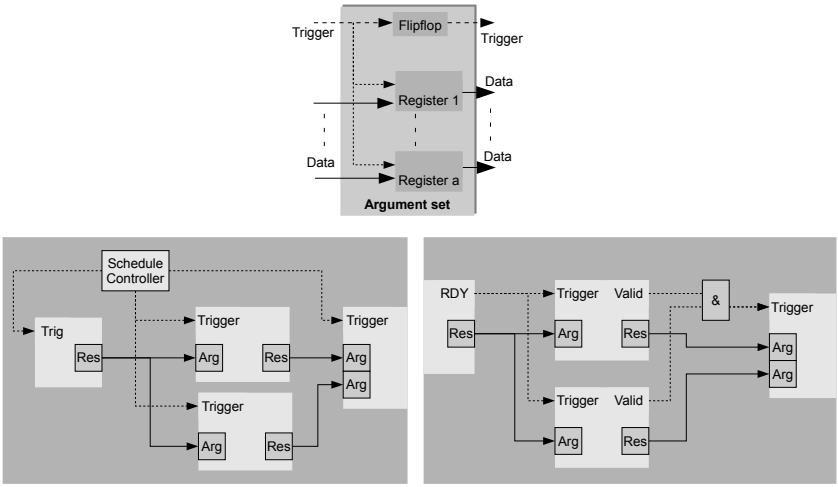


Abbildung 5.5: Synchron getriggerte Komponentenschnittstelle (oben); zeitgesteuert (links) und ereignisgesteuert (rechts) interagierende Komponenten (nach [Yok05]).

die Zeitpunkte der Datenverfügbarkeiten propagieren. Zusätzliche verteilte Steuerlogik ist nötig, um die Datenvaliditätssignale mehrerer Komponenten zusammenzuführen und zum Trigger einer Komponente zu synchronisieren. Da im HSDF pro Iteration jeder Akteur nur einmal feuert, bleiben alle Daten stabil bis zum Ende der Interation in den I/O-Registerstufen der Komponenten liegen, sodass eine zusätzliche Pufferung unnötig ist. Obwohl die Modellierung mit synchron getriggerten Komponenten ob der Interface-Struktur Vorteile bei der Integration von Black-Box-Kompnenten bietet, stellt die Konstruktion eines übergeordneten Steuerpfades auf Systemebene einen Bruch mit den "Good Design Practices"des komponentenorientierten Logikentwurfs dar [KB02].

Die zweite Klasse wird im Metamodell als *DataflowComponent* bezeichnet und verfügt über multiple Ausführungsvorbedingungen, die direkt an die Verfügbarkeit der Eingangsdaten geknüpft sind und damit der Top-Down-Umsetzung des Datenfluss-Paradigmas entsprechen. Dadurch ist es erforderlich, dass das Interface einer solchen Komponente über ein separates Synchronisationssignal zur Übernahme jedes einzelnen Eingangsdatums verfügen muss. Abbildung 5.6 zeigt die Struktur einer Argumentschnittstelle, die das SDF-Paradigma realisiert. Die Datenflusskanäle sind als Argument-FIFOs realisiert,

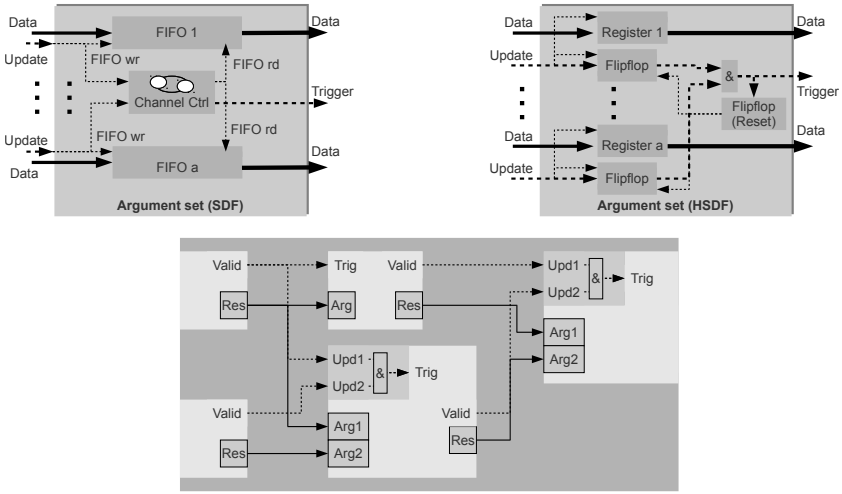


Abbildung 5.6: Datenflussgesteuerte Komponentenschnittstelle (oben) und Komponenteninteraktion (unten).

die von einem Controller (*ChannelCtrl*) überwacht werden, wodurch die lokale Konsumptionsratenanpassung vorgenommen werden kann. Die externe Datenverfügbarkeit wird durch `Update`-Signale angezeigt, die jeweils einen FIFO-Schreibzugriff (*FIFO wr*) auslösen. Das intern synchrone Lesen der FIFOs (*FIFO rd*) und das Erzeugen des internen Trigger-Signals (Feuern des Akteurs, Start der Ausführung) wird durch den Controller vorgenommen.

Für die Umsetzung des HSDF vereinfacht sich diese Struktur, wie in Abbildung 5.6 links dargestellt ist. Zur Argumentpufferung werden lediglich Register benötigt, während die Synchronisation der Update-Signale über rücksetzbare FlipFlop-Strukturen erfolgt. Die datenflussgesteuerte (dataflow-driven) Interaktion (Abbildung 5.6 unten) beruht damit auf einem feingranular verteilten Steuerfluss und der Integration aller zugehörigen Synchronisationsmechanismen in den Komponenten-Interfaces, was sich außerdem positiv auf die Leistungsfähigkeit der Low-Level-Implementierung auswirkt [KB02].

Das Zeitverhalten synchron getriggerter und datenflussgesteuerter Komponenten (HSDF) ist im Diagramm 5.7 gegenübergestellt worden. Zu beachten sind die Unterschiede in den Zeiträumen, für die die Gültigkeit der Argumente an den Eingängen garantiert werden muss, bevor sie mit dem `Trigger`- bzw.

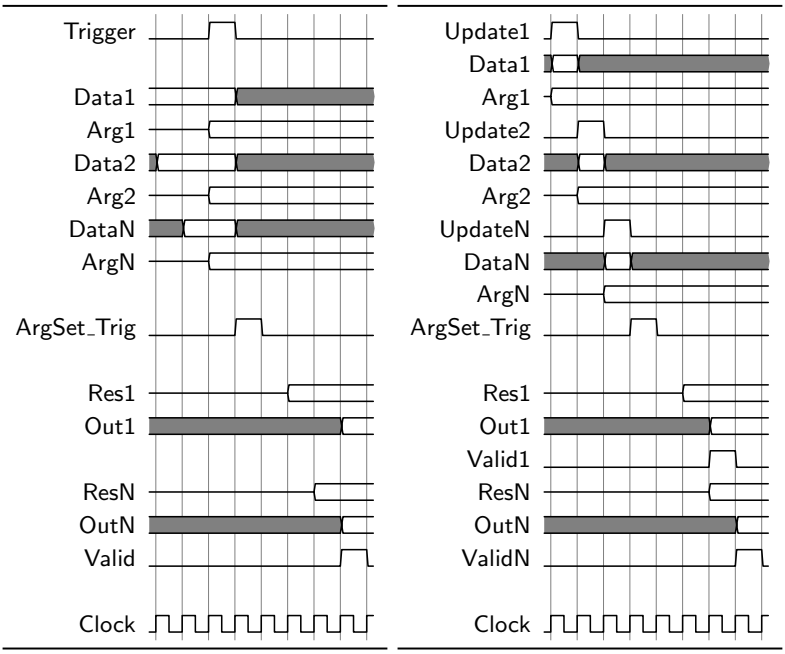


Abbildung 5.7: Zeitverhalten von Komponenten mit synchron getriggertem Interface (links) und mit datenflussgesteuertem Interface (rechts).

`Update`-Signal in die Register übernommen werden können und der interne synchrone Trigger (`ArgSet_Trig`) ausgelöst werden kann.

Generell lässt sich sagen, dass die Ansätze auch gemischt zur Anwendung kommen können, je nach dem, welche Implementierungs- und Integrationsanforderungen das betreffende Teilsystem stellt und welche Robustheit das Verhalten des *PSM* gegen wechselnde Entwurfsentscheidungen und Systemstrukturen während der Verfeinerung aufweisen soll. Die Kompositionsbeziehung zwischen Funktionskomponentenmodellen und ihrer Kernlogik lässt zu, synchron getriggerte Komponenten (v.a. Black-Boxes) in datenflussgesteuerte Schnittstellen zu kapseln, sodass das SDF-Verhalten auf Systemebene hergestellt werden kann, auch wenn bestimmte Teilsysteme stärker eingeschränkte und expliziter gesteuerte Ausführungsmechanismen verlangen.

5.2 „Resource-Sharing“ auf Komponentenebene

Wie in Abschnitt 5.1.3 erläutert wurde, besteht ein hierarchischer Datenflussgraph aus Funktionen unterschiedlicher Komplexität, die durch die Daten, die sie produzieren und konsumieren, verknüpft sind. Wird diese Spezifikation direkt in ein plattformspezifisches Komponentenmodell verfeinert, werden die Funktionen in einen Datenpfad synchroner Logikkomponenten übertragen. Dabei wird jede Funktion nach einer (1:1)-Relation auf eine Logikkomponente abgebildet, die die spezifizierte Funktion ausführt und die Daten an die abhängigen Komponenten propagiert. Dabei werden die Fähigkeiten des FPGA zur vollparallelen Verarbeitung maximal ausgenutzt. Die direkte Repräsentation aller Funktionen in Logik auf dem Chip wird als *Computing-in-Space (CIS)* bezeichnet.

Ein mögliches Entwurfsziel für nicht zeitkritische Teile eines FPGA-Entwurfs ist die Reduktion des Ressourcenbedarfs. Zu diesem Zweck ist es möglich, mehrere gleiche Funktionsblöcke auf einen Hardware-Operator abzubilden (Ressource Allocation) und die zeitlich gestaffelte Ausführung (Scheduling) mittels sequentieller getakteter Logik zu realisieren [JDKR97, Rus11]. Diese Technik ist als *Resource-Sharing* bekannt und wird von heutigen Entwurfs- und Synthese-Tools auf Elementaroperatorebene unterstützt [The11a, Xil08].

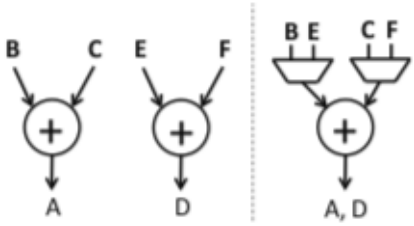


Abbildung 5.8: „Resource-Sharing“ elementarer Operatoren [HCA+12].

Abbildung 5.8 zeigt ein einfaches Beispiel für die Überlagerung zweier Additionen auf einem Addierer. Zusätzliche Logik (Multiplexer und Steuerwerk) sind nötig, um die zwei Additionen sequentiell auf dem selben Hardware-Operator durchführen zu können. Diese (n:1)-Abbildung von Funktionen auf Operatoren und ihre Zeitmultiplex-Ausführung wird auch als *Computing-in-Time (CIT)* bezeichnet.

Der hier vorgestellte Ansatz erweitert das Resource-Sharing auf das Konzept der Abbildung von komplexen Datenfluss-Akteuren auf Komponenten

der plattformspezifischen Modellierungsebene. Verwandte Ansätze sind in [ZFHT12] veröffentlicht worden. Auf dieser Abstraktionsebene werden Systementwurfsentscheidungen bezüglich des *Space-Time-Trade-Offs* getroffen und bewertet, daher ist die Ansiedelung einer Methode zur Modellierung ressourcenminimierender Komponenten hier sinnvoll. Eine weitgehende Automatisierung der Modellgenerierung sorgt zusätzlich für eine Reduktion des Aufwandes während der Design-Space-Exploration und Entwurfsiterationen. In den folgenden Abschnitten wird eine Methode zur Erzeugung von CIT-Komponenten durch Überlagerung von CIS-Komponenten vorgestellt, die während der Verfeinerung des PSM zur Anwendung kommen kann. Dazu werden zuerst die grundlegenden Konzepte ausgehend von der in den vorhergehenden Abschnitten definierten Komponentenstruktur abgeleitet, bevor die Abbildungsvorschrift und die Struktur der resultierenden Multiplex-Komponenten erarbeitet wird. Abschließend wird eine Implementierung dieser Methode zur Anwendung in MATLAB/Simulink vorgestellt und bewertet.

5.2.1 Ressourcen und Kontexte

Sollen Funktionskomponenten des PSM als komplexe Funktionseinheiten zum Zweck des Ressource-Sharing überlagert werden, ist es nötig, die Struktur, wie sie in Abbildung 4.9 definiert wurde, dahingehend zu analysieren, welche Elemente die eigentlichen wiederverwendbaren Berechnungsressourcen darstellen. Zur Abgrenzung der Aspekte der Ausführung einer Funktion von der eigentlichen ausführenden Logik wird hier der Begriff des Kontexts eingeführt (vgl. [ZFHT12]).

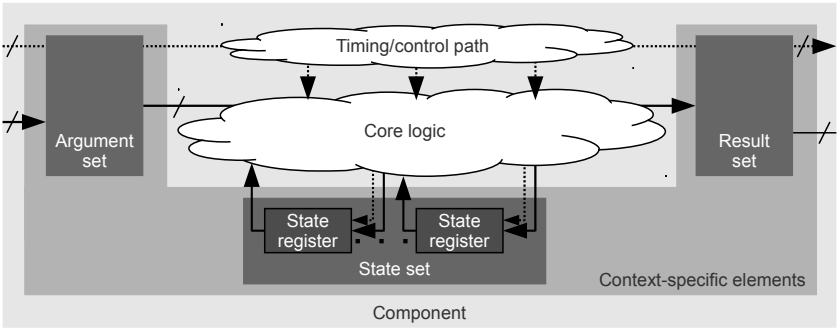


Abbildung 5.9: Klassifizierung der Interna einer Komponente bezüglich des funktionalen Kontextes.

Der *Kontext einer Funktion* ordnet einer Funktion eine Menge von Daten zu, die genau ein Ausführung eines SDF-Akteurs charakterisieren. Wie in Abbildung 5.9 dargestellt, sind die Argumente und die Resultate, sowie eventuelle Zustandsvariablen charakteristisch für den Kontext einer Funktion. Da diese Daten an entsprechende Register in der Komponente gebunden sind, sind die Eingangs- und Ausgangs-, sowie die Zustandsregistermengen (Argument, Result, State set) kontextspezifische Elemente (*context-specific elements*). Im Unterschied dazu wird die Kernlogik (*core logic*) der Komponente als wiederverwendbare konstante Ressource betrachtet, die für die Ausführung verschiedener Kontexte zur Verfügung steht.

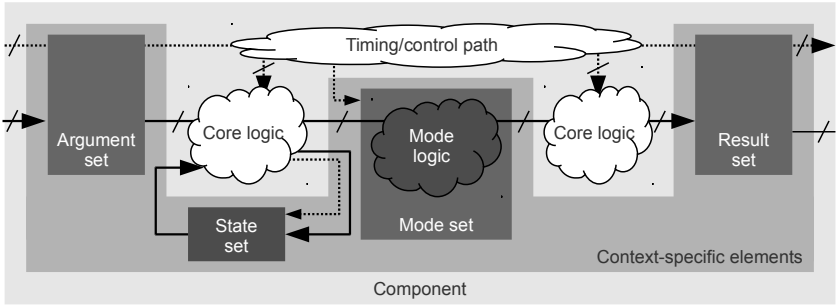


Abbildung 5.10: Der *Modus* als Teil des funktionalen Kontextes.

Das Konzept des Kontextes wird an dieser Stelle erweitert. Da es sich bei Komponenten um größere Logikgebilde handelt, die eine komplexere Funktionalität ausführen, soll der Wiederverwendungsgrad erhöht werden, indem ermöglicht wird, dass nicht nur identische sondern auch ähnliche Funktionen zur Ausführung gebracht werden können. Dazu wird, wie in Abbildung 5.10 dargestellt ist, zugelassen, dass ein kleiner Teil der Kernlogik funktionsabhängig parametrisiert und in Form des *Modus* als kontextspezifisch betrachtet wird.

Abbildung 5.11 zeigt das resultierende Metamodell der Beziehungen zwischen dem spezifizierenden DFG und dem PSM. Die Beziehung zwischen der Funktion des DFG und der CIS-Komponente entspricht der in Abbildung 5.4 angegebenen (1:1)-Relation. Eine entsprechende Beziehung zu CIT-Komponenten resultiert in einer (2:n)-Kardinalität, entsprechend der mehrfachen Bindung von Funktionen auf dieselbe Berechnungsressource. Eine (1:1)-Relation zur bijektiven Abbildung von DFG-Elementen auf Elemente der plattformspezifischen CIT-Komponenten ist nur über den Kontext möglich.

Die Kontexte beschrieben damit „virtuelle Komponenten, deren Überlagerung (*superimposedContexts*) die CIT-Komponente bestimmt. Diese enthält außerdem einen Multiplex-Controller, der das Scheduling der Kontexte zur Laufzeit übernimmt.

Zusammenfassend sei der Kontext einer Funktion $f \in F$ des in Abschnitt 5.1.1 definierten hierarchischen DFG definiert als:

$$Context(f) = \{ArgumentSet_f, ResultSet_f, StateSet_f, ModeSet_f\}$$

bestehend aus:

$$\begin{aligned} ArgumentSet_f &= \{A : \exists! d : A \to d, d \in arguments(f)\} \\ ResultSet_f &= \{R : \exists! d : R \to d, d \in results(f)\} \\ StateSet_f &= \{S : \exists! d : S \to d, d \in D(f)\} \end{aligned}$$

mit S als Zustandsspeicher für eine Zustandsvariable innerhalb des hierarchischen Funktionsblockes f, und

$$ModeSet_f = \{M : M \to f^M \subset F(f)\}$$

mit M als plattformspezifische Beschreibung einer Teilfunktion f_M innerhalb der hierarchischen Funktion f.

Für eine Überlagerung der Kontexte einer Menge von Funktionen muss eine Ähnlichkeit gelten, die wie folgt beschrieben werden kann: Seien die Funktionen $f_1 \ldots f_n \in F$ und $f_1^M \subset F(f_1), \ldots, f_n^M \subset F(f_n)$ ihre Teilfunktionen, dann muss mit

$$F(f_1) - f_1^M = \ldots = F(f_n) - f_n^M$$

die Identität der Funktionen ohne ihre Modi gelten, sodass diese durch eine wiederverwendbare Kernlogik *CoreLogic* implementiert werden können. Außerdem müssen die resultierenden Argument-, Resultat-, Zustands- und Modus-Mengen dieselbe Mächtigkeit aufweisen. Daraus ergibt sich die Definition einer CIT-Komponente des PSM:

$$c_{CIT} = (CoreLogic, \bigcup_{i=1}^{n} (Context(f_i), MuxController)$$

Abbildung 5.12 zeigt die resultierende innere Struktur einer CIT-Komponente mit Elementen der überlagerten Kontexte, und den Multiplexer (MUX)- und Demultiplexer (DEMUX)-Strukturen, die von einem zusätzlichen Steuerpfad (Mux-Controller) zum Zwecke des Kontextwechsels angesteuert werden müssen.

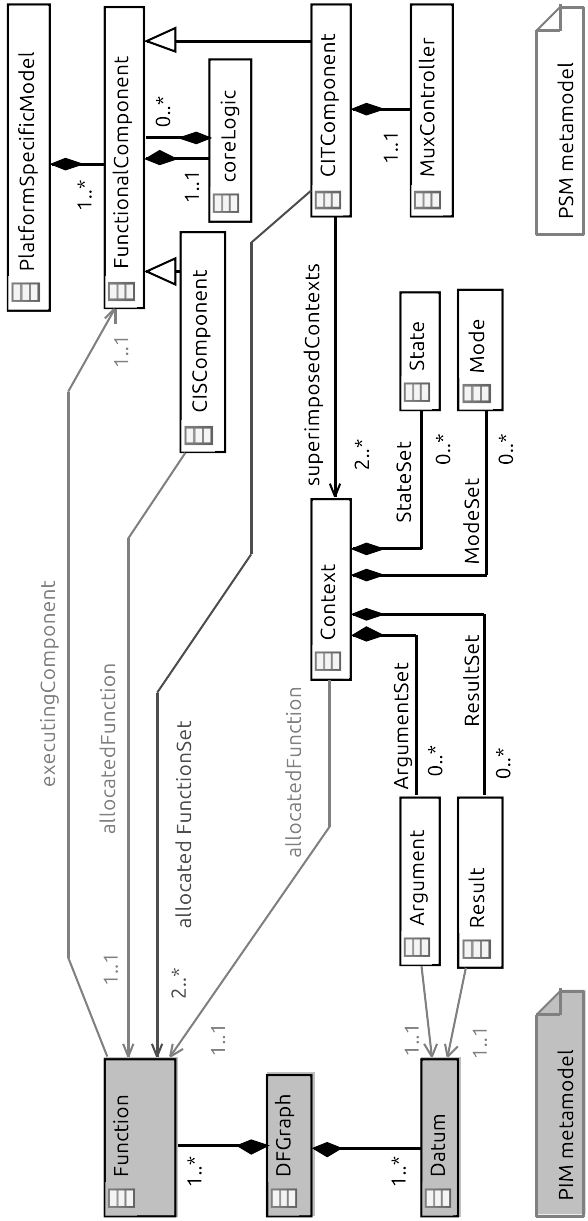


Abbildung 5.11: Metamodell der Beziehungen zwischen DFG und Kontexten von *Computing-in-Space-* bzw. *Computing-in-Time*-Funktionskomponenten (UML).

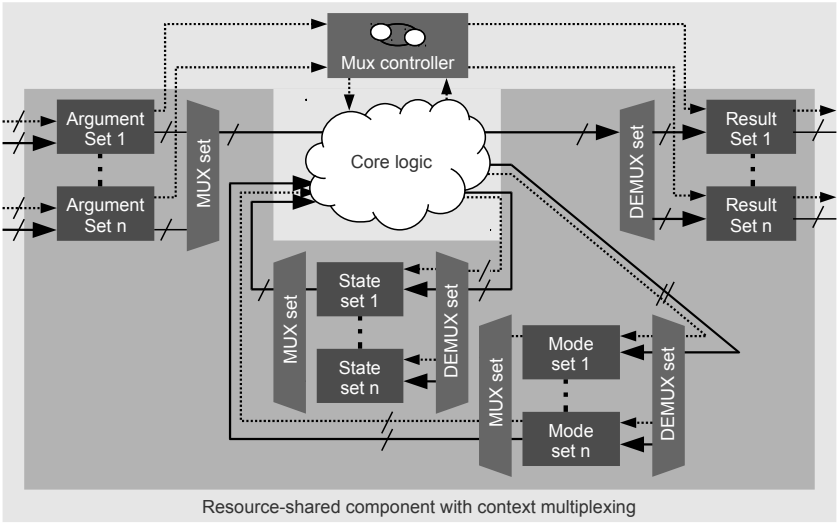


Abbildung 5.12: Komponente mit überlagerten Kontexten zur Wiederverwendung der Kernlogik.

5.2.2 Automatische Modellsynthese von Daten- und Steuerpfad

Diese Methode zur Synthese von Zeitmultiplex-Komponenten während der Verfeinerung plattformspezifischer Modelle für FPGA-Entwürfe lässt sich unter bestimmten Voraussetzungen automatisieren, sodass folgende Modelltransformationen vorgenommen werden können, um Daten- und Steuerpfad der CIT-Komponente zu erzeugen: Für alle zu überlagernden Funktionsblöcke müssen folgende Voraussetzungen gelten:

- Verfügbarkeit einer vollständig I/O-gepufferten Komponente mit synchron getriggertem oder datenflussgesteuertem Argument-Registersatz
- identische Kernlogik
- identifizierte Zustandsvariablen (Register) innerhalb der Kernlogik
- eindeutige Abgrenzung der kontextspezifischen Moduslogik von der statischer Kernlogik mit identischen Schnittstellendefinitionen

Für eine CIT-Komponente mit n Kontexten, wie in Abbildung 5.12 gezeigt, müssen folgende Aktivitäten zur Synthese des Datenpfades durchgeführt werden:

- Einfügen der statischen Kernlogik
- Einfügen der n Argument-Registersätze, Anlegen von Multiplexern (a MUX mit je n Eingängen) und Verdrahtung zu den Eingängen der Kernlogik
- Einfügen der n Resultat-Registersätze, Anlegen von Demultiplexern (a DEMUX mit je n Ausgängen) und Verdrahtung zu den Ausgängen der Kernlogik
- Einfügen der $n * s$ Zustands-Register, Anlegen von Multiplexern (s MUX mit je n Eingängen) mit Verdrahtung zu den Zustandseingängen der Kernlogik und Anlegen von Trigger-Demultiplexern (s DEMUX mit je n Ausgängen) mit Verdrahtung zu den Zustandsausgängen der Kernlogik
- Einfügen der $n * m$ Modus-Blöcke, Anlegen von Multiplexern für alle o Ausgänge der Modusblöcke ($\sum_{j=1}^{m} o_j$ MUX mit je n Eingängen) und Anlegen von Trigger-Demultiplexern (m DEMUX mit je n Ausgängen)

Der Steuerpfad besteht aus dem Controller, der das Zeitmultiplex-Schema (Scheduling) der Funktionsausführung steuern muss (MuxController). Die Konfiguration eines Kontextes wird über einen globalen Index bestimmt, mit dem die Verbindungsstrukturen (MUX/DEMUX) des Datenpfades geschaltet werden. Der zeitliche Ablauf wird anhand der Verfügbarkeitssignale der Argument-Mengen und der Synchronisationssignale der Kernlogik bestimmt.

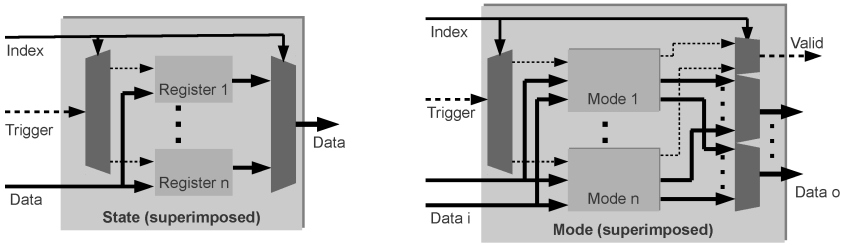


Abbildung 5.13: Struktur eines überlagerten Zustandes (links) und eines überlagerten Modus (rechts).

Der Controller setzt ein dynamisches Scheduling um, das einen Kontext konfiguriert und die Kernlogik triggert, sobald die Datenverfügbarkeit angezeigt wird und die Kernlogik verfügbar ist. Eventuelle Konkurrenz wird durch Priorisierung aufgelöst. Abbildung 5.14 zeigt die Struktur des Multiplex-Controllers.

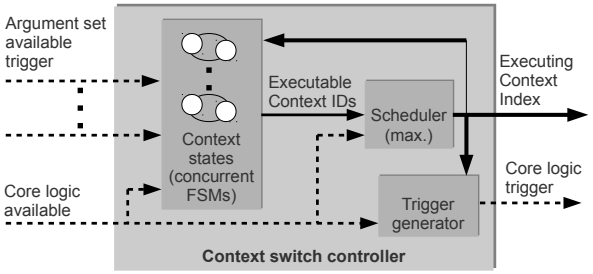


Abbildung 5.14: Struktur des Kontext-Zeitmultiplex-Controllers.

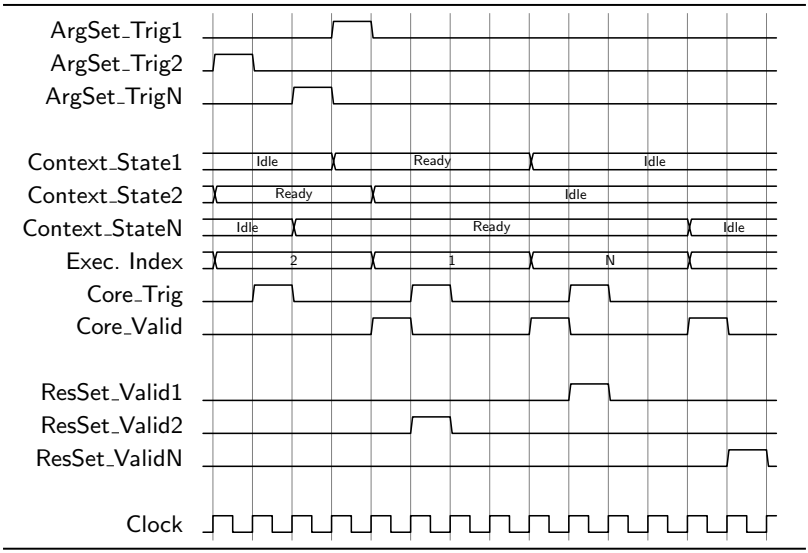


Abbildung 5.15: Verhalten der zeitmultiplexten Ausführung von Funktionen auf Resource-Sharing-Komponenten.

Das Ausführungszeitverhalten, das sich damit für eine Komponente mit überlagerten Kontexten ergibt, ist als Beispiel im Timing-Diagramm 5.15 dargestellt. Es ist zu erkennen, wie die beliebigen Zeitpunkte der Datenverfügbarkeiten durch die Trigger-Signale, die die Argumentregistersätze verlassen (*ArgSet_Trig* - vgl. Abschnitt 5.1.3) angezeigt werden. Diese sorgen für einen Zustandswechsel der betreffenden Kontexte im Multiplex-Controller. Das Anlegen des jeweiligen Index-Wertes konfiguriert die gesamte Komponente für die Ausführung des jeweiligen Kontextes auf der Kernlogik, die mit dem *Core_Trig*-Signal gestartet und deren Ende mit *Core_Valid* angezeigt wird. Das prioritätsgesteuerte Scheduling sorgt für die resultierende Ausführungsreihenfolge, die bei der Ergebnisausgabe *ResSet_Valid* zu beobachten ist.

Der Aufwand E (effort) für die Konstruktion einer solchen Komponente im Zuge der PIM-to-PSM-Transformation kann wie folgt abgeschätzt werden:

$$
\begin{aligned}
E = \mathcal{O}&\left(n * (a + r + s + m) + s + m + \sum_{j=1}^{m} o_j \right) \\
&+\mathcal{O}\left(n * (a + r + s + \sum_{j=1}^{m} (i_j + o_j)) + s + \sum_{j=1}^{m} o_j \right) \\
&+\mathcal{O}(n)
\end{aligned}
$$

Dabei ist $n \in \mathbb{N}$ die Vielfachheit der überlagerten Kontexte, $a, r, s, m \in \mathbb{N}$ jeweils die Anzahl der Argumente, Resultate, Zustandsvariablen und Modi der Funktionen und $o, i \in \mathbb{N}$ die Anzahl der Eingänge und Ausgänge der Modi. Der erste Term beschreibt dabei die Generierungs- und Parametrisierungsschritte dieser kontextspezifischen Elemente in Abhängigkeit ihrer jeweiligen Vielfachheiten und der Vielfachheit der Überlagerung. Der zweite Term beschreibt den Verbindungsaufwand und der letzte Term den linearen Aufwand zur Synthese des Multiplex-Controllers entsprechend der Vielfachheit der Kontext-Überlagerung. Der konkrete Aufwand, der sich in Koeffizienten und Konstanten der Aufwandsabschätzung ausdrücken würde, ist von der verwendeten Modellierungsplattform und der Implementierung der Modelltransformation abhängig.

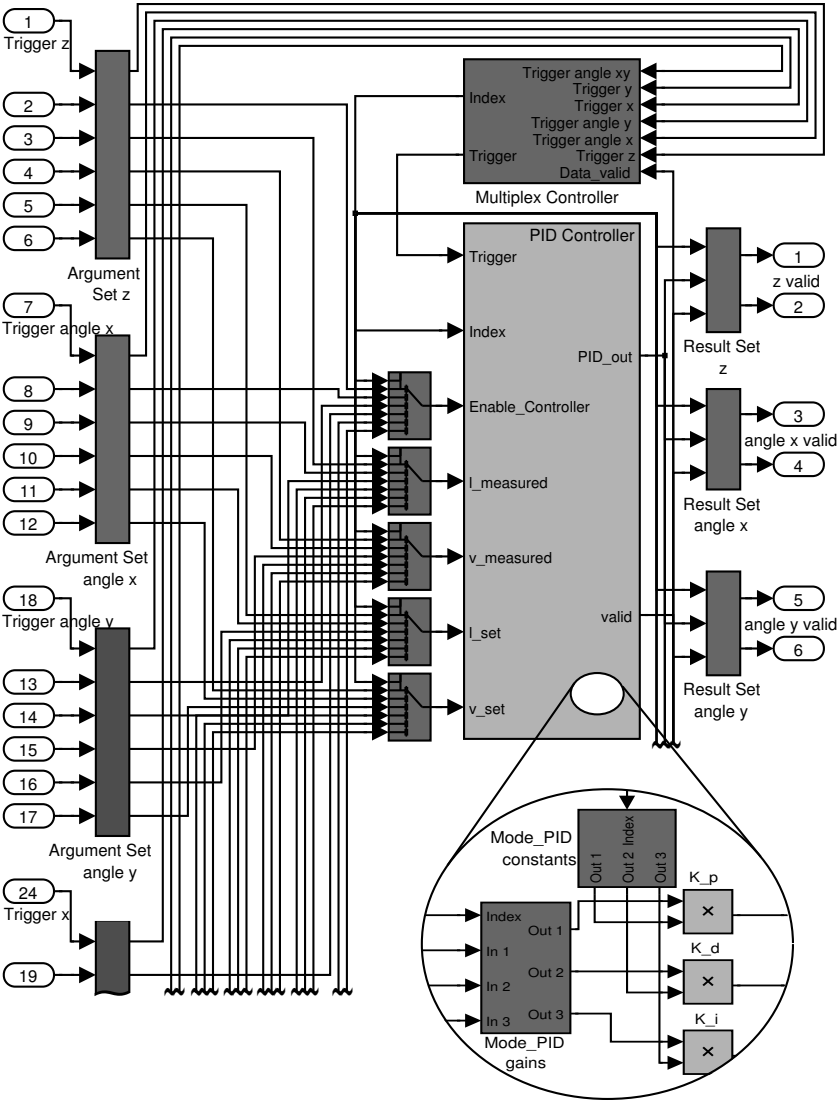


Abbildung 5.16: Ausschnitt eines mit KALI generierten Modells einer Resource-Sharing-Komponente in Matlab/Simulink.

Konzeptuelles Beispiel: Ressource-Sharing-Methode KALI in Simulink

Die Automatisierung der Methode wurde exemplarisch unter der Bezeichnung KALI sowohl für synchron getriggerte, als auch für datenflussgesteuerte Komponenten zur FPGA-orientierten Modellierung mit MATLAB/Simulink implementiert [Bot12]. Ein Beispiel für ein resultierendes Komponentenmodell ist in Abbildung 5.16 zu sehen.

Die Simulink-Implementierung reduziert den manuellen Entwurfsaufwand bei der plattformspezifischen Modellierung auf

$$E = \mathcal{O}(n * (s + m)).$$

Dies entspricht der a-priori-Identifikation der s Zustände und m Modi in den n zu überlagernden Funktionsblöcken, was eine anwendungsspezifische Design-Entscheidung darstellt. Dieser Aufwand könnte durch die Integration umfangreicherer Modellanalysemethoden, wie etwa die in [BR97] angesprochene direkte Komponenteninstanziierung und Komponentenidentifikation, weiter reduziert werden. Die quantitative Bewertung dieser Methode anhand des Modellierungsbeispiels eines komplexen Regelungssystems erfolgt in Abschnitt 6.3.

5.2.3 Feasibility-Abschätzung

Zur Einschätzung, ob der Einsatz des Resource-Sharing überhaupt gerechtfertigt ist, muss die Frage beantwortet werden, ob die Überlagerung von n Kontexten gegenüber der parallelen n-fachen Implementierung die gewünschten Einsparungen erzielen kann. Eine kritische Rolle dabei spielt das zuvor beschriebene, bezüglich mehrerer Kriterien proportionale Anwachsen der nötigen Steuer- und Verbindungslogik in der zu erstellenden Resource-Sharing-Komponente. Folgende Ungleichung bezüglich der on-Chip-Ressourcen von parallel zu implementierenden Komponenten $Res(c_{CIS})$ und den wesentlichsten ressourcenintensiven Anteilen einer CIT-Komponente muss erfüllt bleiben:

$$\begin{aligned}\sum_{j=1}^{n} Res(c_{CIS})_j > & Res(CoreLogic_{c_{CIT}}) + \sum_{j=1}^{n} Res(ModeSet_j) \\ & + n * Res(ArgumentSet, StateSet, ResultSet) \\ & + Res(MuxCtrl_{c_{CIT}}) + Res(MUX) \end{aligned} \tag{5.1}$$

Dabei sind die Anteile wie folgt zu benennen[1]:

- der Ressourcenbedarf der CIS-Komponente $Res_{c_{CIS}}$ in LUTs und FFs ist durch Prototyping zu bestimmen,
- der Ressourcenbedarf der Kernlogik $Res_{CoreLogic}$ und der Modi $Res_{ModeSet}$ der c_{CIT}-Komponente in LUTs und FFs ist durch Prototyping zu bestimmen,
- der Ressourcenbedarf der Argument-, Resultat- und Zustandsmengen in FFs ist über die jeweilige Bit-Breite $bits()$ des Datums zu berechnen:

$$Res_{ArgumentSet,StateSet,ResultSet} = \sum_{j=1}^{a} bits(A_j) + \sum_{j=1}^{s} bits(S_j) + \sum_{j=1}^{r} bits(R_j)$$

- der Ressourcenbedarf der zusätzlichen Multiplexer-Logik in LUTs ist über den Ressourcenbedarf der modellierten Multiplexer in Abhängigkeit der Anzahl ihrer Eingänge und deren Bit-Breiten $Res_{MUX}(k, bits)$ wie folgt abzuschätzen:

$$\begin{aligned} Res(MUX) = &\sum_{j=1}^{a} Res_{MUX}(n, max(bits(A_j)) \\ &+ \sum_{j=1}^{s} Res_{MUX}(n, max(bits(S_j)) \\ &+ \sum_{k=1}^{m} \sum_{j=1}^{o} Res_{MUX}(n, bitwidth(O_j)) \end{aligned}$$

Hier gelten wiederum $n \in \mathbb{N}$ als die Vielfachheit der parallel zu implementierenden Komponenten bzw. der zu überlagernden Kontexte, $a, r, s, m \in \mathbb{N}$ als die Anzahlen der Argumente A, Resultate R, Zustände S und Modi, sowie $o \in \mathbb{N}$ als die Anzahl der Modusausgänge O. Aus diesen Betrachtungen lässt sich ableiten, dass das Resource-Sharing-Konzept natürlich von möglichst großen, wiederverwendbaren Kernlogiken profitiert. Dazu sollten die Modi mit Bedacht, möglichst klein und, unter Betrachtung ihres Einflusses auf die Komplexität der Multiplexer im Datenpfad, mit möglichst wenigen Ports

[1] für die im Rahmen dieser Arbeit zur Anwendung gekommene Implementierung von KALI in MATLAB/Simulink und die Ziel-FPGA-Hardware Spartan-3-4000 sind Messreihen zum Ressourcenbedarf a) des Multiplex-Controllers der Quelle [Bot12] und b) der Multiplexer dem Anhang A.1 zu entnehmen

gewählt werden. Die Latenz der resultierenden CIT-Komponente wächst proportional mit dem Grad der Kontextüberlagerung und fällt entsprechend bei der Bestimmung des Echtzeit-Budgets ins Gewicht.

5.2.4 Bemerkungen zur Resource-Sharing-Methode

Die fortschreitende Weiterentwicklung plattformspezifischer Modellierungswerkzeuge betrifft auch den Aspekt des Resource-Sharing. So wird die Wiederverwendung von Operatoren und identischen Modulen z.B. vom MathWorks Simulink HDL Coder, oder vom National Instruments LabVIEW FPGA Module (hier unter der Bezeichnung „re-entrant VI“) unterstützt. Es ist zu erwarten, dass diese Konzepte in Zukunft noch weiter entwickelt werden. Die Betrachtung der Wiederverwendung im Rahmen des Komponentenbegriffs ist ein Beitrag dieser Arbeit. Die Charakterisierung des Kontext-Begriffs dient der Systematisierung der Komponentenwiederverwendung auch im Hinblick auf künftige Anwendungsgebiete. Die naheliegenste FPGA-Technologie, die die statische Zuordnung zwischen Funktion und Berechnungsressource weiter auflöst, ist die Dynamic Partial Reconfiguration (DPR), die den Austausch von Logikmodulen auf der FPGA-Fläche zur Laufzeit des on-Chip-Systems erlaubt. Aktuelle FPGA-Entwicklungswerkzeuge unterstützen bereits die Erstellung und Synthese von DPR-Plattformen. Die in dieser Arbeit getroffene Definition des Modus als kontextspezifischer austauschbarer Teil einer statischen Ausführungskomponente legt die Grundlage für eine Modellbeschreibung von DPR-Komponenten. Über diese Formalisierung kann in die Entwicklung von modellbasierten Entwurfsmethoden für DPR-Systeme eingestiegen werden.

5.3 Schnittstellenlogik zur Off-Chip-Kommunikation

Sowohl Automatisierungssysteme mit einer physisch verteilten Anordnung von Sensoren und Aktoren und sensornaher Datenverarbeitung, als auch komplexe, in Hardware zu realisierende Hochleistungsanwendungen unterliegen harten physischen Beschränkungen wie I/O-Pin-Anzahlen und Chipfläche. Für solche Systeme können mehr oder weniger stark gekoppelte Multi-FPGA-Anordnungen, wie sie in Abschnitt 3.1.2 beschrieben wurden, als Plattformvarianten in Betracht gezogen werden. Ziel eines modellbasierten Entwurfs für solche Systeme ist es, so lange wie möglich eine geschlossene Modelldarstellung und simulative Validierung des Gesamtsystemverhaltens zu ermöglichen.

Für das plattformspezifische Systemmodell bedeutet dies, dass die Partitionierung eines Anwendungsmodells für die verteilte Plattform und die Kommunikation zwischen den Partitionen mit berücksichtigt werden muss. Wie in Abbildung 5.17 gezeigt wird, besteht ein wesentlicher Aspekt der plattformspezifischen Modellierung in der Behandlung der durch die Partitionierung aufgetrennten Signalpfade zwischen den Komponenten. Diese Signalpfade sind genau die, deren Daten- und Steuerflüsse über die Off-Chip-Kommunikationsinfrastruktur der Plattform realisiert und rekonstruiert werden müssen.

In den folgenden Abschnitten wird die Struktur der plattformspezifischen Modellierung von verteilten FPGA-Entwürfen analysiert und die Grundlagen für eine systematische und automatisierte Generierung von applikationsspezifischen Kommunikationsstrukturen auf Modellebene erarbeitet. Abschließend wird eine Implementierung dieser Methode zur Anwendung in MATLAB/Simulink vorgestellt und bewertet.

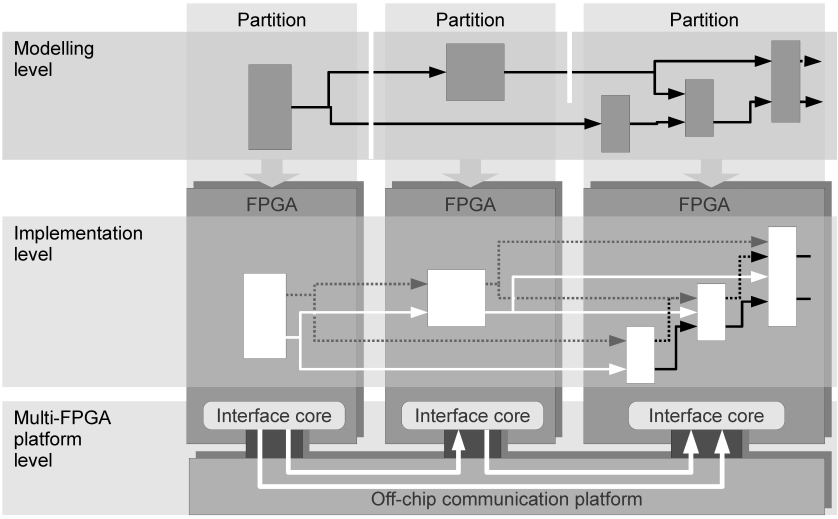


Abbildung 5.17: Gesamtanwendung als verteilter FPGA-Entwurf auf einem Multi-FPGA-System.

5.3.1 Plattform, Partitionen und Komponenten

Zur Identifikation der Einflussfaktoren auf die Generierung applikationsspezifischer Zugriffslogik auf der plattformspezifischen Modellierungsebene müssen zuerst die Abhängigkeiten und Beziehungen zwischen den Elementen des plattformspezifischen Modells und den daraus zu erstellenden FPGA-Designs geklärt werden.

Die Herausforderungen für das Design der Zugriffslogik beruhen auf zweierlei Umständen. Erstens kann die Zugriffslogik, obwohl sie exklusiven Zugriff auf die Signalpfade der plattformspezifischen Schnittstelle nimmt, nicht als wiederverwendbarer Teil der Plattform betrachtet werden, da die Schnittstelle zur Applikation aus den Signalpfaden zu den jeweiligen Funktionskomponenten besteht und damit von Partition zu Partition und von System zu System variabel ist. Zweitens kann die Zugriffslogik nicht als Teil der Applikation betrachtet werden, da in der Spezifikation der Gesamtapplikation keine Informationen zur Kommunikationsplattform enthalten sind, die an dieser Stelle im Top-Down-Verfeinerungsprozess zu benutzen wären.

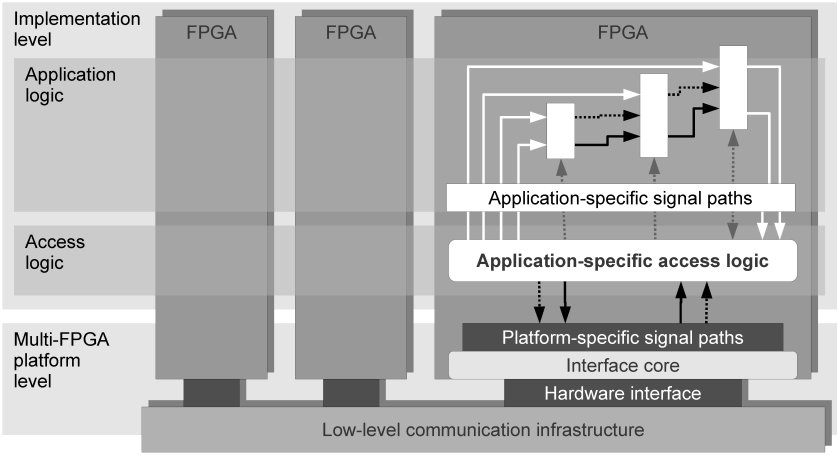


Abbildung 5.18: Applikationsspezifische Protokolllogik zur Intermodulkommunikation in Multi-FPGA-Systemen.

Abbildung 5.18 zeigt die Implementierungsstruktur einer Partition auf einer Multi-FPGA-Plattform, in der die Zugriffslogik in einem applikationsspezifischen Logikblock (*Application-specific access logic*) zusammengefasst ist.

Dieser Block ist im Unterschied zum Off-Chip-Kommunikationskontroller dem applikationsspezifischen Implementierungsteil zugeordnet und nimmt die Abbildung der aufgetrennten Signalpfade von und zu den Funktionskomponenten (*application-specific signal paths*) auf die plattformspezifischen Signalpfade (*platform-specific signal paths*) vor [EPT99]. Für den Entwurfsprozess bedeutet das, dass der plattformspezifische Entwurf des verteilten Systems um neue applikationsspezifische Logik ergänzt werden muss, für deren Erzeugung Bottom-Up-Informationen über die Zielplattform einbezogen werden müssen. Der Aufwand zur Erzeugung dieser Logik ist auf HDL-Ebene vergleichsweise hoch und bleibt auch auf der Ebene der plattformspezifischen Modellierung erhalten, da die benötigten Strukturen aufgrund ihres applikationsspezifischen Charakters weder vom erhöhten Abstraktionsgrad der Modellebene, noch von der Wiederverwendbarkeit von Plattformkomponenten profitieren können. Das Bestreben, diesen Aufwand zu verringern, motiviert die Suche nach automatischen Synthesemöglichkeiten im modellbasierten Entwurf.

Die Analyse der Elemente und Beziehungen in plattformspezifischen Modellen für Multi-FPGA-Systeme geschieht in Abbildung 5.19 anhand eines Metamodells in Form eines Klassendiagramms der UML. Aufbauend auf der in Abbildung 4.10 vorgestellten Metamodellstruktur für Multi-FPGA-Systeme sind auch hier Partitionsmodelle (*PartitionModel*) als Teile des plattformspezifischen Gesamtmodells die Grundlage zur Schaffung der Einzel-FPGA-Entwürfe.

Wesentlich für die Betrachtung der Off-Chip-Kommunikation ist die Tatsache, dass jedem Partitionsmodell genau ein Modellblock zur Schnittstellenabstraktion der Kommunikationsinfrastruktur zugeordnet ist, der als Black-Box-Modell zur Verfügung steht (*OffChipComBlackBox*) und bei der Codegenerierung eine vorimplementierte Plattformkomponente (*OffChipComCore*) instantiiert, wie durch die Assoziation *instantiatedCore* verdeutlicht wird. Die Beziehungen zwischen Off-Chip-Kommunikationskontroller und Multi-FPGA-Plattform wurden bereits in Abschnitt 4.1.3 und Abbildung 4.7 erläutert.

Die Funktionskomponenten *FunctionalComponent* der jeweiligen Partition stehen bezüglich des Bedarfs nach Off-Chip-Datenaustausch in einer (n:1)-Beziehung zu dieser exklusiven Ressource, wodurch sich der Bedarf nach spezifischer Logik zur Zugriffsregelung ergibt. Durch die Partitionierung kann jeder Funktionsblock jetzt als Teile seiner Argument- und Resultat-Mengen über Argumente bzw. Resultate verfügen, die auf entfernten Partitionen produziert (*remotelyProduced*) bzw. konsumiert (*remotelyConsumed*) werden müssen. Die applikationsspezifischen Signalpfade, die diese Daten mit der Zugriffslogik verbinden, werden von entsprechenden Sende- bzw. Empfangsmechanismen (*SendFacility, ReceiveFacility*) zusammengefasst. Diese Mechanismen

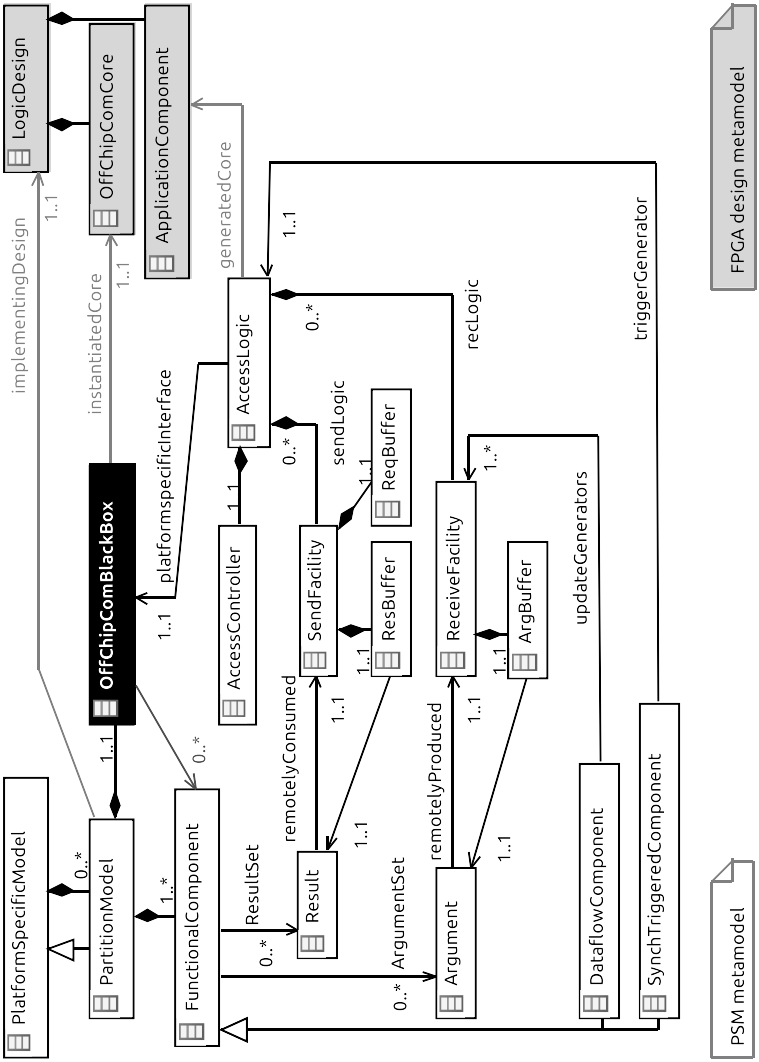


Abbildung 5.19: Metamodell der Beziehungen zwischen Funktionskomponenten und der Off-Chip-Kommunikationsschnittstelle in einem verteilten FPGA-plattformspezifischen Modell (UML).

stellen neben Puffern (*ResBuffer, ArgBuffer*) die Verbindungselemente dar, die entsprechend der benötigten Vielfachheit repliziert oder parametrisiert werden und dadurch eine (1:1)-Beziehung mit den zu übertragenden Daten herstellen können. Die Zusammenfassung der Einzelmechanismen zur Sende- bzw. Empfangslogik (*sendlogic, recLogic*) macht diese zum Teil der Zugriffslogik-Komponente, die mittels eines Zugriffskontrollers (*AccessController*) über das plattformseitig fest definierte plattformspezifische Schnittstellenprotokoll auf Signalpfade der Off-Chip-Kommunikation zugreift. Resultat ist eine top-down entworfene applikationsspezifische Design-Komponente der Zugriffslogik (*generatedCore*).

Zusammenfassend sei die Struktur der applikationsspezifischen Zugriffslogik einer Partition p des PSM mit Funktionskomponenten $c_{PSM} \in p$ für Multi-FPGA-Systeme definiert als:

$$AccessLogic(p) = \{SendLogic_p, RecLogic_p, AccessController_p\}$$

bestehend aus:

$$\begin{aligned} sendLogic_p = &\{SendFacility : \exists! R : SendFacility \\ &\rightarrow R, R \in ResultSet(c_{PSM})\} \\ recLogic_p = &\{ReceiveFacility : \exists! A : ReceiveFacility \\ &\rightarrow A, A \in ArgumentSet(c_{PSM})\} \end{aligned}$$

Zur Rekonstruktion des Synchronisationsverhaltens müssen je nach dem auf der Partition verwendeten Synchronisationsparadigmas der Applikationskomponenten zwei Verfahren unterschieden werden. Für datenflussgesteuerte Komponenten werden die benötigten Verfügbarkeitssignale zusammen mit den empfangenen Daten durch die Empfangsmechanismen erzeugt (die Rolle *updateGenerators* der *ReceiveFacility*). Für synchron getriggerte Komponenten hängt der Trigger-Zeitpunkt von der Verfügbarkeit aller benötigten Daten (lokal und entfernt produziert) ab und muss daher an zentraler Stelle und mit globalem Wissen von der Zugriffslogik vorgenommen werden (die Rolle *triggerGenerator* der *AccessLogic*). In den folgenden Beispielimplementierungen wird dieser Unterschied genauer beleuchtet.

5.3.2 Automatische Modellsynthese der Zugriffslogik

Für das FPGA-plattformspezifische Modell müssen folgende Voraussetzungen gelten, um die automatische Generierung der applikationsspezifischen Zugriffslogik zu unterstützen:

- Applikationsfunktionalität als Komponentenmodell - Komponenten mit synchron getriggertem oder datenflussgesteuertem Interface nach den Definitionen in den Abschnitten 4.2.2 und 5.1.3 mit korrekt identifizierten Daten- und Synchronisationssignalen
- Partitionsmodell - Aufteilung der Applikationskomponenten in Partitionen, die den Ziel-FPGAs zugeordnet werden können (d.h. keine Logik in den partitionsübergreifenden Datenpfaden)
- Plattformentscheidung - automatische Generierung für eine bestimmte unterliegende Kommunikationsinfrastruktur

Auf Basis eines derartig strukturierten Modells ist die Generierung der Zugriffslogik für jede Partition möglich.

Konzeptuelles Beispiel: Zugriffslogik für shared-memory-basierte Intermoulkommunikation - VISHNU für SHIVA

Diese Methode zur Synthese von applikationsspezifischer Zugriffslogik während der Verfeinerung plattformspezifischer Modelle für FPGA-Entwürfe wurde exemplarisch unter der Bezeichnung VISHNU für die FPGA-orientierte Modellierung realisiert [Bot10, Bot11]. Im folgenden werden die generierten Strukturen für die shared-memory-basierte Kommunikationsinfrastruktur SHIVA (siehe Abschnitt 4.1.2) vorgestellt. Unter Berücksichtigung beider Ansätze der Interaktion zwischen Applikationskomponenten (synchron getriggert bzw. datenflussgesteuert) werden in den Abbildungen 5.20 und 5.22 zwei Varianten vorgestellt, die sich in der Struktur der Empfangslogik unterscheiden. Dabei beschreiben A die entfernt produzierten Argumente und R die entfernt konsumierten Resultate der in der lokalen Partition enthaltenen Funktionskomponenten F, sowie $a, r, f \in \mathbb{N}$ ihre jeweiligen Anzahlen.

Plattformspezifische Anbindung an SHIVA: Zur Anbindung der applikationsspezifischen Funktionalitäten von VISHNU an die Schnittstelle der Shared-Memory-basierten Kommunikationsplattform SHIVA sind die im Folgenden beschriebenen Konzepte zu realisieren, die in den Abbildungen 5.20 und 5.22 dargestellt sind. Da es sich bei SHIVA um eine exklusive Speicherschnittstelle

handelt, ist an dieser Stelle zusätzlich zu dem Serialisieren der applikationsspezifischen Anfragen durch Sende- bzw. Lese-Logik (*Access scheduler*) auch die zeitliche Abfolge von Schreib- und Lese-Zugriffen zu gewährleisten. Dies wird durch einen zentralen Steuer-Automat (*SHIVA access controller*) realisiert, der die Verfügbarkeit der Schnittstelle über das `Busy`-Signal prüft, das Zugriffsrecht zwischen Schreib- und Lese-Logik verteilt und damit ungültige und konkurrierende Zugriffe über den gemeinsamen Adressbus verhindert. Der Aufwand zur Erstellung des SHIVA-Zugriffskontrollers ist konstant - dieses Element kann sogar wiederverwendet werden.

Sendelogik: Die Sendelogik ist für beide Varianten gleich gestaltet und erfordert für jede Partition

- Einfügen einer Anordnung von Pufferelementen für die r Resultate, die extern konsumiert werden sollen (*Remotely consumend results buffer set*),
- Einfügen eines Multiplexers für die Daten (*Data MUX*) mit r Eingängen,
- Einfügen von r Ein-Bit-Registern für die `Valid`-Signale (*Request buffer*),
- Generieren der Zugriffssteuerung (*Access scheduler*) zur Serialisierung der potentiell r gleichzeitig ankommmenden zu übertragenden Daten und Erzeugung einer Sequenz von Sende-Anfragen mit physischen Adressen für das Shared-Memory-Interface, die in Interaktion mit dem SHIVA-Zugriffskontroller abgearbeitet werden.

Der Erstellungsaufwand für die Sendelogik einer Partition ergibt sich aus der r-fachen Vielfachheit der Elemente und kann mit $E = \mathcal{O}(r)$ für sowohl Platzierung als auch Verdrahtung der Elemente abgeschätzt werden.

Die Generierung der Adressen bestimmt die Konfiguration und die Kommunikationleistungsparameter der SHIVA-Plattform, wie in Abschnitt 4.1.2 beschrieben wurde. Mittels der globalen Informationen aus dem plattformspezifischen Gesamtmodell können die Adressierungstabellen der VISHNU-Sendelogiken nach zwei wesentlichen Strategien definiert werden:

- *Minimierung des Kommunikationsspeicherbedarfs* - dabei wird die Gesamtanzahl der zu übertragenden Daten betrachtet und die entsprechende Anzahl von benötigten Speicherzellen zu gleichen Teilen auf die Speichersegmente der Module verteilt. Der lokale Adressraum eines jeden SHIVA-internen Speichers benötigt dann

folgende Adressbreite:

$$width(address_{SHIVA}) = \left\lceil log_2 \left(\frac{1}{p} \sum_{j=1}^{p} r_j \right) \right\rceil$$

Dabei ist p die Anzahl der Partitionen und r_j die Anzahl der Ausgänge der Partitionen ($p, r, j \in \mathbb{N}$). Die Adressen werden daraufhin so vergeben, dass möglichst viele Daten eine Adresse auf ihrem Empfängermodul erhalten, d.h. dass latenzarme Remote-write/Local-read-Transaktionen durchgeführt werden können. Darüber hinaus werden die Adressen entsprechend freier Speicherzellen vergeben und erhöhte Kommunikationlatenzen zugunsten des minimierten Speichers in Kauf genommen.

- *Minimierung der Kommunikationlatenz* - hierbei wird das Maximum der Partitionseingänge betrachtet, um den lokalen Speicherbedarf zu bestimmen. Damit wird sichergestellt, dass jede SHIVA-Instanz die für die lokale Partition bestimmten Daten auch im lokalen Speichersegment ablegen kann, und damit ausschließlich latenzarme Remote-write/Local-read-Transaktionen durchgeführt werden können. Die lokale Adressbreite für die SHIVA-Speicher berechnet sich dann durch:

$$width(address_{SHIVA}) = \left\lceil log_2 \left(\max_{j=1 \to p} (a_j) \right) \right\rceil$$

Dabei ist p die Anzahl der Partitionen und a_j die Anzahl der Eingänge der Partitionen ($p, a, j \in \mathbb{N}$). Die Adressen werden dann so vergeben, dass jedes Datum eine Adresse im Speichersegment seines Empfängermoduls bekommt. Aufgrund der geforderten Symmetrie des Distributed-Shared-Memory-Adressraumes entsteht bei Systemen mit Partitionen mit sehr ungleichem Off-Chip-Kommunikationaufkommen ein entsprechender Overhead in Form von ungenutztem SHIVA-Speicher, der zugunsten der minimalen Kommunikationstransaktionslatenz in Kauf genommen werden muss.

Empfangslogik mit zentraler Datenflusssynchronisation: Ein kritisches Szenario für Systeme mit synchron getriggerten Applikationskomponenten ergibt sich, wie in Abbildung 5.20 dargestellt, wenn durch die Partitionierung für bestimmte Komponenten Abhängigkeiten von sowohl lokalen, als auch entfernt produzierten Daten entstehen. Eine konzeptuelle Variante für die Interface-Logik sieht vor, dass diese als Datenflussmanager für *synchron getriggerte Applikationskomponenten*, die über externe Datenabhängigkeiten verfügen, agiert.

Da die externen Abhängigkeiten einen Teil der Vorbedingungen zur Ausführung der jeweiligen Komponente darstellen, bestimmt die Verfügbarkeit der externen Daten den Zeitpunkt mit, zu dem die jeweilige lokale Komponente mit der Berechnung beginnen kann. Das heißt, dass die externe und die lokale Datenverfügbarkeitssignalisierung synchronisiert werden müssen, um die eigentlichen Komponenten-Trigger-Signale als Teil des lokalen Steuerflusses zu erzeugen. Um die benötigten Informationen zur Verfügung zu stellen, müssen die lokalen Datenverfügbarkeitssignale zur Schnittstellenlogik geführt werden, wo sie als Vorbedingungen (`Preconditions`) in die Trigger-Logik für die Komponenten mit externen Datenabhängigkeiten eingehen.

Die interne Struktur der automatisch generierten VISHNU-Logik mit zentraler Datenflusssteuerung wird in Abbildung 5.20 gezeigt. Charakteristisch ist hier die Empfangslogik, deren Kern der Funktionstriggergenerator (*Fct. trigger generator*) ist. Basierend auf den `PRECOND`-Signalen, die die erfüllten lokalen Vorbedingungen anzeigen, und den adressgefilterten `RemoteUpdateNotify`-Signalen, mit denen das SHIVA-Interface die Verfügbarkeit von externen Argumenten anzeigt, trifft dieses Modul die Entscheidung zum Starten einer lokalen Applikationskomponente. Dazu werden zuerst die externen Daten mittels einer Reihe von Leseanfragen (`ReadRequest`) aus dem Shared-Memory angefordert und mittels eines Demultiplexers (*Data DEMUX*) den Argument-Puffern (*Remotely produced argument buffer set*) übergeben.

Ein nachgeschalteter Funktionsdemultiplexer (*Function DEMUX*) verbindet diese Speicherelemente mit den Ausgangsports zu der jeweiligen Applikationskomponente, die bei vollständiger Datenverfügbarkeit mittels eines generierten *TRIGGER*-Signals gestartet wird. Die Komplexität des Funktionstriggergenerators hängt damit nicht nur von der Gesamtanzahl a der eingehenden Argumente ab, sondern auch von der Anzahl f an lokalen Funktionen, die diese Daten benötigen. Die Besonderheit bei diesem Ansatz ist jedoch, dass der Argumentregistersatz nicht alle entfernt produzierten Argumente der Partition halten können muss, sondern nur die maximale Anzahl dieser Argumente einer lokalen Applikationskomponente. Die Argumente werden also erst bei Bedarf nach Erfüllen aller Vorbedingungen für eine bestimmte Komponente aus dem Shared-Memory in den Registersatz geladen. Dies spart wesentliche Speicher-Ressourcen auf dem Chip, da für den Trigger-Generator nur 1 Bit breite Statussignale gespeichert werden müssen. Der Aufwand zur Erstellung dieser Empfangslogik hängt neben der Gesamtanzahl der entfernt produzierten Argumente auch von der Anzahl der an der Off-Chip-Kommunikation beteiligten lokalen Funktionskomponenten f und dem Maximum der von einer der Komponenten erwarteten entfernt produzierten Argumente ab und lässt sich für jede Partition mit $E = \mathcal{O}(a + f + \max_{i=1 \to f}(a_i))$ sowohl für Platzierung als auch Verdrahtung abschätzen.

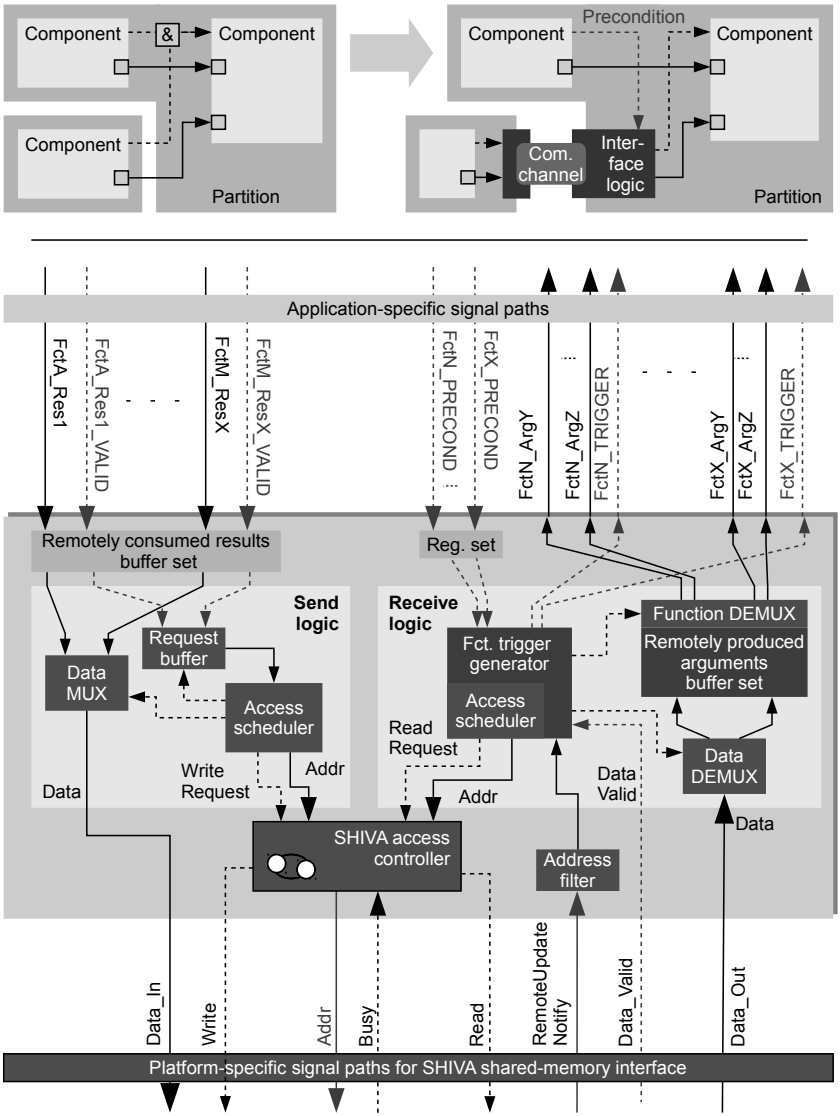


Abbildung 5.20: Synchronisation von lokal und entfernt produzierten Argumenten (o.) und Zugriffslogik zur Off-Chip-Kommunikation mit zentraler integrierter Datenflusssynchronisation (u.).

Allerdings leiden Durchsatz und Latenz unter der strengen Sequentialisierung von Datenverfügbarkeitsprüfungen, Speicher auslesen und Starten der Komponenten. Das Timing-Diagramm in Abbildung 5.21 verdeutlicht diese Abläufe.

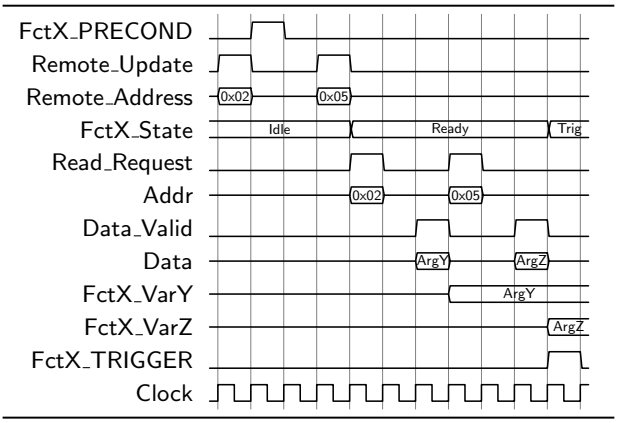


Abbildung 5.21: Verhalten der datenfluss-synchronisierenden Leselogik.

Empfangslogik ohne Datenfluss-Synchronisation: Für *datenflussgesteuerte Applikationskomponenten* besteht das Problem der zentralen Synchronisation lokaler und externer Argumente nicht, da das Komponenteninterface dies selbst mittels der datenspezifischen Update-Signale bewerkstelligt.

Der Empfangsteil der generierten Zugriffslogik in Abbildung 5.22 kann dementsprechend deutlich einfacher ausfallen, da die Synchronisations- und Puffer-Logik bereits dezentral in den Komponenten verfügbar ist. Die über `RemoteUpdateNotify` angezeigten Verfügbarkeiten der externen Daten werden hier nach der Adressfilterung (*Address filter*) direkt gepuffert (*Availability buffer*), um daraufhin von der Zugriffssteuerung (*Access scheduler*) in Shared-Memory-Leseanfragen (`ReadRequest`) umgewandelt. Jedes ausgelesene Datum wird sofort per Demultiplexer (*Data DEMUX*) in das jeweilige Ausgangspufferelement geladen (*Remotely produced arguments buffer set*). Dabei erzeugt die Zugriffssteuerung aus `Data_Valid` mittels Demultiplexing (*Update DEMUX*) das dazugehörigen `Update`-Signal. Bei dieser Lösung sind daher genau *a* Ausgangspuffer nötig, eins für jedes extern erzeugte Argument einer lokalen Applikationskomponente . Datum

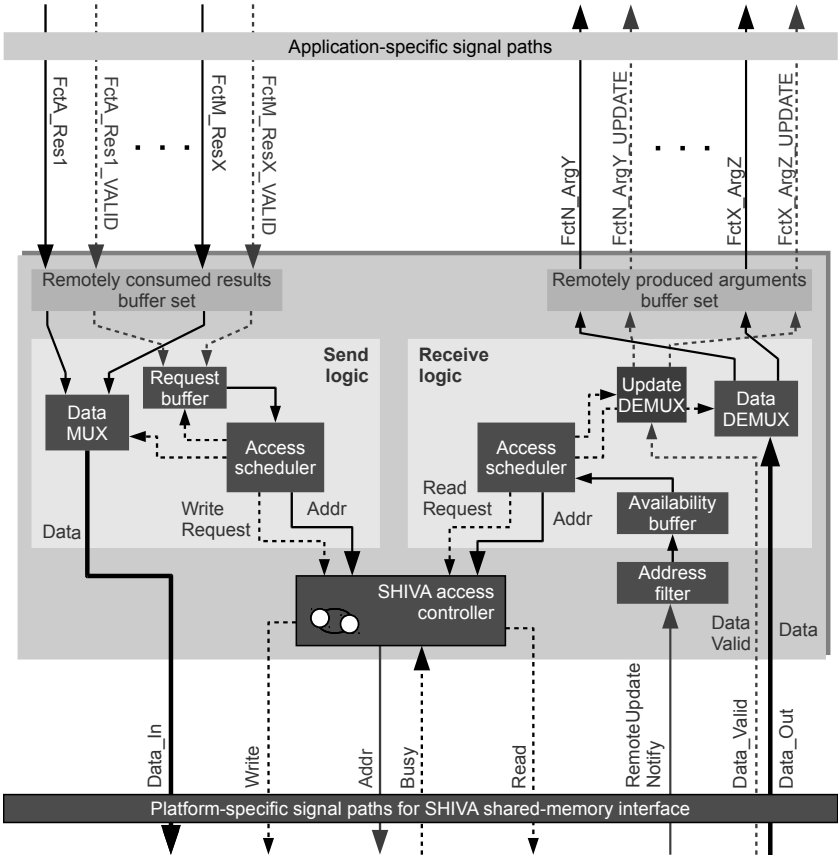


Abbildung 5.22: Zugriffslogik zur Off-Chip-Kommunikation ohne Datenfluss-Synchronisation.

und `Update`-Signal werden ohne weitere Verzögerung an das Interface der jeweiligen Applikationskomponente angelegt. Der Aufwand zur Erstellung dieser Empfangslogik hängt nur von der Anzahl der entfernt produzierten Argumente aller lokalen Funktionskomponenten ab und lässt sich für jede Partition mit $E = \mathcal{O}(a)$ abschätzen.

Das unmittelbare Auslesen der Daten und die fehlenden zentralen Synchronisationsmechanismen wirken sich positiv auf den Durchsatz der Zugriffslogik

und auf die Latenz der Gesamtapplikation aus. Der Zeitverlauf dieser Aktionen ist im Timing-Diagramm in Abbildung 5.23 dargestellt.

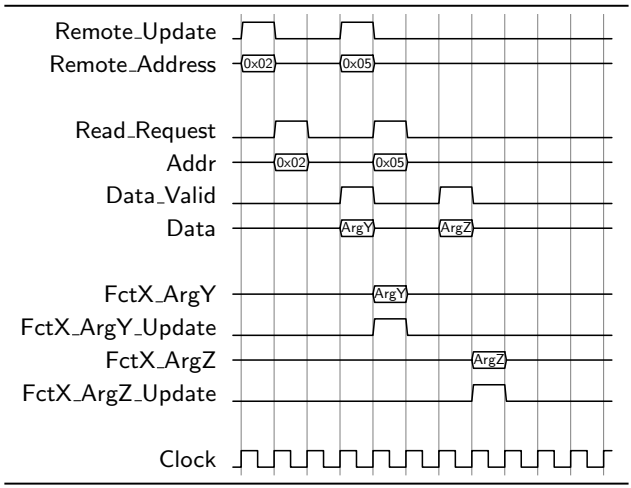


Abbildung 5.23: Verhalten der Leselogik ohne Datenfluss-Synchronisation.

Der Gesamtaufwand E (effort) für die Generierung der applikationsspezifischen Zugriffslogik im Zuge der PIM-to-PSM-Transformation kann wie folgt:

$$E = \mathcal{O}\left(\sum_{j=1}^{p}(a_j + f_j + \max_{i=1\to f}(a_i) + r_j)\right) \tag{5.1}$$

$$+ \mathcal{O}\left(\sum_{j=1}^{p}(a_j + f_j + \max_{i=1\to f}(a_i) + r_j)\right) + \mathcal{O}(p) \tag{5.2}$$

für die Variante für mit zentraler Datenflusssteuerung, und

$$E = \mathcal{O}\left(\sum_{j=1}^{p}(a_j + r_j)\right) + \mathcal{O}\left(\sum_{j=1}^{p}(a_j + r_j)\right) + \mathcal{O}(p) \tag{5.3}$$

für die Variante ohne Datenflusssynchronisation abgeschätzt werden. Dabei ist p die Anzahl der Partitionen, f die Anzahl der an der Off-Chip-Kommunikation beteiligten Funktionskomponenten einer Partition und r_j, a_j die Anzahl der entfernt konsumierte Resultate und produzierte Argumente der jeweiligen Partitionen ($p, f, a, r, j \in \mathbb{N}$). Der erste Term beschreibt dabei

jeweils die Generierungs- und Parametrisierungsschritte der Elemente der Sende- und Empfangslogik in Abhängigkeit der jeweiligen Vielfachheit der extern generierten oder konsumierten Daten. Der zweite Term beschreibt den Verbindungsaufwand. Der letzte Term bezieht sich auf den linearen Aufwand zur Erzeugung der statischen Partitionsgrundstrukturen und der Instanziierung der dazugehörigen Black-Box-Module des abstrakten Kommunikationssimulationsmodells, die nur die Platzierung und Verdrahtung eines einzelnen Blocks pro Partition erfordern. Der konkrete Aufwand, der sich in Koeffizienten und Konstanten der Aufwandsabschätzung ausdrücken würde, ist von der verwendeten Modellierungsplattform und der Implementierung der Modelltransformation abhängig.

VISHNU in Simulink: Die Automatisierung der VISHNU-Methode wurde exemplarisch sowohl für synchron getriggerte, als auch für datenflussgesteuerte Komponenten zur FPGA-orientierten Modellierung mit MATLAB/Simulink implementiert [Bot10, Bot11]. Ein Beispiel für ein resultierendes partitioniertes Modell ist in Abbildung 5.24 zu sehen. Die Simulink-Implementierungen reduzieren den manuellen Entwurfsaufwand bei der plattformspezifischen Modellierung für Multi-FPGA-Designs auf die Strukturierung der Partitionen und die Konfiguration des Generators, was mit

$$E = \mathcal{O}(p)$$

approximiert werden kann. Die Demonstration dieser Methode anhand des Modellierungsbeispiels eines komplexen Regelungssystems erfolgt in Abschnitt 6.4.

5.3.3 Bemerkungen zur Zugriffslogik-Generierungsmethode

Die Generierung von Zugriffslogik zur Off-Chip-Kommunikation ist eine Methode, die auf der Modellentwurfsebene sehr gut angesiedelt ist. Der Entwurf der Gesamtapplikation kann hier auch nach der Partitionierung und mit Anbindung an die Kommunikationsinfrastruktur geschlossen simuliert und getestet werden, bevor daraus die einzelnen FPGA-Designs abgeleitet und synthetisiert werden. Die Definition von regelmäßigen wiederkehrenden Strukturen und die Extraktion aller nötigen Zuordungen aus dem Gesamtmodell machen die vollständige Automatisierung dieser Methode auf der Modellebene möglich.

Die zwei entwickelten und untersuchten Ansätze für Systeme auf der Basis von synchron getriggerten bzw. datenflussgesteuerten Applikationskomponenten

verfügen über gewisse Eigenschaften, die Diskussionen über ihre optimale Anwendbarkeit motivieren. Der Einsatz von VISHNU mit zentraler Synchronisation bedingt im Fall von sowohl lokalen als auch externen Datenabhängigkeiten immer eine Redesign-Iteration der Applikationspartitionen, um den Steuerpfad so zu modifizieren, dass konsistente `Precondition`-Signale zur Verfügung stehen. Dies entfällt bei datenflussgesteuerten Komponenten, vorausgesetzt,

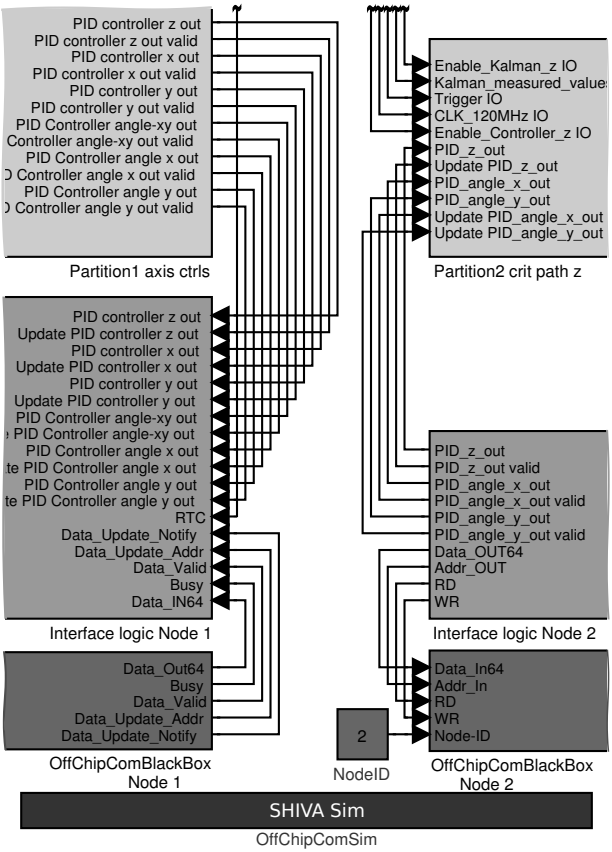


Abbildung 5.24: Ausschnitt eines automatisch mittels der VISHNU-for-SHIVA-Methode generierten verteilten multi-FPGA-plattformspezifischen Modells in MATLAB/Simulink.

die Gesamtapplikation ist bereits in dieser Weise entworfen worden. Andernfalls müssen die Komponenten einem Redesign-Zyklus unterworfen werden.

Die plattformspezifische Modellierung dient auch der Beschleunigung der Design-Space-Exploration durch Modellierung und simulative Bewertung verschiedener Lösungsvarianten. Gerade für verteilte Lösungen ist unter Umständen die Untersuchung verschiedener Partitionierungen oder Kommunikationsplattformen nötig. Ein vollständig datenflussorientierter Entwurf bietet für diese Methode den robusteren und mit geringerem Entwurfsaufwand behafteten Ansatz, da komponentenspezifisches Synchronisationsverhalten und off-chip-kommunikationsspezifisches Verhalten besser voneinander getrennt sind.

5.4 Einordnung in die plattformspezifische Modelltransformation

Abbildung 5.25 zeigt eine detaillierte Einordnung der zuvor erarbeiteten Konzepte und Methoden in den Prozess der Transformation von einer plattformunabhängigen Beschreibung in ein FPGA-plattformspezifisches Modell. Der Prozess stellt damit eine Verfeinerung der *PIM-to-PSM-Transformation* dar, die als ein zentraler Aspekt der modellbasierten Entwurfsprozesse für eingebettete System im Allgemeinen (Abbildung 2.8) und für Multi-FPGA-Systeme im Speziellen (Abbildung 4.11) charakterisiert wurde.

Ausgehend von einem funktional validierten plattformunabhängigen Modell (*Platform-independent model*) wird durch Modularisation die Modellierung von Applikationskomponenten vorbereitet. Der Entwurf eines komponentenorientierten Systems (*Component System Design*) geschieht unter Zugriff auf die plattformspezifische Modellierungsebene, die Modellrepräsentationen synthetisierbarer Artefakte bereitstellt. Die Verfügbarkeit von wiederverwendbaren IP-Cores erlaubt die Deklaration von *Black-Box*-Komponenten. Andere Komponenten müssen spezifisch entworfen werden (*Custom Component Design*), was sowohl die Verfeinerung der funktionalen Beschreibung zu synthetisierbaren Operatoren (*Operator refinement*), als auch die Schaffung einer wohldefinierten Komponentenschnittstelle (*Component interface*) und eines internen Steuerpfades (*Internal control path*) beinhaltet. Entsprechend müssen die Komponenten auch auf Systemebene durch einen zu entwerfenden globalen Steuerpfad (*System level timing/control path*) verknüpft werden.

Als Resultat des Komponentenentwurfs entsteht ein plattformspezifisches Modell eines FPGA-Entwurfs (*Platform-specific FPGA design model*). Die Va-

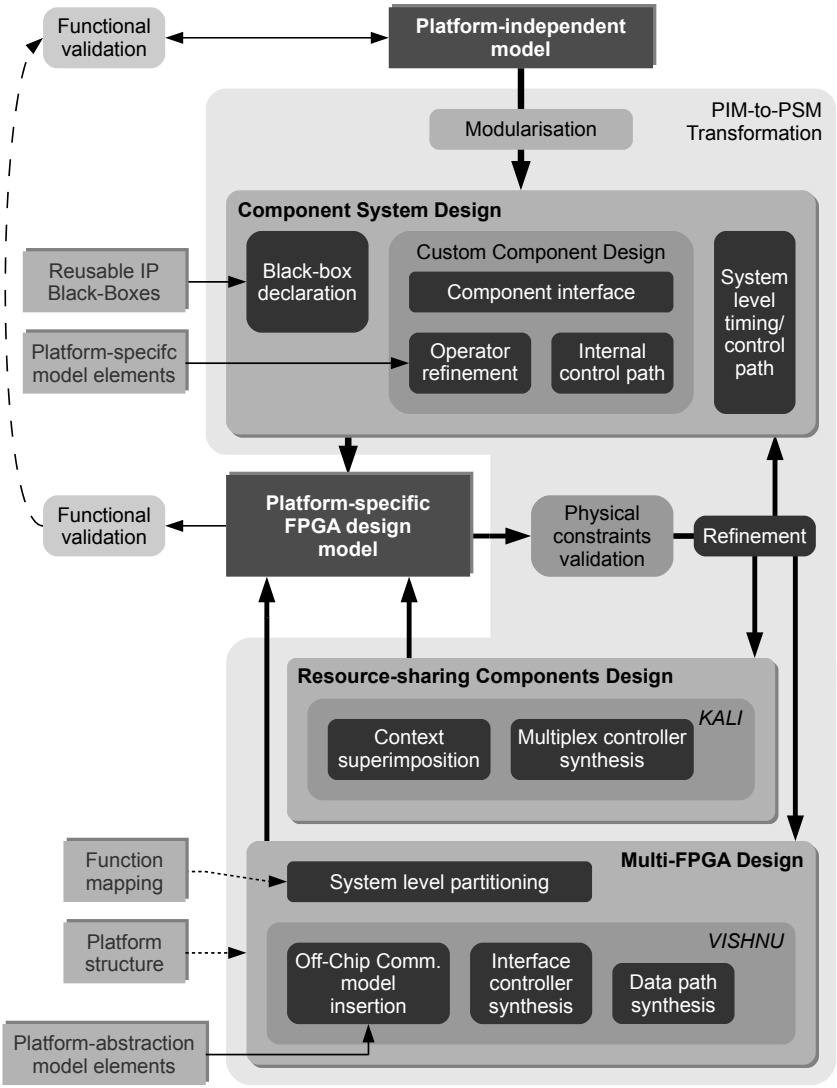


Abbildung 5.25: Vom Funktionsmodell zum Implementierungsstrukturmodell - die PIM-to-PSM-Transformation im Detail.

lidität dieses Modells als Beschreibung einer möglichen Implementierung hängt erstens natürlich vom Erfolg einer funktionalen Validierung (*Functional validation*) durch Simulation gegen das plattformunabhängige Spezifikationsmodell ab, und zweitens von der Einhaltung der physikalischen Beschränkungen, die der Lösung durch die Plattform und die Einsatzumgebung auferlegt werden. Die entprechende Validierung (*Physical constraints validation*) kann entweder auf der Hardware-Ebene durch prototypische Implementierung, Test und Bewertung, oder auf der Modellebene mittels Virtual Prototyping unter Zuhilfenahme von a-priori-Leistungsdaten (Frequenzen, Latenzen, Durchsätze, Chip-Ressourcen-Bedarf, etc.)[2] geschehen.

Die Ergebnisse der Realisierbarkeitsbewertung des plattformspezifischen Modells motivieren weitere Verfeinerungen und Design-Iterationen. Diese können sowohl den Komponentenentwurf, als auch weiterführende Systementwurfsentscheidungen betreffen. Entwurfsentscheidungen, die für Systeme aus der Anwendungsdomäne der Mess- und Regelungstechnik von Bedeutung und daher Gegenstand dieser Arbeit sind, betreffen die Ressourcenminimierung und den Multi-FPGA-Entwurf. Ressourcenminimierung kann durch Schaffung von „resource-shared"-Komponenten durch Zusammenfassung und sequentielle Ausführung mehrerer gleicher Funktionsmodule geschehen. Diese Modelltransformation umfasst die Überlagerung der Datenpfade (*Context superimposition*) und die Synthese eines spezifischen Multiplexcontrollers (*Multiplex controller synthesis*). Sie wird durch die im Rahmen dieser Arbeit entwickelte und implementierte Methode KALI automatisiert. Der Entwurf für eine Multi-FPGA-Plattform erfordert die Partitionierung des Applikationsmodells (*System level partitioning*) und die Schaffung von Verbindungen zur Kommunikationsplattformabstraktion. Die nötigen Verteilungsinformationen sind das Ergebnis eines multikritiellen Optimierungsprozesses, der in der Übersicht in Abbildung 2.8 angedeutet ist. Die Informationen werden für diese Transformation als gegeben vorausgesetzt werden. Ausgehend von einem partitionierten PSM umfasst die Modelltransformation die Erzeugung neuer Datenpfade von den Applikationskomponenten zu den Off-Chip-Kommunikationschnittstellen (*Data path synthesis*) und die Synthese spezifischer Zugriffskontroller, sowie das Einfügen der Simulationsmodells der Kommunikationsplattform (*Off-chip com. model insertion*), das eine geschlossene simulative Validierung des Gesamtsystems gewährleisten soll. Diese Vorgänge wurden durch die im Rahmen dieser Arbeit entwickelte und implementierte Methode VISHNU automatisiert.

[2] z.B.: Xilinx System Generator unterstützt die Abschätzung des Ressourcenbedarfs auf Modellebene; Simulink HDL Coder erlaubt die Rücknotierung der Latenzen der kritischen Pfade ins PSM

Die Ansiedelung dieser Methoden im modellbasierten Entwurf erlaubt die automatische Generierung applikationsspezifischer Strukturen, die einen hohen Entwurfsaufwand erfordern und nicht von der klassischen Wiederverwendung von Entwurfskomponenten (z.B. aus Bibliotheken) profitieren. Dadurch können die Design- und Redesign-Zyklen zur Erstellung plattformspezifischer Lösungsvarianten auf Modellebene verkürzt werden.

5.5 Fazit

In diesem Kapitel wurde ein Reihe von Konzepten vorgestellt, die im modellbasierten Top-Down-Entwurfsprozess für (Multi-)FPGA-Systeme Anwendung finden sollen. Der modellbasierte Entwurfsprozess selbst verringert die Implementierungskomplexität durch Abstraktion von den unter der Modellierungsplattformebene liegenden Implementierungsdetails und -vorgängen. Das macht die Ebene des plattformspezifischen Modells zu dem Ort, an dem ein Großteil der Erforschung des Entwurfsraumes (Design-Space-Exploration) stattfindet und die PIM-to-PSM-Transformation zu dem Prozess, in dem Systementwurfsentscheidungen umgesetzt werden.

Aufgrund der in der zugrundeliegenden Anwendungsdomäne als Spezifikation gebräuchlichen Funktionsblockbeschreibung und des komponentenorientierten Modellierungs- und Implementierungsansatzes dient das Paradigma des grobgranularen synchronen bzw. homogenen synchronen Datenflusses als plattformspezifische Ausführungssematik, die durch den Steuerfluss zwischen den Komponenten realisiert werden muss. In Erweiterung des Metamodells der plattformspezifischen Modellierungsebene wurde die Definition datenflussgesteuert interagierender Komponenten erarbeitet. Dabei wurde auf die Unterscheidung bezüglich des Verhaltens der Komponentenschnittstellen wertgelegt - synchron getriggert, entsprechend der meisten, für global synchrone Entwürfe vorgesehenen, wiederverwendbaren IP-Komponenten, bzw. datenflussgesteuert, für Komponenten, die ihre Ausführung bezüglich global asynchroner Datenverfügbarkeitszeitpunkte selbst triggern. Die Methode des komponentenbasierten Ressource-Sharing beruht auf der Trennung von Strukturelementen, die unabdingbar zur Erhaltung des funktionalen Kontextes sind, von der eigentlichen Berechnungsstruktur, sodass mittels Komponentenüberlagerung Einsparungen von Chip-Ressourcen bei der FPGA-Implementierung erreicht werden können. Die Definitionen wurden auf der Basis des plattformspezifischen Modellierungsmetamodells erarbeitet. Als Neuerung ist hierbei der Ansatz der Moduslogik zu werten, die den flexibleren Einsatz des Resource-Sharing auf der grobgranularen Komponentenebene erlaubt. Als konzeptuelles Beispiel wurde eine Implementierung dieser

Methode zur Anwendung in MATLAB/Simulink beschrieben und bewertet. Ausgehend von einem als grobgranulares Datenfluss-System entworfenen und validierten Gesamtmodell erfordern Multi-FPGA-Entwürfe das Auftrennen der Datenflussverbindungen und die Abbildung derselben auf eine Intermodulkommunikationsinfrastruktur, eine Aktivität, die hochgradig anwendungsfallspezifisch ist, und daher auch auf Modellebene mit hohem Entwurfsaufwand auf niedrigem Abstraktionsgrad verbunden ist. Die entwickelte Methode automatisiert die Generierung der Zugriffslogik, was anhand einer konzeptuellen Beispielimplementierung zur Anwendung in MATLAB/Simulink gezeigt und bewertet wurde.

Die erarbeiteten Konzepte lassen sich in den iterativen Prozess der PIM-to-PSM-Transformation einordnen, tragen durch ihre Automatisierung zur Verringerung des Aufwandes der plattformspezifischen Modellierung bei. Die dadurch erreichte Verkürzung der (Re-)Design-Zyklen dient der Erhöhung der Effizienz der Design-Space-Exploration für Systementwürfe, die auf (Multi-)FPGA-Plattformen ausgerichtet sind.

6 Modellbasierter hardware-orientierter Entwurf in der Anwendung

Die in den vorherigen Kapiteln erarbeiteten und beschriebenen Ansätze zur modellbasierten Entwicklung von verteilten FPGA-basierten Lösungen sollen in diesem Kapitel für Problemstellungen der Anwendungsdomäne der Nano-Positioniertechnik eingesetzt werden. Ein wesentliches Einsatzgebiet für hochleistungsfähige eingebettete Recheneinheiten ist der geschlossene Regelkreis zur Stabilisierung des Bewegungszustandes der Nano-Positioniereinrichtung. Die angestrebte Kombination von hoher Präzision und schnellem Positionier- und Messbetrieb kann nur mittels komplexer Regelungssysteme erreicht werden. Die Rechenlast der benötigten Algorithmen muss in minimalen und harten Echtzeitschranken bewältigt werden können, da die Reglerausführungszeit direkt die Abtastrate, die Regelabweichung und damit die resultierende Prozessqualität beeinflusst [ZKA+12]. Zum effektiven Einsatz eines Regelungssystems ist dessen performante Implementierung auf einem eingebetteten Echtzeitverarbeitungssystem nötig, die, je nach Randbedingungen, den Entwurf applikationsspezifischer, möglicherweise verteilter Plattformlösungen erfordert.

In diesem Kapitel wird gezeigt, wie aus einer validierten Regelungsspezifikation unter Anwendung modellbasierter Methoden spezifische Hardware-Implementierungen zur Aus-führung auf (Multi-)FPGA-Systemen erzeugt werden. Als Basis für den Entwurf der Reglung und die Durchführung der modellbasierten Entwicklungsschritte dient die etablierte Entwicklungsumgebung TheMathworks MATLAB/Simulink [The11b]. Einleitend wird das modellierte Regelungssystem für Nano-Positionier- und -Messmaschinen vorgestellt und charakterisiert. Ausgehend von diesem als Spezifikation zu betrachtenden Modell wird die modellbasierte Überführung in FPGA-Implementierungsmodelle in drei Hauptaspekten beleuchtet. Zum ersten steht der Entwurf und die quantitative Bewertung von hardware-orientierten Funktionskomponenten im Vordergrund. Darauf aufbauend werden die auf Modellebene erzielbaren Effekte der Resource-Sharing-Methode KALI analysiert. Drittens wird an einem ausgewählten repräsentativen Szenario die Implementierung von verteilten Multi-FPGA-Lösungen unter Anwendung und Bewertung der Schnittstellengenerierungsmethode VISHNU demonstriert. Abschließend werden ver-

schiedene Implementierungsszenarien zusammengefasst, die effektive Realisierungen unter verschiedenen Randbedingungen als Ergebnisse der Design-Space-Exploration darstellen.

6.1 Spezifikation des Regelungssystems

Die Möglichkeiten von Entwurfswerkzeugen wie MATLAB/Simulink erlauben den Entwurf von Regelungssystemen auf verschiedenen Abstraktionsebenen. Ein *System*-Modell enthält dabei [DF05]:

- das Modell des zu regelnden Prozesses - *das Streckenmodell*,
- das Umgebungsmodell und die Instrumentierung, und
- das Modell des Reglers als *Device under Development*.

Während das Streckenmodell von sehr abstrakter Form sein kann und nur die Transferfunktion oder das funktionale Verhalten des zu regelnden Prozesses beschreiben muss, ist das Teilmodell des Reglers der Teil, der den modellbasierten Entwicklungsprozess vom Reglerentwurf bis zur validierten eingebetteten Implementierung zu durchlaufen hat [MSF12].

Gegenstand der Betrachtungen wird im Folgenden das Modell eines 3D-Trajektorienfolgereglers sein, wie er für Hochpräzisionspositionier- und -messanlagen verwendet wird. Das betreffende System wurde in [MFAA09, ZAM+10] vorgestellt und bezüglich modellbasierter Implementierungen auf verteilten eingebetteten Plattformen untersucht [MSF12].

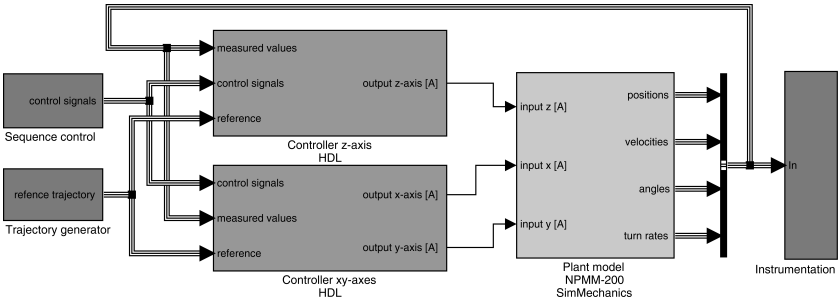


Abbildung 6.1: Top-level des Simulationsmodells des Regelungssystems einer NPMM.

Abbildung 6.1 zeigt das aus drei Partitionen bestehende Systemmodell. Eine Partition enthält das Streckenmodell (*Plant model*), ein SimMechanics-Modell des Positionieraufbaus der NPMM. Dieser Aufbau erhält Stellgrößen in Form von Strömen, die elektromagnetische Aktoren für alle drei räumlichen Achsen ansteuern. Die lateralen Positions- und Geschwindigkeitswerte, ebenso wie Winkel und Winkelgeschwindigkeiten werden vom Streckenmodell als Messwerte ausgegeben. Eine andere Partition enthält ein abstraktes Modell der Ablaufsteuerung (*Sequence control*) zur Steuerung der generellen Betriebsmodi der Regelung, den Bahnplaner (*Trajectory generator*) zur Erzeugung der Referenzwerte, sowie die Instrumentierung zur Überwachung der Simulation und Validierung des Entwurfs. Eine weitere Partition enthält die eigentlichen Reglermodule (*Controller*), die auf eine Hardware-Beschreibung (HDL) zur Implementierung auf FPGAs abgebildet werden sollen. Die Regler für jede Achse bestehen aus modifizierten Proportional-Intregral-Differential (PID)-Reglern, gekoppelt mit Störbeobachtern, und ergänzt um zusätzliche Winkel-Regler zur Kompensation möglicher Verdrehungen um die jeweiligen Achsen.

In Abbildung 6.2 wird exemplarisch das Modell des Reglers für die x-Achse genauer unter die Lupe genommen. In diesem Reglermodul werden für den modifizierten *PID-Regler* zuerst die Regelabweichungen für sowohl Position als auch für die als separate Eingangsgröße zugeführte 1. Ableitung Geschwindigkeit bestimmt, bevor diese den jeweiligen Verstärkungsgliedern zugeführt werden. Mittels des Positionsfehlers wird eine zusätzliche nichtlineare Verstärkung bestimmt, bevor das Signal auf den eigentlichen diskreten Integrator gegeben wird. Eine Stellgrößenbeschränkung nach der Summation der P-,I- und D-Anteile und eine Anti-Wind-Up-Funktion komplettieren die Beschreibung des modifizierten PID-Reglers. Ein *Kalman-Filter* dient als Störbeobachter für nichtmessbare Effekte - in dieser Anwendungsdomäne sind dies hauptsächlich Reibkräfte [AZA08]. Der Filter arbeitet auf den aktuell gemessenen Positionswerten und dem akkumulierten Ausgabewert des letzten Regelschrittes, um seinen internen Zustand zu aktualisieren und den Reibkompensationswert für den aktuellen Regelschritt zu bestimmen. Der komplette Ausgabewert für diese Achse ist die Summe der PID- und Kalman-Filter-Ausgaben und repräsentiert einen Strom zur Ansteuerung eines elektromagnetischen Aktuators.

Die spezifizierte Abtastrate des Gesamtprozesses beträgt 10 kHz, so dass dieses Regelungs- und das Streckenmodell mit einem Basiszeitintervall (*base sample time* in Simulink) von $T\_Control = 100\mu s$ simuliert wird. Die Positionsdaten für die Regler können Werte im Bereich von $\pm 20mm$ mit Subnanometer-Auflösung annehmen. Das Spezifikationsmodell wird durchgängig mit dem Gleitkommadatentyp doppelter Genauigkeit `DOUBLE` mit 64 Bit Breite simuliert. Damit wird man den geforderten Wertebereichen

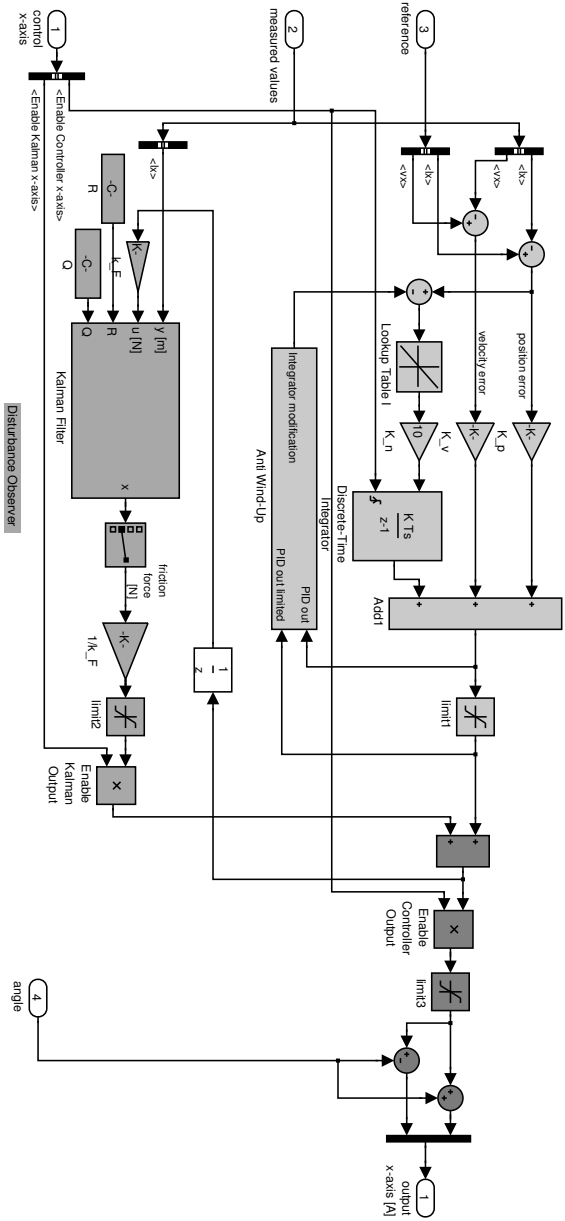


Abbildung 6.2: Detailansicht des Spezifikationsmodells der Reglerfunktionen für die x-Achse.

innerhalb der Algorithmen und damit den Anforderungen der Nanometer-Präzisionstechnik gerecht, ohne auf diesem Abstraktionsgrad den zusätzlichen Aufwand einer Fixed-Point- oder Integer-Realisierung in Kauf nehmen zu müssen.

MATLAB/Simulink erlaubt die heterogene Modellierung und Integration von Modellen mit verschiedenen Abstraktionsgraden, was hauptsächlich durch Polymorphie unterstützt wird. Funktionen und Modellblöcke arbeiten ebenso auf Skalaren wie auf Vektoren und Matrizen und handhaben verschiedene Datentypen transparent für den Entwickler, wodurch die Schaffung von komplexen Modellen möglich ist, die die eigentlichen Implementierungsdetails verbergen. Allerdings gewinnen diese Details an entscheidender Bedeutung, wenn eingebettete Implementierungen angestrebt werden, besonders, wenn es sich um Hardware-Beschreibungen handelt. Einige Funktionen, wie z.B. die Division, können nicht trivial auf binäre Logik abgebildet werden. Für die Überführung eines Simulink-Modells in eine Hardware-Implementierung via VHDL müssen die von der plattformspezifischen Modellierungsebene vorgeschriebenen Restriktionen bezüglich der verfügbaren Elemente und Konstruktionsregeln erfüllt werden. Für Simulink und die angebundenen Tools, wie der MATLAB-eigene Simulink HDL Coder [The11a] oder der FPGA-herstellerspezifische Xilinx System Generator [Xil11], bedeutet dies, dass nur eine Teilmenge der Modellelemente direkt auf VHDL und, noch spezifischer, auf synthesefähige VDHL-Konstrukte abgebildet werden kann. Alle komplexeren Funktionen müssen unter Umständen durch Ersatzkonstrukte auf einem niedrigeren, hardware-näheren Abstraktionsniveau nachgebildet werden. Für die HDL-synthetisierbare Modellierung in Simulink besteht generell die Einschränkung, für den Datenpfad auf Integer- oder Fixed-Point-Datentypen zurückzugreifen. MATLAB stellt mit der Fixed-Point Toolbox [The10] die nötigen Mittel zur Durchführung und zur simulativen Validierung der Datentyptransformation zur Verfügung. Methoden zur analytischen Bestimmung der Festkomma-Konfiguration von Reglern sind z.B. in [PS09, FCV05] zu finden. Genauere Ausführungen zu diesen beiden Aspekten der HDL-konformen Modellierung des Datenpfades, auch an Simulink-Beispielen dieses Reglersystems illustriert, wurden in [MSF12] zusammengetragen.

Eine komplette Ansicht aller Funktionsmodule der x- und y-Achsen-Regelung mit Winkelkorrektur, sowie der z-Achsen-Regelung mit x- und y-Winkelkorrekturen mit spezifizierten Festkommadatentypen ist in Abbildung A.7 im Anhang zu finden.

6.2 Komponentenorientierter plattformspezifischer Entwurf

Ausgehend von dem vorgestellten Spezifiaktionsmodell wird in diesem Abschnitt ein komponentenorientierter FPGA-plattformspezifischer Entwurf des Reglersystems erarbeitet. Die Umsetzung der Definitionen zur Komponentenstruktur für die plattformspezifische Modellierungsebene in Abschnitt 4.2.2, sowie die Designkonzepte für datenflussorientierte Komponentensysteme in Abschnitt 5.1 sorgen für eine konsistente Abbildung von Struktur und Verhalten vom plattformspezifischen Modell auf das resultierende Hardware-Design. Die Fragen, die PSM bezüglich des Verhaltens der zu generierenden eingebetteten Hardware-Lösung zu beantworten hat, betreffen:

- Abbildung der Ausführungszyklen auf der Hardware relativ zum tatsächlichen Echtzeit-Intervall, und daraus folgend
- Definition eines Zeitpunktes zur konsistenten Übernahme der Eingangswerte,
- konsistente Integration von Daten- und Steuerpfad externer Black-Box-Komponenten in die modellierte Schaltung,
- robuste Integration der generierten Schaltungen in die umgebende Schaltung oder Peripherie, durch
- Mechanismen zur konsistenten Übernahme der Eingangswerte und Übergabe der Ausgangswerte an die Chip-Umgebung.

Im Verlauf dieses Abschnitts soll daher zuerst auf die nötige Änderung der Simulationssemantik von der funktionalen Simulation auf eine transaktions- bzw. zyklusorientierte Simulation und die damit verbundenen Modellkonstrukte für Multi-Raten-Simulationen eingegangen werden, die nötig sind, um die angesprochenen Aspekte auf Modellebene ausdrücken zu können.

Bei der Definition der Komponenten ist die Entscheidung über die spätere Realisierung der Funktionen zu treffen. Generell wird zwischen der expliziten Top-Down-Modellierung der Komponente und der Bottom-Up-Referenzierung von bestehenden Implementierungskomponenten unterschieden, auf die an dieser Stelle exemplarisch eingegangen wird. Danach werden plattformspezifische Modellentwürfe von Einzelkomponenten des Regelungssystems mit entkoppelten Schnittstellen, sowie optimiertem Timing-Design vorgestellt und bewertet.

6.2.1 Transformation der Simulationssemantik

Die Änderungen im Verhalten, die ein Modell während der Transformation vom funktionalen Spezifikationmodell zum plattformspezifischen Implementierungsmodell zu durchlaufen hat, beziehen sich hauptsächlich auf eine veränderte Zeitbetrachtung.

Im Simulationsmodell des Reglungssystems beschreibt die Simulationsabtastzeit (base sample time) das Interval zwischen den Zeitpunkten, zu denen die Änderungen relevanter physikalischer Größen beobachtet werden - die Reglerabstastzeit `T_Control`. Die Stellgrößen und die Reaktion der Strecke werden „zeitlos“ zu einem Simulationsszeitpunkt berechnet. In der Realität markiert der Beginn des Intervals `T_Control` den Zeitpunkt der Erfassung der Eingangsdaten, bevor diese dem Reglermodul zugeführt werden. Die Berechnung der Stellgrößen und deren Ausgabe innerhalb einer Periode von `T_Control` ist die Bedingung der *Echtzeit-Ausführung* für ein zeitdiskretes Regelungssystem [CM05]).

Für die plattformspezifische Modellierung hingegen ist die Simulation entscheidender Phasen in der Berechnung der Reglerfunktionen auf der Ausführungsplattform nötig. Dies erfordet die Einführung einer neuen, höher aufgelösten Zeitbasis, die eine *transaktionsorientierte Simulation* bzw., mit zunehmender Verfeinerung, eine *zyklusorientierte Simulation* der Berechnungsschritte innerhalb des Intervalls `T_Control` erlaubt. Die neue Simulationszeitbasis wird mit `T_Clk` bezeichnet und muss mit

$$\texttt{T\_Clk} = \frac{1}{n} * \texttt{T\_Control}$$

mit $n \in \mathbb{N}$ ein Bruchteil des Echtzeitintervalls sein. Abbildung 6.3 illustriert die Beziehungen der Simulationen mit Reglerabtastrate (*Control simulation sample rate*) mit Zeitbasis `T_Control` und mit Ausführungstaktrate (*Cycle-oriented simulation base sample rate*) mit Zeitbasis `T_Clk`. Außerdem wird die Repräsentation des Beginns des Echtzeit-Intervalls bezüglich des Ausführungstaktes mittels eines Trigger-Signals gezeigt, von dem ausgehend die Ausführung von m Zeitschritten der Reglerfunktionalität simuliert wird.

Im Prinzip muss `T_Clk` nicht feingranularer sein, als zur Simulation der m entworfenen Berechnungsschritte nötig ist, d.h. $T\_Clk \leq 1/m * T\_Control$, $m \in \mathbb{N}$. Natürlich wird die realisierte Schaltung auf dem FPGA mit deutlich höherem Takt arbeiten, aber eine *takt-(zyklus)-akkurate Simulation* von mit mehreren MHz getakteter Logik in einem Regelungssystem mit einer Abtastrate im kHz-Bereich ist auf der modellbasierten Systementwurfsebene nicht sinnvoll. Die sukzessive Erhöhung der Auflösung von `T_Clk` mit

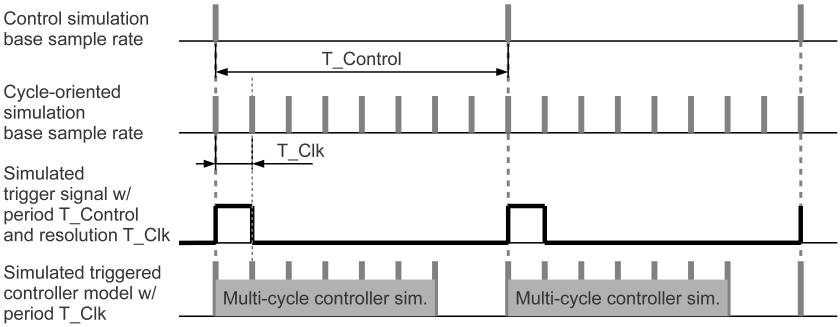


Abbildung 6.3: Vervielfachung der Simulationszeitauflösung zur Simulation der Reglerausführung innerhalb der Reglerabtastzeit.

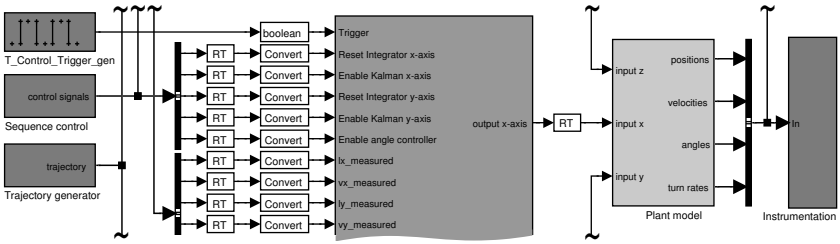


Abbildung 6.4: Ausschnitt des Systemmodells mit Modifikationen zur Multi-Rate-Simulation des Reglermodells.

zunehmender Verfeinerung von Daten- und Steuerpfad des PSM und dem Übergang zur zyklusorientierten Simulation erlaubt es, die Balance zwischen Verhaltensvalidierung Simulationsaufwand zu finden - „so akkurat wie nötig und so schnell wie möglich“.

Da jede Verringerung des Simulationszeitintervalls eine Erhöhung der Simulationskomplexität bedeutet, empfiehlt sich die Konstruktion eines *Multi-Raten-Modells (multi-rate model)*, um diesen Effekt zu minimieren. Alle abstrakten und plattformunabhängigen Partitionen des Systemmodells können damit mit der niedrigen Reglerabtastrate `T_Control` simuliert werden, während der Regler selbst als plattformspezifische Partition mit `T_Clk` simuliert wird. Die Trennung zwischen den Partitionen im Simulink-Modell erfolgt, wie in Abbildung 6.4 gezeigt ist, mittels Ratentransistionsblöcken - *Rate Transition (RT)*. Zusätzlich sind die HDL-spezifischen Datentypkonvertierungen (*Convert*) zu

erkennen. Entscheidend für die weiterführende Entwicklung des plattformspezifischen Entwurfs ist außerdem die Erzeugung des Echtzeit-Trigger-Signals (*T_Control_Trigger_gen*), das den festen Zeitpunkt zur Datenübernahme mit dem Beginn des Echtzeit-Intervalls markiert und als Grundlage für den Steuerpfad des komponentenbasierten Entwurfs dient.

6.2.2 Komponenten- und Schnittstellenentwurf

Für den Top-Down-Entwurf wird der Datenpfad der Komponente auf das Abstraktionsniveau der plattformspezifischen Modellierungsebene verfeinert, so dass die automatische Überführung aller Modellelemente in eine synthesefähige Hardware-Beschreibung möglich ist. Abbildung 6.5 zeigt die Kernlogik einer synthesefähigen PID-Regler-Komponente. Zu beachten sind dabei die Ersetzungen 1) des Look-Up-Tables für die nichtlineare I-Verstärkung (variable I-Gain) durch das in Abbildung A.6 gezeigte Ersatzkonstrukt (lineare Interpolation), sowie 2) des Resettable-Integrator-Blocks durch den diskreten Integrator mit rücksetzbarem Akkumulator, im Vergleich zur Reglerspezifikation in Abbildung 6.2. Details zu diesen Ersatzkonstrukten sind [MSF12] zu entnehmen. Generell ist festzuhalten, dass die Notwendigkeit solcher Ersatzkonstrukte vom Umfang und dem Abstraktionsgrad der verfügbaren modellbasierten Entwurfsplattform abhängt[1].

Unabhängig davon bzw. darüber hinausgehend ist die generelle Vorgehensweise zum Erstellen von robusten Komponenten, wie in den Grundlagen in Abschnitt 3.3 erläutert wurde, das Entkoppeln des komponenteninternen Designs und des Zeitverhaltens (Timing) von dem des umgebenden Designs. Dies geschieht entsprechend der in Abschnitt 4.2.2 eingeführten Komponentenstruktur durch *Argument- und Resultat-Registerstufen*[2] an den Eingangs- und Ausgangsports und den *Entwurf eines komponenteninternen Steuerpfades*, um die synchrone Datenpropagation bezüglich der Taktperiode `T_Clk` zu entwerfen. Die funktionale Integration mit anderen Komponenten und umgebenden Designs wird durch Synchronisationssignale etabliert, die hier mit `Trigger` und `Valid` bezeichnet werden. Das `Trigger`-Signal gibt den Zeitpunkt der Übernahme der Eingabedaten in die Argument-Register an, und damit, wann die neuen Werte entsprechend des inneren Timing-Regimes duch die Komponentenlogik propagiert werden. Das `Valid`-Signal wird über den internen Steuerpfad der Komponente aus dem Trigger abgeleitet. Abbildung 6.5

[1] Umfang und Leistungsfähigkeit der Simulink-eigenen Modellbibliotheken und Codegeneratoren (HDL Coder, etc.) haben sich mit den fortschreitenden Versionen von MATLAB geändert und werden auch künftig erweitert werden.

[2] Modellrepräsentation eines Registers in Simulink ist das *Unit Delay (1/z)*

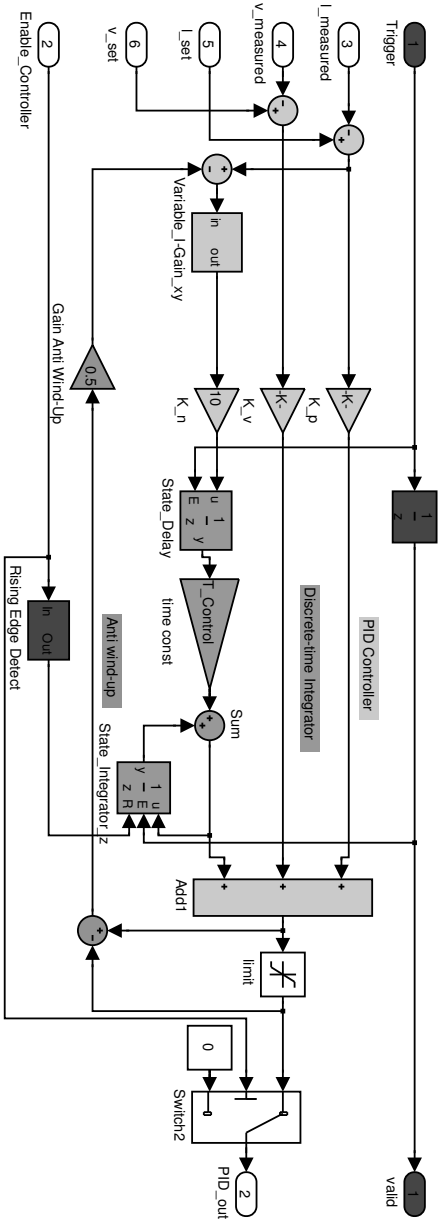


Abbildung 6.5: Modell des internen Daten- und Steuerpfades einer PID-Reglerkomponente.

zeigt den PID-Regler-Entwurf mit dem inneren Steuerpfad zur Timing-Synchronisation. Kritisches Element im Datenpfad ist der „Discrete-time Integrator (Forward-Euler)“, der ein Register im Vorwärtspfad besitzt, das zu einem definierten Zeitpunkt, genau einmal pro `T_Control`-Intervall, Werte übernehmen muss[3]. Das `Trigger`-Signal wird daraufhin lediglich um einen weiteren Takt verzögert[4], bevor der rücksetzbare Akkumulator[5] den aktuellen Wert übernehmen und das `Valid`-Signal den Zeitpunkt anzeigen kann, zu dem die aus den zum Trigger-Zeitpunkt übernommenen Eingaben erzeugten Ausgaben in die Output-Register geschrieben werden dürfen.

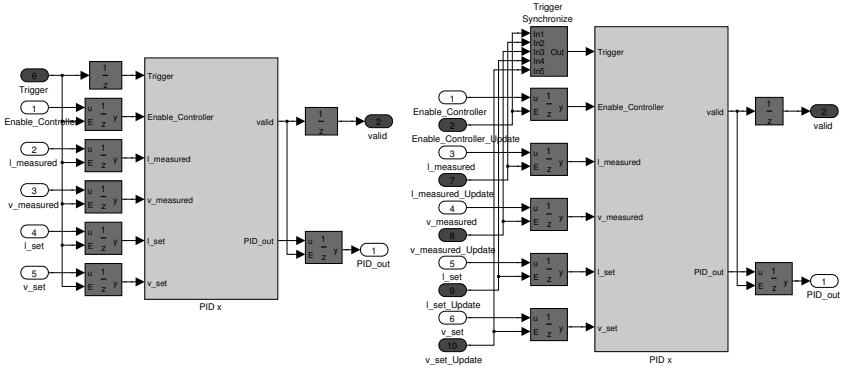


Abbildung 6.6: Schnittstellen mit Argument- und Resultat-Registerstufen - synchron getriggert (links), datenflussgesteuert (rechts).

Abbildung 6.6 zeigt dazugehörige Argument- und Resultat-Registerstufen nach den Konzepten in Abschnitt 5.1.3. Links im Bild ist eine *synchron-getriggerte Argument-Registerstufe* dargestellt, die die anliegenden Daten mit der steigenden Flanke des `Trigger`-Signals in alle Register übernimmt. Im Unterschied dazu zeigt für die rechts dargestellte *datenflussgesteuerte Argument-Registerstufe* kein `Trigger`-Signal die gleichzeitige Verfügbarkeit der Daten an den Registereingängen an. Jedes Datum verfügt dagegen über ein eigenes `Update`-Signal, das dessen Verfügbarkeit anzeigt und die Übernahme ins jeweilige Argument-Register veranlasst. Das `Trigger`-Signal für die innere Komponentenlogik wird über eine zusätzliche Trigger-Synchronisationslogik erzeugt, sobald das letzte Eingangsdatum verfügbar ist.

[3] Register mit Enable-Port = Enabled Unit Delay;
[4] synchrones Register = einfaches Unit Delay
[5] Register mit Enable-Port und Reset = Enabled Resettable Unit Delay

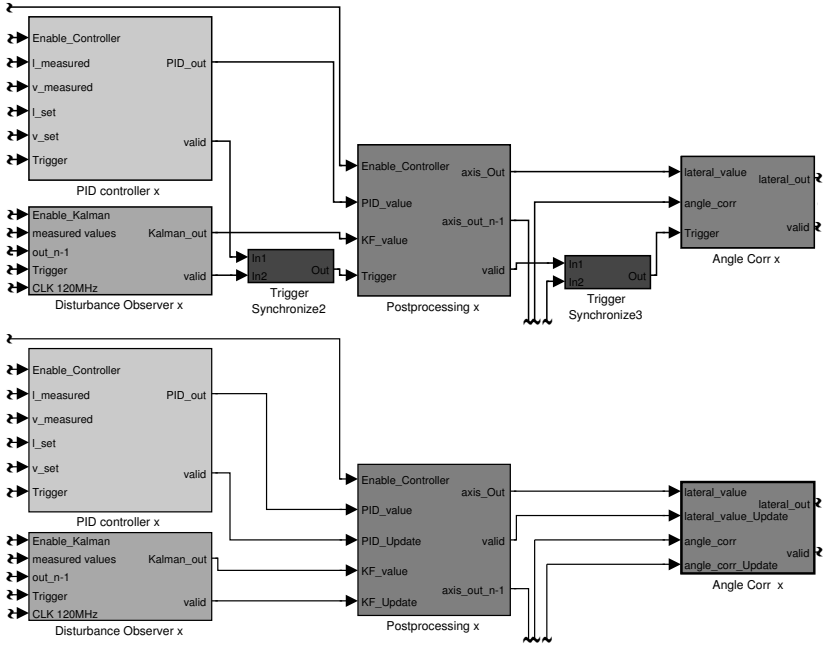


Abbildung 6.7: Beispiele des x-Achsenreglers mit synchron getriggertem (oben) und mit datenfluss-synchronisiertem Steuerpfad (unten).

Das datenflussorientierte Zusammenspiel derartig konstruierter Komponenten im Gesamtsystem ist exemplarisch in Abbildung 6.7 gezeigt. Die Ereignisse, die entscheidend für die Auslösung der jeweiligen Verarbeitung sind, sind die Flanken der `Valid`-Signale, die von den Komponenten zusammen mit den jeweiligen Daten durch das System propagiert werden. Zum Zwecke der Gegenüberstellung finden hier die zwei verschiedenen Synchronisationsmechanismen Anwendung. Der obere Teil der Abbildung zeigt die verbundenen synchron getriggerten Komponenten der x-Achsen-Regelung. Deren Trigger sind vor der zweiten und dritten Verarbeitungsstufe zu synchronisieren, da sichergestellt werden muss, dass der Komponenten-Trigger erst nach dem Anliegen des Ergebnisses mit der höchsten Latenz ausgelöst wird. Die Komponenten der zweiten und dritten Verarbeitungsstufe im unteren Teil der Abbildung verfügen über datenfluss-gesteuerte Interfaces und benötigen keine weiteren Synchronisationmechanismen auf Systemebene. Ein Gesamtmodell

des komponentenbasiert entworfenen Reglersystems ist in Abbildung A.8 im Anhang zu finden.

6.2.3 Latenzoptimierung innerhalb der Komponenten

Die maximale Taktrate, mit der eine Hardware-Komponente betrieben werden kann, hängt von ihrem längsten Pfad kombinatorischer Logik ab[6] und kann durch Einfügen zusätzlicher Register in diesen Pfad erhöht werden. Das Ziel minimaler Latenz $l = \frac{c}{f} \rightarrow min.$ (siehe Abschnitt 3.2) beschränkt allerdings die maximal sinnvolle Erhöhung der Taktrate bzw. Frequenz f durch die damit einhergehende Erhöhung der Taktanzahl c.

Abbildung 6.8 zeigt das Re-Design der PID-Reglerkomponente aus Abbildung 6.5, deren Datenpfad jetzt Registerstufen enthält. Um der erhöhten Latenz Rechnung zu tragen, muss natürlich der Synchronisationspfad auch entsprechend verzögert werden. Im Allgemeinen ist zwischen der möglichen Takterhöhung, der damit verbundenen Latenzerhöhung und auch dem erhöhten Ressourcenbedarf durch die zusätzlichen Register abzuwägen.

6.2.4 Black-Box-Komponenten

Die Bottom-Up-Referenzierung von verfügbaren Implementierungskomponente und deren Einbindung auf Modellebene geschieht mittels *Black-Box*-Modulen. Für das vorliegende Reglermodell wird das Kalman-Filter-Modul auf diese Weise realisiert. Aufgrund eines vorausgegangenen ausführlichen Komponenten-Prototypings konnte entschieden werden, dass für diese Komponente keine direkte Hardware-Implementierung realisierbar ist [MKG+13]. Daher wurde sie mittels des Floating-point-DSP-Softcores LiSARD, der am Fachgebiet Rechnerarchitektur & Eingebette Systeme der TU Ilmenau entwickelt wurde, umgesetzt [PKM+11, PKMF11a].

Der LiSARD-Softcore verfügt über ein eigenes Interface zur asynchronen Kommunikation mit der umgebenden Logik. Damit die VHDL-Komponente im Zuge der Codegenerierung korrekt instantiiert werden kann, muss dieses Interface vom Black-Box-Modul auf Modell-ebene exakt repliziert werden. Der folgende Code-Ausschnitt zeigt das VHDL-Interface der LiSARD-Kalman-Filter-Implementierungskomponente.

[6] Prototyping mit dem Simulink HDL Coder Workflow Advisor bietet die Möglichkeit, nach erfolgreicher Synthese und Mapping den kritischen Pfad im Modell anzeigen zu lassen (Back-Annotation).

Abbildung 6.8: Modell einer auf minimale Latenz optimierten PID-Reglerkomponente mit zusätzlichen Registerstufen.

```
...
-- Component Declarations
Component KalmanFilter_LISARD_BlackBox
  Port(
    clk          : in  std_logic;
    -- LiSARD-specific interface
    CLK_120MHz : out  std_logic;
    Trigger      : in  std_logic;
    Valid        : out std_logic;
    -- Kalman filter function ports
    y            : in  std_logic_vector(57 downto 0);
      -- sfix58_En55
    u            : in  std_logic_vector(58 downto 0);
      -- sfix59_En55
    KF_out       : out std_logic_vector(46 downto 0);
      -- sfix47_En44
  );
end Component;
...
```

Zur korrekten Instanziierung der VHDL-Komponente während der Codegenerierung und anschließender Syntheseprozesse ist die Schaffung eines interface-kompatiblen Modellblocks notwendig. Abbildung 6.9 zeigt das Black-Box-Modul und den Konfigurationsdialog. Deutlich zu erkennen sind die Synchronisationsignale `Trigger` und `Valid`, sowie ein zusätzlicher Clock-Port `CLK_120MHz`, über den der vom Softcore vorausgesetzte interne Takt zu Verfügung gestellt wird. Über den Konfigurationsdialog wird die Generierung eines weiteren Takteingangs `clk` verfügt, über den, zwecks Synchronisation der Datenübergabe zum Softcore, der Takt der umgebenden Logik an die LiSARD-Komponente übergeben wird. Die Verwendung von Black-Box-Komponenten bedeutet eine explizite Trennung von Simulationssemantik und Ausführungssemantik, die mit zunehmender Verfeinerung deutlicher zutage tritt, und derer man sich bei der Modellierung und Simulation bewusst sein muss. Das Innere der Kalman-Filter-Black-Box, das in Abbildung 6.10 dargestellt ist, enthält nur eine abstrakte Funktionsbeschreibung, die eine korrekte Berechnung simuliert und vom umgebenden konkreten Design abgekapselt ist. Diese ist hier als atomares Subsystem im Zentrum modelliert. Die Eingangssignale werden durch Datentypkonvertierungsblöcke an den Datentyp `DOUBLE` angepasst, der im Inneren der abstrakten Funktion benutzt wird. Die für die LiSARD-Implementierung spezifischen Signale `Trigger` und `Valid` haben für die interne Simulation der Kalman-Filter-Funktion keine Bedeutung. Sie werden jedoch benutzt, um die Übergabe der Werte in die Register

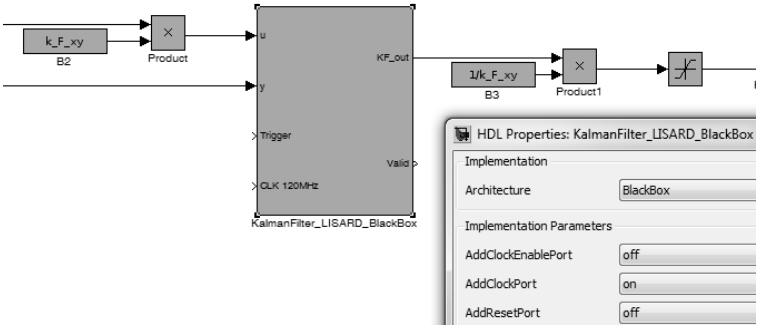


Abbildung 6.9: Deklaration des Kalman-Filter-Blocks als Black-Box.

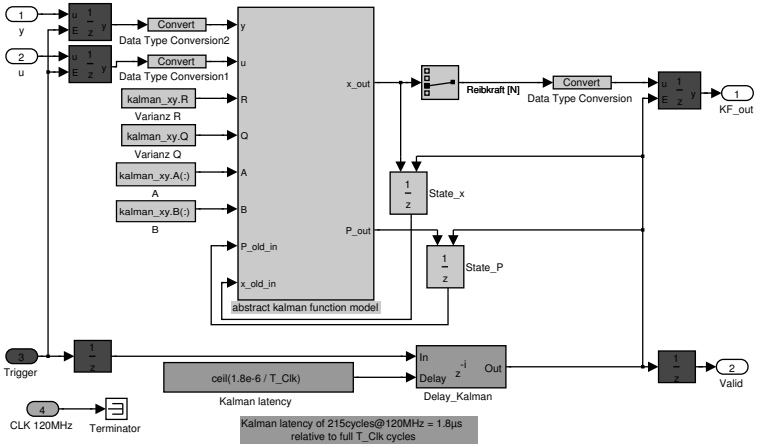


Abbildung 6.10: Interna des Black-Box-Blocks für die funktionale Simulation der Kalman-Filter-Implementierung.

auszulösen, und um mit Hilfe des abgebildeten `Variable-Delay`-Blocks ein realistisches Berechnungslatenzverhalten der Komponente zu erzeugen. Dies geschieht, indem die Latenz des Kalman-Filter-Algorithmus auf dem LiSARD-Core mit der Formel

$$Delay_{Kalman} = \left\lceil \frac{Latency_{KFonLiSARD}}{T\_Clk} \right\rceil$$

auf volle Simulationszeitintervalle mit der Basis `T_Clk` abgebildet wird. Für $\texttt{T\_Clk} \geq Latency_{KFonLiSARD}$ wird die Kalman-Filter-Funktion der Verarbeitungskette mindestens einen Takt Verzögerung hinzufügen, für zunehmend kleinere `T_Clk` jeweils ein Vielfaches dieser Periode. Für eine definierte Simulationsperiode $\texttt{T\_Clk} = 1/f_{clk}$, die der der Hardware-Clock entsprechen würde, könnte die während der IP-Core-Entwicklung [MKG+13] bestimmte Latenz von 215 Zyklen taktgenau simuliert werden, was diese Black-Box auch für *Cycle-accurate simulation* verwendbar macht.

6.2.5 Ergebnisse des komponentenbasierten Entwurfs

Der Entwurf vollständig I/O-gepufferter Komponenten macht die isolierte prototypische Synthese der Komponenten möglich und erlaubt daher die Gewinnung von spezifischen Ressourcen- und Timing-Informationen, mit denen Systementwurfsentscheidungen hinsichtlich optimaler Implementierungsstrukturen getroffen werden können.

Tabelle 6.1 zeigt die Ergebnisse des Prototypings für die Komponenten des im Anhang in Abbildung A.8 dargestellten Gesamtmodells. Die Synthese wurde für den auf der GECKO3-Plattform verbauten Xilinx Spartan3-4000 durchgeführt. Für die Komponenten wurde VHDL-Code mittels des MATLAB 2011a Simulink HDL Coder generiert und mit Xilinx ISE 13.4 synthetisiert. Der jeweilige Ressourcenbedarf und die Frequenzen sind Ergebnisse des Syntheseprozesses[7], während die Latenzen durch Verhaltenssimulation bestimmt wurden. Wie der Tabelle zu entnehmen ist, können mit den in Abbildung 6.8 illustrierten Änderungen die Frequenzen für die jeweiligen PID-Regler um ca. Faktor 3 gesteigert werden, was trotz gleichzeitiger Verdopplung der benötigten Ausführungstakte um Faktor 2 eine Reduktion der absoluten Ausführungszeit ein Drittel bedeutet. Im Falle der Decoupling-Komponente kann eine Latenzverringerung um 20% erreicht werden. Offensichtlich ist, dass alle derartigen Änderungen mit einem erhöhten Bedarf an Flip-Flops einhergehen, die sich allerdings nicht entscheidend auf den Gesamtressourcenbedarf der Komponenten auswirken.

Mit den vorgestellten Reglerkomponenten ist es nun möglich, bereits auf Modellebene Systeme zu erstellen, die auch auf der Hardware ein definiertes Ein- und Ausgabeverhalten mit vorhersagbarer Latenz haben. Die Änderungen im Simulationsparadigma hin zur Multi-Raten-Simulation mit einer Periode,

[7] Frequenz und Takte der Ausführung des Kalman-Filter-Algorithmus auf dem LiSARD-Softcore sind in Klammern angegeben; zusätzlich benötigt jeder Kalman-Filter noch 6% BRAM-Ressourcen für den Programmspeicher des LiSARD-Core

die sich an der Latenz der Logik orientiert, stellen eine unabdingbare Voraussetzung dar.

Komponente	Logik-Ressourcen (FF/LUTs/Slices)	DSP-Ress.	Freq. max. [MHz]	Latenz [cyc.]	Lat. min. [ns]
PIDs					
x+y–Achsen	312/1481/2%	18%	12,4	3	242
lat.–opt.	799/1459/3%	18%	37,8	6	159
z–Achse	304/847/1%	8%	22	3	134
lat.–opt.	810/765/2%	8%	71,3	6	84
x+y–Winkel	254/616/1%	10%	22,7	3	132
lat.–opt.	699/736/2%	10%	66,0	6	91
z–Winkel	266/1228/2%	20%	12,8	3	234
lat.–opt.	746/1332/3%	20%	37,5	6	160
Störbeobachter (Kalman-BB.)					
x+y–Achsen	8387/10867/30%	–	33,9(80)	3(215)	2776
z–Achse	8366/10548/29%	–	36,5(80)	3(215)	2770
Postprocessing					
x+y	229/141/< 1%	–	87,5	2	23
z	148/57/< 1%	–	164,3	2	12
Angle Corr. x+y	131/104/< 1%	–	164,0	2	12
Decoupling z	192/2098/3%	34%	24,7	2	81
lat.–opt.	353/2112/3%	34%	44,5	3	67

Tabelle 6.1: Ergebnisse der Rapid-Prototyping-Synthese der Komponenten des Reglersystems - vor und nach der Latenzoptmierung (lat.-opt.).

Die Latenz des längsten kritischen Pfades wird bestimmt durch die maximale Latenz der kritischen Pfade von z-Achsen-Reglung und x-Achsen-Regelung $max(l_{critPath\_z}, l_{critPath\_x})$[8] mit

$$\begin{aligned} l_{critPath\_z} =& max(max(l_{Kalman\_z}, l_{PID\_z}) + l_{Postproc\_z}, l_{PID\_angleX}) \\ &+ l_{Decoupl\_z} \\ l_{critPath\_x} =& max(max(l_{Kalman\_x}, l_{PID\_x}) + l_{Postproc\_x}, l_{PID\_angleZ}) \\ &+ l_{AngleCorr\_x} \end{aligned}$$

[8] die x- und y-Achsen-Regelungen sind identisch, ebenso die x- und y-Winkelregler.

Ausgehend von dem Gesamtmodell in Abbildung A.8, das ein System bestehend aus den in Tabelle 6.1 charakterisierten Komponenten beschreibt, lassen sich die in Tabelle 6.2 zusammengefassten prototypischen Aussagen über die resultierenden FPGA-Designs des Gesamtreglersystems treffen.

Entwurf	Logik-Res. (FF/LUTs/ Slices)	DSP-Ress.	BRAM	Freq. max. [MHz]	Latenz [cyc.]	Lat. min. [ns]
I/O-Reg., sync.trig.	28310/43535/ 115%	92%	18%	11,8(80)	8(215)	3365
lat.-opt. sync.trig.	30846/41310/ 112%	90%	18%	31,6(80)	11(215)	3035
lat.-opt. df-gest.	30796/41311/ 112%	90%	18%	33,9(80)	9(215)	2952

Tabelle 6.2: Prototyping-Syntheseergebnisse für den komponentenbasierten Entwurf des Gesamtreglersystems

Die Latenz des kritischen Pfades des latenzoptimierten Systems verringert sich um ca. 16% gegenüber dem ursprünglichen Komponentenentwurf. Die Latenz der Kalman-Filter ist dominant und kompensiert die positiven Effekte der dazu parallel ausgeführten PID-Regler-Komponenten, deren Optimierung die Steigerung der Ausführungsfrequenz des Gesamtsystems um nahezu Faktor 3 ermöglichte. Es ist zu erkennen, dass durch die in diesem Fall überflüssige zusätzliche Trigger-Synchronisationslogik der kritische Pfad im System mit datenflussgesteuert integrierten Komponenten um 2 Takte verkürzt werden konnte. Zu bemerken ist außerdem, dass die Syntheseergebnisse der Gesamtsysteme nicht den aufsummierten Summen der Einzelkomponenten entsprechen. Dies liegt an den Optimierungsverfahren der unteren Implementierungsschritte, die u.a. dafür sorgen, dass, sofern nicht anderweitig erzwungen, Funktionen, die nicht mehr verfügbare DSP-Ressourcen belegen würden, mittels FFs/LUTs realisiert werden. Die Ergebnisse verdeutlichen dennoch, dass der Ressourcenbedarf des Gesamtentwurfs die Ressourcen eines der verfügbaren FPGAs überschreiten würde und motivieren daher weitergehende Fragestellungen nach der Anwendung erweiterter Methoden des Resource-Sharings zur Verringerung des Ressourcenbedarfs nicht zeitkritischer Teile des Entwurfs bzw. der Verteilung auf Multi-FPGA-Plattformen.

Wie außerdem deutlich wurde, hängt die Leistungsfähigkeit von top-down-entworfenen Komponenten sehr vom hardware-orientierten Entwurf des internen Timings ab und erfordert daher zusätzlichen Modellierungsaufwand

und den Transfer von Know-how von der Hardware-Beschreibungsebene auf die plattformspezifische Modellierungsebene. Gegen Ende dieser Arbeiten verfügbare Tools, wie Simulink HDL Coder 2012, bieten an dieser Stelle jedoch bereits automatisierte Unterstützung für das Design des komponenteninternen Datenpfades.

6.3 Komponentenbasiertes Resource-Sharing im plattformspezifischen Entwurf

Wenn die Minimierung des Ressourcenbedarfs ein Design-Ziel ist, dann ist es möglich, den Systementwurf bereits in frühen Phasen des Entwicklungsprozesses darauf auszurichten. In diesem Fall heißt das, bereits auf Modellebene Transformationen vorzunehmen, die das Ziel verfolgen, möglichst wenig strukturelle Redundanz im resultierenden FPGA-Design zu erzeugen. Im Gesamtkomponentenmodell des Reglersystems in Abbildung A.8 fallen eine Vielzahl identischer oder ähnlicher Komponenten auf. Diese entsprechen den Funktionsmodulen des Reglerentwurfs, die nach dem *Computing-in-Space*-Paradigma 1:1 auf Hardware-Komponenten abgebildet wurden, um nebeneinander auf dem FPGA implementiert zu werden. Ziel dieses Abschnitts ist nun die Erstellung von „Resource-Sharing"-Komponenten, die eine n:1-Abbildung von Berechnung und Berechnungslogik nach dem *Computing-in-Time*-Paradigma herstellen. Dazu ist die Charakterisierung der statischen Kernlogik nötig, sowie die Erzeugung eines neuen Steuerpfades, der die Korrektheit der zeitlich gestaffelten Ausführung der verschiedenen Funktionsläufe (Zeitmultiplexing) sicherstellt. Dies geschieht mittels der in Abschnitt 5.2 vorgestellten KALI-Methode, deren quantitativen Bewertung in Sachen Ressourcenminimierung und Reduktion des Modellierungsaufwands anhand repräsentativer Komponenten des Reglersystemmodells im Folgenden behandelt wird.

6.3.1 Resource-Sharing für atomare Operatoren

Die Anwendung der KALI-Methode zur Generierung von Zeitmultiplex-Strukturen ist nicht nur auf Komponenten komplexerer Logik beschränkt, sondern kann ebenso auf elmentarere Operatoren bzw. Modellblöcke angewandt werden, die auf der Hardware jedoch beträchtliche Ressourcen benötigen würden. Ein gutes Beispiel hierfür ist die Matrix-Multiplikation, die in der Decoupling-Komponente der z-Achsen-Regelung benötigt wird. Insgesamt 9 Multiplikationen und 3 Additionen müssen durchgeführt werden, die sich auf

verschiedene Ressourcenkombinationen abbilden lassen. Die Syntheseergebnisse sind in Tabelle A.2 aufgeführt und werden im Diagramm in Abbildung 6.11 graphisch veranschaulicht.

Zu erkennen ist der generelle Trend der Verringerung der benötigten DSP-Ressourcen durch die Multiplizierer. Die Zusammenfassung der Addierer bringt hingegen nach Gleichung 5.1 keine wesentlichen Vorteile - durch den zusätzlichen Steuerpfad erhöht sich die Anzahl der benötigten Flip-Flops und LUTs sogar. Das führt dazu, dass ein Ressourcenminimum bei der Variante mit 9-facher Zusammenfassung der Multiplikationen auf einen Multiplikator und mit paralleler Berechnung der Additionen zu beobachten ist. Die letzte Variante mit maximal möglichem Resource-Sharing weist dagegen eine Kompensation der Resourcenzusammenfassung durch die zusätzliche generierte KALI-Logik und maximale Latenz auf und stellt damit nicht das erreichbare Optimum dar.

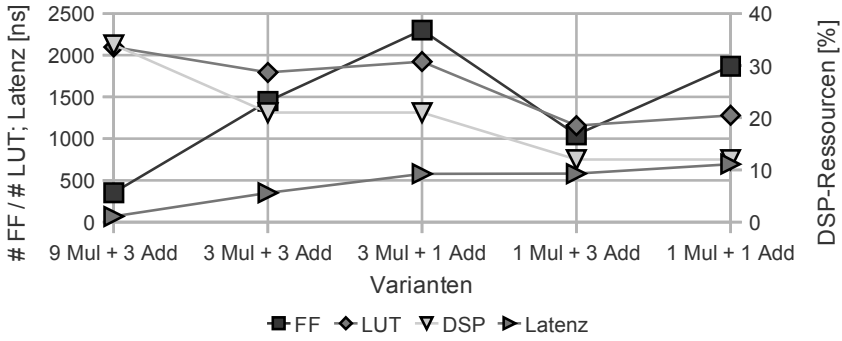


Abbildung 6.11: Syntheseergebnisse des zeitmultiplexten Entwurfs der Decoupling-Komponente mit Matrixmultiplikation - grafische Übersicht.

6.3.2 Resource-Sharing für generische parametrisierbare Komponenten

Die Anwendung der automatisierten KALI-Methode für komplexe Komponenten erfordert, wie in Abschnitt 5.2 erarbeitet wurde, die Charakterisierung der kontextspezifischen Elemente der zu überlagernden Komponenten und die Unterteilung in:

- wiederverwendbare statische Kernlogik,
- kontextspezifische Argument- und Resultat-Mengen,
- kontextspezifische Zustandsspeicherung für zustandsbehaftete Komponenten - *States* - und
- kontextspezifische Funktionslogik - *Modes*.

Ziel des Resource-Sharing ist es, eine möglichst maximale Ressourceneinsparung auf der Chip-Ebene zu erreichen. Dies erfordert einen möglichst großen bzw. komplexen Anteil statischer Kernlogik, und, wenn überhaupt nötig, möglichst kleine Modi.

Im vorgestellten Regelungssystem sind die PID-Regler-Komponenten prädestiniert für die Anwendung der Resource-Sharing-Methode. Eine strukturelle Analyse der PID-Regler-Modelle ergibt, dass sie sich ausschließlich in ihren Reglerverstärkungen unterscheiden, das heißt, in den `Gain`-Blöcken für K_p, K_v und K_n. Für die Regler der x-y-Ebene kommt noch die nicht-lineare Verstärkungskennlinie für den I-Anteil hinzu.

Abbildung 6.12 zeigt den Entwurf einer generischen, parametrisierbaren PID-Kernlogik mit den zusammengefassten Status- und Modus-Blöcken. Die Besonderheit hier ist, dass die `Gain`-Blöcke der Reglerverstärkungen (Multiplikation mit Konstanten) durch explizite Multiplikationen ersetzt wurden, denen die Reglerverstärkungskoeffizienten als zweites Argument zugeführt werden. Auf diese Weise können die Koeffizienten zu sehr kleinen *Modi* zusammengefasst werden. Im Modell sind die entsprechenden Blöcke mit dem Präfix `Mode_` gekennzeichnet, sodass sie der automatisierten Modellsynthese unterzogen werden können. Die ressourcenintensiven Multiplizierer bleiben somit Teil der wiederverwendeten Kernlogik.

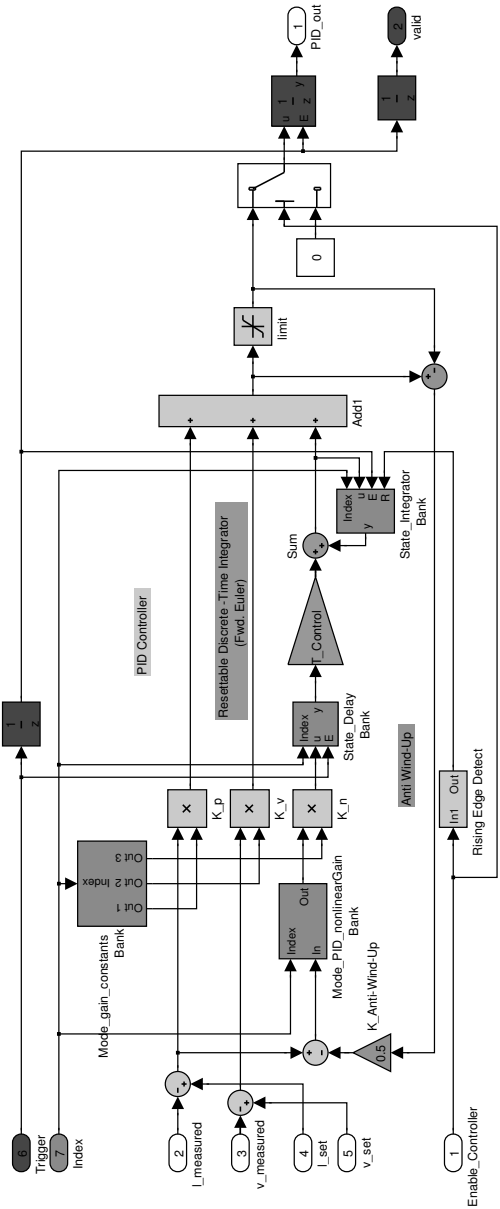


Abbildung 6.12: Generalisierte PID-Regler-Komponente mit Parametrisierung über Zustandsspeicher und modusabhängige Module für Regler-Konstanten.

Skalierung des Resource-Sharing

Das Skalierungsverhalten des Ressourcenbedarfs von Resource-Sharing-Komponenten wird im folgenden anhand des z-Achsen-PID-Reglers illustriert. Abbildung 6.13(links) stellt den steigenden Resourcenbedarf einer n-fach parallelen Implementierung (parallel PID) der einer entsprechenden resource-shared Implementierung (RS-PID) gegenüber. Zu beobachten ist, wie in Abschnitt 5.2.3 formuliert, das Ansteigen der Ressourcen der Resource-Sharing-Komponenten aufgrund der wachsenden Multiplexer-Datenpfad- und Steuerlogik zusätzlich zur konstant bleibenden Größe der Kernlogik. Dennoch ist dieser Anstieg deutlich geringer, was den Einsatz des Resource-Sharing-Verfahrens für diese Komponente nach Gleichung 5.1 als lohnenswert einstuft. Das Ende der y-Achse markiert mit ca. 55000 LUTs/FFs die verfügbaren Ressourcen auf dem Spartan-3-4000-FPGA, wodurch deutlich wird, dass für die Beschränkungen dieser Ziel-Hardware die Anzahl potentiell zu implementierender PID-Funktionen per resource-shared Komponente um mindestens zwei Zweierpotenzen gegenüber der parallelen Implementierung gesteigert werden kann. Allerdings wächst die Latenz der sequentiellen Ausführung proportional mit der Vielfachheit der Überlagerung, sodass die Latenz schnell zum limitierenden Entwurfsfaktor werden kann.

Abbildung 6.13(rechts) zeigt das Verhältnis des jeweils auf die Anzahl der Komponenten/Kontexte normierten relativen Ressourcenbedarfs für die Ausführung einer PID-Funktionalität. Für die flache parallele Implementierung bleibt dieser quasi konstant. Für die resource-shared Implementierungen nimmt er asymptotisch bis auf das Niveau, das den Anstieg des Multiplexer-Overheads in den Resource-Sharing-Komponenten charakterisiert, ab.

Wie in Anhang A.1 dokumentiert ist, verhält sich das Anwachsen des Modellierungsaufwands in Simulink ebenso linear. Der aus Tabelle A.3 abgeleitete Anteil der Modellblöcke, die für den Multiplexing-Overhead im plattformspezifischen Modell der jeweiligen Resource-Sharing-Komponenten benötigt werden, ist in Diagramm 6.14 in Form der roten Linie abgetragen.

Dem gegenübergestellt ist der Anteil des tatsächlichen Ressourcenbedarfs der Multiplexing-Logik auf der Hardware. Es ist zu erkennen, dass der Modellierungsaufwand für diese Logik anteilmäßig sogar größer ist, als ihr Anteil an der Nutzung der Chip-Fläche, was den überproportionalen plattformspezifischen Entwurfsaufwand dieser Logik auf der plattformspezifischen Modellierungsebene verdeutlicht. Dies wiederum unterstreicht den Wert einer automatischen Modellgenerierung, wie sie durch die implementierte KALI-Methode umgesetzt wird.

Abbildung 6.13: Skalierung des Ressourcenbedarfs und der Latenz von parallel implementierten und resource-shared PID-Komponenten (links) und normierter relativer Ressourcenbedarf eines PID-Kontextes (rechts).

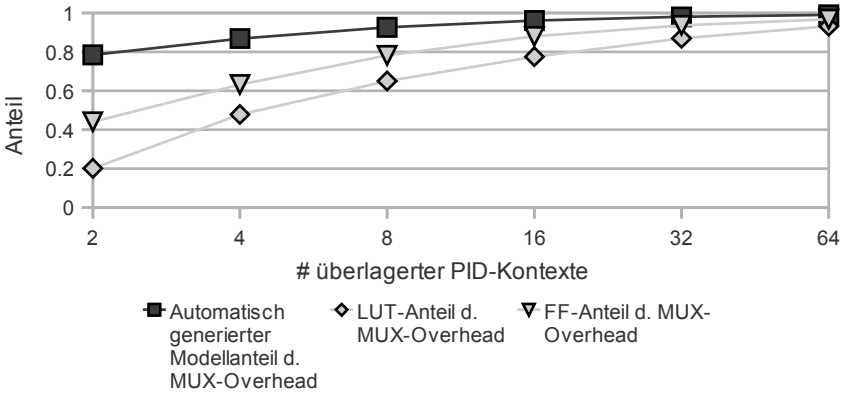


Abbildung 6.14: Vergleich der Anteile der automatisch generierten Multiplexer-Logik an der Modellkomplexität und am Ressourcenbedarf der resource-shared PID-Komponenten.

Anwendung auf die PID-Regler des Gesamtmodells

Die folgenden Ergebnisse illustrierenden die Effektivität des Ressourcen-Multiplexings am konkreten Anwendungsbeispiel. Dazu wurden die sechs PID-Regler getrennt vom Regelungsgesamtsystem synthetisiert.

Die nach der Generierung durch KALI entstandenen Interna der beiden Modus-Blöcke `Mode_gain_constants` und `Mode_PID_nonlinearGain` sind in Abbildung A.1 dargestellt und weisen die typischen, über ein Index-Signal gesteuerten Multiplexer-Strukturen auf.

Zu beachten ist an dieser Stelle, dass sich die Fixed-Point-Datentypen des generischen PID-Entwurfs deutlich von denen der originalen PID-Funktionsblöcke unterscheiden. Das liegt daran, dass die generische Kernlogik der Komponente allen Datenbreiten aller Kontexte gerecht werden muss. Hervorgehoben ist dieser Aspekt an den Schnittstellen der Modus-Blöcke in Abbildung A.1, bei denen deutlich wird, wie sich die Überlagerung der Fixed-Point-Typen in Sachen Datenbreite und Präzision auswirkt.

In der Tabelle 6.3 sind die Syntheseergebnisse für den auf der GECKO3STACK-Plattform verbauten Spartan3-4000-FPGA aufgeführt. Dem ursprünglichen komplett parallelen Entwurf aller sechs PID-Regler-Komponenten sind zwei Überlagerungsszenarien gegenübergestellt. Erstens ist die Überlagerung der jeweils zwei identischen PID-Regler für x- und y-Achsen,

sowie x- und y-Winkel aufgeführt, die Resource-Sharing-Komponenten ohne Modi ermöglicht. Zweitens wurde die Überlagerung aller sechs PID-Kontexte zur maximalen Wiederwendung der Multiplikatoren mit Modi, die die unterschiedlichen Verstärkungskoeffizienten und die nichtlineare Verstärkungskennlinie enthalten, analysiert.

PID-Komponenten	Logik-Ressourcen (FF/LUTs/Slices)	DSP-Ress.	Freq. max. [MHz]	Latenz [cyc.]	Latenz theo. [ns]
6-fach parallel (kein Mux.)	4552/6487/15%	84%	37,5	6	160
2 einfach + 2 2-fach mux.	3890/5157/13%	66%	36,1	16	443
6-fach mux. (Const-Modi)	2207/3252/9%	32%	34,1	41	1202

Tabelle 6.3: Syntheseergebnisse der zeitmultiplexten PID-Regler-Entwürfe für das Gesamtregelungssystem.

Die DSP-Ressourcen sind für die Reglerkomponenten die dominanten Anteile, da die Reglerverstärkungen, die die Hauptkomplexität der Komponente ausmachen, auf die speziellen DSP-Multiplizierer des FPGA abgebildet werden. Wie erwartet ist im Falle des Multiplexings der paarweise identischen Komponenten eine Reduktion zu erkennen, die bei den DSP-Ressourcen nahezu ein Drittel beträgt. Die Zahl der Look-up-Tables, mit denen u.a. die restliche Kernlogik realisiert wird, reduziert sich ebenfalls, allerdings nicht in diesem Maße. Da zusätzliche Multiplexerlogik und Register um die Kernlogik generiert werden mussten, steigt die Anzahl der benötigten Flip-Flops und auch, anteilig, die Zahl der LUTs. Deutlicher wird dieser Effekt bei der sechsfachen PID-Multiplex-Komponente. Hier kompensiert die steigende Komplexität der Multiplexer-Schaltungen wesentlich deutlicher die Reduzierung der Kernlogik. Einzig in den DSP-Ressourcen ist eine weitere Verbesserung zu erkennen. Dass diese wiederum nicht linear ausfällt, liegt an der steigenden Bitbreite der benötigten Multiplizierer in der generischen Kernlogik. Durch die Überlagerung der Bitbreiten der Modi und States, wie in Abbildung A.1 illustriert, sind die Bitbreiten der Multiplizierer für die Reglerverstärkungen größer als die der ursprünglichen Multiplizierer.

6.3.3 Ergebnisse für Resource-Sharing-Entwürfe

Die Ergebnisse der Synthese für zwei charakterisische Realisierungen des Trajektorienfolgereglersystems werden in Tabelle 6.4 zusammengefasst und dem ursprünglichen in Abbildung A.8 dargestellten komponentenbasierten Entwurf als Referenz gegenübergestellt. Erstens wurde eine Systemvariante entworfen,

Regler-system	Logik-Ress. (FF/LUTs/ Slices)	DSP-Res.	BRAM	Freq. max. [MHz]	Lat. [cyc.]	Latenz @25MHz [μs]
voll-parallel	30214/41759/ 115%	92%	18%	33,9(80)	227	3.2
Ident. Komp. Sharing	22965/28016/ 79%	78%	12%	33,9(80)	455	6.4
max. Komp.-Sharing	14202/16846/ 49%	42%	6%	31,3(80)	667	9.1

Tabelle 6.4: Syntheseergebnisse der Gesamtreglersysteme mit resource-shared Komponenten.

bei der strukturell identische Komponenten überlagert wurden, sodass auf Modus-Logik verzichtet werden konnte. Die Ressourceneinsparungen relativ zum Referenzentwurf sind mit ca. einem Drittel zu beziffern. Die weniger starke Reduktion der DSP-Ressourcen ist darauf zurückzuführen, dass die Zahl der PID-Reglerkomponenten nur um ein Drittel reduziert wurde. Die Latenz dieses Entwurfes ist um Faktor zwei erhöht. Das zweite analysierte Szenario mit maximal möglicher Anwendung des Ressource-Sharing zeichnet sich durch die Überlagerung aller 6 PID-Komponenten mit entsprechender Einführung der Modus-Logiken, die Überlagerung aller Störbeobachter zu einer 3-fach sequentiellen Kalman-Filter-Ausführung auf einem LiSARD-Core, und dem Einfügen der ressourcenminimalen Decoupling-Komponente aus. Zu bemerken ist, dass die durch die Kalman-Filter dominierte Latenz des kritischen Pfades mehr als verdreifacht wurde. Eine weitere Reduktion der benötigten DSP-Ressourcen kann beobachtet werden. Dass diese unterproportional geringer ausfällt, liegt an der bereits zuvor diskutierten Erhöhung der Bitbreiten der Multiplikationen in der generischen PID-Regler-Kernlogik. Zusammenfassend lässt sich sagen, dass mittels der Methode zur Generierung

von resource-shared Komponenten ein auf der Ziel-Hardware realisierbares plattformspezifisches Modell des Reglersystems erzeugt werden kann. Die massive Nutzung der Komponentenüberlagerung hat ebenfalls repräsentative Resultate gezeigt. Allerdings ist für zunehmende Komponentenüberlagerungen die abnehmende Effektivität durch den größer werdenden Anteil an Steuer- und Verbindungslogik zu beachten, ebenso wie die daraus resultierende überproportional ansteigende sequentielle Ausführungszeit. Ein bewusster Einsatz an entscheidenden Stellen dient dem Erreichen eines optimalen Time-Space-Trade-offs für das zu realisierende System.

6.4 Generierung von Intermodulkommunikationslogik im verteilten plattformspezifischen Entwurf

Wie in den vorangegangen Abschnitten zur experimentellen Untersuchung des Reglersystems und seiner plattformspezifischen Modelle zur FPGA-basierten Implementierung deutlich wurde, ist ein latenzminimaler Systementwurf zu ressourcenintensiv für einen FPGA der Ziel-Hardware, während die möglichen Ressourceneinsparungen mit hohem Latenzzuwachs zu erkaufen sind. Eine Erweiterung des Entwurfsraumes und damit der Möglichkeiten, effiziente Implementierungsvarianten zu finden, ergibt sich aus der künstlichen Vergrößerung der nutzbaren Chip-Ressourcen durch Verteilung der Anwendung auf mehrere FPGAs.

Dieser Abschnitt beschäftigt sich mit der Erstellung des hardware-orientierten Modells eines Multi-FPGA-Designs unter den gleichbleibenden Voraussetzungen der geschlossenen simulativen Validierung im abstrakten Umgebungsmodell. Die Modelltransformationen, ausgehend von einem synthetisierbaren Komponentenmodell des Gesamtregelungssystems, beinhalten eine Partitionierung in auf verschiedene FPGAs abzubildende Teilsysteme, die Einbindung eines Simulationsmodells der Off-Chip-Kommunikation und die Synthese der applikationsspezifischen Kommunikationslogik. Das in Abschnitt 4.1.2 vorgestellte Konzept SHIVA zur shared-memory-basierten Intermodulkommunikation und die darauf zugeschnittene, in Abschnitt 5.3 vorgestellte High-Level-Synthese-Methode VISHNU werden im Folgenden in ihrer Anwendung auf die plattformspezifische Modellierung eines ausgewählten Multi-FPGA-Entwurfs des Reglersystems demonstriert und bewertet.

6.4.1 Partitioniertes Modell

Die Partitionierung des Gesamtreglersystems, die hier exemplarisch verwendet werden soll, beruht auf folgenden Maßgaben:

- je ein Hardware-Modul steht für jede Achsregelung mit lokaler I/O zur spezifischen Erfassung der Messwerte und Ausgabe der Stellgrößen für die jeweilige Achse zur Verfügung,
- die kritischen Ausführungspfade werden zur Latenzminimierung auf je einem Modul untergebracht, die PID-Komponenten entprechend des Lokalitätsprinzips zur Minimierung der Off-Chip-Kommunikation den Partitionen zugeordnet,
- die Steuersignale und Referenzwerte werden auf einer externen Partition generiert und den Reglerpartitionen über die Shared-Memory-Schnittstelle zugeführt.

Da dieses Szenario der Demonstration der VISHNU-Methode dient, wird hier auf die Umsetzung weiterer Entwurfsentscheidungen verzichtet und dahingehend auf Anschnitt 6.5 verwiesen.

Das Innere einer Applikationspartition am Beispiel der für den kritischen Pfad der x-Achsen-Regelung ist in Abbildung 6.15 dargestellt. Die Messwerterfassung von `lx_meas` und `lx_meas`, ebenso wie die Ausgabe der Stellgröße `output x-axis` sind physisch dieser Partition zugeordnet, daher sind diese Signale mit `IO` gekennzeichnet. Alle anderen Eingangswerte werden über die Off-Chip-Kommunikation zur Verfügung gestellt. Da das Zeitverhalten dieser Kommunikationsverbindung nicht explizit bekannt ist, sind alle Komponenten, die über externe Datenabhängigkeiten verfügen, mit einem Datenfluss-Interface ausgestattet, sodass sie sich selbst auf Basis der Datenverfügbarkeit triggern können. Im Falle des Störbeobachters wird die Datenfluss-Synchronisation extern durchgeführt.

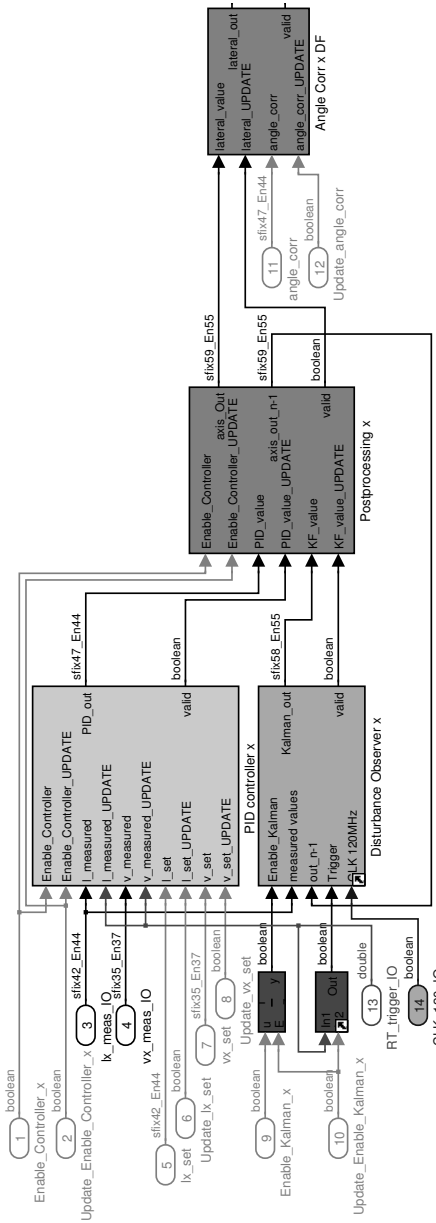


Abbildung 6.15: Interna der Applikationspartition für den x-Achsen-Regler.

6.4.2 Multi-Chip-Struktur-Modell mit Intermodulkommunikation

Für dieses Szenario kommen die in Simulink implementierten Modellgenerierungsmechanismen *VISHNU-for-SHIVA* zum Einsatz, die in Abschnitt 5.3 vorgestellt wurden. Damit wird ein plattformspezifisches Modell von verteilten FPGA-Designs auf der Shared-Memory-Kommunikationsinfrastruktur SHIVA geschaffen, das auf der Multi-FPGA-Hardware GECKO3STACK implementiert werden kann. Da die Interna der eigentlichen VISHNU-Blöcke automatisch generiert werden, sollen sie hier nicht im Fokus der Betrachtungen stehen, die Mechanismen wurden ebenfalls bereits in Abschnitt 5.3 erläutert. Stattdessen wird die Modellrestrukturierung während der Anwendung dieser Methode in den Vordergrund der Betrachtungen gerückt. Folgende strukturelle Änderungen werden während der Transformation im Gesamtmodell vorgenommen:

- Erzeugung von applikationsspezifischen Strukturpartitionen (`APPLICATIONPSECIFIC ...`) für jedes FPGA-Modul,
- Platzierung der jeweiligen Applikationspartition (`APP_...`) und Generierung des spezifischen Zugriffslogik-Moduls (`VISHNU ...`) innerhalb der applikationsspezifischen Partitionen und
- Einbindung des abstrakten Modells der Intermodulkommunikation zur Wiederherstellung eines simulierbaren Gesamtmodells.

Nach der Transformation spiegelt das plattformspezifische Modell nicht mehr die funktionale Komponentenstruktur, sondern die Struktur der verteilten FPGA-Implementierung wider. Der Ausschnitt des resultierenden Gesamtmodells in Abbildung 6.16 (das Gesamtmodell ist in Abbildung A.9 im Anhang abgebildet) zeigt die applikationsspezifischen Module, die mittels des Simulink HDL Coder direkt in HDL-Beschreibungen für die jeweiligen FPGA-Entwürfe zu überführen sind. Sie enthalten die Applikationspartitionen und die jeweils spezifisch generierten VISHNU-for-SHIVA-Logikblöcke (siehe Abbildung 6.17). Zum Zwecke der Validierung der Gesamtapplikation im Rahmen des Regelungsumgebungsmodells ist auch die Simulation der Intermodulkommunikation im verteilten FPGA-System nötig. Dafür fügt die VISHNU-for-SHIVA-Generierung ein taktgenaues Modell des Shared-Memory-Interfaces SHIVA für jede FPGA-Partition ins Gesamtmodell ein.

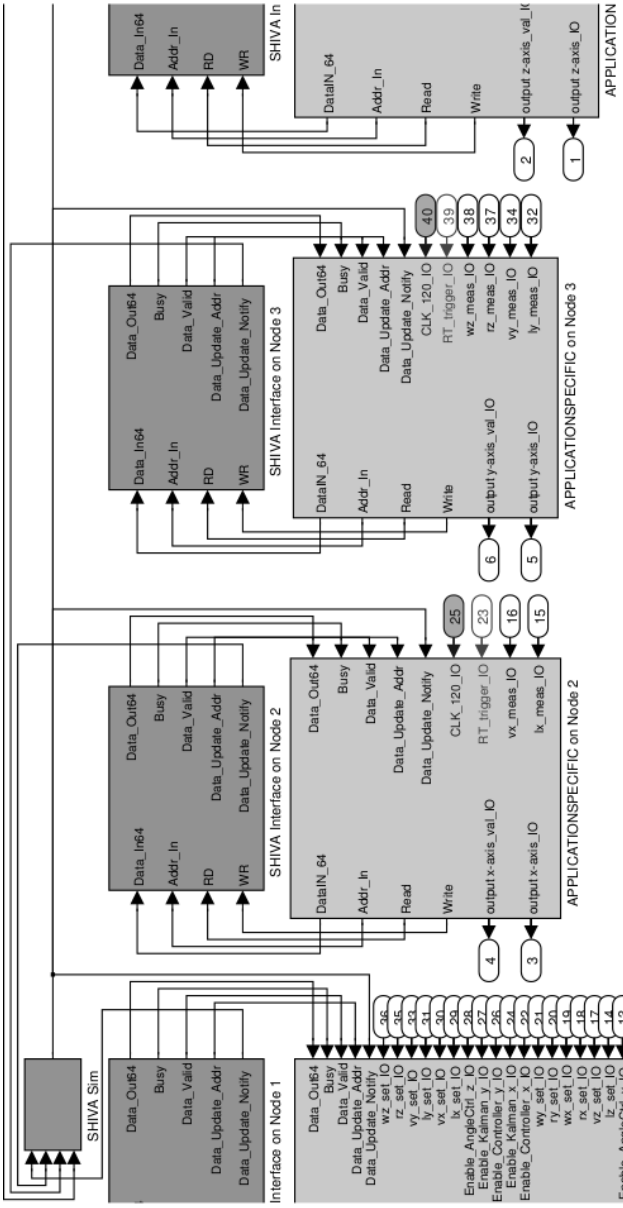


Abbildung 6.16: Ausschnitt aus dem Multi-FPGA-Strukturmodell des partitionierten Mehrachsenreglers mit simulierten Off-Chip-Kommunikationsinterfaces (SHIVA).

6.4.3 Syntheseergebnisse für applikationsspezifische Zugriffslogik

Die Intermodulkommunikation unterbricht den Datenfluss, der ursprünglich zwischen interagierenden Funktionskomponenten bzw. zwischen den Partitionen entworfen wurde und bildet die logischen Datenpfade auf die Shared-Memory-Schnittstelle zur Intermodulkommunikation ab. Abbildung 6.17 zeigt die Applikationspartition der x-Achsen-Regelung mit dem applikationspezifisch generierten VISHNU-Block. Zu erkennen sind die Schnittstellensignale zum SHIVA-Interface (vgl. mit Abbildung 5.22), sowie die lokalen I/O-Signale, die die applikationsspezifische Partition verlassen. Für die Zugriffslogik der jeweili-

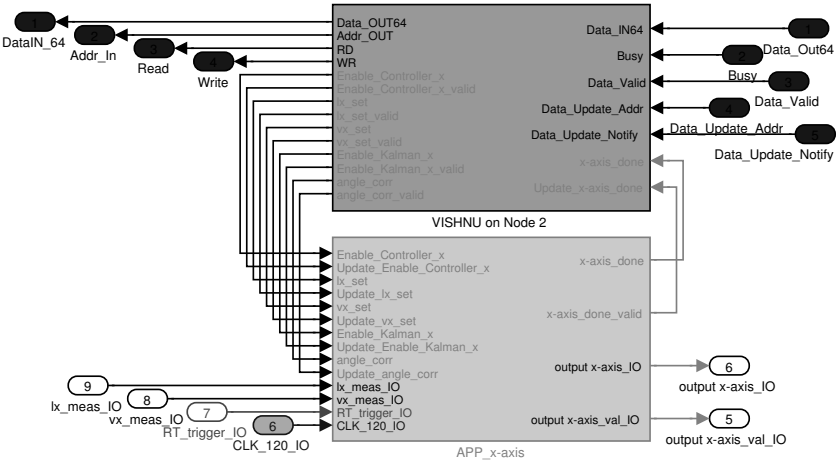


Abbildung 6.17: Innere Ansicht einer applikationspezifischen Strukturpartition des verteilten Implementierungsmodells mit Applikationspartition und automatisch generiertem Kommunikationslogikblock (VISHNU).

gen Partitionen des zuvor vorgestellten partitionierten Gesamtmodells ergeben sich die in Tabelle 6.5 zusammengetragenen Syntheseergebnisse. Zu erkennen sind der in Abhängigkeit der an der Off-Chip-Kommunikation beteiligten Signalpfade steigende, insgesamt jedoch vergleichsweise verschwindend geringe Ressourcenbedarf, und die daraus resultierende sinkende Ausführungsfrequenz. Da diese theoretische Frequenz über der Frequenz der Applikationslogik liegt, beeinflusst die Kommunikationslogik die Leistungsfähigkeit der Logik innerhalb der jeweiligen Partition nicht negativ.

Partition	Logik-Ressourcen (FF/LUTs/Slices)	Frequenz (theo. max.) [MHz]
x-Achse auf Knoten2 - VISHNU für 6 Datenpfade	152/119/< 1%	115,3
y-Achse auf Knoten3 - VISHNU für 9 Datenpfade	155/192/< 1%	84,1
z-Achse auf Knoten4 - VISHNU für 11 Datenpfade	244/223/< 1%	72,3

Tabelle 6.5: Syntheseergebnisse für die applikationspezifische Zugriffslogik VISHNU-for-SHIVA

Die Syntheseergebnisse für die applikationsspezifischen Partitionen mit dem verteilten Reglerentwurf und den jeweiligen Kommunikationslogikblöcken sind in Tabelle 6.6 zusammengefasst. Die Latenz des verteilten Gesamtsystems

Partition	Logik-Ressourcen (FF/LUTs/Slices)	DSP-Res.	Frequenz (theo. max.) [MHz]
x-Achse auf Knoten2	9687/12759/35%	26%	36,8(80)
y-Achse auf Knoten3	10452/14191/39%	46%	37,1(80)
z-Achse auf Knoten4	11066/15004/40%	63%	36,5(80)

Tabelle 6.6: Syntheseergebnisse der applikationsspezifischen Partitionen des für die GECKO3STACK-Plattform entworfenen Gesamtreglermodells

beläuft sich auf 228 Takte und dadurch auf $3,2\mu s$ bei einem Chiptakt von 25 MHz und einem ebenso getakteten Off-Chip-Bus des GECKO3STACK. Dies stellt allerdings nur die Latenz des Reglers ab dem durch den Echtzeittrigger markierten Zeitpunkt zur Übernahme der Messwerte, dem Beginn des Tastintervalls k, dar. Durch die entsprechende Partitionierung wurde sichergestellt, dass keine Off-Chip-Datenübertragungen in die kritischen Ausführungspfade eingebracht wurden. Außerdem gilt dabei die Voraussetzung, dass alle Steuer- und Referenzdaten zu diesem Zeitpunkt bereits an die Reglerpartitionen übertragen wurden. Da die Berechnung derselben allerdings a priori möglich ist, kann dies im Verlauf des Intervals $k-1$ geschehen. Für die Übertragung der Steuer- und Referenzdaten über die SHIVA-Infrastruktur müssen in diesem Szenario 126 Bustakte $= 5,04\mu s$ veranschlagt werden. Die Ergebnisse der zu

Vergleichszwecken wiederum für den Spartan3-4000 durchgeführten Synthese zeigen, dass sich der Bedarf an Ressourcen pro Chip deutlich entspannt hat, ebenso wird eine bessere Nutzung der speziellen DSP-Ressourcen ermöglicht[9]. Untersuchungen haben gezeigt, dass dieser Systementwurf auch für eine Plattform mit FPGA vom Typ Spartan3-1500[10] mit je halber Kapazität synthetisiert werden kann (vgl. Tabelle A.5).

Interessant zu betrachten ist an dieser Stelle noch die Modellkomplexität für den Entwurf einer Applikation für ein verteiltes FPGA-System. Die Interface-Logik stellt bekanntlich einen beträchtlichen Anteil des plattformspezifischen Entwicklungsaufwandes dar. Die Zahlen in Tabelle 6.7 zeigen dementsprechend, wieviel Design-Aufwand in diese Logik im Vergleich zur eigentlichen Applikationslogik dieses Beipsielszenarios zu investieren ist. Als Maß für den Aufwand auf der Ebene der plattformspezifischen Modellierung wird hier die Anzahl der benötigten elementaren Modellblöcke herangezogen. Es

Partition	Modell-blöcke	davon in VISHNU
krit. Pfad x-Achse mit VISHNU-on-GECKO1	1178	469 (39,8%)
krit. Pfad y-Achse mit VISHNU-on-GECKO2	1647	643 (39%)
krit. Pfad z-Achse mit VISHNU-on-GECKO3	1945	769 (39,5%)

Tabelle 6.7: Analyse der Modellkomplexität des verteilten Entwurfs

entfallen durchschnittlich knapp 40% der Modellblöcke für jede Partition auf die VISHNU-Logik. Dem gegenüber stehen die Zahlen aus den Tabellen 6.5 und 6.6, die zeigen, dass die VISHNU-Logik nach der Synthese nur unter 2% der applikationspezifischen Logik des verteilten Reglerentwurfs ausmachte. Diese Diskrepanz verdeutlicht, in welchem Maße die Produktivität dieses Entwicklungsprozesses von der automatischen Generierung der Schnittstellenlogik profitiert.

[9] die Summe über alle Knoten würde ca. 125% Nutzung der DSP-Multiplizierer ergeben. Das Erschöpfen der DSP-Ressourcen eines einzelnen FPGA bedeutet nicht unbedingt, dass ein Entwurf nicht realisiert werden kann - ggf. werden alle anderen Funktionen mittels LUTs implementiert, bis auch diese erschöpft sind, was meist zu langsameren Ausführungstaktraten führt. Beim Multi-Chip-Entwurf können wesentlich mehr (in diesem Fall alle) Multiplikationen mittels der DSP-Multiplizierer umgesetzt werden.)

[10] Spartan3-4000: Kapazität 27 K Slices, 96 DSP-Multiplizierer; Spartan3-1500: Kapazität 13 K Slices, 32 DSP-Multiplizierer [Xil09]

Zusammenfassend lässt sich sagen, dass mittels der Methode zur Generierung von applikationspezifischer Off-Chip-Kommunikationslogik auf einer verteilten Ziel-Hardware realisierbare plattformspezifische Modelle des Reglersystems erzeugt werden können. Prototypische Realisierungen haben gezeigt, dass die VISHNU-Logik leistungsfähig und ressourcensparend realisiert wird, sodass ihr Einfluss auf eine Revision der Partitionierungsentscheidung aus Ressourcengründen eher als gering einzuschätzen ist. Als wesentlicher ist der Eintrag möglicher zusätzlicher Kommunikationslatenzen in den kritischen Ausführungspfad zu betrachten, was bei entsprechenden Partitionierungsentscheidungen zu berücksichtigen ist. Generell eröffnet die automatische Generierung der Zugriffslogik die Möglichkeit, effizient verteilte Modelle von Multi-FPGA-Realisierungen zu erzeugen und den Prozess der Design-Space-Exploration in diese Richtung entscheidend zu beschleunigen.

6.5 Repräsentative Ergebnisse der Design-Space- Exploration für Multi-FPGA-Implementierungen

In den vorigen Abschnitten sind Modelle des Regelungssystems vorgestellt worden, anhand derer die Anwendung der plattformspezifischen Entwurfsmethoden demonstriert wurde. In diesem Abschnitt repräsentative Vertreter einer Vielzahl von Systementwürfen vorgestellt werden, die gezielte Realisierungen von Systementwurfsentscheidungen während der Design-Space-Exploration widerspiegeln. Die Kriterien, nach denen Systementwurfsentscheidungen getroffen werden, hängen auch von den nichtfunktionalen Anforderungen und Beschränkungen der Systemumgebung ab, z.B. Echtzeitschranken, Verfügbarkeit von Berechnungsressourcen, physikalische Randbedingungen.

6.5.1 Systementwurfskriterien und Realisierungsvarianten

Die für diesen Reglersystementwurf bestimmenden nichtfunktionalen Beschränkungen und die daraus resultierenden Systementwurfskriterien sind:

Latenz kritischer Pfade: Die Latenzen der längsten Verarbeitungspfade entscheiden über die Realisierbarkeit des Systems im Rahmen des verfügbaren Echtzeitbudgets. Für ein Regelungssystem wird das Echtzeitbudget durch die Abtastperiode festgelegt als die Zeit zwischen zwei Messzeitpunkten, in der die Reglerfunktion berechnet und die Stellgrößenausgabe erfolgt sein müssen.

Entwurfsziel ist generell die Minimierung der Latenz der kritischen Pfade, hauptsächlich durch die Minimierung der Berechnungslatenzen, aber auch durch die Vermeidung zusätzlicher Latenzen, wie sie z.B. durch Kommunikation eingebracht werden.

I/O-Beschränkungen und -Lokalität: Eine kritische physikalische Beschränkung ist die Verfügbarkeit von I/O-Ressourcen zur Ein- und Ausgabe von Mess- Steuerwerten sowie Stellgrößen. Speziell für eingebettete Hardware bedeutet dies z.B. die begrenzte Möglichkeit, A/D- und D/A-Wandler als Sensor- und Aktor-Schnittstellen direkt an einem Chip anzuschließen. Ein Argument für verteilte Systeme beruht auf der Erhaltung der Lokalität der kritischen Pfade zwischen Messwerterfassung, Verarbeitung und Ausgabe auf je einem Chip mit Übertragung der Daten nicht zeitkritischer Pfade über die Intermodulkommunikation.

On-Chip-Ressourcen: Ein für FPGA-Realisierungen signifikantes Kriterium ist die begrenzte Logik-Kapazität, die den auf einem Chip realisierbaren Funktionsumfang bzw. die Leistungsfähigkeit dieser Realisierungen limitiert. Um die Realisierung trotz Ressourcenbeschränkung zu ermöglichen, bieten sich die Partitionierung des Gesamtentwurfs zu einer Multi-FPGA-Realisierung oder die Reduzierung der Zahl der benötigten Berechungsstrukturen durch Resource-Sharing an. Generell ist eine Minimierung des Ressourcenbedarfs als erstrebenswert anzusehen, dieses Entwurfsziel konkurriert jedoch mit dem Ziel der Latenzminimierung.

Die im Folgenden vorgestellten Realisierungsvarianten beruhen auf einer heuristischen Optimierung bezüglich der soeben diskutierten Kriterien.

- *LatMin* - Referenzszenario mit latenzoptimierten Komponenten - Primär: minimale Gesamtlatenz (Abb. A.8);
- *1Chp-ResFit* - Ein-Chip-Realisierung - Primär: Verringerung des Ressourcenbedarfs bis zur Ein-Chip-Realisierung, Sekundär: minimal mögliche Latenzerhöhung;
- *1Chp-ResMin* - Ein-Chip-Realisierung; Primär: minimaler Ressourcenbedarf (Abb. **??**);
- *2Chp-(XYZ)(6xPID)* - Zwei-Chip-Realisierung mit Trennung der latenzkritischen Pfade für x-, y- und z-Achsen von den 6 PID-Reglern - Primär: Latenzminimierung, Sekundär: Ressourcenminimierung abseits der kritischen Pfade;

- *2Chp-(XY)(Z)* - Zwei-Chip-Realisierung mit Trennung der Funktionen für xy-Ebene und der z-Achse - Primär: Latenzminimum im kritischen Pfad zwischen Messwerterfassung und Stellgrößenausgabe, Sekundär: Ressourcenminimierung abseits der kritischen Pfade;
- *3Chp-(X)(Y)(Z)* - Drei-Chip-Realisierung mit Trennung der Achsen - Primär: Lokalität, Sekundär: Latenzminimierung, Tertiär: lokale Ressourcenminimierung abseits der kritischen Pfade;

Die Entwürfe beruhen auf der Tatsache, dass die Sollwerte von einem auf einer vom eigentlichen Regler getrennten Systempartition implementierten Trajektoriengenerator erzeugt und über die Intermodulkommunikation zur Regler-Partition übertragen werden. Das Zeitschema in Abbildung 6.18 beschreibt die Synchronisation zwischen Messwertaufnahme (DAQ), Erzeugung (Generate) und Übertragung (Transfer) der Sollwerte und Berechnung des Reglers (Controller) während des Echtzeitintervalls zwischen zwei Tastschritten (T_Control). Da die Bahnplanung laut Spezifikation nicht von aktuellen

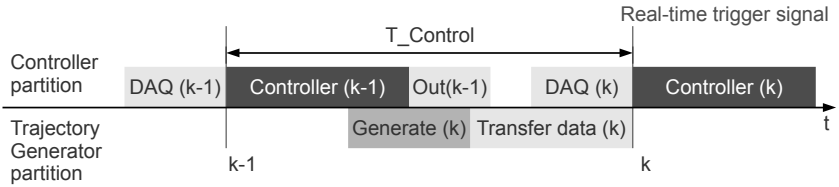


Abbildung 6.18: Ablauf eines Tastschrittes für die Multi-FPGA-Reglerimplementierung.

Messwerten abhängig ist, werden die benötigten Sollwerte bereits vor dem Echtzeittrigger (Real-time trigger signal), der die Verfügbarkeit der aktuellen Messwerte des Tastschrittes anzeigt, übertragen, sodass diese Kommunikationslatenz hierfür nicht Teil der kritischen Latenz der Reglerausführung wird.

6.5.2 Ergebnisse

Die vorgestellten Szenarien wurden plattformspezifisch unter Einsatz der Methoden KALI und VISHNU für die Intermodulkommunikationsplattform SHIVA modelliert und für die FPGA von Typ Xilinx Spartan3-4000 der Multi-FPGA-Plattform GECKO3STACK synthetisiert [Sch12]. Die Validierung der plattformspezifischen Reglermodelle erfolgte entwurfsbegleitend zu den sukzessiven Verfeinerungen gegen das Spezifikationsmodell des Reglers. Die

Validierung wurde durch Simulation und Co-Simulation im Rahmen des abstrakten Umgebungs- und Streckenmodells durchgeführt. Dabei kamen neben der transaktionsorientierten und zyklusgenauen Simulation des plattformspezifischen Modells in Simulink auch die Software-Cosimulation der generierten Hardware-Beschreibungen mittels VHDL-Simulation und Hardware-Cosimulationen (Hardware-in-the-Loop (HIL)) der resultierenden FPGA-Implementierungen zum Einsatz.

Abbildung 6.19 zeigt beispielhaft ein Strukturdiagramm der partitionierten Realisierungsvariante *3Chp-(X)(Y)(Z)* und den dazugehörige Schedule der Ausführung der verteilten Reglerimplementierung. Zu erkennen ist die Partitionierung nach zu regelnden Achsen, sodass die Messwertaufnahme (DAQ) und Stellgrößenausgabe (OUT) lokal auf den jeweiligen FPGAs geschehen kann. Die massive Kommunikation zur Referenzwertübertragung über die Module SHIVA und VISHNU ist, wie in Abbildung 6.18 verdeutlicht wurde, zu diesem Zeitpunkt bereits abgeschlossen und nicht Teils des Ausführungsschedules der Reglerfunktionalität. Daher ist die Intermodulkommunikation während der Reglerberechnung auch minimal - nur das Resultat der Winkelkorrektur muss übergeben werden. Zu erkennen ist außerdem die gestaffelte Ausführung der verschiedenen PID-Regler-Kontexte auf den überlagerten PID-Komponenten auf den Partitionen 2 und 3. Der kritische Pfad setzt sich zusammen aus den Ausführungen des Störbeobachters, der Postprocessing- und der Decoupling-Funktionen der z-Achsenregelung und ist minimal.

Die Dokumentationen der übrigen Szenarien sind dem Anhang A.3.2 und [Sch12] zu entnehmen. Die Ergebnisse sind in Tabelle A.6 im Anhang zusammengefasst und im Diagramm 6.20 grafisch gegenübergestellt. Für jede Variante ist der Ressourcenbedarf bezüglich Slices und DSP-Multiplizierern abgetragen. Für Multi-FPGA-Realisierungen sind die jeweiligen Teilergebnisse für jeden FPGA nebeneinander abgetragen und werden durch einen Balken für die theoretisch akkumulierten Ergebnisse (Slices acc. bzw. DSPs acc.) ergänzt. Zusätzlich zeigt das Diagramm die Grenzen der Ressourcenkapazität des verwendeten FPGA, sowie der beiden Typen mit den nächstgeringeren Kapazitäten, sodass eine Einschätzung der Realisierbarkeit des Systems für wirtschaftlichere Plattformen ermöglicht wird. Zusätzlich ist die Latenz der Realisierungen zwischen Messwertaufnahme und Ende der Stellgrößenberechnung für einen (Multi-)FPGA-Systemtakt von 25 MHz abgetragen. Zu erkennen ist, dass die Leistungsfähigkeit der Zwei-Chip-Entwürfe von der gewählten Partitionierung abhängt - die Ressource-Shared-Realisierung der PID-Regler auf einer getrennten Partition bedingt allerdings den Eintrag zusätzlicher Kommunikationslatenz in den kritischen Ausführungspfad. Latenzarm zeigt sich auch der Drei-Chip-Entwurf, der

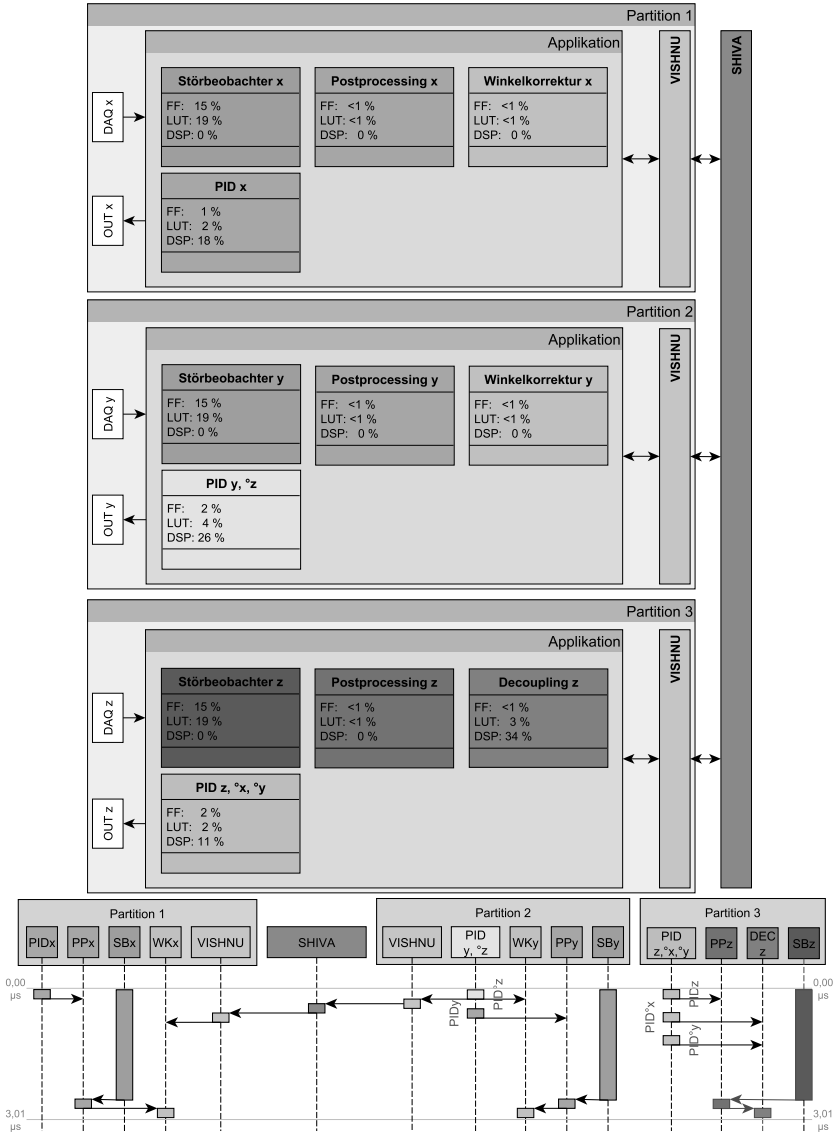


Abbildung 6.19: Struktur und Schedule der 3-Chip-Realisierung *3Chp-(X)(Y)(Z)* des Trajektorien-folgeregelungssystems.

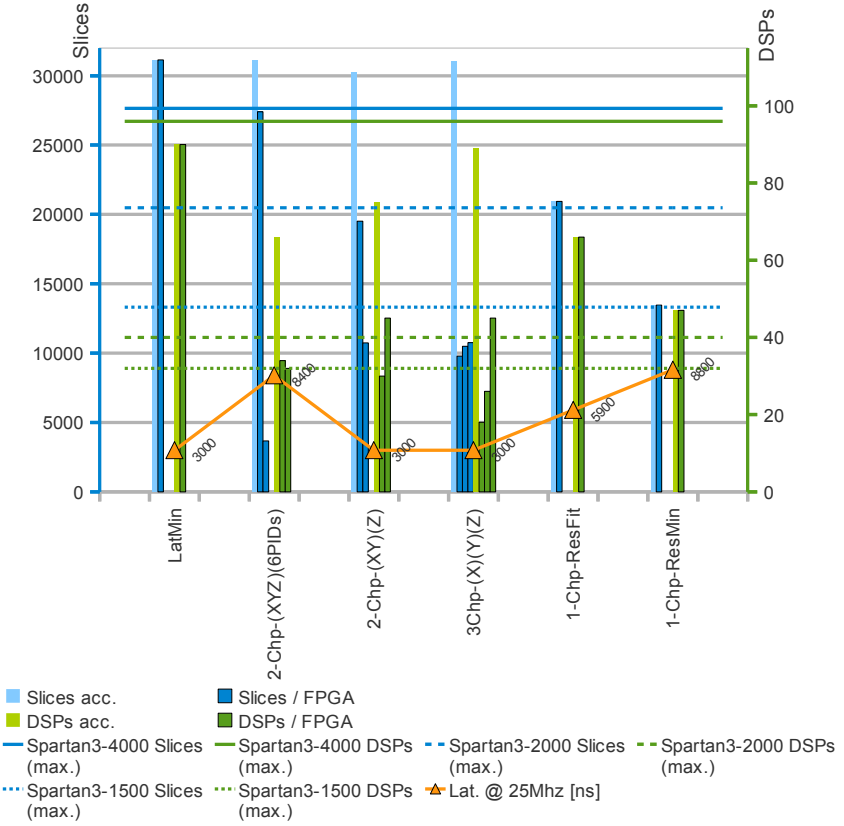


Abbildung 6.20: Quantitativer Vergleich der repräsentativen Implementierungsszenarien des Trajektorienfolgeregelungssystems.

zusätzlich eine Realisierung auf FPGAs mit weniger als der halben Kapazität erlaubt (siehe Tabelle A.5). Die Realisierbarkeit der Ein-Chip-Entwürfe beruht auf dem Resource-Sharing, wodurch sich die Latenz vergleichsweise stark erhöht. Solange eine Latenz von ca. 9 μs vertretbar ist, erlaubt der ressourcenminimale Entwurf eine Realisierung als Teilsystem Seite an Seite mit weiterer Funktionalität auf einem Chip. Mit Ausnahme des Referenzszenarios zur Bestimmung des absoluten Latenzminimums ist die Validität aller

vorgestellten Realisierungsvarianten für die GECKO3STACK-Plattform gegeben.

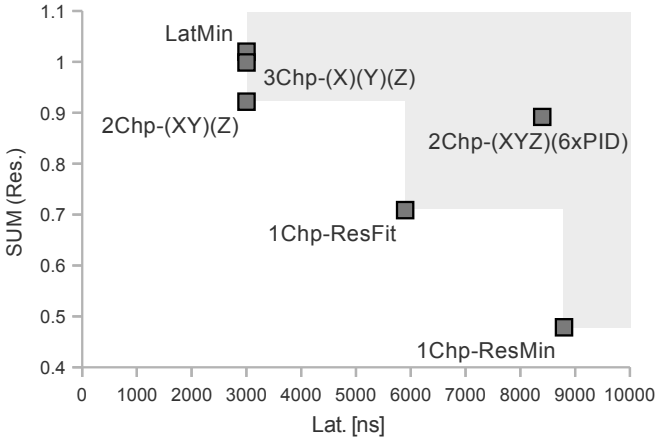


Abbildung 6.21: Qualitativer Vergleich (Pareto-Optimalität) der repräsentativen Implementierungsszenarien des Trajektorienfolgeregelungssystems.

Die Gegenüberstellung der Optimalität bezüglich der verknüpften Kriterien Ressourcenbedarf und Latenz erfolgt in Abbildung 6.21. Dabei ist der Ressourcenbedarf als normierte gewichtete Summe der Slices und der DSP-Ressourcen mit

$$SUM(Res.) = w_1 * \frac{Slices}{max_{Spartan3-4000}(Slices)} + w_2 * \frac{DSPs}{max_{Spartan3-4000}(DSPs)}$$

und $w_1 + w_2 = 1$ dargestellt. In dieser Pareto-Darstellung wird deutlich, dass das Szenario *2Chp-(XYZ)(6xPID)* aus der Gruppe der bezüglich verknüpfter Kriterien optimalen Realisierungen herausfällt, da es bezüglich beider Kriterien von anderen Szenarien dominiert wird. Die Einbeziehung zusätzlicher hier nicht erfasster Randbedingungen zur Implementierungen im Rahmen eines Gesamtsystems und des zu regelnden Prozesses können den gültigen Entwurfsraum weiter einschränken.

Im Verlauf der Verfeinerung des plattformspezifischen Modells nimmt die Modellkomplexität, für die hier die Anzahl der im Gesamtmodell enthaltenen Modellblöcke als Maß gilt, immer weiter zu. Für die repräsentativen Szenarien des Beispielsystems sind die Werte in Tabelle A.7 im Anhang zusammengefasst

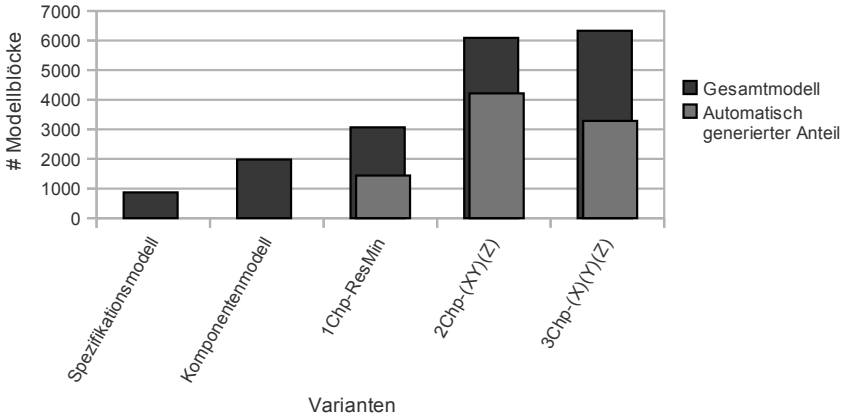


Abbildung 6.22: Vergleich der Komplexitäten der Simulink-Modelle aus dem plattformspezifischen Entwurf des Trajektorienfolgeregelungssystems.

und in Abbildung 6.22 grafisch veranschaulicht. Bei der Umwandlung vom Spezifikationsmodell zum HDL-kompatiblen Komponentenmodell kann bereits eine Verdopplung der Modellkomplexität verzeichnet werden. Die Schaffung des ressourcenminimierten Ein-Chip-Entwurfs erhöht die Komplexität erneut um die Hälfte. Allerdings ist zu verzeichnen, dass 47% der feingranularen plattformspezifischen Modellblöcke dieses Entwurfs auf die automatische Generierung durch die Ressource-Sharing-Methode KALI zurückzuführen sind und daher keinen Anteil am manuellen Entwurfsaufwand haben. Noch deutlicher äußert sich der Vorteil des Einsatzes der automatisierten Methoden am ressourcenminimierten Zwei-Chip-Entwurf, dessen Modellkomplexität zu 69% aus automatisch generierten Ressource-Sharing-Komponenten und durch VISHNU generierter Intermodulkommunikationslogik besteht.

6.6 Fazit

In diesem Kapitel stand die Demonstration der in den vorigen Kapiteln formulierten Konzepte und Methoden im Vordergrund. Gegenstand der Untersuchung war der plattformspezifische Entwurf eines komplexen Regelungssystems aus der Nanopositioniertechnik, das, selbst Gegenstand vorangegangener Forschung und Veröffentlichungen, in Form eines Simulink-Modells als

ausführbare Spezifikation beschrieben wurde. Erste Ergebnisse dokumentieren die Einzelentwicklung von Komponenten entsprechend der erarbeiteten Komponentendefinitionen für grobgranulare Datenfluss-Systeme. Durch explizites Timing-Design als Teil der plattformspezifischen Modellverfeinerung konnten signifikante Leistungssteigerungen für die Einzelkomponenten - bis zu Faktor 2 - erreicht werden. Erste prototypische Synthesevorgänge für das Gesamtsystem offenbarten die Überschreitung der physischen Beschränkungen für eine Einzel-Chip-Implementierung der zugrundegelegten FPGA-Plattform GECKO3STACK und motivierten damit die Anwendung fortgeschrittener Entwurfsmethoden zur Verringerung des Ressourcenbedarfs, bzw. zur Verteilung und Multi-FPGA-Implementierung des Gesamtregelungssystems.

Das Potential zur Ressourceneinsparung mittels der zuvor vorgestellten, für Simulink implementierten Ressource-Sharing-Methode KALI wurde sowohl zur Anwendung auf ressourcenintensive Elementaroperationen, als auch auf der Ebene komplexerer Komponenten quantitativ evaluiert. Die Feasibility der Anwendung dieser Methode auf die bezüglich der Ressourcen dominanten Komponenten des Reglersystems konnte gezeigt werden, wodurch Gesamtsystementwürfe ermöglicht wurden, die mit einer Ressourceneinsparung von bis zu ca. 50% die Ein-Chip-Realisierung des Reglersystems erlauben. Als Alternative zu Resource-Shared-Entwürfen wurde die Erstellung verteilter Entwürfe für die Multi-FPGA-Plattform GECKO3STACK mit der Off-Chip-Kommunikationsplattform SHIVA untersucht, um die automatische Generierung der Zugriffslogik mittels der für Simulink implementierten Methode VISHNU zu demonstrieren. Der verteilte Entwurf machte den Vorteil von Applikationskomponenten mit Datenfluss-Interface deutlich. Sie erleichterten die Integration mit der Off-Chip-Kommunikationsschnittstelle duch Minimierung des manuellen Redesign-Aufwands für die Applikationspartitionen.

Mittels des kombinierten Einsatzes beider Methoden konnte im Rahmen der Design-Space-Exploration eine Reihe von Implementierungsszenarien des Gesamtreglers erstellt werden, deren repräsentative Vertreter bezüglich der Kriterien Latenz und Chipressourcenkonsumption gegenübergestellt wurden. Eine abschließende Analyse der Modellkomplexität während des zunehmend plattformspezifischen Verfeinerungsprozesses offenbart eine starke Zunahme von Modellanteilen, die die eigentlichen funktionalen Komponenten um rein ausführungsplattformspezifische Daten- und Steuerpfadlogik ergänzen - in den analysierten Szenarien beträgt der Anteil zwischen 47 und 69%. Diese Zahlen verdeutlichen, in welchem Maße der Entwurfsaufwand durch Einsatz der automatisierten Methoden während der PIM-to-PSM-Transformation reduziert werden kann.

7 Zusammenfassung und Ausblick

Der Entwurf verteilter eingebetteter Systeme ist aufgrund steigender Plattformleistungsfähigkeit und daraus resultierender steigender Komplexität der realisierbaren Systeme zunehmend nur unter Einsatz von modellbasierten Entwicklungsmethoden zu realisieren. In der Mess- und Automatisierungstechnik ist die Untersuchung des Reglerentwurfs im Zusammenhang mit dem zu regelnden Prozess im Rahmen der simulativen Validierung und des virtuellen Prototypings wesentlicher Bestandteil des modellbasierten Entwurfsprozesses. Zur Realisierung eines geschlossenen Entwicklungsprozesses vom ausführbaren Spezifikationsmodell zum ausführbaren Code wird die Entwicklung von zielplattformspezifischen Modellierungswerkzeugen, -bibliotheken und Code-Syntheseverfahren vorangetrieben. Die zunehmende Leistungsfähigkeit von FPGAs und die Verfügbarkeit von FPGA-basierten eingebetteten Plattformen und -modulen motiviert dabei die Entwicklung von Generierungsmechanismen für Hardware-Beschreibungssprachen. Dabei ergänzen sich die Bemühungen sowohl von Seiten der namhaften Hersteller von Modellierungs-, Simulations- und Entwicklungswerkzeugen, wie u.a. The Mathworks, als auch von Seiten der FPGA-Hersteller wie Xilinx oder Altera. Dadurch kann der modellbasierte Entwicklungsprozess auch für FPGA-Plattformen durch die Integration der Tool-Chains und Methoden weiter geschlossen werden.

In Anwendungsbereichen mit außergewöhnlichen Anforderungen, wie die den Hintergrund dieser Arbeit darstellende Realisierung von Regelungssystemen für die Nano-Positioniertechnik, spielt eine umfassende Design-Space-Exploration eine wichtige Rolle, um Systementwurfsentscheidungen hinsichtlich ihrer Machbarkeit bewerten zu können. Die wesentlichsten Kriterien sind hier die Umsetzbarkeit eines FPGA-Entwurfs unter Ressourcenbeschränkungen und Latenzschranken, der gegebenenfalls auf verteilte bzw. Multi-FPGA-Plattformen zu implementieren ist. Die in dieser Arbeit zusammengetragenen Methoden des modell- und plattformbasierten Entwurfs beschreiben aktuelle Meet-in-the-Middle-Entwurfsansätze, die die Systementwurfskomplexität durch ein erhöhtes Abstraktionsniveau und automatisierte Transformationen sowie den Implementierungsaufwand durch wiederverwendbare Komponenten und Architekturen reduzieren sollen.

Im Zentrum dieser Ansätze steht die Definition einer Plattformebene, die das Architekturmodell für die plattformspezifischen Beschreibungen des Systementwurfsprozesses festlegt. Die Erarbeitung einer solche Entwurfsplattformebene und dort anzusiedelnder Methoden für den modellbasierten komponentenorientierten Entwurf von eingebetteten Echtzeit-Signalverarbeitungs- und Regelungssystemen zur Implementierung auf Multi-FPGA-Plattformen ist Gegenstand dieser Arbeit. Die Methoden wurden in einen MATLAB/Simulink-basierten Entwicklungsprozess integriert, der die Demonstration ihrer Anwendung anhand von Beispielszenarien aus dem Kontext der Reglerentwicklung für die Nano-Positioniertechnik ermöglichte.

7.1 Diskussion der Ergebnisse

Kern der methodischen Ansätze dieser Arbeit ist die Formulierung eines Metamodells, das die Strukturen und ihre Beziehungen während der Transformation vom plattformunabhängigen Spezifikationsmodell zum plattformspezifischen Systemmodell festlegt. Dieses Metamodell umfasst:

- *das Metamodell der Multi-FPGA-Hardware-Plattform* - es erfasst die Struktur von Multi-FPGA-Hardware, die Abstraktion ihrer Kommunikationsinfrastruktur sowie die Struktur von verteilten Komponentensystemen. Diese Definitionen wurden unter anderem von Eigenentwicklungen und Erfahrungen auf den Gebieten der modularen skalierbaren Hardware-Plattformen und der Intermodulkommunikation motiviert, wie durch die konzeptuellen Studien belegt wurde. Als entscheidend für die Entkopplung zwischen Systementwurf und Multi-FPGA-Hardware ist die Schaffung einer Abstraktionsebene für die Intermodulkommunikation zu sehen, die eine flexible Integration in verschiedene Systemarchitekturen und Entwurfskontexte erlaubt.
- *das Metamodell der plattformspezifischen Modellierungsebene* - es legt die Struktur für die komponentenbasierten Systeme der mess- und regelungstechnischen Anwendungsdomäne fest. Dazu enthält es die Abbildungen des Datenflussgraphen als Spezifikationsmodell auf nach dem synchronen Datenflussparadigma interagierende plattformspezifische Komponentenstrukturen. Außerdem werden die Abbildungen dieser Komponentenmodelle auf Implementierungskomponenten der Hardware-Plattformebene ausgedrückt. Zwei wesentliche Komponententypen wurden unterschieden - synchron getriggerte und datenflussgesteuerte Komponenten. Erstere bilden die Struktur der meisten wiederverwendbaren IP-Cores ab und

erleichtern die Bottom-up-Integration von Black-Box-Komponenten in die plattformspezifische Modellebene; letztere bieten die Möglichkeit zur konsistenten und effizienteren Top-Down-Realisierung der datenflussorientierten Systeme aus der Anwendungsdomäne.

- *Erweiterungen zum plattformspezifischen Metamodell* - sie definieren spezielle Komponentenstrukturen, die eine automatische Modellsynthese zulassen, und betreffen:
  - *komponentenbasiertes „Resource-Sharing“* - die Trennung von kontextspezifischen Elementen und dem eigentlichen Berechnungskern zur sequentiellen Ausführung von multiplen spezifizierten Funktionen auf einer gemeinsamen Strukturkomponente. Unterstützt durch den als Neuerung zu wertenden Ansatz der „Moduslogik“ ist diese Methode des Resource-Sharing auf der Ebene des grobgranularen Komponentenentwurfs gewinnbringend einsetzbar.
  - *Zugriffslogik zur Off-Chip-Kommunikation* - die applikationsspezifische Abbildung von partitionsübergreifenden Datenflussverbindungen auf eine Intermodulkommunikationsinfrastruktur.

  Die Integration beider Ansätze zur automatischen Modellsynthese in einem plattformspezifischen Entwurfskontext konnte exemplarisch durch in MATLAB/Simulink® realisierte Tools gezeigt werden.

Als Demonstrationsbeispiele für die Anwendung der vorgestellten Methoden dienten aus dem Kontext des SFB622 übernommene, im Rahmen einer Design-Space-Exploration erstellte Entwürfe zur modellbasierten multi-FPGA-orientierten Realisierung von 3D-Trajektorienfolgeregelungssystemen. Von der Simulation der funktionalen Spezifikation des Reglers bis zur taktgenauen Simulation des plattformspezifischen Modells ist die Validierung im heterogenen Umgebungsmodell während der Verfeinerungen und Modelltransformationen eine wesentliche Charakteristik dieses modellbasierten Entwurfs.

- *Komponenten-Prototyping* - Die Anwendung des komponentenorientierten Entwurfs führte zu konsistent synthetisierbaren und testbaren Hardware-Beschreibungen der Reglerkomponenten und des Gesamtsystems und erlaubte Aussagen über Leistungsfähigkeit und Ressourcenbedarf, was die Entwicklung und Anwendung fortgeschrittener plattformspezifischer Entwurfsansätze motivierte.
- *Methodenevaluierung* - Das Potential zur Ressourceneinsparung mittels der für Simulink implementierten Ressource-Sharing-Methode wurde sowohl zur Anwendung auf ressourcenintensive Elementaroperationen, als auch auf komplexere Reglerkomponenten

quantitativ evaluiert. Dadurch wurden Gesamtsystementwürfe ermöglicht, die mit einer Ressourceneinsparung von bis zu ca. 50 % bezüglich der parallelen Implementierung Ein-Chip-Realisierungen auf der Demonstratorplattform möglich machen. Der Entwurf verteilter Anwendungsszenarien erlaubte die Implementierung des Reglersystems ohne die Latenzeinbußen, die ein Resource-Sharing-Entwurf mit sich bringt. Während des Entwurfs wurde der Vorteil von Applikationskomponenten mit Datenfluss-Interface deutlich, da sie die Generierung zusätzlicher Puffer- und Synchronisationslogik zwischen serialisierter Übertragung und parallelisierter Ausführung unnötig machen. Dadurch wird die Integration der Off-Chip-Kommunikationsschnittstelle mit dem grobgranularen Datenflussparadigma deutlich effektiver.

- *Design-Space-Exploration* - Mittels des kombinierten Einsatzes der Methoden konnte im Rahmen der Design-Space-Exploration für Realisierungen des 3D-Trajektorienfolgereglers eine Reihe von repräsentativen Implementierungsszenarien geschaffen und bezüglich der Kriterien Latenz und Chipressourcenkonsumption gegenübergestellt werden.

Eine abschließende Analyse der Modellkomplexität als Maß für den Entwurfsaufwand während des zunehmend plattformspezifischen Verfeinerungsprozesses offenbart eine starke Zunahme von Modellanteilen, die die eigentlichen funktionalen Komponenten um rein ausführungsplattformspezifische Daten- und Steuerpfadlogik ergänzen - in den analysierten Szenarien beträgt der Anteil ca. 47 bis 69 %. Diese Zahlen verdeutlichen, wie groß die Kluft zwischen den von klassischer Wiederverwendung profitierenden Modellanteilen auf der einen Seite und den plattform- und infrastrukturspezifischen Anteilen auf der anderen Seite ist. Dies zeigt, in welchem Maße der Entwurfsaufwand durch Einsatz der automatisierten Strukturgenerierungsmethoden während der PIM-to-PSM-Transformation reduziert werden konnte.

7.2 Weiterführende Arbeiten und Ausblick

Ausgehend vom Stand der zuvor beschriebene Entwicklungen lassen sich weiterführende Arbeiten in zwei Kategorien einteilen - künftige Weiterentwicklungen gemäß des fortschreitenden Standes der Technik und Erweiterungen in den umgebenden Entwurfskontext.

Der Fortschritt auf dem Gebiet der eingebetteten Hardware wird auch weiterhin von höherer Integrationsdichte und steigender Systemkomplexität gekennzeichnet sein. Das zieht auch eine Änderung der Systemgranularität

nach sich - Strukturen, die vor kurzem noch aus Platzgründen auf mehreren Chips verteilt implementiert werden mussten, werden in naher Zukunft in Form eines Teilsystem in einem komplexeren System-on-Chip unterzubringen sein. Die automatische Anbindung modellierter Funktionskomponenten an etablierte on-Chip-Kommunikationssysteme geschieht prinzipiell nach den gleichen Grundsätzen wie die in dieser Arbeit bezüglich der Off-Chip-Kommunikation formulierte Methode. Die Abbildung von modellbasiert entworfenen Komponenten und Teilsystemen auf System-on-Chip- oder Network-on-Chip-Architekturen wird dann ein wesentlicher plattformspezifischer Entwurfsschritt sein. Die Technik der dynamischen partiellen Rekonfiguration (DPR) ist mittlerweile Stand der FPGA-Technologie und stellt eine erweiterte Möglichkeit des Resource-Sharing dar, bei dem zur Laufzeit größere Teile der Chip-Fläche zum zeitlich gestaffelten Austausch ganzer Komponenten oder Teilsysteme umkonfiguriert werden. Die in dieser Arbeit entwickelte komponentenbasierte Resource-Sharing-Methode mit dem Konzept der Modus-Logik zeigt einen Ansatz für die Modellierungstechnik zum Entwurf von DPR-Systemen auf.

Die Schaffung automatisierter Modelltransformationen soll der Beherrschung und der Reduktion des Entwicklungsaufwandes in einem durchgehenden modellbasierten Entwurfsprozess dienen. Die Inhalte dieser Arbeit konzentrieren sich auf die automatisierte Umsetzung von Systementwurfsentscheidungen im Rahmen der plattformspezifischen Modellierung und dienen der *Verkürzung der Entwurfszyklen.* Die entstandenen Modellkomponenten, die formulierten Entwurfsrichtlinien und die implementierten Modellsyntheseansätze können als Basis für den Ausbau FPGA-plattformspezifischer Bibliotheken und Tools für die Anwendungsdomäne der Mess- und Regelungstechnik dienen. Ein folgerichtiger Schritt für die in dieser Arbeit auf Modellebene untersuchten Syntheseansätze wäre die Verlagerung in die Ebene der Modell-zu-Code-Transformation, was die Usability der modellbasierten Entwicklung durch eine weitere Anhebung des Abstraktionsgrades verbessern würde.

Ein wesentlicher Teil des Systementwurfs besteht in der effizienten Eingrenzung des Entwurfsraumes für die Design-Space-Exploration mit dem Ziel der *Verringerung der Design-Iterationen.* Die in dieser Arbeit als Teile der modellbasierten Entwurfsplattformebene definierten Metamodelle für die funktionale Spezifikation, die Plattformarchitektur und das plattformspezifische Systemmodell stellen die Grundlage für die Anbindung an ein Framework zur Architekturabbildung und -optimierung dar. Während die Struktur der Ausführungsplattform und die Abbildungs- und Verteilungsvorschriften im Rahmen dieser Arbeit heuristisch ermittelt wurden, würde der Entwurfsprozess von weiterführenden Arbeiten bezüglich der Automatisierung dieses multikriteriellen Optimierungsprozesses profitieren.

A Anhang

A.1 Resource-Sharing - Prototyping

# LUTs	Eingänge	2	3	4	5	6	7	8
Bitbreite								
16		17	33	51	67	72	89	99
32		33	65	99	131	136	177	196
48		49	97	147	195	200	291	308
64		65	129	195	259	264	353	388
80		81	161	243	323	328	483	516

Tabelle A.1: Syntheseergebnisse für Simulink-Multiplexer.

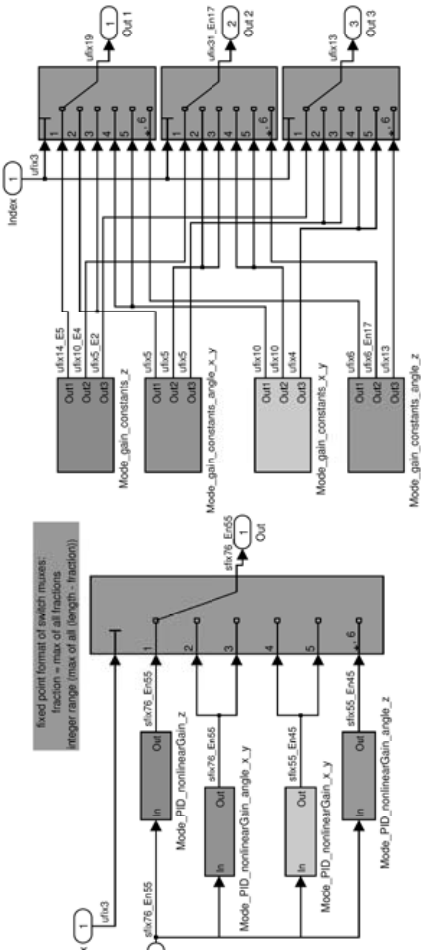


Abbildung A.1: Interna der Modus-Blöcke mit Multiplexern für Regler-Konstanten aller sechs PID-Kontexte.

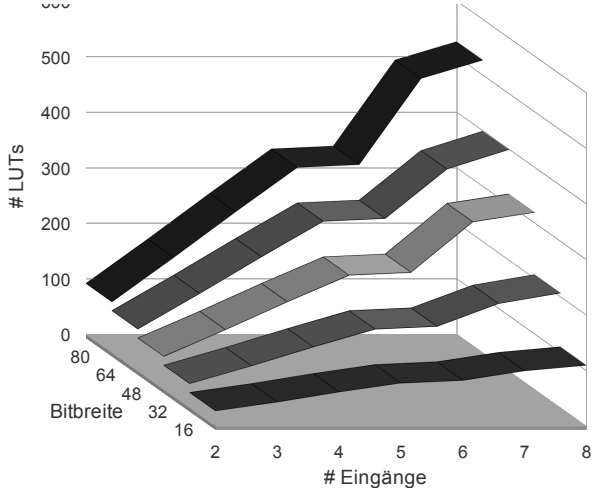


Abbildung A.2: Syntheseergebnisse für Simulink-Multiplexer.

Decoupling-Block der z-Achsenregelung

Decoupling z lat.-min.	Logik-Ress. (FF/LUTs/ Slices)	DSP-Ress.	Frequ. (MHz)	Takte	Lat. (ns)
ohne Mux. 9 Mul.+ 3 Add.	352/2098/3%	34%	44,5	3	67
3 Mul.+ 3 Add.	1447/1793/5%	21%	48,3	17	351
3 Mul.+ 1 Add.	2301/1923/6%	21%	48,3	28	579
1 Mul.+ 3 Add.	1046/1156/3%	12%	44,8	26	580
1 Mult. + 1 Add.	1869/1280/5%	12%	44,6	31	695

Tabelle A.2: Syntheseergebnisse des zeitmultiplexten Entwurfs der Decoupling-Komponente mit Matrixmultiplikation.

Resource-Sharing PID-Regler z-Achsenregelung

Anzahl PIDz	1	2	4	8	16	32	64
FF (RS-PID)	656	1173	1783	3001	5439	10314	20058
FF (par. PID)	656	1312	2624	5248	10496	20992	41984
LUT (RS-PID)	1669	2090	3198	4771	7422	12792	24505
LUT (par. PID)	1669	3366	6732	13336	26672	53088	106176
FF (Kernlogik)	656	656	656	656	656	656	656
LUT(Kernlogik)	1669	1669	1669	1669	1669	1669	1669
FF (MUX-O.hd)	0	517	1127	2345	4783	9658	19402
LUT (MUX-O.hd)	0	421	1529	3102	5753	11123	22836
Freq. (RS) (MHz)	47	39	32	25	23	17	16
Freq. (par.) (MHz)	47	47	47	47	47	45	45
Latenz (RS) (ns)	128	363	808	1971	4333	11167	23945
Latenz (par.) (ns)	128	128	128	128	128	133	133

Tabelle A.3: Syntheseergebnisse für ressource-shared PID-Regler der z-Achsenregelung.

Anzahl PIDz	Gesamt (Parallel)	Gesamt (RS_Komp.)	PID	States	Modi	Kernlogik
2	259	413	232	92	51	89
4	517	675	336	166	81	89
8	1033	1199	544	314	141	89
16	2065	2247	960	610	261	89
32	4129	4343	1792	1202	501	89
64	8257	8535	3456	2386	981	89

Tabelle A.4: Modellierungsaufwand für ressource-shared PID-Regler.

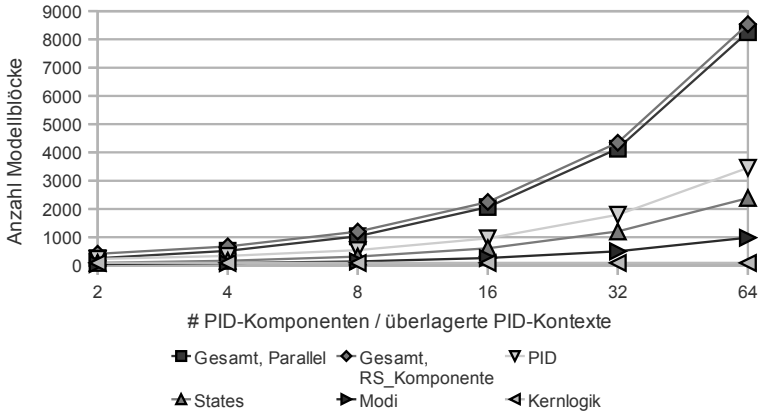


Abbildung A.3: Modellierungsaufwand für ressource-shared PID-Regler z-Achsenregelung.

A.2 Modellierung des Regelungssystems

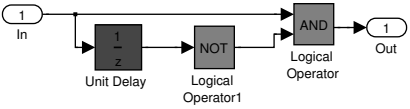


Abbildung A.4: Flankenerkennung „Rising Edge Detect“.

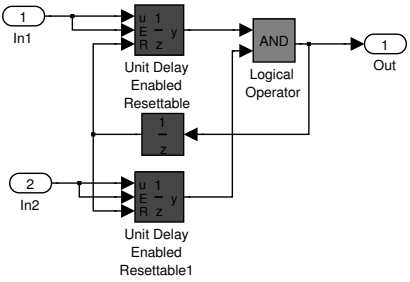


Abbildung A.5: Trigger-Synchronisation „Trigger Synchronize“.

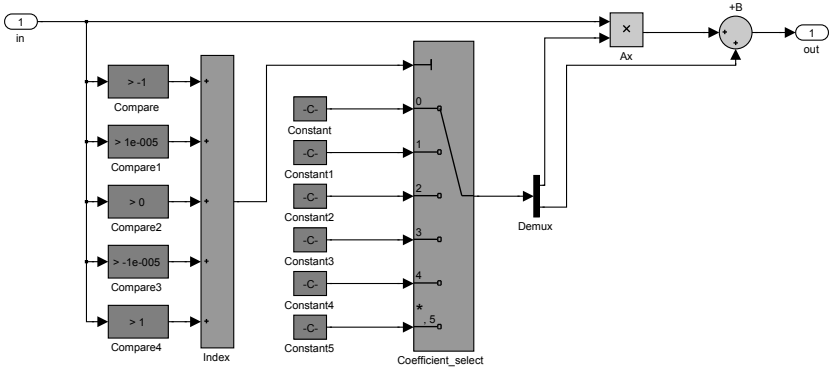


Abbildung A.6: Linearer Interpolator als Ersatzkonstrukt für Look-Up-Table der nichtlinearen Verstärkungen der PID-Regler.

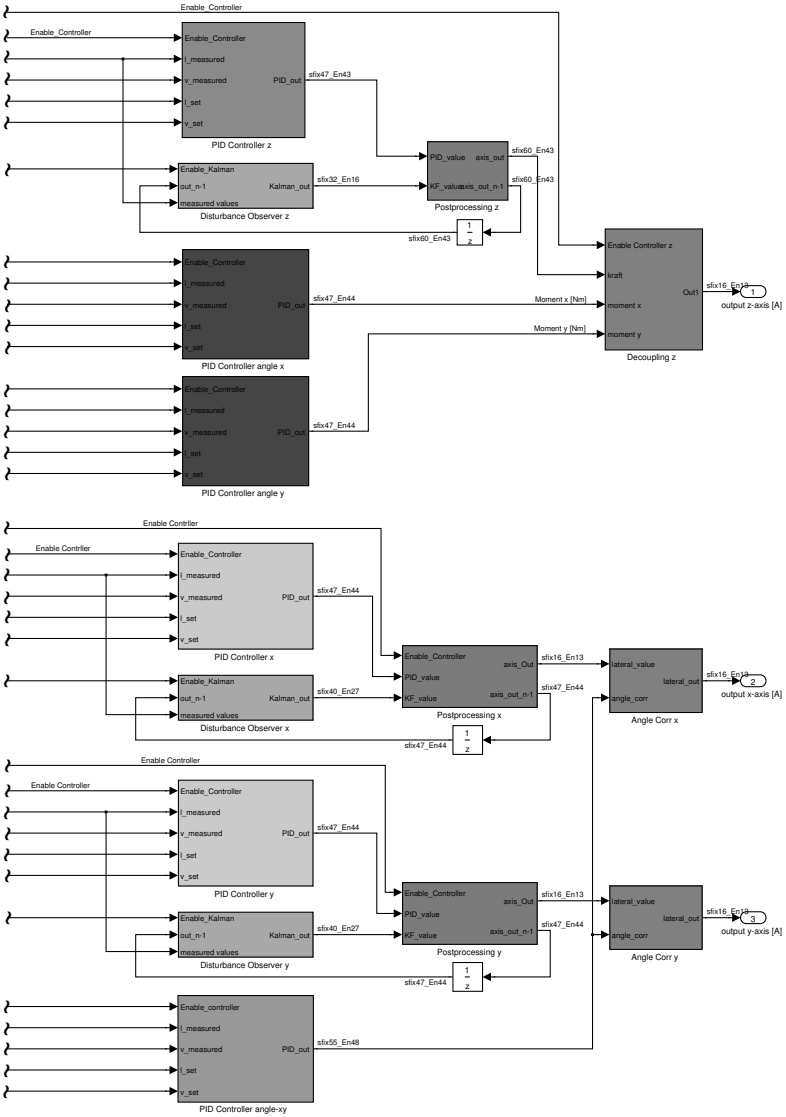


Abbildung A.7: Datenpfad des 3D-Trajektorienfolgereglersystems mit spezifizierten Funktionsmodulen und Festkommadatentypen.

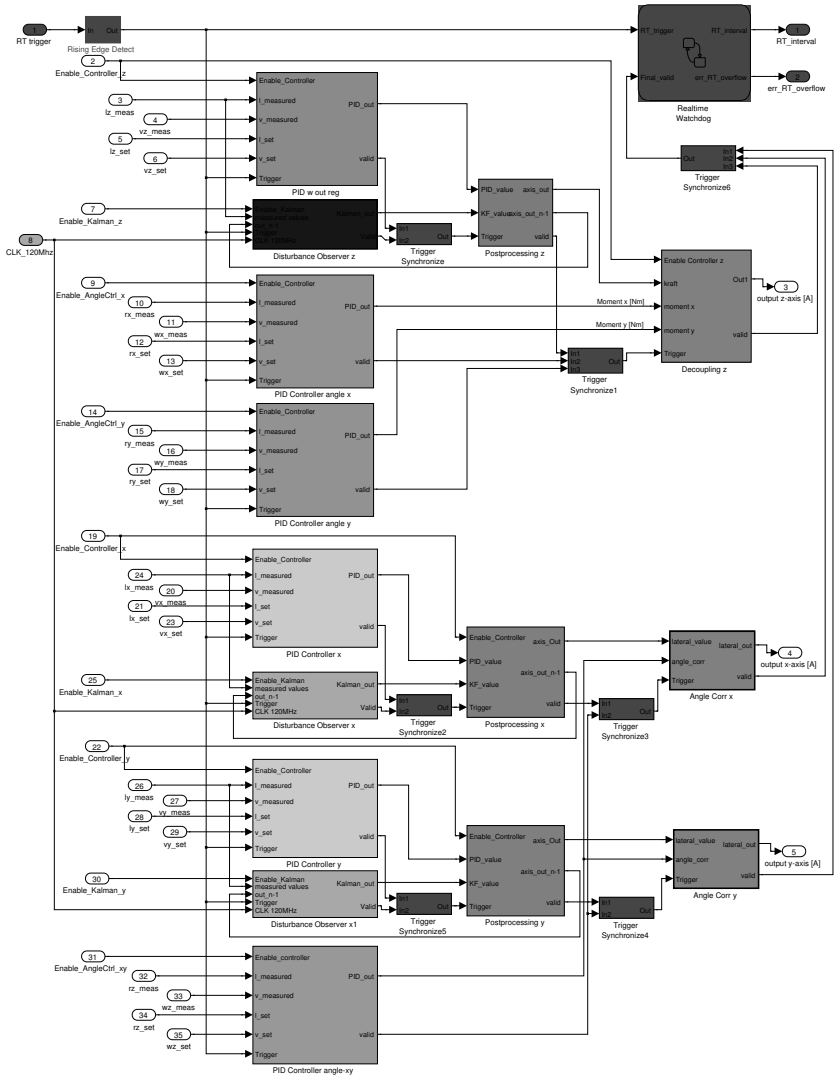


Abbildung A.8: Komponentenmodell des 3D-Trajektorienfolgereglers mit synchron getriggerten Komponenten.

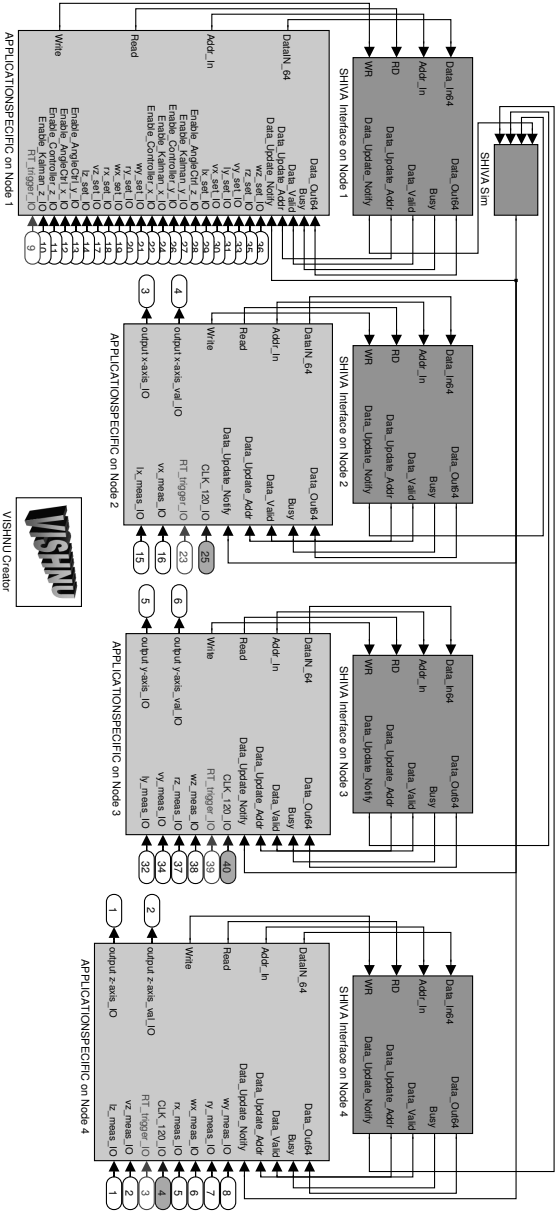


Abbildung A.9: Multi-FPGA-Strukturmodell mit simulierter Off-Chip-Kommunikation (SHIVA).

Partition	Logik-Ressourcen (FF/LUTs/Slices)	DSP-Ressourcen	Frequenz (MHz)
x-Achse auf Knoten2	9703/12700/72%	78%	18,5(120)
y-Achse auf Knoten3	10448/15005/85%	93%	17,9(120)
z-Achse auf Knoten4	11046/16103/87%	90%	36,5(120)

Tabelle A.5: Syntheseergebnisse der applikationsspezifischen Partitionen (Teilapplikation+VISHNU) für alternativen FPGA Spartan3-1500

A.3 Plattformspezifische Multi-FPGA-Modelle

A.3.1 Syntheseergebnisse

Regler-system	Logik-Res. (Slices)	DSP-Res.	BRAM	Freq. (max.) [MHz]	Lat. [cyc.]	Lat. @25MHz [μs]
LatMin	112%	90%	18%	31,6(80)	225	3,0
1Chp-ResMin	48%	47%	6%	30,6(80)	667	8,8
1Chp-ResFit	77%	66%	12%	30,8(80)	447	5,9
2Chp-(XY)(Z)					225	3,0
xy-Eb. (Kn. 2)	70%	30%	12%	33,9(80)		
z-Achse (Kn. 3)	38%	45%	6%	36,5(80)		
Kommunikation					126	5,04
2Chp-(XYZ) (6xPID)						8,4
xyz-A. (Kn. 2)						
krit. Pfade	98%	34%	18%	33,9(80)		
PIDs (Kn. 3)						
6x mux. PIDs	13%	32%	6%	29,8(80)		
Kommunikation					126	5,04
3Chp-(X)(Y)(Z) LatMin-ResMin					228	3,0
x-Achse (Kn. 2)						
krit. Pfad	35%	18%	6%	33,9(80)		
y-Achse (Kn. 3)						
2x mux. PIDs	37%	26%	6%	33,9(80)		
z-Achse (Kn. 4)						
3x mux. PIDs	38%	45%	6%	36,5(80)		
Kommunikation					126	5,04

Tabelle A.6: Syntheseergebnisse der Reglersysteme für Multi-FPGA-Plattform GECKO3STACK mit Spartan3-4000.

A.3.2 Strukturen und Schedules

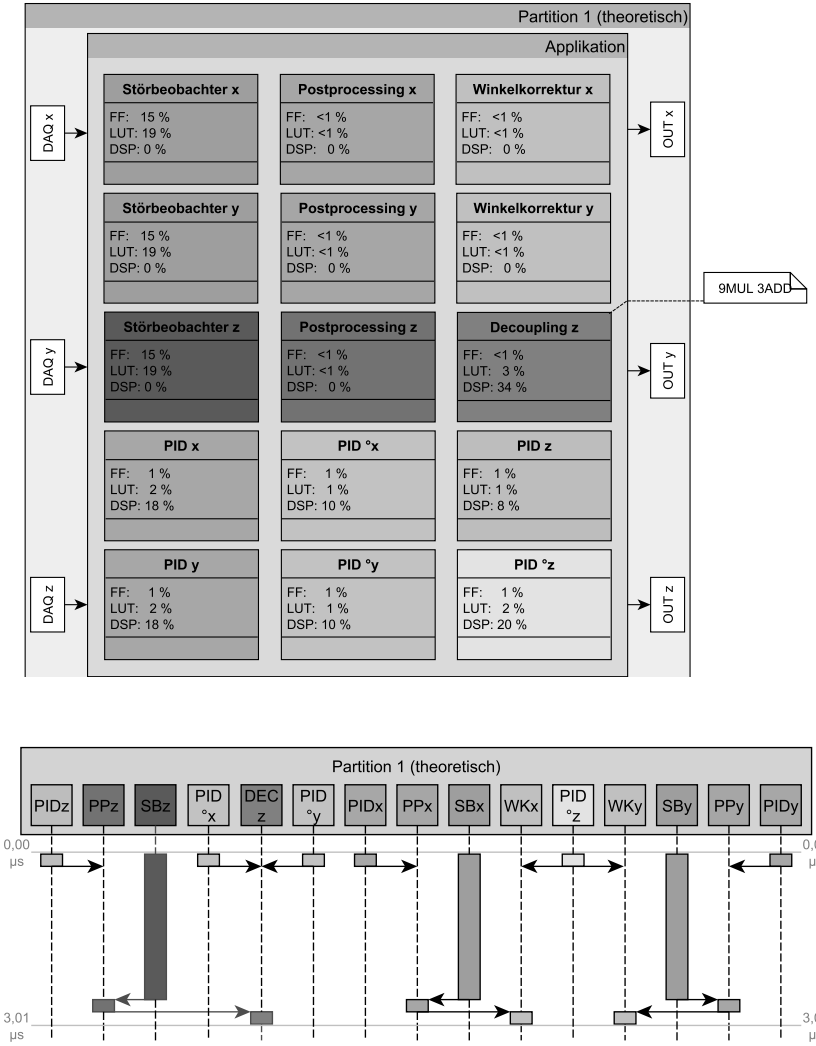
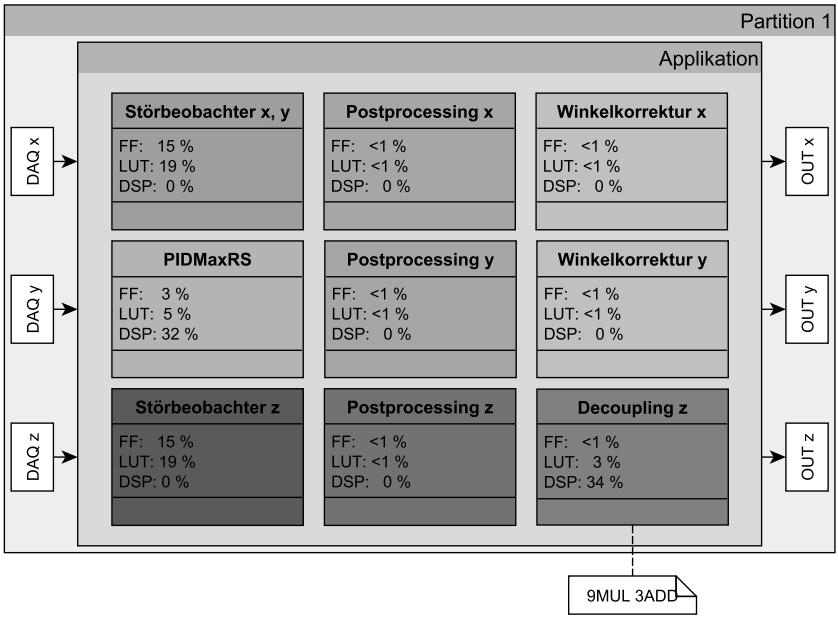


Abbildung A.10: Struktur und Schedule der latenzminimierten Realisierung *LatMin* des Trajektorienfolgeregelungssystems.


Partition 1
Applikation
DAQ x
DAQ y
DAQ z
Störbeobachter x, y
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing x
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur x
FF: <1 %
LUT: <1 %
DSP: 0 %
PIDMaxRS
FF: 3 %
LUT: 5 %
DSP: 32 %
Postprocessing y
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur y
FF: <1 %
LUT: <1 %
DSP: 0 %
Störbeobachter z
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing z
FF: <1 %
LUT: <1 %
DSP: 0 %
Decoupling z
FF: <1 %
LUT: 3 %
DSP: 34 %
OUT x
OUT y
OUT z
9MUL 3ADD

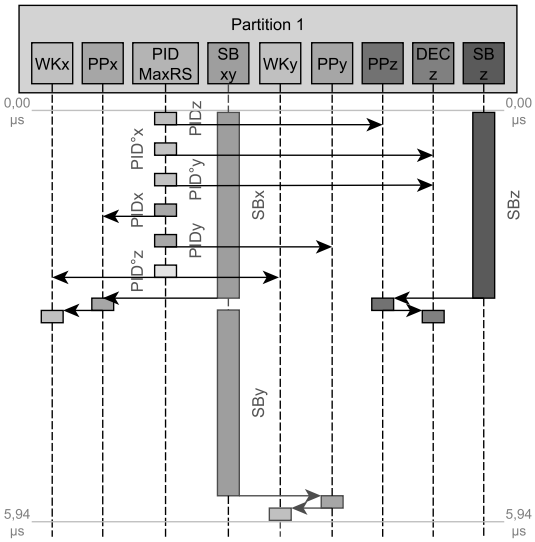
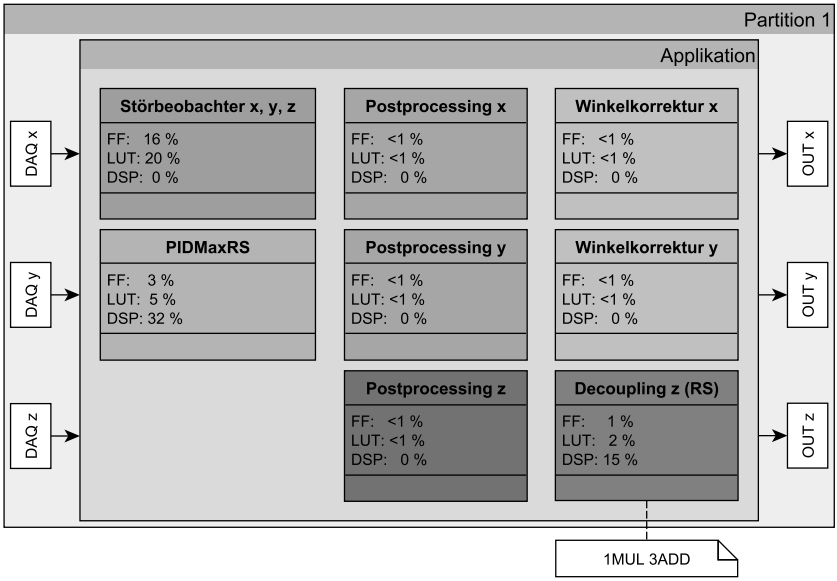


Abbildung A.11: Struktur und Schedule der 1-Chip-Realisierung *1Chp-ResFit* des Trajektorienfolgeregelungssystems .


Partition 1
Applikation
DAQ x
DAQ y
DAQ z
Störbeobachter x, y, z
FF: 16 %
LUT: 20 %
DSP: 0 %
PIDMaxRS
FF: 3 %
LUT: 5 %
DSP: 32 %
Postprocessing x
FF: <1 %
LUT: <1 %
DSP: 0 %
Postprocessing y
FF: <1 %
LUT: <1 %
DSP: 0 %
Postprocessing z
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur x
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur y
FF: <1 %
LUT: <1 %
DSP: 0 %
Decoupling z (RS)
FF: 1 %
LUT: 2 %
DSP: 15 %
1MUL 3ADD
OUT x
OUT y
OUT z

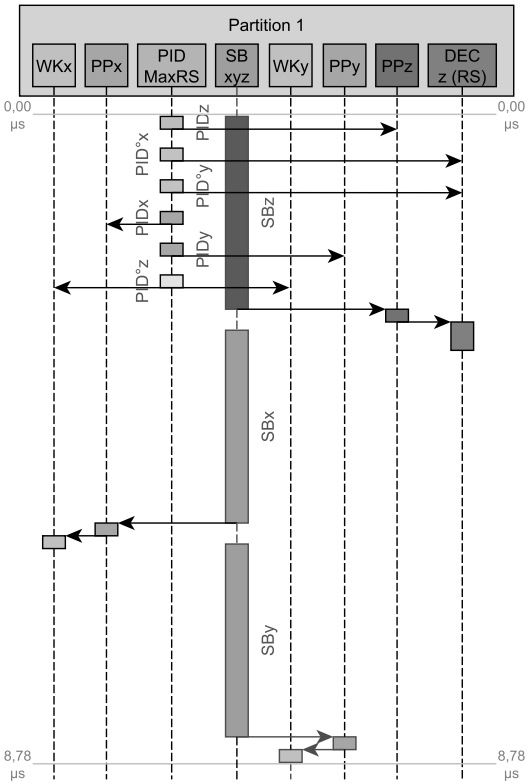
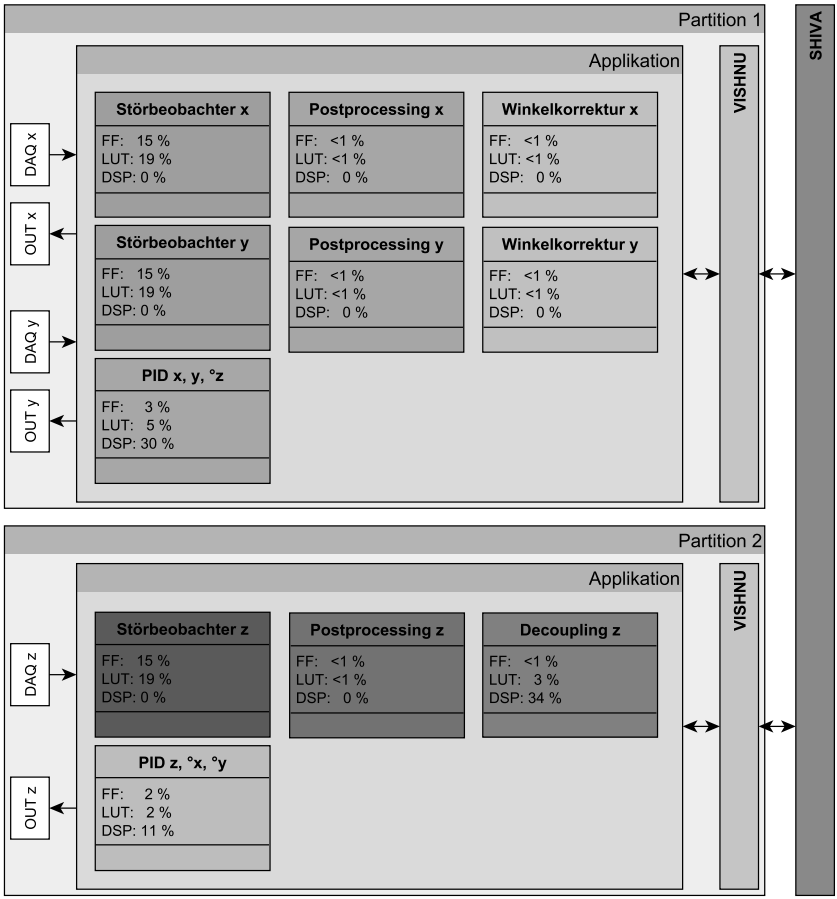


Abbildung A.12: Struktur und Schedule der 1-Chip-Realisierung *1Chp-ResMin* des Trajektorienfolgeregelungssystems.


Partition 1
Applikation
VISHNU
SHIVA
DAQ x
OUT x
DAQ y
OUT y
Störbeobachter x
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing x
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur x
FF: <1 %
LUT: <1 %
DSP: 0 %
Störbeobachter y
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing y
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur y
FF: <1 %
LUT: <1 %
DSP: 0 %
PID x, y, °z
FF: 3 %
LUT: 5 %
DSP: 30 %
Partition 2
Applikation
VISHNU
DAQ z
OUT z
Störbeobachter z
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing z
FF: <1 %
LUT: <1 %
DSP: 0 %
Decoupling z
FF: <1 %
LUT: 3 %
DSP: 34 %
PID z, °x, °y
FF: 2 %
LUT: 2 %
DSP: 11 %

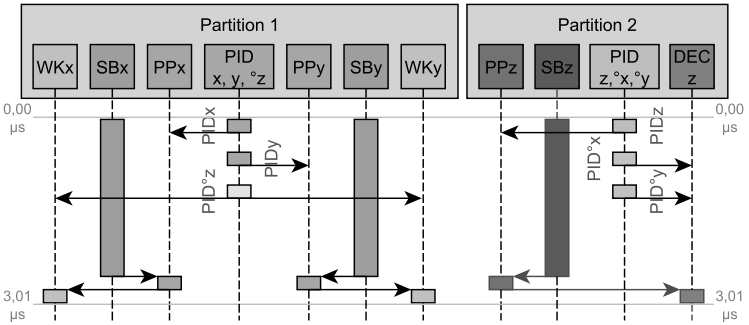
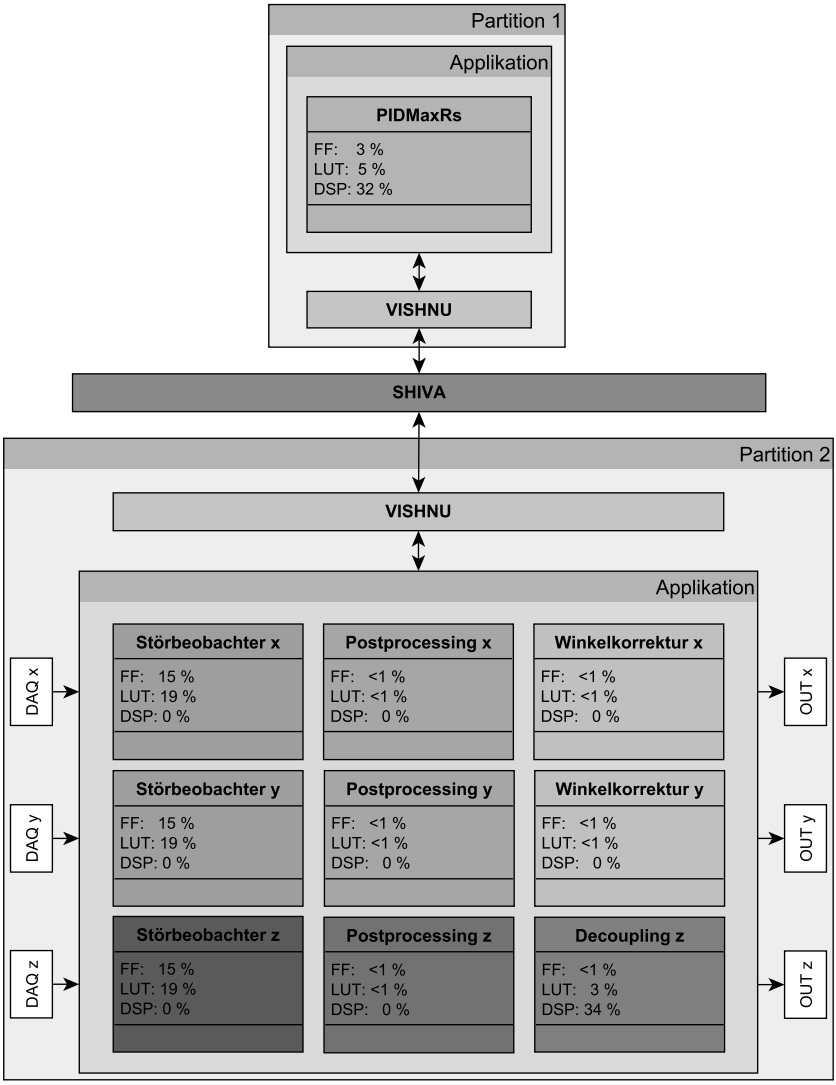


Abbildung A.13: Struktur und Schedule der 2-Chip-Realisierung *2Chp-(XY)(Z)* des Trajektorienfolgeregelungssystems.


Partition 1
Applikation
PIDMaxRs
FF: 3 %
LUT: 5 %
DSP: 32 %
VISHNU
SHIVA
Partition 2
VISHNU
Applikation
DAQ x
DAQ y
DAQ z
Störbeobachter x
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing x
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur x
FF: <1 %
LUT: <1 %
DSP: 0 %
Störbeobachter y
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing y
FF: <1 %
LUT: <1 %
DSP: 0 %
Winkelkorrektur y
FF: <1 %
LUT: <1 %
DSP: 0 %
Störbeobachter z
FF: 15 %
LUT: 19 %
DSP: 0 %
Postprocessing z
FF: <1 %
LUT: <1 %
DSP: 0 %
Decoupling z
FF: <1 %
LUT: 3 %
DSP: 34 %
OUT x
OUT y
OUT z

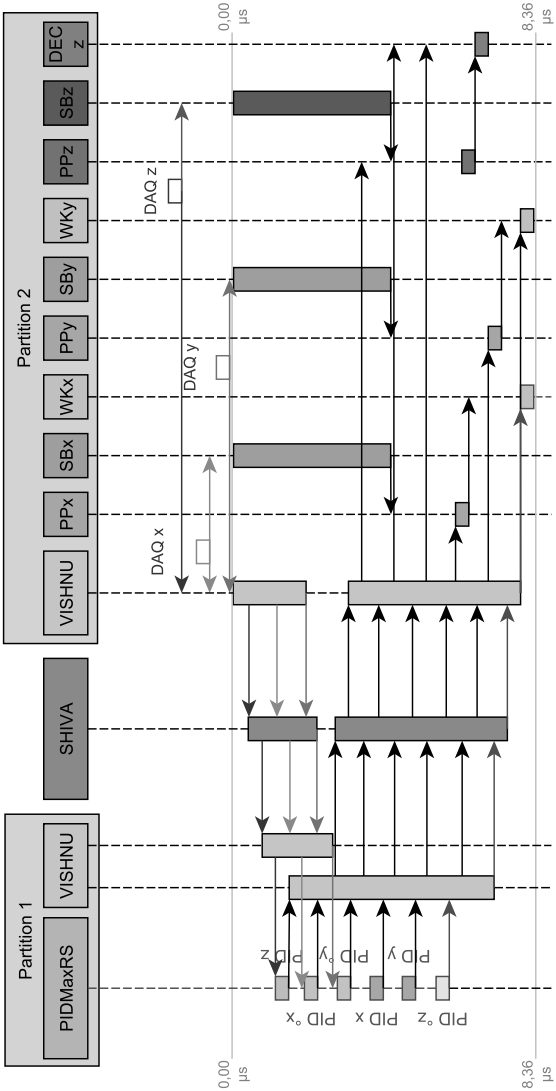


Abbildung A.14: Schedule der 2-Chip-Realisierung *2Chp-(XYZ)(6xPID)* des Trajektorienfolgeregelungssystems.

A.3.3 Modellkomplexität

Regler-Modell	Modell-blöcke	davon automatisch generiert
plattformunabhängiges Spezifikationsmodell	867	-
HDL-konformes Komponentenmodell	1979	-
latenzoptimiertes Komponentenmodell	2012	-
ressourcenminimaler Ein-Chip-Entwurf	3000	1431 (47%)
latenz-/ressourcen-minimaler 2-Chip-Entwurf	6091	4214 (69%)
latenz-/ressourcen-minimaler 3-Chip-Entwurf	6325	3287 (52%)

Tabelle A.7: Analyse der Modellkomplexität des verteilten Entwurfs

Abkürzungsverzeichnis

A/D	Analog-zu-Digial
AFM	Atomic Force Microscopy
ASIC	Application-specific Integrated Circuit
ASIP	Application-specific Instruction Set Processor
ASP	Application-specific Processor
CIS	Computing-in-Space
CIT	Computing-in-Time
CTR	Compile-time Reconfiguration
D/A	Digital-zu-Analog
DEMUX	Demultiplexer
DFG	Datenflussgraph
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor
DUT	Device-under-Test
FF	Flip-Flop
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine (endlicher Automat, Zustandsmaschine)
GALS	Globally Asynchronous Locally Synchronous
HDL	Hardware Description Language
HIL	Hardware-in-the-Loop
HMI	Human-Machine-Interface
HSDF	Homogeneous Synchronous Data Flow

I/O	Input/Output
IP	Intellectual Property
LUT	Look-up Table
MUX	Multiplexer
NPMM	Nano-Positionier- und -Messmaschine
NUMA	Non-uniform Memory Access
PID	Proportional-Intregral-Differential
PIM	Platform-independent Model
PSM	Platform-specific Model
PXI	PCI eXtension for Instrumentation
RTL	Register-Transfer Level
RTR	Run-time Reconfiguration
SDF	Synchronous Data Flow
SFB	Sonderforschungsbereich
SoB	System-on-Board
SoC	System-on-Chip
SiP	System-in-Package
VHDL	VHSIC Hardware Description Language

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AIS09] Agarwal, Ankur ; Iskander, Cyril ; Shankar, Ravi: Survey of network on chip (Noc) Architectures & Contributions. In: *Journal of Engineering, Computing and Architecture* 3 (2009), Nr. 1

[Alt11] Altera Corporation: *DSP Builder Handbook Volume 1 : Introduction to DSP Builder*. `http://www.altera.com/literature/hb/dspb/hb_dspb_intro.pdf`. Version: 2011

[Amt10] Amthor, Arvid: Modellbasierte Regelung von Nanopositionier- und Nanomessmaschinen. In: *Fortschritt-Berichte Reihe 8: Meß-, Steuerungs- und Regelungstechnik, Band 1179*. Düsseldorf : VDI Verlag, 2010

[Ash08] Ashenden, Peter J.: *Digital Design - An Embedded Systems Approach using VHDL*. Burlington, USA : Morgan Kaufman, 2008

[AZA08] Amthor, Arvid ; Zschäck, Stephan ; Ament, Christoph: Position control on nanometer scale based on an adaptive friction compensation scheme. In: *2008 34th Annual Conference of IEEE Industrial Electronics*. Orlando, USA : IEEE, 2008, S. 2568–2573

[BBCM06] Balmelli, L ; Brown, D ; Cantor, M ; Mott, M: Model-driven systems development. In: *IBM Systems Journal* 45 (2006), Nr. 3, S. 569–585

[BC12] Barry, Peter ; Crowley, Patrick: *Modern Embedded Computing*. Waltham, USA : Morgan Kaufman, 2012

[BFRV92] Brown, S. ; Francis, R. ; Rose, J. ; Vranesic, Z.: *Field Programmable Gate Arrays*. New York, USA : Springer, 1992

[BM06] Bjerregaard, Tobias ; Mahadevan, Shankar: A survey of research and practices of Network-on-chip. In: *ACM Computing Surveys* 38 (2006), Juni, Nr. 1, S. 1–51

[BM11] Brandel, Oliver ; Müller, Marcus: *GECKO3STACK*. `http://labs.ti.bfh.ch/gecko/wiki/systems/gecko3stack`. Version: 2011

[Bol02] Bolsens, Ivo: Challenges and opportunities for FPGA platforms. In: *FPL'02 Proceedings of the 12th International Conference on Field-Programmable Logic and Applications - Reconfigurable Computing Is Going Mainstream*. London, UK : Springer, 2002, 391–392

[Bot10] Both, Johannes: *Anwendungsspezifische Kommunikationsstrukturen in Multi-FPGA-Systemen*, Technische Universität Ilmenau, Bachelorarbeit, 2010

[Bot11] Both, Johannes: *VISHNU II*, Technische Universität Ilmenau, Projektarbeit, 2011

[Bot12] Both, Johannes: *Ressourcenminimierte Komponenten im modellbasierten Logik-Entwurf - Automatisierte Generierung von anwendungsspezifischen Zeitmultiplex-Strukturen*, Technische Universität Ilmenau, Masterarbeit, 2012

[BR97] Bringmann, Oliver ; Rosenstiel, Wolfgang: *Resource sharing in hierarchical synthesis*. 1997. – 318–325 S.

[Bra10] Brandel, Oliver: *GECKO3STACK - Aufbau und Inbetriebnahme eines Multi-FPGA-Systems*, Technische Universität Ilmenau, Diplomarbeit, 2010

[CBP+05] Carloni, Luca P. ; Bernardinis, Fernando D. ; Pinello, Claudio ; Sangiovanni-Vincentelli, Alberto L. ; Sgroi, Marco: Platform-Based Design for Embedded Systems. In: Zurawski, R (Hrsg.): *The Embedded Systems Handbook*. Florida : CRC Press, August 2005, Kapitel 22

[CDSVS02] Carloni, L.P. ; De Bernardinis, F. ; Sangiovanni-Vincentelli, A.L. ; Sgroi, M.: The art and science of integrated systems design. In: *Proceedings of the 28th European Solid-State Circuits Conference, 2002. ESSCIRC 2002*. Florence, Italy, 2002, 25 – 36

[CH06] Cofer, R. C. ; Harding, Ben: *Rapid System Prototyping with FPGAs*. Oxford, UK : Elsevier, 2006

[CM05] Caspi, Paul ; Maler, Oded: From control loops to real-time programs. In: *Handbook of networked and embedded control systems*. Boston, USA : Birkhäuser, 2005, S. 395–418

[CMSSV99] Carloni, L.P. ; McMillan, K.L. ; Saldanha, A. ; Sangiovanni-Vincentelli, A.L.: A methodology for correct-by-construction latency insensitive design. In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, IEEE, 1999, 309–315

[DB04] Daene, Bernd ; Berger, Falk: A Multiprocessor DSP System for a High Throughput Control Application. In: *EDERS-2004: The European DSP Education and Research Symposium, 16th November 2004, Birmingham, UK*, 2004

[DF05] Däne, Bernd ; Fengler, Wolfgang: A Case Study for Partitioned Modelling of a Control System. In: *The 24rd IASTED International Conference on Modelling, Identification, and Control. Innsbruck, Austria, Feb. 16-18* (2005), S. 525–530

[dSP12a] dSPACE GmbH: *Real-Time Interface (RTI).* `http://www.dspace.com/de/gmb/home/products/sw/impsw/realtimeinterf.cfm`. Version: 2012

[dSP12b] dSPACE GmbH: *RTI FPGA Programming Blockset.* `http://www.dspace.com/de/gmb/home/products/sw/impsw/rti_fpga_programming_blockset.cfm`. Version: 2012

[EPT99] Eisenring, M. ; Platzner, M. ; Thiele, L.: Communication synthesis for reconfigurable embedded systems. In: Lysaght, Patrick (Hrsg.) ; Irvine, James (Hrsg.) ; Hartenstein, Reiner (Hrsg.): *Field Programmable Logic and Applications - Lecture Notes in Computer Science.* Berlin/Heidelberg : Springer, 1999, S. 205–214

[ETZ00] Eisenring, M. ; Thiele, L. ; Zitzler, E.: Conflicting criteria in embedded system design. In: *IEEE Design \& Test of Computers* 17 (2000), Nr. 2, S. 51–59

[FCV05] Fang, Zhengwei ; Carletta, J.E. ; Veillette, R.J.: A methodology for FPGA-based control implementation. In: *IEEE Transactions on Control Systems Technology* 13 (2005), November, Nr. 6, S. 977–987

[GAGS09] Gajski, Daniel D. ; Abdi, Samar ; Gerstlauer, Andreas ; Schirner, Gunar: *Embedded System Design: Modeling, Synthesis and Verification.* New York, USA : Springer, 2009

[GK83] Gajski, Daniel D. ; Kuhn, Robert H.: Guest Editors' Introduction: New VLSI Tools. In: *Computer* 16 (1983), Dezember, Nr. 12, S. 11–14

[GR94] Gajski, D.D. ; Ramachandran, L.: Introduction to high-level synthesis. In: *IEEE Design & Test of Computers* 11 (1994), Januar, Nr. 4, S. 44–54

[Gro08] Grout, Ian: *Digital Systems Design with FPGAs and CPLDs*. Oxford, UK : Elsevier, 2008

[Hau02] Hausotte, Tino: *Nanopositionier- und Nanomessmaschine*, Technische Universität llmenau, Dissertation, 2002

[HCA+12] Hadjis, Stefan ; Canis, Andrew ; Anderson, Jason H. ; Choi, Jongsok ; Nam, Kevin ; Brown, Stephen ; Czajkowski, Tomasz: Impact of FPGA architecture on resource sharing in high-level synthesis. In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*. New York, USA : ACM Press, Februar 2012, S. 111

[Hei12] Heinz, Christian: *Entwicklung einer seriellen High-Speed-Verbindung zur Intermodulkommunikation in einem FPGA-Verbundsystem*, Technische Universität Ilmenau, Projektarbeit, 2012

[HNN06] Hansson, Hans ; Nolin, Mikael ; Nolte, Thomas: Real-Time in Embedded Systems. In: Zurawski, Richard (Hrsg.): *Embedded Systems Handbook*. Boca Raton, USA : Taylor&Francis Group, 2006, S. 2.1–2.32

[HS06] Henzinger, T. ; Sifakis, Joseph: The embedded systems design challenge. In: Misra, Jayadev (Hrsg.) ; Nipkow, Tobias (Hrsg.) ; Sekerinski, Emil (Hrsg.): *FM 2006: Formal Methods*. Berlin/Heidelberg : Springer, 2006, S. 1–15

[HS07] Henzinger, Thomas A. ; Sifakis, Joseph: The Discipline of Embedded Systems Design. In: *Computer* 40 (2007), Oktober, Nr. 10, S. 32–40

[IAB06] IABG: *Das V-Modell XT*. Ottbrunn, 2006

[ITR11] ITRS: *International Technoloy Roadmap for Semiconductors - 2011 Edition - DESIGN*. `http://www.itrs.net/Links/2011ITRS/2011Chapters/2011Design.pdf`. Version: 2011

[JDKR97] Jerraya, Ahmed ; Ding, Hong ; Kission, Polen ; Rahmouni, Maher: *Behavioural Synthesis and Component Reuse with VHDL*. Norwell, USA : Kluwer Academic Publishers, 1997

[Jes12] Jesko, Christian: *Shared-Memory-Kommunikation in einem Multi-FPGA-System - Abstraktion einer Ringtopologie*, Technische Universität Ilmenau, Studienarbeit, 2012

[JMH+01] Jäger, G. ; Manske, E. ; Hausotte, T. ; H.-J., Büchner ; Grünwald, R ; Schott, W.: Nanomeasuring Technology - Nanomeasuring Machine. In: *16th Annual Meeting of ASPE*. Crystal City, Arlington, Virginia, 2001, S. 23–27

[JMHB00] Jäger, G. ; Manske, E. ; Hausotte, T. ; Büchner, H.-J.: Nanomeßmaschinen zur abbefehlerfreien Koordinatenmessung. In: *Technisches Messen* 67 (2000), Nr. 7-8, S. 319–323

[JMHS02] Jäger, G. ; Manske, E. ; Hausotte, T. ; Schott, W.: Operation and analysis of a nanopositioning and nanomeasuring machine. In: *Proceedings of the 17th Annual Meeting of the American Society for Precision Engineering*. St. Louis, USA, 2002, 299–304

[KB02] Keating, M. ; Bricaud, P.: *Reuse Methodology Manual*. 3. Auflage. Norwell, USA : Kluwer Academic Publishers, 2002

[KGGV07] Krstic, Milos ; Grass, Eckhard ; Gürkaynak, Frank K. ; Vivet, Pascal: Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook. In: *IEEE Design & Test of Computers* 24 (2007), September, Nr. 5, S. 430–441

[KNRSV00] Keutzer, K. ; Newton, A.R. ; Rabaey, J.M. ; Sangiovanni-Vincentelli, A.: System-level design: orthogonalization of concerns and platform-based design. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19 (2000), Nr. 12, S. 1523–1543

[Kra04] Krasner, Jerry: *Model-based design and beyond: Solutions for today's embedded systems requirements*, EMBEDDED MARKET FORECASTERS American Technology International, Report, 2004. `http://www.embedded-forecast.com`

[Kra10] Krahn, Alexander: *Intermodulkommunikation in einem Multi-FPGA-System SHIVA - Shared memory Interface for Versatile Application*, Technische Universität Ilmenau, Bachelorarbeit, 2010

[KSH07] Kulmala, Ari ; Salminen, Erno ; Hamaalainen, T.: Evaluating large system-on-chip on multi-FPGA platform. In: Vassiliadis, Stamatis (Hrsg.) ; Berekovic, Mladen (Hrsg.) ; Hämäläinen, Timo (Hrsg.): *Embedded Computer Systems: Architectures, Modeling, and Simulation - Lecture Notes in Computer Science*. Springer, 2007, S. 179–189

[KTR07] Kuon, Ian ; Tessier, Russell ; Rose, Jonathan: FPGA Architecture: Survey and Challenges. In: *Foundations and Trends in Electronic Design Automation* 2 (2007), Februar, Nr. 2, S. 135–253

[LAD04] Lemaitre, J ; Alliot, Sylvain ; Deprettere, Ed: On the (re-) use of IP-components in re-configurable platforms. In: Vassiliadis, Stamatis (Hrsg.): *Computer Systems: Architectures, Modeling, and Simulation - Lecture Notes in Computer Science*. Berlin/Heidelberg : Springer, 2004, S. 264–273

[LAD06] Lemaitre, Jérôme ; Alliot, Sylvain ; Deprettere, Ed: Requirements for Interfacing IP-Components in Re-configurable Platforms. In: *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 43 (2006), Juni, Nr. 2-3, S. 173–184

[LBT+11] Leong, C. ; Bexiga, V. ; Teixeira, J. P. ; Bugalho, R. ; Ferreira, M. ; Rodrigues, P. ; Silva, J. C. ; Lousã, P. ; Varela, J. ; Teixeira, I. C.: Automatic adjustment of a medical imaging data acquisition system to unknown delays in the input communication channels. In: *Analog Integrated Circuits and Signal Processing* 70 (2011), September, Nr. 2, S. 213–227

[LP06] Lavagno, Luciano ; Passerone, Claudio: Design of Embedded Systems. In: Zurawski, Richard (Hrsg.): *Embedded Systems Handbook*. Boca Raton, USA : Taylor&Francis Group, 2006, S. 3.1–3.24

[LSYM08] Lee, Jason ; Shannon, Lesley ; Yedlin, Matthew J. ; Margrave, Gary F.: A multi-FPGA application-specific architecture for accelerating a floating point Fourier Integral Operator. In: *2008 International Conference on Application-Specific Systems, Architectures and Processors*, Ieee, Juli 2008, S. 197–202

[LTT11] Lau, Kung-Kiu ; Taweel, Faris M. ; Tran, Cuong M.: The W Model for Component-Based Software Development. In: *2011*

37th EUROMICRO Conference on Software Engineering and Advanced Applications, Ieee, August 2011, 47–50

[LZ06] Lavagno, Luciano ; Zurawski, Richard: Embedded Systems: Toward Networking of Embedded Systems. In: Zurawski, Richard (Hrsg.): *Embedded Systems Handbook*. Boca Raton, USA : Taylor&Francis Group, 2006, S. 1.1–1.17

[Mar11] Marwedel, Peter: *Embedded Systems Design - Embedded Systems Foundation of Cyber-physical Systems*. 2. Auflage. Heidelberg : Springer, 2011

[MBKF11] Müller, Marcus ; Brandel, Oliver ; Krahn, Alexander ; Fengler, Wolfgang: Shared-memory communication in distributed SoCs on multi-FPGA systems. In: Edacentrum (Hrsg.): *edaWorkshop 11 Proceedings, Dresden, May 10-12, 2011*. Berlin : VDE Verlag GmbH, 2011, S. 57–62

[MC07] Monmasson, E. ; Cirstea, M.N.: FPGA Design Methodology for Industrial Control Systems - A Review. In: *IEEE Transactions on Industrial Electronics* 54 (2007), August, Nr. 4, S. 1824–1842

[MFAA09] Müller, Marcus ; Fengler, Wolfgang ; Amthor, Arvid ; Ament, Christoph: Model-driven development and multi-processor implementation of a dynamic control algorithm for nanopositioning and nanomeasuring machines. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 223 (2009), S. 417–429

[MG09] Maurmaier, Mathias ; Göhner, Peter: Model-driven Development in Industrial Automation - Automating the Development of Industrial Automation Systems using Model Transformations. In: *ICINCO 2009 - Sixth lnternational Conference on Informatics in Control, Automation and Robotics*, 2009, S. 244–249

[Mic11] MicroLab: *Gecko3 System-on-Chip Codesign Environment*. `http://labs.ti.bfh.ch/gecko/wiki/`. Version: 2011

[MKG+13] Müller, Marcus ; Klöckner, Johannes ; Gushchina, Irina ; Pacholik, Alexander ; Fengler, Wolfgang ; Amthor, Arvid: Performance evaluation of platform-specific implementations of numerically complex control designs for nano-positioning applications. In: *International Journal of Embedded Systems (IJES)* 5 (2013), Nr. 1-2, S. 95–105

[MM03] Miller, Joaquin ; Mukerji, Jishnu: *MDA Guide Version 1.0. 1.* `http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf`. Version: 2003

[MS06] Mitić, Milica ; Stojčev, Mile: An overview of on-chip buses. In: *Facta universitatis - series: Electronics and Energetics* 19 (2006), Nr. 3, S. 405–428

[MSCL06] Mak, T. S. T. ; Sedcole, P. ; Cheung, Peter Y. K. ; Luk, Wayne: On-FPGA communication architectures and design factors. In: *International Conference on Field Programmable Logic and Applications FPL'06.*, 2006, S. 1–8

[MSF12] Müller, Marcus ; Schwannecke, Hans-Christian ; Fengler, Wolfgang: From Control Design to FPGA Implementation. In: Prof. Subhas Chakravarty (Hrsg.): *Technology and Engineering Applications of Simulink.* inTech, 2012, S. 129–148

[Nat12a] National Instruments Corporation: *NI LabVIEW FPGA Module.* `http://www.ni.com/labview/fpga/`. Version: 2012

[Nat12b] National Instruments Corporation: *NI LabVIEW Real-Time Module.* `http://www.ni.com/labview/realtime`. Version: 2012

[NM10] Nicolescu, Gabriela ; Mosterman, Pieter J.: *Model-based Design for Embedded Systems.* Boca Raton, USA : Taylor&Francis Group, 2010

[PKM+11] Pacholik, A. ; Klöckner, J. ; Müller, M. ; Gushchina, I. ; Fengler, W.: LiSARD: LabVIEW Integrated Softcore Architecture for Reconfigurable Devices. In: *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig '11).* Cancun, Mexico : IEEE Computer Society CPS, 2011, S. 442–447

[PKMF11a] Pacholik, A. ; Klöckner, J. ; Müller, M. ; Fengler, W.: LiSARD : Programmierbarer Rechenkern für rechenintensive Echtzeitdatenverarbeitung mit PXI-RIO. In: Jamal, R (Hrsg.) ; Heinze, R (Hrsg.): *Virtuelle Instrumente in der Praxis - Begleitband zum 16. VIP-Kongress.* München, 2011, S. 217–221

[PKMF11b] Percle, Brandon ; Klöckner, Johannes ; Manske, Eberhard ; Fengler, Wolfgang: Signal and data processing in high-precision measuring machines - a case study of NPMM-200. In: *Innovation in mechanical engineering - shaping the future*

/ Internationales Wissenschaftliches Kolloquium. Technische Universität Ilmenau, Faculty of Mechanical Engineering ; 56 (Ilmenau) : 2011.09.12-16, 2011, S. 16

[Poo10] Poovendran, R: Cyber-Physical Systems : Close Encounters Between Two Parallel Worlds. In: *Proceedings of the IEEE* 98 (2010), Nr. 8, S. 1363–1366

[PP05] Pretschner, Alexander ; Phillips, Jan: Methodological Issues in Model-Based Testing. In: Broy, Manfred (Hrsg.) ; Jonsson, Bengt (Hrsg.) ; Katoen, Joost-Pieter (Hrsg.) ; Leucker, Martin (Hrsg.) ; Pretschner, Alexander (Hrsg.): *Model-based testing of Reactive Systems*. Heidelberg : Springer-Verlag, 2005, S. 281–291

[PS09] Pasricha, Ruchi ; Sharma, S: An FPGA-Based Design of Fixed-Point Kalman Filter. In: *DSP Journal* 9 (2009), Nr. 1, S. 1–9

[PSG+12] Poovendran, R ; Sampigethhaya, K ; Gupta, S ; Kumar, S ; Lee, I ; Prasad, K V. ; Corman, D ; Paunicka, J L.: Special Issue on Cyber Physical Systems. In: *Proceedings of the IEEE* 100 (2012), Nr. 1, S. 6–12

[QBJ05] Quan, Wen ; Bo, Liu ; Jianmin, He: Platform-based synthesis design methodology for system-on-chips. In: *6th International Conference on Electronic Packaging Technology*, IEEE, 2005, 153–156

[RC03] Royal, Andrew ; Cheung, Peter Y.: Globally Asynchronous Locally Synchronous FPGA Architectures. In: Cheung, Peter (Hrsg.) ; Constantinides, George A. (Hrsg.): *Field Programmable Logic and Application - Lecture Notes in Computer Science* Bd. 2778. Berlin/Heidelberg : Springer, 2003, S. 355–364

[RS11] Radojevic, Ivan ; Salcic, Zoran: *Embedded Systems Design based on Formal Models of Computation*. Heidelberg : Springer, 2011

[Rus11] Rushton, Andrew: *VHDL for Logic Synthesis*. 3. Auflage. Chichester, UK : John Wiley & Sons, 2011

[Sch12] Schwesinger, Folker: *Plattformspezifischer Entwurf für Multi-FPGA-Systeme - Design-Space-Exploration einer komplexen regelungstechnischen Anwendung*, Technische Uiversität Ilmenau, Masterarbeit, 2012

[SIO07] SIOS Messtechnik GmbH: *Nanopositionier- und Nanomessmaschine*. `http://www.sios.de/DEUTSCH/PRODUKTE/NMM.HTM/NPMM.pdf`. Version: 2007

[SKK12] Sztipanovits, J ; Koutsoukos, Xenofon ; Karsai, Gabor: Toward a Science of Cyber-Physical System Integration. In: *Proceedings of the IEEE* 100 (2012), Nr. 1, S. 29–44

[SLS10] Stepniewska, Marta ; Luczak, Adam ; Siast, Jakub: Network-on-Multi-Chip (NoMC) for Multi-FPGA Multimedia Systems. In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, IEEE, September 2010, 475–481

[SQC02] Schäfer, B. C. ; Quigley, S. F. ; Chan, A. H. C.: Scalable implementation of the Discrete Element Method on a Reconfigurable Computing platform. In: *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream - Lecture Notes in Computer Science*. Berlin, Heidelberg : Springer, 2002, S. 925–934

[SS10] Sass, Ron ; Schmidt, Andrew G.: *Embedded Systems Design with Platform FPGAs*. Burlington, MA, USA : Morgan Kaufman, 2010

[SVCDS04] Sangiovanni-Vincentelli, A ; Carloni, L ; De Bernardinis, Fernando ; Sgroi, Marco: Benefits and challenges for platform-based design. In: *Proceedings of the 41st annual Design Automation Conference DAC '04*, 2004, 409–414

[SWM+06] Saleh, R. ; Wilton, S. ; Mirabbasi, S. ; Hu, A. ; Greenstreet, M. ; Lemieux, G. ; Pande, P.P. ; Grecu, C. ; Ivanov, A.: System-on-Chip: Reuse and Integration. In: *Proceedings of the IEEE* 94 (2006), Juni, Nr. 6, S. 1050–1069

[The10] The MathWorks, Inc.: *Simulink Fixed Point 6 User Guide*. `http://www.mathworks.de/help/pdf_doc/fixpoint/fp_blks.pdf`. Version: 2010

[The11a] The MathWorks, Inc.: *Simulink HDL Coder User's Guide R2011b*. Juni 2011

[The11b] The MathWorks Inc.: *MATLAB/Simulink Documentation*. `http://www.mathworks.com/help/toolbox/simulink/`. Version: 2011

[VG02] Vahid, Frank ; Givargis, Tony: *Embedded System Design - A Unified Hardware/Software Approach*. New York, USA : John Wiley & Sons, 2002

[VHKBW09] Vogel-Heuser, Birgit ; Kegel, Gunther ; Bender, Klaus ; Wucherer, Klaus: Global Information Architecture for Industrial Automation. In: *Automatisierungstechnische Praxis (atp)* 51 (2009), Nr. 1, S. 108–115

[Vig10] Vigenshow, Uwe: *Testen von Software und Embedded Systems*. 2. Auflage. Heidelberg : dpunkt.verlag GmbH, 2010

[WH03] Wuttke, Heinz-Dietrich ; Henke, Karsten: *Schaltsysteme*. München : Pearson Studium, 2003

[Xil00] Xilinx, Inc.: *Xilinx Design Reuse Methodology for ASIC and FPGA Designers*. 2000

[Xil08] Xilinx, Inc.: *Synthesis and Simulation Design Guide*. 2008

[Xil09] Xilinx, Inc.: *Spartan-3 FPGA Family Data Sheet*. 2009

[Xil11] Xilinx, Inc.: *Xilinx System Generator for DSP User Guide*. 2011

[Yok05] Yokoyama, Takanori: An aspect-oriented development method for embedded control systems with time-triggered and event-triggered processing. In: *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS05)*, 2005, 302–311

[ZAM+10] Zschäck, Stephan ; Amthor, Arvid ; Müller, Marcus ; Klöckner, Johannes ; Ament, Christoph ; Fengler, Wolfgang: Integrated system development process for high-precision motion control systems. In: *2010 IEEE International Conference on Control Applications*. Yokohama : IEEE, September 2010, S. 344–350

[ZFHT12] Zebelein, Christian ; Falk, Joachim ; Haubelt, Christian ; Teich, Jürgen: A Model-based Inter-process Resource Sharing Approach for High-level Synthesis of Dataflow Graphs. In: *Electronic System Level Synthesis Conference (ESLsyn)*, 2012, S. 17–22

[ZHCY09] Zhou, Bo ; Hu, X. S. ; Chen, Danny Z. ; Yu, Cedric X.: A multi-FPGA accelerator for radiation dose calculation in cancer treatment. In: *2009 IEEE 7th Symposium on Application Specific Processors*, IEEE, Juli 2009, 70–79

[ZKA+12] Zschäck, Stephan ; Klöckner, Johannes ; Amthor, Arvid ; Ament, Christoph ; Fengler, Wolfgang: Regelung von Nanopositionier- und Nanomessmaschinen mittels einer modularen FPGA-Plattform. In: *3. Landshuter Symposium Mikrosystemtechnik*. Landshut, 2012, S. 267–276

[ZZ07] Zurbrügg, Matthias ; Zimmermann, Christoph: *Gecko3 - New generation of the Microlab HW / SW co-design Platform*, University of Applied Sciences Berne, Diploma thesis, 2007