Eine FPGA/DSP-Entwicklungsplattform für eingebettete audiosignalverarbeitende Echtzeitsysteme

vorgelegt von
Diplom-Ingenieur
Marco Beyer
aus Hennigsdorf

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr.-Ing. Heinrich Klar
Gutachter: Prof. Dr.-Ing. Hans-Ulrich Post
Gutachter: Prof. Dr.-Ing. Hans Liebig

Tag der wissenschaftlichen Aussprache: 12. Dezember 2003

Berlin 2003

D 83

Danksagung

Die vorliegende Promotionsarbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter unter der Leitung von Herrn Prof. Dr.-Ing. Hans-Ulrich Post am Institut für Technische Informatik der Technischen Universität Berlin, Fachgebiet Rechnertechnologie.

Ich danke Herrn Prof. Dr.-Ing. Hans-Ulrich Post für die zahlreichen wissenschaftlichen Anregungen, fachlichen Ratschläge und die vielen Diskussionen, Herrn Prof. Dr.-Ing. Hans Liebig für die Übernahme des zweiten Gutachtens, Herrn Prof. Dr.-Ing. Heinrich Klar für den Vorsitz in der Prüfungskommission, Dipl.-Inform. Carsten Gremzow für die vielen fachlichen Gespräche und die Unterstützung bei der Digitalfotografie, Dr.-Ing. Irenäus Schoppa für die Anregungen bei der Arbeit mit FrameMaker und Dr.-Ing. Thomas Flik, Dipl-Ing. Dirk Pflug, Dipl.-Ing. Sasa Vulinovic und Silvia Rabe für das Korrekturlesen der Arbeit.

Ganz besonders danke ich meiner Frau Antje für ihre Geduld und die moralische Unterstützung und meinem kleinen Sohn Julius, der zur willkommenen Abwechslung im Dissertationsalltag beitrug. Allen an dieser Stelle nicht genannten, die zum Gelingen dieser Arbeit beigetragen haben, gilt an dieser Stelle mein besonderer Dank.

Marco Beyer Berlin, im Dezember 2003

Kurzfassung (Abstract)

Die vorliegende Dissertation beschreibt den Entwurf einer Plattform für die Entwicklung von eingebetteten audiosignalverarbeitenden Echtzeitsystemen, bestehend aus der Kombination eines digitalen Signalprozessors (DSP) und einer programmierbaren applikationsspezifischen Hardware (FPGA). Für diese Plattform wurde eine integrierte FPGA/DSP-Entwicklungsumgebung implementiert. Besonderes Augenmerk richtet sich hierbei auf die Hardware/Software-CoVerifikation in Form einer Hardware/Software-CoSimulation mittels VHDL-Board-Level-Simulation und auf die Interface-Synthese, der automatisierten Generierung von Schnittstellen zwischen Hardware und Software.

Problem eines aus Hardware und Software bestehenden Systems ist die Verifikation der Hardware/Software-Schnittstellen und des Zusammenspiels von Hard- und Software im Gesamtsystem. Bestehende Ansätze verknüpfen in einer Hardware/Software-CoSimulation einen HDL-Simulator mit der Simulation der Software-Ausführung mit Hilfe einer synchronisierenden Software-Komponente. In dieser Arbeit wird der Ansatz untersucht, alle Teile des Systems gemeinsam in einer VHDL-Simulation zu verifizieren, wobei für den DSP, anstelle der oft in anderen Arbeiten eingesetzten Interface-Modelle (Busfunktionsmodelle), ein zyklengetreues VHDL-Modell des DSP-Core inklusive des Zeitverhaltens der Schnittstellen verwendet wird. Übergreifend erlaubt die zyklengenaue Simulation ein frühzeitiges Erkennen von Fehlern, die insbesondere bei der Simulation von sicherheitskritischen Anwendungen auf höheren Abstraktionsebenen nicht identifiziert werden können. Damit der Software-Entwickler nicht auf die gewohnten Debug-Möglichkeiten für die Software des DSP verzichten muss, ist im Rahmen dieser Arbeit für den HDL-Simulator ModelSim von Mentor Graphics eine entsprechende graphische Erweiterung implementiert worden. Dies ist eine wichtige Voraussetzung für das effektive Testen eines Algorithmus, der partitioniert in Hardware und Software arbeitet.

Ein weiterer Bestandteil der Arbeit behandelt die Interface-Synthese und dabei die automatische Generierung von synthesefähigen VHDL-Modulen für die hardwareseitige Schnittstellenkommunikation und zugehöriger Softwarefunktionen, formuliert in C/Assembler, aus einer abstrakten Beschreibung in XML. Als Eingabesprache eignet sich XML aufgrund des plattform- und softwareunabhängigen Austausches von Daten zwischen verschiedenen Programmen und Rechnern in einem einheitlichen, allgemein verwendbaren, herstellerunabhängigen Format. Mit Hilfe von Style-Sheets können XML-Dokumente ebenso von Web-Browsern dargestellt und ausgedruckt werden und daher auch bei der Dokumentation eine wichtige Rolle spielen.

Die Tauglichkeit des FPGA/DSP-Entwicklungssystems für die Entwicklung von Anwendungen aus dem audiosignalverarbeitenden Bereich konnte an einem praktischen Beispiel eines Prototypen für einen MPEG1-Layer3-Player erfolgreich demonstriert werden.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
ASSP	Application Specific Standard Product
ASPP	Application Specific Programmable Product
AT	Advanced Technology
ATA	AT Attachment
ATAPI	AT Attachment Packet Interface
BFM	Bus Functional Model
CAD	Computer Aided Design
CFG	Control Flow Graph
CFSM	Co-design Finite State Machine
CLB	Configurable Logic Block
CPLD	Complex PLD
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DES	Data Encryption Standard
DFT	Discrete Fourier Transform
DLL	Dynamic Link Library/Digital-Locked Loop
DMA	Direct Memory Access
DSP	Digital Signal Processor/Digital Signal Processing
DTD	Document Type Definition
EDA	Electronic Design Automation
EEPROM	Electrically Erasable ROM
EPGA	Embedded FPGA Core
EPP	Enhanced Parallel Port
EPROM	Erasable Programmable ROM
FIFO	First-In First-Out
FIR	Finite Impulse Response
FFT	Fast Fourier Transformation
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
FPSLIC	Field Programmable System Level Integration Circuit
FSM	Finite State Machine

GPP	General Purpose Processor
GTL	Gunning Transceiver Logic
HDL	Hardware Description Language
HLL	High-Level Language
HTML	Hypertext Markup Language
IBIS	Input/Output Buffer Information Specification
ICE	In-Circuit Emulation
IDE	Integrated Design Environment/Integrated Drive Electronics
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
ILP	Instruction Level Parallelism
IP	Intellectual Property
IPC	Interprocess Communication
ISO	International Standard Organization
ISP	In System Programmability
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator
JTAG	Joint Test Action Group
LC	Logic Cell
LCD	Liquid Crystal Display
LSB	Least Significant Bit
LUT	Look-Up Table
LVTTL	Low Voltage TTL
MAC	Multiply-And-Accumulate
MFLOPS	Million Floating-Point Operations Per Second
MIPS	Million Instructions Per Second
MPEG	Moving Picture Experts Group
MOPS	Million Operations Per Second
MSB	Most Significant Bit
NRE	Non-Recurrent Engineering
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCM	Pulse Code Modulation
PLD	Programmable Logic Device
PLI	Programming Language Interface

X

PLL	Phase-Locked Loop
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTK	Real-Time Kernel
RTL	Register Transfer Level
RTOS	Real-Time OS
SHARC	Super Harvard Architecture
SDF	Standard Delay Format/Synchronous Data Flow
SDK	Software Developer's Kit
SGML	Standard Generalized Markup Language
SIA	Semiconductor Industry Association
S/PDIF	Sony/Philips Digital Interface Format
SoC	System-on-Chip
SoPC	System-on-programmable-Chip
SPLD	Simple PLD
SRAM	Static RAM
STG	Signal Transition Graph
TCL	Tool Command Language
TK	Tool Kit
TTL	Transistor-Transistor Logic
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHDL-AMS	VHDL For Analog And Mixed Signals
VHPI	VHDL PLI
VHSIC	Very High-Speed Integrated Circuit
VITAL	VHDL Initiative Towards ASIC Libraries
VLIW	Very Long Instruction Word
VSIA	Virtual Socket Interface Alliance
W3C	World Wide Web Consortium
WAV	Wave Audio File Format
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Die Anforderungen an die Leistungsfähigkeit der eingebetteten elektronischen Systeme steigen stetig. Gefordert werden ein geringer Platzbedarf, ein geringer Stromverbrauch, eine hohe Taktrate sowie eine hohe Zuverlässigkeit. Die zur Verfügung stehende Zeit für die Entwicklung eines elektronischen Systems verkürzt sich dagegen kontinuierlich aufgrund des großen Konkurrenzdruckes in diesem Marktsegment und immer kürzerer Produktlebenszyklen. Ein zu spät auf dem Markt erhältliches Produkt kann enorme Auswirkungen auf den Erfolg und damit auf den Umsatz haben (Time-to-Market). Die Entwicklung elektronischer Systeme, die durch wachsende Systemkomplexitäten und die genannten Anforderungen gekennzeichnet ist, kann nur durch eine strukturierte Vorgehensweise beherrschbar und mittels (teil-)automatisierter Werkzeuge realisierbar gestaltet werden.

Die Einsatzgebiete für eingebettete Systeme und damit gekoppelte Entwurfsthematiken – HW/SW-CoDesign, HW/SW-CoVerifikation etc. – lassen sich in allen Bereichen finden, in denen elektronische Systeme den Alltag der Menschen bestimmen. Im Bereich der Unterhaltungselektronik werden sie z. B. in digitalen Kameras, CD- und DVD-Playern eingesetzt. Im Bereich der Telekommunikation sind sie in digitalen Endgeräten, wie Mobiltelefonen, oder auch in den Vermittlungsstellen der Telekommunikationsunternehmen zu finden. Auch im Automobilbereich kommen eingebettete Systeme bei der Motorregelung, der Bremsensteuerung oder der Fahrzeugnavigation zum Einsatz.

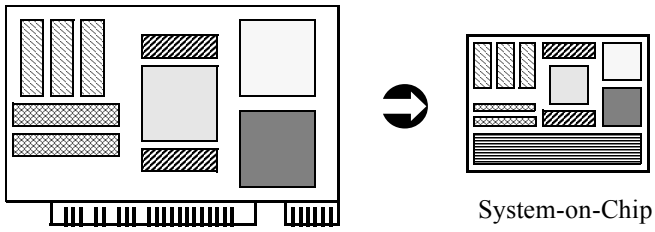


Bild 1-1. Gegenwärtiger Trend: System-on-Chip (SoC)

Die nahe Zukunft liegt in eingebetteten **Ein-Chip-Systemen** (**SoC** - System-on-Chip), die spezifische Prozessorarchitekturen mit eingebetteter Software und zusätzlichen Co-Prozessoren integrieren (Bild 1-1). Ein SoC bedeutet immer eine hohe Transistorintegration in einem Chip, eine große Anzahl an I/O-Pins, hohe Taktraten und eine kurze Zeit zur Markteinführung. Die Vorteile sind die geringe Größe, das geringe Gewicht, die niedrigen Kosten und die geringe Verlustleistung. Der Weg zur Integration der diskreten ICs in ein SoC geht über IPs (Intellectual Property): Firm-, Soft- und Hard-IPs.

Im Jahr 1999 enthielten ASICs (Application Specific Integrated Circuit) 150 KBit Gatter oder 400 KBit Speicher und die Entwürfe umfassten typisch 10.000 Zeilen VHDL-Code.

Aktuell (2002) können 500 KBit Gatter bzw. 1 MBit Speicher in einen ASIC bei 100.000 Zeilen VHDL-Code integriert sein. Wirtschaftlich möglich sind jedoch fünfmal so große und komplexe ICs [73]. Es handelt sich hierbei um den sogenannten **„Productivity Gap“**, d. h., zwischen der durch die Technologie möglichen und durch den Entwickler in einem vernünftigen Zeitrahmen realisierbaren Komplexität der Anwendungen und Architekturen klafft eine immer größer werdende Lücke (Bild 1-2). Zudem verkürzt sich der Produktzyklus und damit auch die zur Verfügung stehende Entwicklungszeit stetig. Diese Gegensätze können nur über eine signifikante Steigerung der Effizienz des Entwicklungsablaufs gelöst werden, die vor allem durch Abstrahieren, Automatisieren und Wiederverwenden zu erreichen ist.

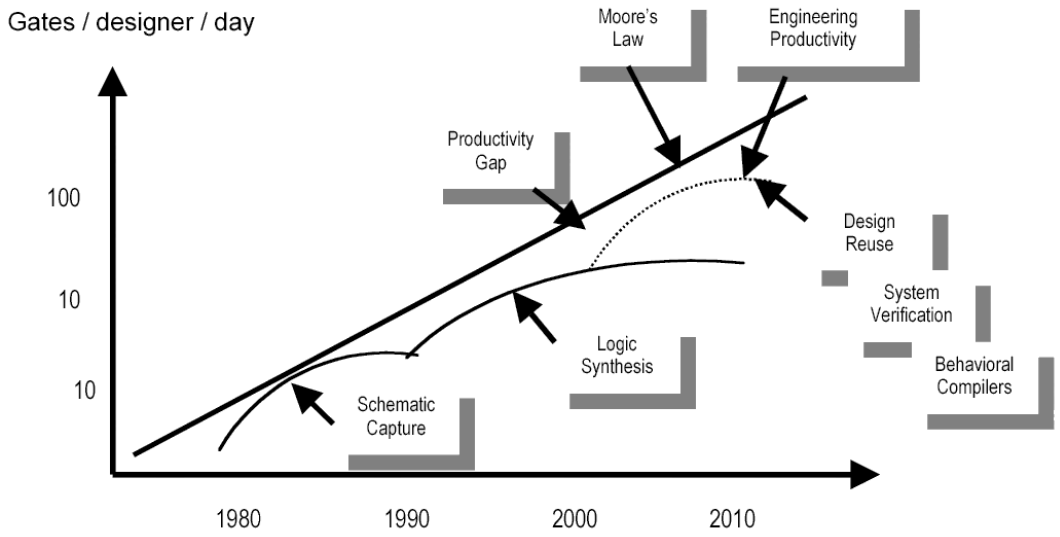


Bild 1-2. Der sogenannte „Productivity Gap“ (Quelle: [138])

Die „Roadmap“ der SIA (Semiconductor Industry Association – der Verband der US-amerikanischen Halbleiterhersteller) [104] beziffert für das Jahr 2014 die Größe von Prozessoren mit 4,3 Mrd. Transistoren und gibt die Kapazität von Speicherschaltungen (DRAM) mit 48 GBit an, die in Technologien mit minimalen Strukturbreiten von 0,034 µm (34 nm) hergestellt werden. Auch die Versorgungsspannung der Schaltungen sinkt bis 2014 auf 0,5 V [32, 72]. Die Probleme, die sich dabei ergeben, sind die sinnvolle Nutzung der verfügbaren Leistungssteigerung und die Sicherstellung der Fehlerfreiheit der Schaltungen.

Eine mögliche Lösung dieser Probleme stellt die **Wiederverwendung** („Design Reuse“) von bereits entwickelten und verifizierten Komponenten dar (IP). Durch das Zusammenfügen dieser Komponenten kann ein komplexeres System in kürzerer Zeit implementiert werden, da eine Neuentwicklung den Entwurfsablauf verlängern würde.

Eine weitere Lösung steht mit parametrisierbaren Komponenten für reguläre Blöcke (Addierer, Multiplizierer, Komparatoren, Zähler, Speicher) zur Verfügung. Dedizierte Makrogeneratoren bieten zu den in einer Bibliothek gespeicherten Komponenten eine Eingabemaske für die Spezifizierung der Parameter und generieren daraus mit Zeitwerten behaftete Simulationsmodelle in einer HDL-Beschreibung und ein Symbol für die Einbindung in einen schematischen Schaltplan bzw. eine Referenz auf ein Entity in der HDL-

Bibliothek. Als Parameter wird zumeist die Bitbreite vorgegeben und die Ein-/Ausgänge sind wählbar. Auch können spezielle Architekturen für die interne Implementierung der Komponente ausgewählt werden. Beispielsweise ist ein Addierer in Carry-Ripple- oder Carry-Select-Technik zu realisieren, mit entsprechenden Vor- und Nachteilen hinsichtlich Geschwindigkeit und Flächenaufwand. Aufgrund der Programmierbarkeit der Hardwarestrukturen eignet sich eine Parametrisierung in besonderem Maße für den Entwurf bei FPGAs. Hierbei kann die Ausnutzung der Logikzellen eines FPGA bei Verwendung der Makrogeneratoren verbessert werden, zudem verringert sich der Entwicklungsaufwand, wenn vordefinierte Module zur Verfügung stehen. Jeder FPGA-Anbieter hat ein entsprechendes Programm in seine Softwarewerkzeuge integriert, z. B. Xilinx (CoreGenerator, vormals LogiBlox), Actel (ACTgen) etc.

Mit wachsender Komplexität der digitalen Systeme, deren Größe weiterhin mit der von Gordon Moore vor 34 Jahren prognostizierten Steigerung zunimmt (Faktor 4 alle 3 Jahre), steigt auch der notwendige Anteil der Verifikation der Systeme im Entwicklungsprozess. Der Anteil der Verifikation an der gesamten Entwicklungszeit liegt bei etwa 70% [9] bis 80% [26]. Dieser Umstand erfordert zunehmend formale Verifikationsmethoden, die Simulation spielt in der Praxis heutzutage jedoch die primäre Rolle. Die **HW/SW-CoSimulation** ist für eingebettete Systeme entscheidend im Verifikationsprozess. Die Anwendungen, bei denen eingebettete Systeme zum Einsatz kommen, werden häufig in Echtzeit ausgeführt. Zunehmend sind Software-Funktionen mittels anwendungsspezifischer Hardware realisiert. Diese Entwicklungen führen zu einer verstärkten Erhöhung der Fehlermöglichkeiten an den Schnittstellen zwischen Hardware und Software. Damit die korrekte Funktionsweise des Systems sichergestellt ist, muss stets die Software auf Konflikte mit der Hardware überprüft werden. Das Problem ist, wie die Funktionalität der HW/SW-Schnittstellen bei Einhaltung des Zeitrahmens für ein Projekt verifiziert werden soll. In der Praxis hat sich bisher eine funktionale Verifikationsmethodik durchgesetzt, die Interface-Modelle (BFM - Bus Functional Model) verwendet. Sie reduzieren den Bedarf an Rechenleistung und daraus folgend die erforderlichen Testzyklen, da nur das Verhalten des Prozessors auf den externen Schnittstellen und die Zusammenarbeit mit der Peripherie modelliert ist. Die bei den Interface-Modellen vorhandene Abbildung auf eine hohe Abstraktionsebene kann in einem frühen Stadium der Entwicklung zur Aufdeckung von Systemfehlern genutzt werden. Im weiteren Projektverlauf ist es mit ihnen möglich, spezifische Peripheriefunktionen zu überprüfen. Geht es in die Detailphase des Projektes, kommt im Verifikationsprozess ein Befehlssatz-Simulator (ISS - Instruction Set Simulator) zum Einsatz. Damit kann die Funktionalität des gesamten Systems inklusive des Prozessorkerns sichergestellt werden. Für Optimierungs- und Verifikationszwecke ist es dann sinnvoll, eine nano/pico-Sekundengenaue Simulation des Systems durchzuführen. Dies erlaubt das Aufdecken von Fehlern, die auf den höheren Abstraktionsebenen nicht zu identifizieren sind, was besonders bei der Simulation von sicherheitskritischen Anwendungen von Bedeutung ist.

Für die HW/SW-CoSimulation erscheint die in der Literatur erwähnte Board-Level-Simulation auch für „System on Chip“-Entwürfe geeignet. Frühere Arbeiten in diesem Bereich beschreiben die Möglichkeit eines Einsatzes der Hardwarebeschreibungssprache VHDL und geben Hinweise zur Optimierung der VHDL-Modelle hinsichtlich der Leistungseigenschaften der Simulation [52, 53, 54]. Zusätzlich wurden in Board-Level-Simulationen

lediglich Interface-Modelle der Komponenten integriert [30, 97] bzw. es wurde auf die Simulation von Software-Treibern auf eventuell vorhandenen Prozessoren hingewiesen [52].

Im Bereich der digitalen Signalverarbeitung setzen sogenannte **digitale Signalprozessoren** (DSP) den Quasistandard bei der Bewältigung unterschiedlichster Algorithmen aus den Bereichen Audio/Video, Kommunikation, Netzwerke etc. Die Anwendungen beinhalten beispielsweise Funktionen wie Spracherkennung und Sprachkomprimierung, Komprimierung digitaler Musik- und Videodaten sowie weitere Funktionen für die Breitband-Datenübertragung. Ihre Leistungsfähigkeit in diesen Segmenten erlangen sie durch Spezialisierung auf häufig vorkommende Programmkonstrukte, wie z. B. der MAC-Operation (Multiply-And-Accumulate). Ausgehend von den Universalprozessoren, wie Intel Pentium, Transmeta Crusoe, o. ä., sind bestimmte Daten- und Kontrollpfade und Ausführungseinheiten (ALU, Multiplizierer) auf die Besonderheiten der digitalen Signalverarbeitung hin optimiert. Dabei spielt die Laufzeit und somit auch der Durchsatz pro Zeiteinheit eine herausragende Rolle. Aber auch unterstützte Datenformate, Verlustleistung u. ä. tragen ihren Teil dazu bei. DSPs spielen immer dann ihre Stärke aus, wenn es darum geht, komplizierte Algorithmen möglichst in Echtzeit auf eine große Datenmenge anzuwenden.

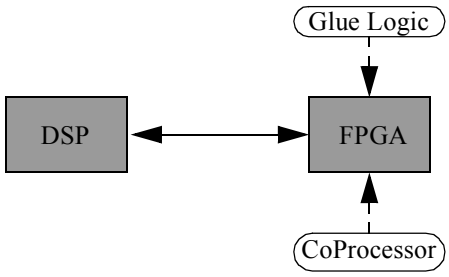


Bild 1-3. Erweiterung des Einsatzgebietes der FPGAs.

In den letzten Jahren hat sich die **FPGA**-Technologie für eingebettete Systeme sowohl als Prototyp-Hardware als auch in konkreten Produkten etabliert. FPGAs eignen sich traditionell ideal für die Implementierung der Verbindungslogik („Glue Logic“), z. B. für durchzuführende Formatkonvertierungen oder auch zur Verknüpfung unterschiedlichster Schnittstellen, die benötigt werden, um Prozessoren mit Echtzeit-Peripherie zu verbinden, z. B. Modems, A/D-Umsetzer und digitale Empfänger-/Sendebausteine. Mit den Möglichkeiten der Realisierung von signalverarbeitenden Algorithmen wird der traditionelle Einsatz von FPGAs auf die Auslagerung von rechenzeitaufwendigen Funktionen der digitalen Signalverarbeitung erweitert (Bild 1-3). Begünstigt wird dies durch eine hochgradig parallele Implementierung eines Algorithmus im FPGA und durch eine stetig wachsende Anzahl von Logikgattern in neuen FPGA-Generationen.

Derzeit stehen bereits FPGAs mit nahezu acht Millionen nutzbarer Gatteräquivalente, z. B. XC2V8000 von Xilinx [140], zur Verfügung, in die nahezu alle wesentlichen Teile der umgebenden Hardware eines DSP untergebracht werden können. Davon ausgenommen sind bisher und in naher Zukunft analoge Baugruppen und große Speicher. Daher ist es günstig, die auf digitale Signalverarbeitung und Fließkomma-Arithmetik optimierten DSPs in Kombination mit in Hardware realisierten applikationsspezifischen Erfordernisse einer

Anwendung einzusetzen. Trotz des hohen Preises für FPGAs eignen sie sich als Ausgangsbasis einer SoC-Entwicklung, aber auch für konkrete Produkte eines kleinen Marktsegmentes mit entsprechend niedrigen Stückzahlen.

Darüber hinaus erlaubt der Einsatz von FPGAs, integriert in SoCs, eine Wiederprogrammierbarkeit, Adaptierbarkeit und Rekonfigurierbarkeit nach der Chip-Produktion, die die Produktivität der Chipentwicklung wesentlich steigern. Industrielle Anwendungen enthalten zumeist Hardware-Blöcke, die Standards und Protokolle implementieren, die sich jedoch häufig zur Entwicklungszeit in der Standardisierungsphase befinden, so dass die Spezifikation noch nicht feststeht. Zudem kann ein und dieselbe Hardwareplattform für verschiedene Applikationen eines Anwendungsgebietes oder für mehrere Kunden in kurzer Zeit spezifisch angepasst werden.

Die Hauptmotivation für diese Arbeit stellt die **Kombination von DSP und FPGA** dar, bei der das FPGA als CoProzessor für den DSP arbeiten kann oder auch im FPGA extern notwendige zusätzliche Hardwarebaugruppen nebst Verbindungslogik flexibel, sprich programmierbar, zusammengefasst werden können. Die erreichte zusätzliche Flexibilität bedingt eine höhere Komplexität in der Entwicklung für dieses System, so dass geeignete Entwurfswerkzeuge benötigt werden, die in ein integriertes FPGA/DSP-Entwicklungssystem als Vorstufe der Entwicklung für ein ASIC einzubetten sind. Denn die Leistungsfähigkeit einer Kombination von FPGA und DSP lässt sich nur dann effizient nutzen, wenn eine entsprechende Software-Unterstützung zur Verfügung steht.

1.2 Zielstellung

Mit dieser Arbeit wird *FADES* (FPGA/DSP-Entwicklungssystem) als Plattform für den Entwurf von eingebetteten audiosignalverarbeitenden Echtzeitsystemen vorgestellt. Eine spezielle Architektur in Form eines DSP mit eingebetteter Software (RTK - Real-Time Kernel) wird um die Fähigkeiten eines variablen Co-Prozessors in Hardware (FPGA) erweitert. Eine Spezialisierung auf die digitale Audiosignalverarbeitung erfolgt mittels digitaler Audioschnittstellen. Das Entwicklungssystem ist als Ausgangsbasis für die Entwicklung eines integrierten eingebetteten Systems (SoC) in einem Chip (ASIC) für die digitale Echtzeitverarbeitung von Audiosignalen anzusehen.

Zur Unterstützung des Entwurfsflusses ist in die Entwicklungssoftware eine Bibliothek wiederverwendbarer Komponenten integriert. Diese wird erweitert um Komponenten, die für die Interface-Synthese, speziell der Generierung von Modulen zur Kommunikation zwischen Hardware und Software, zur Verfügung stehen.

Eine umfangreiche HW/SW-CoSimulation deckt die virtuelle Verifikation des gesamten Systems ab und wurde bereits in [29] veröffentlicht. Für die HW/SW-CoSimulation gibt es Lösungen, die die Hardware-Simulation und die Software-Simulation getrennt ausführen und bei denen eine externe Software-Komponente die Synchronität zwischen beiden Simulationen sicherstellt. Aufgrund der existenten Beschränkungen einer getrennten Simulation wird hier der Ansatz verwendet, die Software-Simulation in die Hardware-Simulation zu

integrieren, indem ein selbst entwickeltes zyklengetreues und Pin-Timing-genaues VHDL-Modell des eingesetzten DSP in die VHDL-Simulation einbezogen wird. Damit läuft die Simulation komplett innerhalb eines Simulatorkerns. Das für den DSP entwickelte VHDL-Modell ist bezüglich der Genauigkeit zwischen „cycle accurate“ und „nano-second accurate“ einzuordnen [101]. Um die Simulationszeiten niedrig zu halten, bedurfte es eines optimierten Modells für den verwendeten DSP.

Ein Hardware-Prototyp des Gesamtsystems, realisiert als Leiterplatine mit diskreten Komponenten, dient schließlich der HW/SW-In-Circuit-CoVerifikation, bei der das FPGA als Hardware-Emulator dient. Ist die Zieltechnologie ein SoC, realisiert in einem ASIC, handelt es sich um ein „Rapid Prototyping“. Tritt ein Problem beim Testen im Hardware-Prototyp auf, kann rückgekoppelt der theoretische Hintergrund in der HW/SW-CoSimulation mit den an den Problemfall angepassten Eingabedaten untersucht werden, sofern es keine Möglichkeit gibt, direkt im Hardware-Prototyp die Ursache des Problems zu finden.

Die HW/SW-CoSimulation kann auf jeden Fall durchgeführt werden, auch wenn die Hardware zur HW/SW-In-Circuit-CoVerifikation nicht zur Verfügung steht. Der Test und die Inbetriebnahme der Software erfolgt bereits parallel zur Entwicklung und Produktion eines möglichen ASIC („Virtual Prototyping“).

Im Gegensatz zu anderen Arbeiten [15, 61] wird mit dieser Arbeit kein Entwicklungssystem für die Systemebene vorgestellt, die im Allgemeinen unterschiedliche Hardware-Architekturen bedienen. In [61] werden beispielsweise verschiedene alternative Konfigurationen einer variablen, parametrisierbaren SoC-Plattform (Prozessor, Speicher, Systembusse, Peripherie) hinsichtlich ihres Leistungsverbrauchs verglichen. Das Ziel hierbei ist es nicht, einen absoluten Wert des Leistungsverbrauchs einer Konfiguration zu ermitteln, sondern einen relativen Vergleich zwischen mehreren Alternativen im Entwurfsraum zu ermöglichen. *FADES* dagegen ist speziell auf die Architektur ausgerichtet, die einen DSP mit einem FPGA kombiniert.

Der Begriff HW/SW-CoDesign wird in dieser Arbeit hauptsächlich im Sinne von Co = concurrent (nebenläufig) verwendet. Die Software für den DSP, als spezielle Prozessorarchitektur, ist parallel zur zu implementierenden applikationsspezifischen Hardware im FPGA zu entwickeln. Der Begriff bezeichnet hier nicht den gemeinsamen Entwurf von Hardware und Software in einer einheitlichen Beschreibungssprache, wofür z. B. SytemC [106] oder ESTEREL [27] verwendet werden.

1.3 Eigener Beitrag

Der Entwurfsprozess bei der SoC-Entwicklung muss möglichst integriert in einer konsistenten Umgebung und durchgängig bezüglich der Simulation, Synthese und Analyse auf den unterschiedlichen Entwurfsebenen erfolgen. In dieser Arbeit werden deshalb mehrere Bereiche betrachtet und geeignete Methoden für den anvisierten Einsatzbereich vorgestellt. Folgende Teile wurden im Einzelnen behandelt und untersucht:

- Für die SoC-Entwicklung wurde eine integrierte HW/SW-Entwicklungsumgebung konzipiert und implementiert.
- Es wird mit dieser Arbeit gezeigt, dass eine auf diese Weise genutzte VHDL-Simulation zur HW/SW-CoSimulation für den Anwendungsbereich eingesetzt werden kann. Sie bietet aufgrund der Zyklentreue und dem genauen Pin-Timing den besten virtuellen Verifikationsgewinn, dafür ist diese Vorgehensweise gegenüber allen anderen die langsamste. In der digitalen Signalverarbeitung werden jedoch oft sich wiederholende Berechnungen auf variable Eingangswerte angewandt, so dass sich die Verifikation des Zusammenspiels zwischen Hardware und Software auf wenige dieser Zyklen beschränken kann. Der zeitliche Aufwand für die HW/SW-CoSimulation wird anhand von Anwendungsbeispielen dokumentiert.
- Weiterhin wird mit dieser Arbeit gezeigt, dass die angewandte HW/SW-CoSimulation in Kombination mit der implementierten HW/SW-In-Circuit-CoVerifikation ideal für eine rasche Entwicklung von Prototypen aus dem adressierten Anwendungsbereich ist. Diese bietet den besten praktischen Verifikationsgewinn, da sie gegenüber allen anderen Vorgehensweisen die schnellste ist, ohne dabei Einbußen hinsichtlich der Sichtbarkeit der Systeminterna hinnehmen zu müssen.
- Der Aspekt der Wiederverwendung („Design Reuse“) ist mit einer Bibliothek bereits implementierter und getesteter Komponenten im Entwurfsfluss berücksichtigt. Die Komponenten sind zum Teil speziell auf die implementierte Struktur des Systems ausgerichtet (Interface-Synthese), andere dagegen sind architekturunabhängig.
- Als Anwendungsfall für das Entwicklungssystem wurde ein auch im industriellen Umfeld interessanter MPEG1-Layer3-Player implementiert, anhand dessen die Praxistauglichkeit des Systems nachgewiesen wird.

1.4 Struktur der vorliegenden Arbeit

Kapitel 2 befasst sich mit dem derzeitigen Stand der Technik auf den von dieser Arbeit tangierten Sachgebieten. Der Schwerpunkt liegt bei SoCs, eingebetteten Systemen und deren Entwicklung (HW/SW-CoDesign). Kapitel 3 gibt einen detaillierten Überblick über die für die HW/SW-CoSimulation verwendete VHDL-Board-Level-Simulation. Es werden sowohl grundlegende als auch spezifische Eigenschaften dargelegt. Kapitel 4 gibt einen Überblick über die Eigenschaften der Schnittstellen einer FPGA/DSP-Kombination als Grundlage für die Interface-Synthese. Darüber hinaus werden die Möglichkeiten der Kombination von FPGA und DSP für die SoC-Entwicklung im anvisierten Anwendungsbereich und die zugehörigen technischen Aspekte erörtert. Systemkonzepte von *FADES* sind in Kapitel 5 und implementierungstechnische Details der Hardware und der Software des Entwicklungssystems in Kapitel 6 aufgeführt. Synthese- und Simulationsergebnisse für unterschiedliche Anwendungsbeispiele und der MPEG1-Layer3-Player werden in Kapitel 7 vorgestellt.

Die Zusammenfassung dieser Arbeit mit einer Diskussion der Ergebnisse und einem Ausblick auf weiterführende Arbeiten befindet sich in Kapitel 8. Im Anhang sind abschließend ausgewählte Quelltextauszüge, Schaltpläne und -layouts der Arbeit beigefügt.

2 Stand der Technik

In diesem Kapitel werden zunächst die Alternativen für die Realisierung eines Systems für die digitale Signalverarbeitung vorgestellt und Merkmale dieser „Bausteine für den Hardwareentwurf“ vergleichend gegenübergestellt. Detaillierter wird die Virtex-Familie von Xilinx als Vertreter der FPGA-Bausteine und die ADSP-2106x-Familie von Analog Devices als Vertreter der DSP-Bausteine behandelt, da diese Hauptbestandteile des im Rahmen dieser Arbeit entwickelten FPGA/DSP-Entwicklungssystems sind. Anschließend wird der Stand der Technik bei SoCs, eingebetteten Systemen im Allgemeinen und dem notwendigen HW/SW-CoDesign erläutert. Zum Ende dieses Kapitels werden mit VHDL und XML zwei Beschreibungssprachen eingeführt, die im weiteren Verlauf der Arbeit eine wesentliche Rolle spielen.

2.1 Bausteine für den Hardwareentwurf

2.1.1 ASIC

Der traditionelle Ansatz einen Algorithmus zu berechnen ist der, ein ASIC zu verwenden, d. h. die Operationen in Hardware ausführen zu lassen. Weil diese ASICs speziell für eine gegebene Berechnung entworfen und implementiert werden, sind sie unschlagbar schnell und effizient in dieser Disziplin. Der Nachteil ist, dass nach der Fertigung des ASIC keine Änderungen mehr möglich sind. Dies bedingt einen Neuentwurf und eine Neuanfertigung, wenn auch nur kleinste Teile der Schaltung bei geänderten Randbedingungen modifiziert werden müssen. Das ist ein sehr kostspieliger Prozess, da zu bedenken ist, dass auch bereits ausgelieferte Chips auszutauschen sind. Auch die Entwicklungskosten, als NRE-Kosten (Non-Recurrent Engineering) bezeichnet, spielen eine wichtige Rolle bei der Entscheidung für oder wider eine Implementierung als ASIC. Erst eine Fertigung in großen Stückzahlen rechtfertigt die Investitionen und die relativ lange Entwicklungszeit. Die Stückzahlen, ab denen kundenspezifische Entwicklungen rentabel sind, erreichen immer neue Größenordnungen [49]. Soll eine kurze Time-to-Market angestrebt werden, so sind Zwischenlösungen mit programmierbaren Bauelementen gängige Praxis.

Eine Abwandlung vom ASIC stellen die ASSPs (Application Specific Standard Product) dar. ASIC und ASSP sind insofern gleichwertig, als sie für eine spezifische Applikation entwickelt wurden. ASICs werden im Unterschied zu ASSPs jedoch für nur einen Auftraggeber gefertigt. Die realisierte Applikation stellt bei ASSPs in der Regel einen Standard dar (z. B. PCI, MP3-Dekoder) und sie werden von mehreren Kunden erworben.

Einen interessanten Aspekt stellt der Einsatz von Prozessorkernen als IP in ASICs dar. Die zusätzliche Möglichkeit der Modifikation des Verhaltens über Änderungen der Software erleichtert die Implementierung von komplexen Algorithmen und Standards (MPEG,

DVD), die in der Standardisierungsphase ständigen Detailwechseln unterliegen und oft Anpassungen noch während der Entwicklung erfordern. Eine programmierbare Version ist erheblich flexibler als eine verdrahtete Schaltung.

2.1.2 Mikroprozessor

Mikroprozessoren (GPP - General Purpose Processor) sind sehr viel flexibler als anwendungsspezifische Bausteine (ASICs). Das Hauptmerkmal ist ihre Programmierbarkeit. Sie arbeiten eine Befehlsfolge ab, um eine Berechnung durchzuführen. Mit einer Änderung von Befehlen innerhalb der Software kann die Funktionalität des Gesamtsystems beeinflusst werden, ohne die darunterliegende Hardware anpassen zu müssen. Der Nachteil dieser Flexibilität ist die geringere Leistungsfähigkeit. Sie liegt weit unter der, die mit einem ASIC möglich ist. Jeder Befehl wird, vereinfacht ausgedrückt, zunächst aus dem Speicher geladen, dekodiert und erst dann ausgeführt. Dazu kommt, dass der Befehlssatz bereits bei der Herstellung des Prozessors festgelegt wird. Jede Funktion muss danach mit den zur Verfügung stehenden Befehlen realisiert werden. Dies wiederum kann unter Umständen zu einem zusätzlichen Programmieraufwand führen.

Mikrocontroller stellen eine Unterart der Mikroprozessoren dar. Dies sind Prozessoren, die für die Steuerung von Prozessen optimiert sind. Der Code ist kontrollflussdominiert, d. h., es gibt viele Sprünge, Verzweigungen und überwiegend logische Operationen. Der Datendurchsatz hingegen ist relativ gering. Im Gegensatz zu Mikroprozessoren sind auf dem Chip weitere Spezialkomponenten integriert, wie Timersysteme, Schnittstellenkomponenten, Analog/Digital-Umsetzer etc.

2.1.3 ASIP

ASIPs (Application Specific Instruction Set Processor) stellen eine Spezialisierung von Mikroprozessoren dar. Sie sind speziell für eine bzw. für eine Klasse von Anwendungen hinsichtlich der Randbedingungen Leistungsfähigkeit, Kosten und Leistungsaufnahme optimiert und unterscheiden sich sehr stark voneinander. ASIPs bieten eine Lösung zwischen den beiden Extremen ASIC und GPP. Ein ASIP ist eigentlich ein „abgespeckter“ GPP, der jedoch aufgrund der (beschränkten) Programmierbarkeit flexibler ist als ein ASIC. Der aufwendige Entwurf eines ASIP ist in die Kosten-/Nutzenanalyse einzubeziehen, da neben dem Prozessorkern auch das Software-Entwicklungssystem implementiert werden muss.

Die Entwicklung eines ASIP beinhaltet die Charakterisierung der für den Anwendungsbereich typischen Operationen, den Entwurf einer geeigneten Prozessorarchitektur und die Generierung eines Instruktionssatzes neben der Implementierung eines zugehörigen, möglichst optimierenden Compilers. Dies entspricht eines der zentralen Probleme des HW/SW-CoDesigns.

ASIPs spielen vorwiegend in den Bereichen eine wichtige Rolle, in denen der Algorithmus vorher sicher feststeht (z. B. JPEG) oder in denen höchste Leistungsanforderungen bei gro-

ßen Stückzahlen gefordert sind. Beispiele für Anwendungen sind im Bereich Mobiltelefon, „Set-Top“-Boxen, Grafikbeschleuniger und allgemein Multimedia-Massenanwendungen zu finden.

2.1.4 DSP

Digitale Signalprozessoren sind Mikroprozessoren, die auf die Aufgaben im Bereich der digitalen Signalverarbeitung zugeschnitten sind. Der Code für DSPs umfasst viele arithmetische Operationen, vor allem Multiplikationen und Additionen. Es kommen wenige Verzweigungen vor, wenn doch, dann mit sehr gut vorhersagbaren Sprungzielen. DSPs weisen eine hohe Nebenläufigkeit auf und verarbeiten dabei sehr große Datenmengen. Die Programmiersprache C ist auch bei DSPs zum Standard geworden, jedoch sind zeitkritische Teilprogramme in Assembler zu programmieren. Weil in den meisten signalverarbeitenden Systemen komplexe mathematische Operationen auf zeitkritische Signale (Echtzeit) angewandt werden, besitzen DSPs angepasste Architekturen, um sich wiederholende, numerisch aufwendige Berechnungen zu beschleunigen.

DSPs basieren üblicherweise auf einer Architektur, die getrennte Speicher für Befehle und Daten aufweist und dafür separate Speicherbusse zur Verfügung stellt. Dies ermöglicht eine Parallelisierung von Daten- und Befehlstransfer und damit eine Steigerung der Rechenleistung. Allerdings bringt diese Architektur einen Kostennachteil mit sich, da für die separate Ausführung von Befehls- und Datenleitungen mehr Pins und ein höherer Aufwand bei der Chipherstellung aufzubringen ist. Eine Lösung dieses Problems stellt eine Architektur dar, bei der extern Befehle und Daten auf einem gemeinsamen Bus transferiert werden, intern jedoch weiterhin separate Busse zur Verfügung stehen. Der Großteil der DSPs besitzt eine MAC-Einheit, die im Idealfall die Multiplikation, die Addition und das Speichern des Ergebnisses in einem einzigen Taktzyklus ermöglicht. Ob diese Operation dabei auf Festkomma- oder Fließkomma-Zahlen angewandt wird, ist ein wichtiges Unterscheidungsmerkmal der DSP-Familien. Des Weiteren unterstützen DSPs Hardware-Loops (zero overhead loops), das schnelle Ausführen von Verzweigungen, Overflow-Schutz (Sättigung) in der ALU, separate Adress-Generatoren für den Programm- und Datenspeicher mit Index-, Modify-Register und Ringpufferverwaltung. Sie besitzen üblicherweise einen Barrel-Shifter und unterstützen ein oder mehrere serielle bzw. parallele I/O-Interfaces oder auch DMA-Kanäle. Die DMA-Kanäle können dazu genutzt werden, mehrere DSPs miteinander zu verbinden und damit Multiprozessor-Architekturen zu bilden.

Seit 1996 gibt es DSPs mit VLIW-Architektur [56], die einen höheren Grad der Parallelität erreichen (ILP - Instruction Level Parallelism) und damit den Durchsatz in Bezug zur Taktrate erhöhen. VLIW stellt eine radikale Abkehr von herkömmlichen DSP-Architekturen dar. Ein VLIW-Prozessor besitzt mehrere Funktionseinheiten, die unabhängig voneinander durch verschiedene Bits im Instruktionswort gesteuert werden. Das Instruktionswort wird dadurch unter Umständen sehr lang, welches einen erhöhten Programmspeicherbedarf erfordert. Das VLIW-Konzept hängt zudem im besonderen Maße von der Qualität des Hochsprachen-Compilers (meistens C) ab, da dieser in der Lage sein muss, den Programm-

code gleichmäßig auf die Funktionseinheiten zu verteilen. Ein Vergleich verfügbarer DSPs mit VLIW-Architektur wurde in [24] publiziert.

Die VLIW-Architektur steht im Gegensatz zur superskalaren Architektur, die heutzutage überwiegend bei GP-Prozessoren, jedoch auch bei DSPs, eingesetzt wird und bei der ebenfalls mehr als ein Befehl pro Takt ausgeführt werden kann. Der Unterschied zwischen superskalaren und VLIW-DSPs ist die Art und Weise der Aufteilung einer Befehlssequenz auf die zur Verfügung stehenden parallelen Funktionseinheiten. Der Compiler übernimmt bei VLIW diese Aufteilung (static scheduling). Bei superskalaren DSPs gibt es spezielle Hardware-Blöcke (instruction dispatcher), die die optimale Reihenfolge der Befehle dynamisch zur Laufzeit unter Beachtung von Datenabhängigkeiten und Ressourcenkonflikten bestimmen (dynamic scheduling). Damit wird die Aufgabe der Ablaufplanung paralleler Instruktionen vom Compiler bzw. Programmierer auf den Prozessor verlagert.

Die bei superskalaren und VLIW-DSPs angegebenen hohen MIPS/MOPS/MFLOPS-Zahlen setzen normalerweise ideale Randbedingungen voraus, die in der Praxis eher selten vorliegen. Ideale Voraussetzungen bedeuten z. B.:

- die Daten sind bereits in den internen Speicher geladen,
- die Pipeline ist gefüllt,
- es gibt keine Interrupts oder Sprünge.

Tabelle 1: Übersicht der 2002 verfügbaren DSP-Chips (Quellen: [18, 21, 22, 23, 45]).

Hersteller	Familie	Fest-, Fließkomma oder Beides	Datenbreite [bits]	Instruktionswortbreite [bits]	Taktrate [MHz]	BDTiMark2000	adressierbarer Speicherbereich	Kernspannung [V]
Agere Systems	DSP16xxx	Fest	16	16/32	240	1150	512K-32M	1,0; 1,6; 2,5
	StarPro 2000	Fest	16	16	250	n/a	512M	1,2
Analog Devices	ADSP-21xx	Fest	16	24	80	240	4M	1,8
	ADSP-219x	Fest	16	24	160	420	32M	2,5
	ADSP-2106x	Fließ	32/40	48	66	250	64M-1G	3,3; 5,0
	ADSP-2116x	Fließ	32/40	48	100	510	63,5 M-1G	1,8; 2,5
	ADSP-2153x	Fest	16	16/32	300	1690	768M	0,9-1,5
	ADSP-TS101S	Beides	8/16/32/40	32	250	n/a	960	1,2

Tabelle 1: Übersicht der 2002 verfügbaren DSP-Chips (Quellen: [18, 21, 22, 23, 45]).

Hersteller	Familie	Fest-, Fließkomma oder Beides	Datenbreite [bits]	Instruktionswortbreite [bits]	Taktrate [MHz]	BDTiMark2000	adressierbarer Speicherbereich	Kernspannung [V]
Hitachi	SH76xx (SH-DSP)	Fest	16	16/32	100	280	160-384M	1,9; 3,3
	SH772x (SH3-DSP)	Fest	16	16/32	200	490	384M	1,8; 1,9; 2,0
	SH775x (SH-4)	Beides	32	16	240	750	448M	1,5; 1,8; 1,9
Infineon	TC1xxx (TriCore 1)	Fest	32	16/32	96	n/a	4G	1,5
LSI Logic	LSI140x (ZSP400)	Fest	16/32	16	200	940	4M	1,8
Motorola	DSP563xx	Fest	24	24	240	710	48M	1,5; 1,8; 2,5; 3,3
	DSP568xx	Fest	16	16	80	110	0-128K	2,5; 3,3
	DSP5685x	Fest	16	16	120	340	36M	1,8
	MSC810x	Fest	16	16	300	3430	4G	1,5; 1,6
Texas Instruments	TMS320F24xx	Fest	16	16/32	40	n/a	384K	3,3; 5,0
	TMS320F28xx	Fest	32	16/32	150	n/a	2M	1,8
	TMS320C3x	Fließ	32	32	75	n/a	64M	3,3; 5,0
	TMS320C54xx	Fest	16	16	160	500	512K-16M	1,5; 1,6; 1,8; 2,5; 3,3; 5,0
	TMS320C55xx	Fest	16	8-48	200	970	16M	1,6
	TMS320C62xx	Fest	16	32	300	1920	52-512M	1,2; 1,5; 1,8
	TMS320C64xx	Fest	8/16	32	600	5320	3G	1,2; 1,4
	TMS320C67xx	Fließ	32	32	167	820	52-512M	1,8; 1,9

Tabelle 1 gibt eine Übersicht der im Jahre 2002 kommerziell erhältlichen DSPs. Die dominierenden Marktteilnehmer bei digitalen Signalprozessoren sind Texas Instruments und Analog Devices Inc. (ADI). Andere Firmen wie Agere Systems (früher Lucent Microelectronics Group und davor AT&T Microelectronics), Motorola, LSI Logic Corp., Intel und andere versuchen Anschluss zu halten. Die programmierbaren Logikbauelemente stel-

len eine leistungsfähige Ergänzung und Erweiterung auf dem Gebiet digitaler Signalverarbeitung dar. Auch wenn dieser Bereich derzeit nur einen kleinen Teil des gesamten DSP-Marktes ausmacht, so wird er doch in den nächsten Jahren überproportional wachsen. Xilinx mit seiner Ende 2000 angekündigten XtremeDSP-Initiative [141] und Altera mit seinen DSP-bezogenen vorgefertigten Blöcken (IP) drängen in den lukrativen DSP-Markt. Im Jahr 2000 lag der Umsatz im DSP-Marktsegment bei rund 6 Mrd. Dollar [48].

2.1.5 Programmierbare Logikbausteine

Die programmierbaren Logikbausteine (PLD) füllen die Lücke zwischen Hard- und Software. Software bietet inhärent die Möglichkeit der Programmierbarkeit. Programme oder Programmteile werden aus dem Speicher in den Rechenkern geladen, ausgeführt und durch neue Programmteile oder andere Programme ersetzt. Hardware bot diese Möglichkeit so lange nicht, bis es programmierbare Logikschaltungen gab, die einmal oder beliebig oft programmierbar waren. Diese erreichen eine potentiell höhere Leistungsfähigkeit als Software und bieten zudem eine größere Flexibilität als Hardware. PLDs werden grob in SPLDs (Simple PLD), CPLDs (Complex PLD) und FPGAs (Field Programmable Gate Array) unterteilt.

- *SPLDs:* SPLDs sind die kleinsten verfügbaren und damit kostengünstigsten Bausteine programmierbarer Logik. Sie besitzen typischerweise 4 bis 22 Makrozellen und können damit einige TTL-Standardbausteine der 7400-Serie ersetzen. Alle Makrozellen sind untereinander verbunden, so dass es kein Platzierungs- und Verdrahtungsproblem gibt. Ein weiterer Nebeneffekt ist die konstante Verzögerung von Eingängen zu Ausgängen, die sich auch nicht mit der Programmierung ändert.

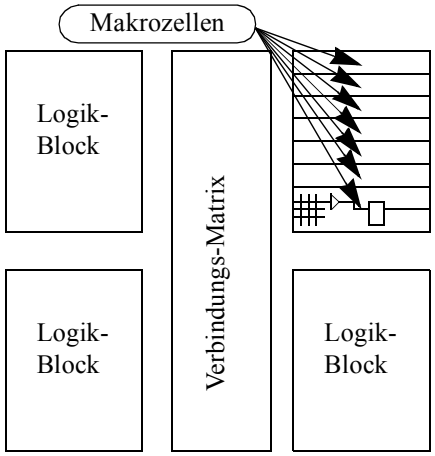


Bild 2-1. Prinzipieller Aufbau eines CPLD.

- *CPLDs:* CPLDs besitzen eine höhere Kapazität als SPLDs und stellen eine Weiterentwicklung derselben dar. Der prinzipielle Aufbau von CPLDs ist in Bild 2-1 dargestellt. Ein typisches CPLD entspricht in etwa 2 bis 64 SPLDs und enthält einige hundert Makrozellen, die in Gruppen von 8 bis 16 Makrozellen pro Logikblock angeordnet sind. Schaltungsmatrizen zwischen den Logikblöcken garantieren eine größere Flexibilität bei der Verdrahtung, aber nicht bei jedem Hersteller und jeder Bausteinfamilie

sind alle Logikblöcke miteinander verbunden. Sollte dies der Fall sein, so gibt es unter Umständen bei komplexen Entwürfen ein Verdrahtungsproblem. Wenn nicht, dann sind die Verzögerungszeiten genau wie bei SPLDs unabhängig von der Programmierung. CPLDs eignen sich besonders für kontrollflussorientierte Schaltungen, und zwar aufgrund ihrer kurzen Signallaufzeiten. Typische Anwendungsgebiete sind Verbindungslogik, Zustandsautomaten (State Machines) sowie Dekoder-, Initialisierungs- und Signal-Handshaking-Funktionen.

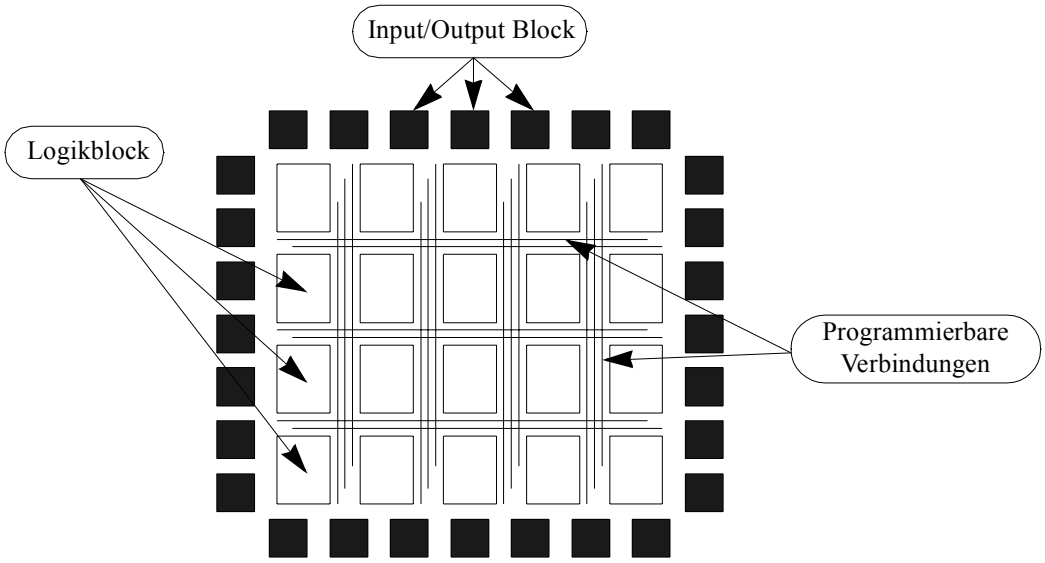


Bild 2-2. Prinzipieller Aufbau eines FPGA.

- *FPGAs:* FPGAs sind programmierbare Speicherbausteine, die eine Matrix aus Berechnungselementen besitzen und deren Funktionalität durch Konfigurationsbits bestimmt wird. Diese Berechnungselemente, auch als Logikblöcke bezeichnet, sind über eine Vielzahl von Verdrahtungskanälen miteinander verknüpft. Außerdem sind alle internen Ressourcen von programmierbaren I/O-Blocks umgeben, wie in Bild 2-2 schematisch dargestellt. Damit können spezifische Schaltungen im FPGA implementiert werden, deren logische Funktionen in den Logikblöcken abgebildet sind und deren Ergebnisse miteinander verknüpft die gesamte Schaltung realisieren. Die meisten heute erhältlichen FPGAs sind SRAM-Bausteine. Bei diesen befinden sich an den Konfigurationsknoten Speicherzellen, die das FPGA konfigurieren, wenn sie programmiert werden. Dies erfordert einen externen Konfigurationsspeicher oder zumindest einen Controller oder Host, damit die Entwurfsdaten in das FPGA geladen werden können.

Die Leistungsfähigkeit von FPGAs im Hinblick auf Platzverbrauch und Verarbeitungsgeschwindigkeit hängt von der in den Logikblöcken integrierten Logik und von der Effizienz ihrer Verdrahtungsarchitektur ab. Ein wichtiges Kriterium dafür liefert ebenso die zu implementierende Schaltung und deren Platzierung und Verdrahtung und damit letzendlich auch die Qualität der Werkzeuge des FPGA-Herstellers.

Da FPGAs in Bezug auf die Schaltungsdichte wesentlich ineffizienter als ASICs sind, muss für eine effiziente Nutzung die Siliziumfläche des FPGA mit Hilfe einer dynamischen Reprogrammierung mehrfach wiederverwendet werden. Das Verhältnis zwischen einem

wahlweise im FPGA oder im ASIC realisierten Entwurf kann nur schwer in Zahlen ausgedrückt werden, da zu viele Parameter zu berücksichtigen sind [47]. Zu den Parametern gehören die verwendete ASIC-Technologie (Vollkundenentwurf, Standardzellen etc.), die FPGA-Architekturmerkmale, die unterschiedlichen Synthesewerkzeuge (u. a. Bibliotheken) und darüber hinaus die Anwendung selbst. Mit dem Erscheinen der XC6200-Familie von Xilinx 1995 gab es über die FastMap genannte Schnittstelle die Möglichkeit, Teile der Schaltung dynamisch zur Laufzeit des Systems auszutauschen, jedoch gab es keine Entwicklungstools, die dies unterstützten. Die Software zur Platzierung und Verdrahtung muss nämlich für diesen Fall speziell angepasst sein, denn die dynamische Umprogrammierung eines Teils der Schaltung darf nicht zu Konflikten innerhalb des Bausteins führen. Die Abgrenzung von Schaltungsteilen erfolgt deshalb immer an den Grenzen ganzer Logikblock-Einheiten des FPGA. Darüber hinaus ist die zeitliche Abfolge des Ein- und Auslagerns von Schaltungsteilen zur Laufzeit des Systems in die rekonfigurierbare Hardware aufgrund der damit verbundenen Verzögerung essentiell für die Entwicklungswerkzeuge. Ohne angepasste Software kann die hardwareseitig vorhandene dynamische Rekonfigurationsmöglichkeit schlussfolgernd nicht genutzt werden.

Tabelle 2 zeigt eine Übersicht aktuell verfügbarer FPGA-Familien und deren Hersteller. Abgekündigte Produkte sind nicht aufgeführt. Für weiterführende Informationen zu den einzelnen Bausteinfamilien sei auf die Datenblätter auf den Webseiten der jeweiligen Hersteller verwiesen [1, 4, 13, 75, 99, 131].

Während Mikroprozessoren im Wesentlichen ein Computing-In-Time, also eine sequentielle Abarbeitung eines Algorithmus, durchführen, basieren strukturierbare Logikschaltungen im Sinne eines klassischen ASIC auf einem Computing-in-Space. Rekonfigurierbare Logikschaltungen erlauben die konsequente Verbindung beider Paradigmen und stellen einen Paradigmenwechsel beim Entwurf mikroelektronischer Systeme dar [50].

Tabelle 2: Übersicht der wichtigsten aktuell verfügbaren FPGA-Familien.

Hersteller	SRAM	Flash	Antifuse
Actel		proASIC proASIC Plus	MX, eX, SX, SX-A Axcelerator
Altera	Flex 6000, Flex 8000, Flex 10K Cyclone, Acex 1K, Apex 20K Apex II, Mercury Stratix, Stratix GX		
Atmel	AT40K, AT6000		
Lattice	ispXPGA, ORCA		
Quicklogic			QuickRAM, QuickPCI QuickSD, QuickFC pASIC /1 /2 /3 Eclipse, Eclipse-II, EclipsePlus
Xilinx	Spartan, Spartan-II Virtex, Virtex-II Pro		

2.2 XILINX Virtex-Familie

Die Virtex-Familie von Xilinx [119] gehört zu den SRAM-basierenden FPGAs, deren Funktionalität über einen Konfigurationsdatenstrom in den internen Speicherstellen programmiert wird. Die in den Bausteinen implementierten Anwendungen können Taktraten bis zu 200 MHz erreichen, einschließlich der I/O-Verzögerung und in Abhängigkeit von der Geschwindigkeitsklasse (speedgrade).

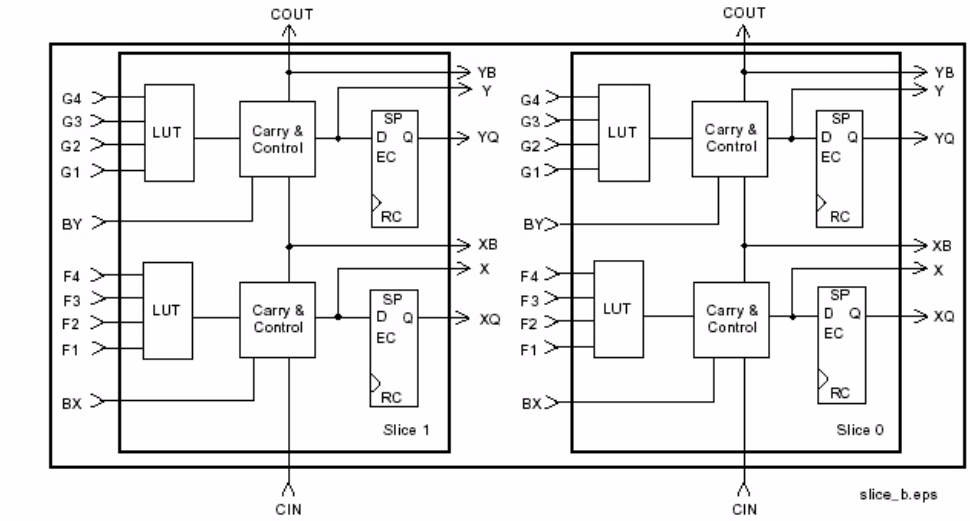


Bild 2-3. 2-Slice Virtex-CLB (Quelle: [119]).

Das Basiselement eines CLB (Configurable Logic Block) ist die Logikzelle, die einen Funktions-Generator (Look-Up-Tabelle - LUT), spezielle Carry-Logik und ein Speicherelement enthält. Zwei Logikzellen sind in einem sogenannten Slice organisiert. Der Funktions-Generator kann jede Funktion mit vier Eingangsbits abbilden, darüber hinaus aber auch als 16x1-Bit synchrones RAM fungieren. Die beiden Look-Up-Tabellen eines Slice lassen sich als 16x2-Bit oder 32x1-Bit synchronem RAM oder als 16x1-Bit dual-port synchronem RAM nutzen. Eine weitere Möglichkeit der Nutzung der Look-Up-Tabelle bietet ein 16-Bit breites Shift-Register, das besonders im Bereich der digitalen Signalverarbeitung Verwendung findet. Die Carry-Logik bietet einen schnellen Pfad für die Propagierung des Übertrags in schnellen arithmetischen Operationen. Pro Slice gibt es einen Carry-Pfad, also insgesamt zwei innerhalb eines CLBs, die jeweils zwei Carry-Bits produzieren können. Die arithmetische Logik enthält außerdem ein XOR-Gatter, so dass sich ein 1-Bit-Volladdierer in einer Logikzelle implementieren lässt. Ein zusätzliches AND-Gatter erleichtert die Implementierung eines Multiplizierers. Das Speicherelement kann als flankengetriggertes D-Flipflop oder als pegelsensitives Latch konfiguriert werden. Der Eingang des Speicherelementes wird aus dem Funktions-Generator oder direkt vom Eingang des Slice gespeist. Asynchrone/Synchrone Preset-/Reset-Eingänge stehen neben Takt- und Takt-Enable-Eingängen ebenfalls zur Verfügung. Ein CLB besitzt insgesamt vier dieser Logikzellen, darüber hinaus aber noch zusätzliche Logik, die es ermöglicht, die Ausgänge der Funktions-Generatoren zu kombinieren. Damit können Funktionen mit fünf oder sechs Eingängen implementiert werden. In Bild 2-3 ist ein Virtex-CLB und in Bild 2-4 eine detailliertere Ansicht eines Virtex-Slice dargestellt.

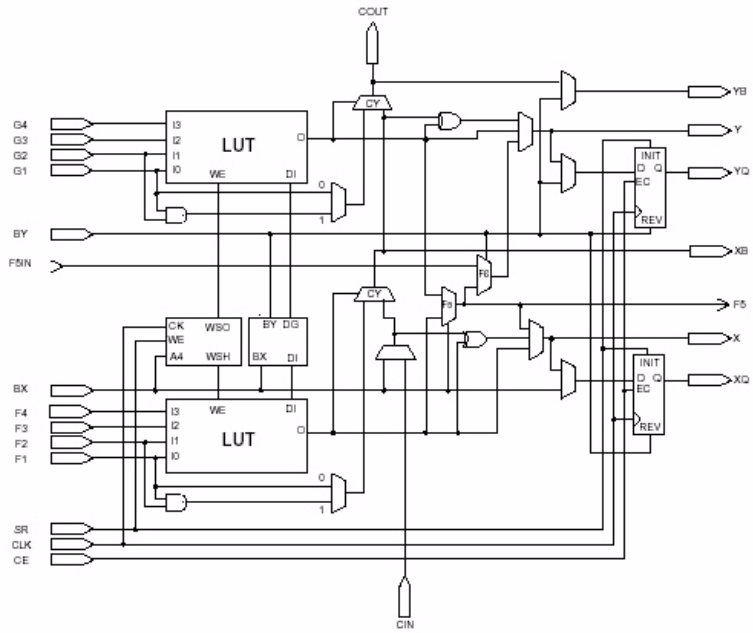


Bild 2-4. Detaillierte Ansicht eines Virtex-Slice (Quelle: [119]).

Die Virtex-FPGAs unterstützen mittels SelectMap-Port eine dynamische partielle Rekonfiguration. Für die notwendige Software-Unterstützung gab es das JBits genannte Java-basierte SDK von Xilinx, das aber nicht mehr offiziell auf der Webseite zum Download angeboten wird, sondern nur noch auf Anfrage erhältlich ist. Die unterstützende Software ist jetzt in die Entwurfsumgebung ISE ab Version 5.2i von Xilinx integriert worden. Eine Liste der am häufigsten gestellten Fragen zu diesem Thema ist in [132] zu finden und ein aktuelles „Application Note“ in [136].

2.3 Analog Devices ADSP-2106x-Familie (SHARC)

Die ADSP-2106x-Familie von Analog Devices [7] gehört zur Gruppe der 32-Bit Fließkomma-DSPs mit 48-Bit Befehlswortbreite und ist seit September 1994 auf dem Markt erhältlich. Sie integrieren neben IEEE Fließkomma- auch Festkomma-Operationen in einer Super-Harvard-Architektur (SHARC). Die DSPs der Familie unterscheiden sich in der Größe des internen Daten- und Programmspeichers und in ihrer unterschiedlichen peripheren Konfiguration. Bild 2-5 zeigt ein Blockschaltbild der ADSP-2106x-Familie. Das Operationswerk besitzt drei parallele arithmetische Einheiten: eine MAC-Einheit, einen Barrel-Shifter und eine ALU, die alle arithmetisch/logischen Operationen in einem Takt ausführen, sofern die Daten ohne Konflikt auf den beiden internen Bussen übertragen werden können bzw. diese schon im Registerfile verfügbar sind. Die Prozessorfrequenz beträgt 33 MHz oder 40 MHz und der DSP erreicht damit bei 33 MHz 33,3 MIPS bzw. durchschnittlich 66 MFLOPS. In einem gemeinsamen Registerfile stehen zwei Registersätze (Bänke) mit jeweils sechzehn 40-Bit-Registern zur Verfügung. Die zweite Bank kann z. B. bei einem Kontextwechsel zwischen zwei Prozessen eingesetzt werden. Der Barrel-Shifter unterstützt die Manipulation einzelner Bits bzw. Bitfelder variabler Länge, eine Rotation und arithmetisch/logische Schiebeoperationen.

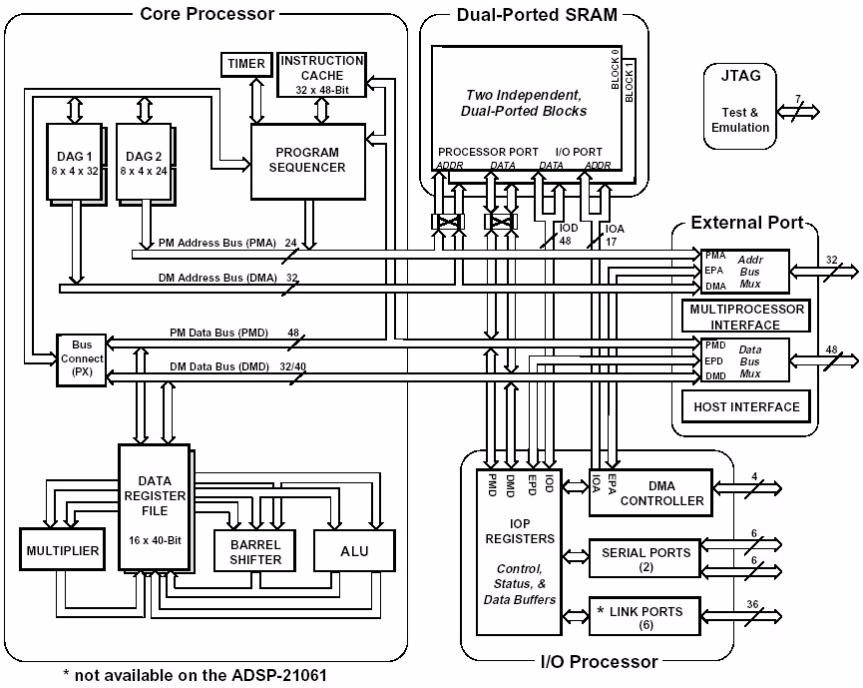


Bild 2-5. Blockschaltbild der ADSP-2106x-Familie (SHARC) (Quelle: [7]).

Die Multipliziereinheit arbeitet sowohl auf Festkomma- als auch auf Fließkomma-Operanden. Fließkomma-Multiplikationen können mit 32x32 ⇨ 40-Bit oder mit 40x40 ⇨ 64-Bit ausgeführt werden, wobei zusätzlich ein 80-Bit-Akkumulator im Multiplikationspfad liegt, aber nur bei Festkomma-Multiplikationen zum Einsatz kommt. Bei Fließkomma-Multiplikationen dagegen ist die ALU für eine anschließende Addition heranzuziehen. Die ADSP-2106x-Familie unterstützt vier Datentypen: 40-Bit und 32-Bit IEEE Fließkomma-, 16-Bit Fließkomma- und 32-Bit Festkomma-Zahlen. Der externe Adressbus ist 32-Bit breit und ermöglicht es, insgesamt 4 Gigawörter Multiprozessor- oder anderen externen Speicher zu adressieren. Intern stehen je nach Modell 512 KByte (4 MBit ADSP-21060), 256 KByte (2 MBit ADSP-21062) oder 128 KByte (1 MBit ADSP-21061) On-Chip-Speicher zur Verfügung. Die auf dem Chip vorhandenen peripheren Einheiten beinhalten einen Timer, zwei serielle Ports, sechs Link-Ports (nicht im ADSP-21061), zehn DMA-Kanäle (sechs beim ADSP-21061) und einen Host-Port. Spezielle Befehle zur Schleifenabarbeitung, insbesondere geschachtelte Schleifen (nested loops), werden im Programmsequenzer unterstützt. Wie für DSPs typisch, wird die Schleifenberechnung mittels Hardware durch einen Schleifenzähler und durch gespeicherte Start- und Endadressen der Schleife beschleunigt. Ein kleiner Instruktionscache mit 32 Einträgen puffert die Instruktionen, die nicht in den Programmsequenzer geladen werden konnten, weil ein Datentransfer über den Programmspeicherbus stattfand. Dieser Konflikt auf dem Programmspeicherbus würde sich ansonsten innerhalb einer Schleife x-mal wiederholen. Es existieren jeweils ein Adress-Generator für den Programm- und für den Datenspeicherbus. Der erste Port des vorhandenen On-Chip-Speichers (dual port) wird über einen Crossbar-Switch für Daten-und Adressbus multiplext. Der zweite Port ist exklusiv für den I/O-Prozessor reserviert. In einer 48-Bit breiten Instruktion sind maximal drei parallele Operationen enthalten.

2.4 System-on-Chip (SoC)

Der Begriff **System-on-Chip** kennzeichnet Halbleiter (ASIC), in denen ein nahezu komplettes System enthalten ist, wobei mindestens ein programmierbarer Prozessorkern die Hauptkomponente darstellt. Neben dem eigentlichen Prozessorkern sind zusätzliche Funktionsgruppen integriert, die über den chipinternen Datenbus angeschlossen sind. Dazu gehören z. B. Schnittstellen-Controller (PCI, USB), Timer (Watch Dog), aber auch zunehmend Sensoren, analoge Schaltkreise oder sogar mikromechanische Elemente. Zu diesen „Standard-Funktionalitäten“ kommt dedizierte kunden- und applikationsspezifische Hardware hinzu, die das SoC zum ASIC macht. Fehlt diese, ist das SoC eher den ASSPs zuzuordnen. Bei einem SoC wird z. B. ein RISC-Kern mit einem DSP-Kern, Speicher und Schnittstellen (Ethernet, seriell, parallel) auf einem Chip integriert.

2.4.1 Wiederverwendung („Design Reuse“)

Damit die Komplexität eines SoC bewältigt und der „Productivity Gap“ überwunden werden kann, kommen heutzutage vermehrt IPs, auch IP-Blöcke, Cores, Macros oder Virtual Components (VC) genannt, zum Einsatz. Laut einem aktuellen ITRS-Report der SIA [72] werden im Jahr 2010 90% der Fläche eines integrierten Schaltkreises mit IPs belegt sein, bei einer Speicherbelegung von 10% und 0% neuentwickelter Logik.

Ein **IP** kennzeichnet ein Objekt, das aus dem Know-How des Erzeugers entstanden ist und dessen Verwendung im Entwurf eines übergeordneten, komplexeren Systems eine aufwendige Neuentwicklung erspart. Sie werden in der Praxis von Dienstleistern eingekauft und ermöglichen im selben Zeitrahmen die Entwicklung einer in der Komplexität gestiegenen Schaltung. Es werden Hard-IPs, Soft-IPs und Firm-IPs unterschieden. In einzelnen Fällen gibt es fließende Übergänge zwischen den drei IP-Klassen, auf die an dieser Stelle nicht weiter eingegangen wird.

- *Hard-IP:* Es handelt sich um ein Hard-IP, wenn die Entwicklung eines IP vollständig abgeschlossen ist, d. h. die Komponente komplett verdrahtet und verifiziert in einer bestimmten Technologie vorliegt. Dazu gehören typisch größere Prozessorkerne, aber auch analoge und Mixed-Signal-Zellen mit einer engen Bindung zur verwendeten Technologie. Sie werden auf Maskenebene beschrieben, z. B. durch GDSII-Daten. Die Übertragbarkeit in eine andere Technologie ist nicht ohne weiteres gegeben und erfordert einen erheblichen Entwicklungsaufwand.
- *Soft-IP:* Die Bezeichnung Soft-IP kennzeichnet ein als synthetisierbare Beschreibung vorliegendes IP formuliert in einer HDL (VHDL oder Verilog). Die technologiespezifische Netzliste muss erst durch Synthesewerkzeuge erzeugt werden. Die anschließende Optimierung, Platzierung und Verdrahtung erfolgt mit geeigneten Programmen. Die Soft-IPs sind häufig parametrisierbar und technologieunabhängig. Sie sind daher sehr flexibel an die eigenen Anforderungen anpassbar. Dafür erfordern sie jedoch eine Verifikation nach der Synthese bzw. der Platzierung und Verdrahtung. Soft-IPs bieten ähnlich den Software-Bibliotheken einen Austausch zwischen verschiedenen Projekten. Wie beim Wechsel von in Assembler geschriebenen Programmen zu Hoch-

sprachen sind jedoch Leistungseinbußen beim Wechsel von handoptimierten Schaltungen zu Soft-IPs in Kauf zu nehmen.

- *Firm-IP:* Bei Firm-IPs existiert eine technologiespezifische Netzliste auf Gatterebene. Dafür wurde bereits die Synthese für die Zieltechnologie, aber noch nicht die Platzierung und Verdrahtung durchgeführt. Zu den Firm-IPs zählen fast alle IPs für FPGAs. Sie lassen sich nur begrenzt in andere Technologien überführen.

Komplexe Kerne, wie Prozessoren, eignen sich nicht als Soft-IP, da das Ergebnis der neu durchzuführenden Synthese einen erheblichen Verifikationsaufwand bedeuten würde. Dies ist umso schwieriger, da die Detailkenntnisse des Anbieters des IP nicht immer zur Verfü-

Tabelle 3: Übersicht der 2002 verfügbaren DSP-Kerne für SoC-Entwürfe (Quellen: [18, 21, 22, 23, 45]).

Hersteller	Familie	Fest-, Fließkomma oder beides	Datenbreite [bits]	Instruktionswortbreite [bits]	Taktrate [MHz]	BDTiMark2000	adressierbarer Speicherbereich	Kernspannung [V]
3DSP	SP-3	Fest	32	32	225	n/a	4G	1,0
	SP-5	Fest	32	32	225	1720	4G	1,0
Clarkspur Design	CD245x	Fest	16/24	16	50	n/a	64K	5,0
DSP Group	TeakLite	Fest	16	16	190	n/a	256K	1,0
	Oak	Fest	16	16	180	n/a	128K	1,0
	Teak	Fest	16	16	180	n/a	8M	1,0
	Palm	Fest	16/20/24	16/32	180	n/a	32M	1,0
	Cedar	Fest	16/24/32	16/32	450	n/a	n/a	1,2
Infineon	Carmel 10xx	Fest	16	24/48	166	n/a	4G	1,8
	TC1MP-S (TriCore 1)	Fest	32	16/32	250	n/a	4G	1,5
LSI Logic	ZSP400	Fest	16/32	16	180	850	256K	1,2
	ZSP600	Fest	16/32	16/32	300	n/a	64M	1,0
StarCore	SC110	Fest	16	16	300	1540	n/a	1,5
	SC140	Fest	16	16	300	3430	n/a	1,5
Super-H	SH-4	beides	32	16	266	830	4G	1,2
	SH-5	Fest	8/16/32/ 64	16/32	400	1560	4G	1,2

gung stehen. Ist ein Prozessorkern als Hard-IP für die verwendete Technologie vorhanden, herrscht hier die größte Entwurfssicherheit. Der Anteil der Soft-IPs wird jedoch in Zukunft steigen, wenn Standards ausgereift und Mechanismen zum Schutz des Know-How verbessert sind.

Die Wiederverwendung eines IP stellt hohe Anforderungen an die Dokumentation und die Überprüfbarkeit der Qualität. Bei der Integration von IPs, insbesondere von Drittanbietern, in ein SoC zusammen mit selbst entwickelten Unterblöcken ist es erforderlich, die korrekte Interaktion zwischen allen Komponenten des Systems zu überprüfen. Besondere Probleme gibt es noch aufgrund fehlender standardisierter Buskonzepte (AMBA [5], Wishbone [128], CoreConnect [37]). Ein Vergleich der vorhandenen Buskonzepte wird in [109] vorgenommen. Es existieren daher verschiedene Gruppen, die sich bemühen, einen Standard für die internen Schnittstellen zu definieren (die wichtigste ist die VSI Alliance - Virtual Socket Interface Alliance). So lange die Standardisierung nicht erfolgt ist, sind die IPs, sofern dies möglich ist, anzupassen und die Änderungen in zusätzlichen kritischen Verifikationsläufen zu überprüfen. Hierbei ist das Vorhandensein von simulierbaren Interface-Modellen wichtig. Ein weiteres Problem ist die Frage, ob ein IP auch die geforderte Lösung für ein Teilproblem darstellt. Beides kann eigentlich nur effizient mittels einer Simulation überprüft werden, sofern ein Simulationsmodell des IP vorhanden ist.

Für den Bereich der digitalen Signalverarbeitung stehen DSP-Kerne in Form von Hard- oder Soft-IPs zur Verfügung. Der OakDSP genannte Prozessorkern von Atmel [14], eine Lizenz der DSP Group, ist ein 16-Bit Festkomma-DSP, der in eigene Entwürfe für ein ASIC bzw. ASSP eingebettet werden kann. Dieser ist in einigen Punkten benutzerdefiniert parametrisierbar, z. B. sind die Register, RAM-, ROM- und I/O-Blöcke variabel. Im Paket enthalten sind unterschiedliche Software- und Hardware-Entwicklungswerkzeuge. Zu ersterem gehört ein optimierender C/C++-Compiler und Debugger, zu letzterem ein auf dem Chip integriertes Emulationsmodul. Für die Simulation des Systems wird ein erweiterbarer Simulator namens ASSYST verwendet. Der Prozessorkern wird meistens eingesetzt, wenn ein günstiges Verhältnis zwischen Kosten und Leistungseigenschaften erforderlich ist. Tabelle 3 auf Seite 21 zeigt eine Übersicht der verfügbaren DSP-Kerne für den Bereich der digitalen Signalverarbeitung für SoC/ASIC-Entwürfe.

2.4.2 Programmierbare SoC (SoPC)

Einen weiteren Trend, der zu beobachten ist, stellen Mischarchitekturen aus den im Kap. 2.1 erläuterten Komponenten dar. Hierbei werden festverdrahtete Standardbauelemente und programmierbare Logik auf einem Chip monolithisch integriert. Für diese Mischarchitekturen hat sich das Akronym **ASPP** (Application Specific Programmable Product) etabliert. Es steht sowohl für in ASICs eingebettete programmierbare Logik (FPGA) als auch umgekehrt für programmierbare Logik, in die komplexe festverdrahtete Unterbaugruppen integriert sind. Solche Architekturen werden üblicherweise als „hybrid“ bezeichnet. Der Vorteil gegenüber einer externen Kopplung ist, dass die zusätzlichen Latenzzeiten entfallen und sich damit der notwendige Kommunikationsaufwand reduziert. Weiterhin ist es von Vorteil, dass der programmierbare Hardwareteil dem eigenen Einsatzgebiet angepasst werden kann.

Im Gegensatz zu einem konventionellen Prozessor, bei dem variable Software auf fester Hardware ausgeführt wird, wird bei einem Prozessor, gepaart mit rekonfigurierbarer Hardware, auch die Hardware ganz oder teilweise an die aktuelle Problemstellung angepasst. Auf diese Weise lassen sich bei bestimmten Anwendungen nennenswerte Verkürzungen der Rechenzeit erreichen. Den Leistungssteigerungen und der Flexibilität, die durch die zusätzliche rekonfigurierbare Hardware erreicht werden können, steht aber eine deutlich kompliziertere Programmierung sowie die Beherrschung der Aufteilung des Algorithmus in Hard- und Software gegenüber. Die Kombination mit programmierbarer Hardware eignet sich am Besten für Anwendungen, in denen entweder sehr viele Berechnungen auf einer kleinen Datenmenge auszuführen sind (Verschlüsselung) oder solche, die auf einer großen Datenmenge vergleichsweise einfache Berechnungen durchführen (Signal- und Bildverarbeitung). Die kontrolldominierten Teile außerhalb der inneren Schleifen der Algorithmen werden in der Regel effizienter auf konventionellen Prozessoren ausgeführt. Deshalb sollte auch immer einer dieser konventionellen CPUs präsent sein, sei es nur als eine der möglichen (Teil-)Konfigurationen der adaptiven Komponente. Bild 2-6 zeigt die verschiedenen Möglichkeiten der Anbindung der rekonfigurierbaren Hardware an einen konventionellen Prozessor (GPP, DSP, Mikrocontroller).

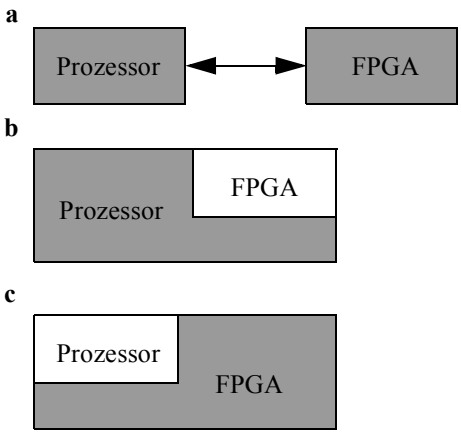


Bild 2-6. Kombination eines Prozessors mit rekonfigurierbarer Hardware (FPGA) mittels **a** externer Anbindung, **b** monolithischer Integration der FPGA-Strukturen in den Prozessorchip oder **c** Integration eines Prozessorkerns als Hard-IP in das FPGA.

Ein Beispiel für in ASICs als IP einzubettende FPGA-Kerne ist **VariCore** von Actel (EPGA - Embedded FPGA) [110]. Sie sind für einen Einsatz in ASIC- und ASSP-basierten SoC-Lösungen geeignet. IBM und Xilinx kündigten mittlerweile an, mit „XBlue“ auch programmierbare Hardware in ASICs zu integrieren [129]. Das Problem dabei ist, wieviel programmierbare Hardware integriert werden muss, um auch zukünftig gewünschte Systemeigenschaften implementieren zu können und damit flexibel im Markt zu sein.

Bei programmierbaren „System-on-Chip“, mit **SoPC** (System-on-programmable-Chip) abgekürzt, wird die programmierbare Logik als Plattform für die Systemintegration auf einem Chip genutzt und stellt eine Konvergenz von ASPP und SoC dar. Der Entwickler realisiert Software-Blöcke in Hardware, die der Prozessor im System als kundenspezifische Befehle nutzen kann. Aus der Sicht des Software-Entwicklers stellen diese in Hardware ausgelagerten Software-Blöcke lediglich Funktionsaufrufe in einer Programmiersprache

(C oder Assembler) dar. Dahinter steht allerdings keine sequentielle Folge von Software-Befehlen, sondern ein Hardware-Block, der die Funktionalität zur Verfügung stellt. SoPCs erlauben in der Regel die Entwicklung von Mikrocontroller-Applikationen, für die eine Vielzahl von IPs zur Verfügung stehen.

Verschiedene Akronyme kennzeichnen derzeit die Lösungen der Hersteller im Bereich programmierbarer SoCs:

- Altera mit seiner Excalibur-Familie spricht von einer Lösung im SoPC-Segment,
- die Firma Cypress Microsystems bezeichnet ihre SoPCs als PSoC (Programmable System-on-Chip),
- Mentor Graphics bietet Synthese- und Verifikationswerkzeuge für FPSoCs (Field Programmable System-on-Chip) an,
- die Firma Triscend bezeichnet ihre SoPCs als CSoC (Configurable System-on-Chip),
- Xilinx nennt seine „Virtex-II Pro“-Bausteine Plattform-FPGAs.

Die verwirrende Vielfalt der Akronyme ist den Marketingabteilungen der Hersteller zu verdanken. Sie kennzeichnen alle mittels programmierbarer Logik (FPGA) realisierte SoCs und lassen sich den applikationsspezifisch programmierbaren Produkten (ASPP) zuordnen. SoPCs können als Konvergenz dreier Technologien angesehen werden: ASICs, programmierbare Logik und Software.

Hauptsächlich sind zwei Möglichkeiten für die Realisierung von SoPCs anhand der Methodik der Integration des Prozessors bzw. der Prozessoren zu unterscheiden. Zum einen kann sie mittels Hard-IP und zum anderen mittels Soft-IP erfolgen. Es sind auch Mischformen denkbar, so kann zu einem als Hard-IP in einem FPGA vorhandenen Prozessor ein Soft-IP hinzugefügt werden.

Ein mit Hard-IPs realisiertes SoPC integriert einen oder mehrere festverdrahtete Prozessorkerne mit definierten Schnittstellen zur benutzerdefiniert programmierbaren umgebenden Hardware. Oder anders ausgedrückt: In einem FPGA als programmierbare Hardware sind ein oder mehrere Prozessorkerne mit definierten Schnittstellen auf dem Chip integriert.

Die Vorteile der Integration der Prozessoren als Hard-IP in programmierbare Logik sind der deutlich geringere Flächenaufwand gegenüber einer Realisierung als Soft-IP, sowie die meist deutlich bessere Leistungsfähigkeit (Frequenz, Verlustleistung). Nachteilig wirkt sich die reduzierte Flexibilität beim Entwickler aus, der die vorgefertigte festverdrahtete Komponente nicht mehr beeinflussen kann. Besonders sinnvoll ist diese Vorgehensweise jedoch, wenn sich die festverdrahtete Komponente nur unzureichend hinsichtlich Flächenaufwand und Zeitverhalten in programmierbarer Logik abbilden lässt. Beispiele dafür bieten die Fließkomma-Arithmetik, Speichermodule (z. B. Cache) oder auch Kommunikationsschnittstellen, deren Frequenzen oberhalb derer liegen, die mit programmierbaren Logikbausteinen heutzutage realisierbar sind. Einzusetzen ist diese Form der SoPCs immer dann, wenn eine hohe Rechenleistung benötigt wird und komplexe Echtzeitbetriebssysteme zum Einsatz kommen.

Von der Firma Xilinx gibt es den mit „**Virtex-II Pro**“ bezeichneten FPGA-Baustein, der bis zu vier IBM PowerPC405-Kerne auf dem Die als Hard-Macro bei 300 MHz integriert. Die Bausteine der **Excalibur**-Familie von Altera vereinen einen ARM-Prozessor als Hard-IP mit Speicher, Peripherie (UART, Interrupt-Controller etc.) und programmierbare Logik auf einem Chip. Auch von Konkurrenzfirmen gibt es mittlerweile entsprechende Produkte. So verwendet Atmel einen 8-Bit RISC-Kern von AVR für die **FPSLIC**-Familie, und Triscend integriert einen ARM7-Kern in ihre FPGAs der **A7**-Familie.

Eine Übersicht aktuell verfügbarer SoPCs, bei denen Prozessorkerne auf dem Chip integriert sind, enthält Tabelle 4. Für weitere Informationen zu den aufgeführten Baustein-Familien sei auf die entsprechenden Datenblätter der Hersteller verwiesen [4, 13, 42, 43, 75, 99, 108, 131].

Tabelle 4: Übersicht der wichtigsten verfügbaren SoPCs, bei denen programmierbare Logik zusammen mit festverdrahteten Baugruppen auf einem Chip monolithisch integriert sind.

Hersteller	Familie	Technologie
Altera	Excalibur	ARM922T 32-Bit RISC
Atmel	AT94K (FPSLIC)	AVR 8-Bit RISC
Cygnal	C8051F	8-Bit 8051 CISC
Cypress Microsystems	PSoC	M8C 8-Bit RISC
Lattice	ORCA FPSC	embedded ASIC macrocells (bus-interfaces)
Quicklogic	QuickMIPS	MIPS32 4Kc 32-Bit RISC
Triscend	E5 A7	accelerated 8-Bit 8051 CISC ARM7TDMI 32-Bit RISC
Xilinx	Virtex-II Pro	Power-PC 32-Bit RISC

Ein zweiter gangbarer Weg für SoPCs stellt die Einbindung von Soft-IPs der Prozessoren in programmierbare Logik dar. Die Vorteile dieses Ansatzes liegen auf der Hand: Bei einem als Soft-IP realisierten Prozessor können alle Elemente parametrisiert in die Implementierung einfließen. Die Anpassungen an die Anforderungen der Aufgabe sind sehr flexibel durchführbar und sind mit denen für ASIPs zu vergleichen. Damit ist die Bandbreite der möglichen Anwendungslösungen größer. Die Nachteile, die es bei einem Prozessor als Soft-IP gibt, wurden bereits im vorhergehenden Absatz und im Kap. 2.4.1 genannt.

Als Soft-IP vorliegende Prozessoren sind mittlerweile reichlich verfügbar [45, 57, 59, 91], z. B. auch der klassische Intel-Mikroprozessor 8051 [2, 34, 44, 92]. Hervorzuheben seien an dieser Stelle lediglich einige davon. Zum Beispiel der eingebettete Prozessor **Nios** der Firma Altera [88] implementiert einen RISC-Prozessor mit 16-Bit Befehlssatz und konfigurierbarer Breite des Datenpfades (16- oder 32-Bit). Er ist für eine Einbettung in eine benutzerdefinierte Hardwareumgebung in einem PLD von Altera geeignet. Im SOPC-Builder genannten Werkzeug lassen sich außerdem IPs von Drittanbietern (z. B. ARM) in die PLDs einbinden. Weitere Blöcke für das Debuggen der Software können zum Soft-IP hinzugefügt werden, z. B. für die In-Circuit-Emulation. Ein besonderes Merkmal dieses Pro-

zessorkerns ist die Möglichkeit, eigene Instruktionen des Prozessors in der Art eines ASIP zu definieren. Damit können häufig vorkommende Operationen und Funktionen in Hardware ausgelagert und damit beschleunigt werden.

Das mit **ARC** bezeichnete und im AllianceCORE-Programm von Xilinx [3] verzeichnete Soft-IP der Firma ARC [11] implementiert einen 32-Bit RISC-Prozessor in zwei Konfigurationen, eine für Basis- und eine für DSP-Aufgaben geeignete Variante. Als Debug-Schnittstelle ist ein Host-Port für den Parallel-Port eines PC eingebaut, alternativ ist JTAG und RS232 möglich. Über diese Schnittstelle lassen sich sowohl Register- und Speicherstellen manipulieren als auch der Prozessorkern stoppen und das Programm Step-by-Step, also Befehl für Befehl, ausführen. Der Befehlssatz umfasst einen Grundstock an Befehlen, zusätzliche Befehle lassen sich wie bei ASIPs applikationsspezifisch definieren.

Ein weiteres Beispiel für einen konfigurierbaren Prozessorkern für SoPCs stellt die **LEON**-Architektur [77] dar. Hierbei handelt es sich um einen 32-Bit-Prozessor als Soft-IP, der kompatibel zur „SPARC V8“-Architektur ist. Der synthetisierbare VHDL-Code des Prozessors wurde unter die LGPL (Lesser GNU Public License) gestellt und steht der Allgemeinheit zur Verfügung. Aufgrund der Kompatibilität zur SPARC-Architektur, lassen sich deren Entwicklungswerkzeuge auch bei LEON einsetzen.

2.5 Eingebettete Systeme

Bevor der Begriff eingebettetes System erläutert werden kann, ist zunächst der Begriff System im technisch orientierten Sinne zu definieren: Unter einem **System** wird die Zusammenführung von Elementen oder auch Untersystemen verstanden, das der Erfüllung von Aufgaben dient und über definierte Schnittstellen mit der Außenwelt verbunden ist. Ein **eingebettetes System** ist ein aus Hardware und Software bestehendes Rechnersystem, das Bestandteil eines übergeordneten größeren Systems ist und meist im Verborgenen einen bestimmten technischen Zweck erfüllt (in einen technischen Kontext eingebettet ist), jedoch für die Funktions- und Leistungsfähigkeit des Gesamtsystems entscheidend ist. Es kann als System-on-Chip oder als Board-Level-System implementiert sein. Eine Realisierung als SoC ist immer dort attraktiv, wo die geplanten Stückzahlen ausreichend hoch sind.

Beispiele für eingebettete Systeme sind Vermittlungssysteme oder Endgeräte in der Telekommunikation (Mobiltelefone), Geräte der Unterhaltungselektronik (CD/MP3-Player) sowie verteilte Systeme in der Automobil- (ABS) und Automatisierungstechnik (Maschinensteuerung). Komplexere eingebettete Systeme sind zumeist heterogen aufgebaut und enthalten Mikrocontroller, DSPs, Co-Prozessoren, analoge und digitale applikations- und kundenspezifische Schaltungen und Speicherbausteine.

Die Funktionalität eines eingebetteten Systems hängt nicht mehr nur von der Hardware, sondern in besonderem Maße auch von der im System implementierten Software ab. Die Vorteile sind die zusätzliche Funktionalität und die Flexibilität in der Entwicklung und im praktischen Einsatz. Allerdings wird durch die Einbeziehung der Software-Entwicklung in den Entwurfsfluss der Lösungsprozess der Aufgabenstellung komplizierter. Es besteht der

Bedarf, Hard- und Software zusammen in einer gemeinsamen Simulationsumgebung zu verifizieren, eine Aufgabe, die heute noch nicht befriedigend gelöst ist.

Bei den mit Prozessorkernen ausgestatteten eingebetteten SoCs handelt es sich in der Regel um autonome Systeme, die auf Signale der Umgebung reagieren und zeitlich komplizierte/kritische Steuerungsfunktionen übernehmen. Damit repräsentieren sie Echtzeit- oder Realzeit-Applikationen mit Interrupts, zeitkritischen DMA-Transfers etc. Auf diesen Systemen kommt häufig ein RTK (Real Time Kernel) oder ein RTOS (Real-Time Operating System) zum Einsatz. Prinzipiell wird die Software in drei Schichten organisiert (Top-Down):

1. *Applikation:* eigentliche Anwendung,
2. *Betriebssystem:* komplexe Organisationsaufgaben, Prozessverwaltung,
3. *Hardwarenahe Routinen (BIOS):* Funktionen, die direkt das Hardwareverhalten beeinflussen (Treiber).

Bei der Integration eines Prozessorkerns in ein eingebettetes SoC und der Ausrichtung auf eine einzige Anwendung ist die Betriebssystemschicht durch einen minimalen RTK ausreichend repräsentiert.

Echtzeit- oder **Realzeitbetrieb** bedeutet in diesem Zusammenhang der Betrieb eines Rechnersystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, und zwar so, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu bestimmten Zeitpunkten anfallen (nach DIN 44300, 1985). Unter einem **Echtzeitsystem** wird ein Rechnersystem verstanden, das eine Reaktion auf äußere Ereignisse in einer Zeitspanne gewährleisten muss, während der diese Ereignisse noch relevant sind.

Es wird zwischen weichen und harten Echtzeitsystemen unterschieden. **Harte Echtzeitsysteme** führen bei Verletzung bestimmter Zeitbedingungen zu einem inakzeptablen Systemverhalten. Beispiele hierfür finden sich in der Fahrzeugtechnik und der Telekommunikation. Bei **weichen Echtzeitsystemen** gibt es kein lebensbedrohliches Risiko durch ein Fehlverhalten des Systems. Beispiele finden sich in den Bereichen Multimedia und Virtual Reality. Welche Zeitspanne mit Echtzeit in Verbindung gebracht werden muss, ist definitions- und applikationsabhängig. Auf jeden Fall ist bei einer Echtzeitanwendung die maximal zulässige (hart oder weiche) Antwortzeit eines Systems auf externe Ereignisse bekannt.

2.6 Hardware/Software-CoDesign

Der Begriff **HW/SW-CoDesign** bezeichnet den integrierten Entwurf von eingebetteten Systemen, die sowohl Hardware- als auch Softwarekomponenten enthalten. Gesucht werden Entwurfsmethoden, die es ermöglichen, Hardware- und Softwarekomponenten gleichzeitig zu entwerfen und dabei Entwurfsalternativen zu untersuchen.

Motiviert wird der Einsatz von Methoden des HW/SW-CoDesigns durch die zunehmende Komplexität und Heterogenität, die durch den Technologiefortschritt und die Vielfalt heutiger Anwendungen begründet sind. Dies gilt besonders im Bereich der eingebetteten Systeme und Echtzeitsysteme. Es bedarf rechnergestützter Werkzeuge auf möglichst vielen Ebenen des Entwurfs, um die Komplexität in den Griff zu bekommen und die Produktivität zu steigern. Gefordert und notwendig ist ein durchgängiger Entwurfsfluss, angefangen von der Systemkonzeption bis hin zur Systemimplementation. Ein zentrales Problem im Entwurfsprozess eines eingebetteten Systems ist die Überwachung und Integration des parallelen Hardware- und Software-Entwurfs. Eine frühe Fehlererkennung erfordert die Kontrolle von Konsistenz und Korrektheit, die umso aufwendiger wird, je detaillierter der Entwurf ausgearbeitet ist.

Problemstellungen des HW/SW-CoDesigns treten sowohl bei GPP-Systemen (PCs, Workstations) als auch bei eingebetteten Systemen und SoCs auf. Bei GPP-Systemen und eingebetteten Systemen werden Prozessoren und Compiler nebst Betriebssystem gemeinsam entwickelt. Entwurfsalternativen gibt es hinsichtlich des Instruktionssatzes, der Anzahl an Pipelinestufen und auch der Kontroll- und Datenpfade. Bei SoCs, die für ein eingebettetes System konzipiert sein können, müssen Entwurfsalternativen hinsichtlich des verwendeten Prozessorkerns und der HW/SW-Partitionierung berücksichtigt werden und es sind das Betriebssystem (RTOS, RTK) und anwendungsspezifische Hardware gemeinsam zu entwickeln.

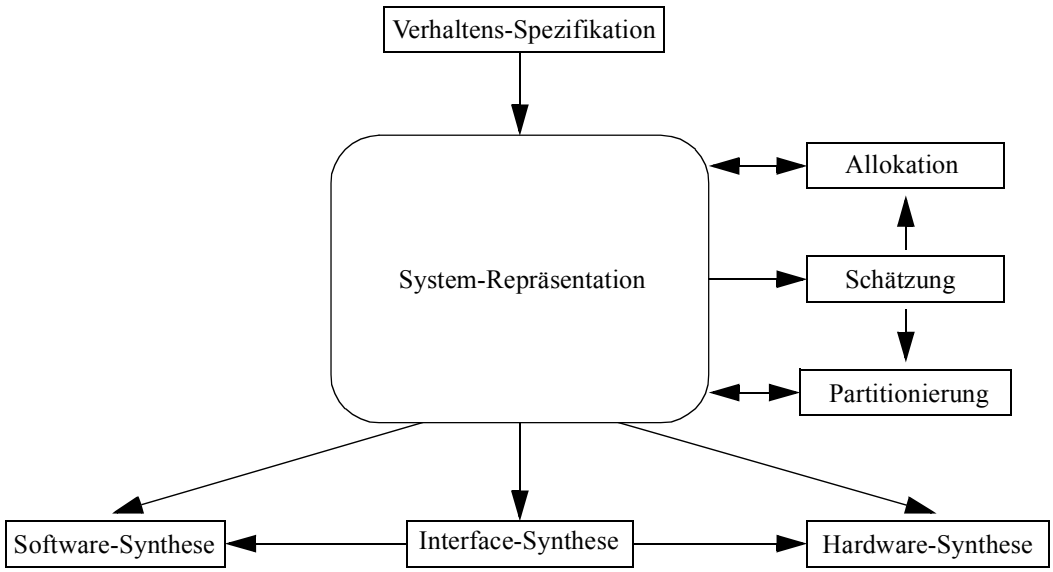


Bild 2-7. Entwurfsablauf im HW/SW-CoDesign.

Grob umrissen wird im HW/SW-CoDesign, ausgehend von einer (ausführbaren) Spezifikation des gewünschten Systemverhaltens und nach einer Systempartitionierung, eine Synthese der Software, der Hardware und der HW/SW-Schnittstellen durchgeführt.

Eine ausführbare Spezifikation des Systems (**Verhaltens-Spezifikation**) liegt vor, wenn sie simulierbar ist und somit die Möglichkeit zur Verifikation bietet. Heutzutage wird in der Regel C/C++ und VHDL eingesetzt, neue Spezifikationssprachen wie z. B. SystemC [106]

haben sich noch nicht durchgesetzt. Mit SystemC wurde eine Sprache basierend auf C/C++ entwickelt, mit der digitale Schaltungen auf Systemebene spezifiziert und zusammen mit Software-Funktionen verifiziert werden können.

In einer sogenannten **Allokationsphase** werden zunächst die Komponenten ausgewählt, die benötigt werden, wie z. B. Speicher, Prozessoren oder anwendungsspezifische integrierte Schaltungen. Diese sind durch Parameter gekennzeichnet, wie Zugriffszeiten bei Speichern, Instruktionen pro Zeiteinheit bei Prozessoren, oder Verlustleistung bei integrierten Schaltungen. Schwierigkeiten ergeben sich aufgrund der Heterogenität der Hardware (ASIC, Mikroprozessor, Mikrocontroller, FPGA) und der Vielzahl, oft gegensätzlicher Bedingungen, z. B. Kosten, Zeit (Entwurf, Herstellung) und Effizienz (Ausführungszeit, Speicherplatzverbrauch, Leistungsverbrauch).

Die Systemspezifikation wird in Hardware- und Softwarekomponenten aufgeteilt. Dieser Vorgang wird als **HW/SW-Partitionierung** bezeichnet. Da jede neue Allokation mit zugehöriger Partitionierung eine mögliche Systemimplementierung erzeugt, müssen diese miteinander verglichen werden. Dazu wird eine **Schätzung** der Eigenschaften der Implementierungsalternativen durchgeführt. Sind alle oder – aufgrund der langen Rechenzeiten – die wichtigsten Systemalternativen untersucht, wird die günstigste Alternative anhand der vorgegebenen Randbedingungen bestimmt und dann die Software- und Hardware-Teile in der Phase der System-Synthese erzeugt.

Eine anschließend durchzuführende **Interface-Synthese** generiert für die Schnittstellen und Protokolle, die zur Kommunikation zwischen den Komponenten eines Systems (Hardware und Software) notwendig sind, die Software-Funktionen und zugehörige Hardwarebeschreibungen. Die partitionierten Softwarekomponenten können auf einem oder mehreren Prozessoren ausgeführt werden. Dafür werden sie mit Hilfe von optimierenden Compilern für den oder die Prozessoren übersetzt. Dies geschieht in der Phase der **Software-Synthese**, in der Verfahren der Codegenerierung und -optimierung zum Einsatz kommen. Typische Werkzeuge dafür sind Compiler für die Hochsprachen C oder C++. Unter Software-Synthese wird in dieser Arbeit nicht die Erzeugung von Code, beispielsweise in C formuliert, aus den partitionierten Software-Teilen verstanden, sondern die davon ausgehende weitere Verfeinerung der Implementation in ein Maschinenprogramm, das für eine gegebene Zielarchitektur übersetzt und optimiert wird. Parallel zur Software-Synthese wird in der Phase der **Hardware-Synthese** die Spezifikation der Hardware-Teile soweit verfeinert, dass sie die Eigenschaften der Implementation auf Systemebene nachbildet. Dazu wird die in der HW/SW-Partitionierung erzeugte synthetisierbare Beschreibung, formuliert in einer HDL, in eine technologiespezifische Netzliste überführt und anschließend auf die Zieltechnologie abgebildet. Auch unter dem Begriff Hardware-Synthese ist in dieser Arbeit, ähnlich wie bei der Software-Synthese, nicht die Erzeugung einer Verhaltensbeschreibung aus den partitionierten Hardware-Teilen zu verstehen. Zur Hardware-Synthese gehören, von einer Verhaltensbeschreibung ausgehend, sowohl die Architektursynthese als auch die Logiksynthese.

Die Systemintegration schließlich führt Hardware- und Software-Teile zusammen und es wird überprüft, ob alle spezifizierten Eigenschaften des Systems erfüllt werden. Der beschriebene Entwurfsfluss im HW/SW-CoDesign ist in Bild 2-7 dargestellt.

Die Anforderungen an EDA-Systeme (Electronic Design Automation) für das HW/SW-CoDesign beinhalten grob umrissen die Unterstützung bei der Untersuchung des Entwurfsraums nach einer günstigen HW/SW-Partitionierung (design space exploration), der gleichzeitigen parallelen Entwicklung von Hard- und Software und der Verifikation mit unterschiedlichen Modellen auf verschiedenen Abstraktionsebenen. Beispiele für solche Systeme sind CoWare [31], POLIS [16] und Ptolemy [98].

Als Systemspezifikation werden beim POLIS-System verschiedene Eingabesprachen akzeptiert, dazu gehören ESTEREL [27], graphische FSMs und ein Verilog- und VHDL-Subset. POLIS ist auf reaktive kontrolldominierte eingebettete Systeme (Mikrocontroller + applikationsspezifische Hardware) spezialisiert, so dass die verwendeten Methodiken nicht oder nur teilweise auf die digitale Signalverarbeitung angewendet werden können. Die zum Beispiel beim POLIS-System verwendeten CFSMs (Co-design Finite State Machine) können nur Aufzählungstypen oder Datentypen enthalten, die vom Typ integer abgeleitet sind. Dies ermöglicht nicht die Modellierung von Festkomma- bzw. Fließkomma-Typen, die jedoch in der digitalen Signalverarbeitung von essentieller Bedeutung sind.

Der Schwerpunkt des CoDesign-Werkzeugs Ptolemy liegt primär auf heterogenen eingebetteten Systemen. Diese werden in Ptolemy in einem heterogenen Systemmodell hierarchisch modelliert, wobei unterschiedliche Beschreibungsformen Verwendung finden (Java, FSM - Finite State Machine, SDF - Synchronous Data Flow etc.). Das Systemmodell besteht aus mehreren Domänen, denen Beschreibungsformen und Berechnungsmodelle zugeordnet sind. Die von Ptolemy adressierten Systeme sind komplex aufgrund der unterschiedlichen Operationen, die darin vereint sind. Dazu zählen Algorithmen der digitalen Signalverarbeitung, Verfahren der Regelungstechnik oder auch die Realisierung von Benutzeroberflächen.

2.6.1 Hardware/Software-Partitionierung

Zu Beginn einer SoC-Entwicklung steht zunächst die Frage der Partitionierung der Aufgabenstellung anhand vorgegebener Optimierungskriterien in einen Hardware- und einen Software-Teil, d. h., es muss entschieden werden, welcher Teil durch Hardware und welcher durch Software-Programme realisiert bzw. ausgeführt wird. Kostengesichtspunkte sprechen für eine reine Software-Realisierung, während Aspekte der Geschwindigkeit eher zu einer reinen Hardware-Realisierung tendieren lassen.

Es gibt zwei Vorgehensweisen für die HW/SW-Partitionierung. Basierend auf einer vollständigen Softwarelösung wird die Spezifikation, die in einer höheren Programmiersprache vorliegt, auf geeignete Hardware-Anteile hin untersucht und inkrementell modifiziert (z. B. COSYMA [38, 65]). Sollte sich dabei keine geeignete Hardware-Abbildung realisieren lassen oder bringt der zusätzliche Hardware-Teil einen zu hohen Kostenaufwand für die Kommunikation mit sich, so wird die gesamte Spezifikation als Programm auf der unveränderlichen Hardware ausgeführt, ohne Geschwindigkeitsgewinn, jedoch zumindest lauffähig. Der zweite Ansatz zur Lösung des Partitionierungsproblems geht von einer vollständig in Hardware realisierten Spezifikation aus. Davon werden dann schrittweise Softwareteile extrahiert, bis alle Randbedingungen erfüllt sind [63]. Diese Vorgehensweise ist

allerdings nur für relativ kleine Schaltungen und eingebettete Systeme realistisch. Denn, ist ein Prozessor im System vorhanden, wird zunächst versucht werden diesen auszulasten, bevor zusätzliche Hardware generiert wird.

Ausgehend von den im vorherigen Absatz beschriebenen Möglichkeiten der HW/SW-Partitionierung, lassen sich die feinkörnige (fine-grained) und die grobkörnige (coarse-grained) HW/SW-Partitionierung unterscheiden. Bei einer feinkörnigen HW/SW-Partitionierung werden einzelne Befehle des Software-Codes in Hardware ausgelagert, beispielsweise die Addition zweier Operanden. Diese Möglichkeit kann nicht angewendet werden, wenn der Prozessor und die Hardware synchron zu demselben Taktsignal arbeiten und die Ausführung eines Befehls ein oder zwei Takte benötigt, da der zusätzliche Kommunikationsaufwand den Durchsatz des Systems verringert. In diesem Fall ist es sinnvoll, eine grobkörnige HW/SW-Partitionierung anzuwenden, bei der ganze C-Funktionsblöcke in Hardware ausgelagert werden. Die Berechnung trigonometrischer Funktionen, eine FFT (Fast Fourier Transformation) oder eine DCT (Discrete Cosine Transform) sind Beispiele dafür.

Für das HW/SW-CoDesign unerlässlich ist das statistische Profiling. Es verhilft, ein Optimum frühzeitig im Entwurfsprozess zu finden. Profiling wurde im Grunde entwickelt, um Software zu optimieren und daraus folgend Applikationen zu beschleunigen. Bei einer HW/SW-Partitionierung kann ein Profiling aufzeigen, welche Teile eines Programms langsamer laufen bzw. öfter als erwartet aufgerufen werden und damit als Kandidaten für eine Auslagerung in Hardware zur Beschleunigung in Frage kommen. Diese Programmteile müssen aber nicht unbedingt auch für eine Auslagerung in Hardware geeignet sein. Aufgrund der auftretenden Latenzzeiten an den HW/SW-Schnittstellen ist z. B. der Datenfluss zwischen Hardware und Software zu minimieren. Sind vom gefundenen Programmteil viele Daten auszutauschen, so kann der zusätzliche Kommunikationsaufwand den Geschwindigkeitszuwachs durch die Auslagerung in Hardware zunichte machen. Das heißt, Profiling gibt Hinweise auf Partitionierungsentscheidungen, sollte aber nicht als einziges Kriterium dafür herangezogen werden.

Ein Beispiel für ein HW/SW-Partitionierungswerkzeug ist der ANSI-C Compiler der BRASS Research Group der Universität Berkeley [60]. Für eine vorgegebene selbstentwickelte Chiparchitektur (Garp) wurde ein ANSI-C Compiler entwickelt, der die Partitionierung und Implementierung der Hardware- und der Software-Teile aus einem homogenen Systemmodell übernimmt. Die Garp-Architektur enthält einen Mikroprozessor mit rekonfigurierbarer Logik als transparenten Co-Prozessor. Beide arbeiten synchron mit derselben Taktfrequenz. Die algorithmische Beschreibung in C wird auf Schleifen hin untersucht, die in Hardware ausgelagert werden können. Dabei sind besondere Techniken für die Abbildung der C-Beschreibung auf die rekonfigurierbare Hardware (FPGA-Struktur) notwendig, für die Compiler-Lösungen aus dem Bereich der VLIW-Prozessoren eingesetzt werden [33]. Die Auslagerung in Hardware kann mehrere Module umfassen, die mit Hilfe eines einfachen synthetisierten Controllers zum entsprechenden Zeitpunkt in die rekonfigurierbare Hardware geladen werden.

Auch beim POLIS-System werden aus partitionierten HW/SW-Teilen einer internen Repräsentation, wofür CFSMs zum Einsatz kommen, Software- und Hardwarebeschrei-

bungen erzeugt. Die generierte Software umfasst dabei ein komplettes RTOS und die generierte Hardware auf RT-Ebene kann in den Formaten BLIF, XNF oder als HDL (VHDL, Verilog) ausgegeben werden. Das XNF-Format von Xilinx ermöglicht direkt ein „Rapid Prototyping“ mittels Xilinx-FPGA. Die HW/SW-Partitionierung selbst ist aufgrund der Komplexität der Automatisierung manuell auszuführen. Dies geschieht dabei entweder durch manuelle Vorgaben innerhalb einer Netzliste oder während der CoSimulation mittels Ptolemy. Bei letzterem können ausgehend von den Simulationsergebnissen manuell Module für die Realisierung in Software oder Hardware festgelegt werden und das Werkzeug überführt in Abhängigkeit von der Domäne manuell oder automatisch die Software-Partition in C/Assembler [95] bzw. die Hardware-Partition in eine HDL-Beschreibung auf RT-Ebene [127].

Generell ist die Tatsache für die Praxis relevanter, dass Software sehr viel billiger ist als Hardware, auch wenn Geschwindigkeitseinbußen hingenommen werden müssen. Es gilt daher: **„Was in Software geht, wird auch programmiert!“**

Ein Grund ist die aufwendigere Implementierung von Hardware. Nur wenn Geschwindigkeits- oder Funktionsforderungen nicht erfüllt werden können, lohnt sich der Aufwand für dedizierte Hardware. Die Entscheidung für oder gegen eine Hardwarelösung ist leichter zu treffen als der enorme Forschungsaufwand glauben macht, der derzeit für die HW/SW-Partitionierung betrieben wird [17]. Prinzipiell ist es naheliegend, dass primäre, eng gekoppelte und parallelisierbare Strukturen mit hoher Datenrate in Hardware und komplexe Kontrollstrukturen mittels Software auf einem Prozessor zu realisieren sind.

Viel wichtiger ist die Suche nach einem für das anvisierte Anwendungsgebiet und die Aufgabenstellung passenden Prozessor bzw. Prozessorkern, deren Aufwand nicht zu unterschätzen ist. Für die Auswahl eines geeigneten DSP müssen beispielsweise die folgenden Randbedingungen berücksichtigt werden [20]:

- verwendete Arithmetik (Fließkomma-, Festkomma-),
- Bitbreite im Datenpfad,
- Rechenleistung (MIPS, Benchmarks),
- Speicherorganisation, Architekturmerkmale,
- verfügbare Entwicklungswerkzeuge (IDE, HLL-Compiler, Assembler),
- Multiprozessorunterstützung und I/O-Schnittstellenausstattung,
- Verlustleistung und Powermanagement,
- Verfügbarkeit und Kosten.

Eine geeignete Auswahl beruht derzeit auf Erfahrungen und Heuristiken der leitenden Entwicklungsingenieure sowie auf Beratungsfirmen, die Benchmarks von Prozessorfamilien für viel Geld verkaufen [19]. Die Werkzeuge zur Untersuchung des Entwurfsraums können hierbei jedoch gute Dienste leisten und den Entwickler beim Auswahlprozess unterstützen.

2.6.2 Interface- und Kommunikations-Synthese

Bevor auf die Interface- und Kommunikations-Synthese eingegangen werden kann, muss zunächst der Begriff der **Synthese** näher erläutert werden: Für die Beschreibung eines Systems gibt es unterschiedliche Sichten, zum einen das Verhalten und zum anderen die Struktur (Bild 2-8). Ein Wechsel von der Verhaltens- in die Struktursicht wird als **Synthese** bezeichnet. Dieser Wechsel kann auf unterschiedlichen Abstraktionsebenen erfolgen, in der Hardware-Domäne namentlich auf der System-, der Architektur- oder der Logikebene. Die Aufgabe der Synthese besteht in der (teil-)automatischen Transformation zwischen den Abstraktionsebenen und Sichten. In der **System-Synthese** wird aus einer Verhaltensbeschreibung eines Systems eine strukturelle Systembeschreibung (Prozessor, Speicher, ASIC) und eine HW/SW-Partitionierung generiert. Die **Architektursynthese** erzeugt eine strukturelle Beschreibung einer Architektur aus einer Verhaltensbeschreibung, mit den Schritten Allokation, Bindung und Ablaufplanung. Das Ergebnis der **Logiksynthese** ist eine logische Schaltung, die aus einer Verhaltensbeschreibung in Form von booleschen Gleichungen oder auch Zustandsautomaten generiert wird. Optimierungsziele der Synthese in der Hardware-Domäne umfassen die Minimierung des Flächen- und Leistungsverbrauchs und die Maximierung der Taktfrequenzen der resultierenden Schaltung bzw. beliebiger Kombinationen davon, denn oft bedingt die Optimierung eines Zielparameters die Verschlechterung eines anderen.

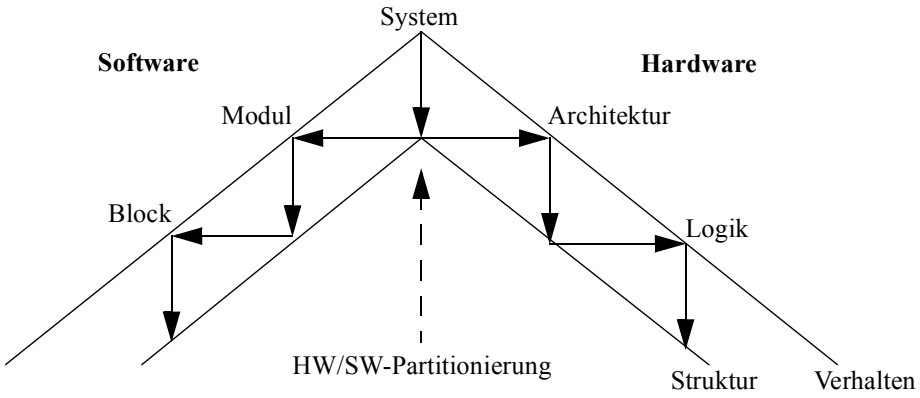


Bild 2-8. Transformationsschritte in der Hardware- und der Software-Domäne auf den unterschiedlichen Abstraktionsebenen.

In der Software-Domäne ist es die Aufgabe der **Synthese auf Modulebene**, aus einer Verhaltensbeschreibung, die in Form einer algebraischen Spezifikation vorliegen kann, eine äquivalente strukturelle Beschreibung, formuliert in einer höheren Programmiersprache (C, C++), zu erzeugen. **Auf Blockebene** wird anschließend die strukturelle Beschreibung der Modulebene in Assembler- oder Maschinensprache übersetzt. Bei der Synthese in der Software-Domäne ist es das Ziel, den resultierenden Maschinencode möglichst kompakt und darüber hinaus effizient hinsichtlich des dynamischen Speicherplatzverbrauchs und der Ausnutzung der Hardware-Ressourcen zu erzeugen.

Unter **Interface-Synthese** wird die Generierung von Schnittstellen auf einer niedrigen Ebene (low level) für die Kommunikation zwischen Hardware und Software verstanden.

Dazu gehören Schaltungsteile und entsprechende Software-Treiberfunktionen, die verschiedene Protokolle der Kommunikation implementieren. Die **Kommunikations-Synthese** findet auf einer höheren Ebene statt und bezeichnet die Generierung von abstrakteren Schnittstellen, die die Kommunikation zwischen Prozessen ermöglichen und die die zur Verfügung gestellten Schnittstellen der Interface-Synthese nutzen. Meistens werden die Interface- und Kommunikations-Synthese nach der System-Synthese zusammen mit der Hardware- und Software-Synthese durchgeführt. Besondere Bedeutung kommen der Interface- und der Kommunikations-Synthese in heterogenen Systemumgebungen zu, wie sie bei SoCs vorliegen. Die Interface-Synthese lässt sich folgendermaßen klassifizieren:

- *Manuelle Interface-Synthese:* Diese wird derzeit am häufigsten verwendet, da hierfür viele Standards, Protokolle und Bussysteme existieren.
- *Library-based Interface-Synthese:* In einer Bibliothek werden komplette getestete Treiber (Software) und Schaltungen (Hardware) organisiert zusammengefasst. Diese Methodik kann angewendet werden, wenn die Zielplattform feststeht und diese wenige Komponenten aufweist.
- *Template-based Interface-Synthese:* Es gibt vordefinierte Code-Fragmente, sowohl software- als auch hardwareseitig, die zur Synthesezeit angepasst werden können, z. B. #define IOAddress <address>. Diese Vorgehensweise bietet gegenüber der library-based Interface-Synthese eine größere Flexibilität.
- *Pattern-based Interface-Synthese:* Mit Hilfe von Pattern werden typische Interface-Schemata beschrieben. Sie stellen jedoch keine konkret vorliegende Implementierung dar. Das zugrunde liegende Wissen wird für die (manuelle) Erstellung von Schnittstellen verwendet.
- *Generator-based Interface-Synthese:* Timing-Diagramme der Interface-Protokolle sind formal beschrieben oder es werden STGs (Signal Transition Graph) verwendet. Hier gibt es aufgrund der Komplexität und der Heterogenität der Komponenten zur Zeit noch wenige Ansätze.
- *Component-based Interface-Synthese:* Es werden Standardschnittstellen zwischen den Komponenten eines SoC generiert. Problem ist die Kompatibilität zwischen verschiedenen Herstellern aufgrund fehlender standardisierter Buskonzepte für IPs (z. B. AMBA, Wishbone, CoreConnect).

Zu unterscheiden sind folgende Schnittstellen, die in der Interface- und der Kommunikations-Synthese zu berücksichtigen sind:

- Hardware/Software,
- Hardware/Hardware,
- Software/Software.

Hierbei ist anzumerken, dass HW/HW-Schnittstellen besonders im Bereich der Wiederverwendung von IPs unterschiedlicher Hersteller mit dementsprechend inkompatiblen Protokollen in einer SoC-Entwicklung notwendig sind.

Beispiele für **Systeme in der Forschung**, die die Interface-Synthese in die Werkzeuge integrieren, sind POLIS und der C-Compiler der Brass Research Group.

Beim **POLIS-System** werden Schnittstellen zwischen Hardware und Software automatisch ím Entwurfsfluss erzeugt und in die synthetisierte Implementation eingebettet. Die SW/SW-Schnittstellen werden direkt in das generierte RTOS integriert und die HW/HW-Schnittstellen werden über einfache Handshake- und Datenleitungen realisiert. Die HW/SW-Kommunikation kann über die im Prozessor vorhandenen I/O-Ports oder über Speicher-I/O (memory mapped) erfolgen und wird automatisch synthetisiert. Dabei kommen Request/Acknowledge-Protokolle zum Einsatz. Damit die Ressourcen optimal genutzt werden können, muss eine Charakterisierung des Prozessors in der Bibliothek von POLIS existieren. Sind keine Architekturmerkmale des Prozessors im System bekannt, so müssen diese manuell vom Entwickler angegeben werden. Ob die entstandene Implementation optimal die I/O-Ressourcen des Prozessors ausnutzt ist ungewiss.

Der **C-Compiler** der Brass Research Group generiert Schnittstellen-Code für den Mikroprozessor immer dann, wenn ein auszulagernder Software-Block im CFG (Control Flow Graph) in Hardware synthetisiert werden konnte. Dazu gehören die folgenden Software-Instruktionen in der angegebenen Reihenfolge:

1. Laden der Konfiguration des Co-Prozessors in die rekonfigurierbare Hardware,
2. Übertragung der Datenwerte,
3. Aktivierung des Co-Prozessors,
4. Übertragung der Ergebniswerte.

Aus der internen Darstellung der Ablaufplanung werden die notwendigen Schnittstellen-Module für die HW/SW-Kommunikation während der Interface-Synthese in Software generiert. Die hardwareseitigen Schnittstellen werden bereits innerhalb der Hardware-Synthese erzeugt.

Beispiele für **kommerzielle Systeme**, die Methoden der Interface-Synthese beinhalten, sind N2C (napkin-to-chip) der Firma CoWare [39] und VCC von Cadence [111] .

Auf der Systemebene bietet das **N2C-System** eine Umgebung für die Entwicklung von Plattformen im Entwurfsprozess eines eingebetteten Prozessors. Die Eingabesprache des Systems ist CoWareC oder SystemC. Die Integration von Komponenten (IP) unterschiedlichen Typs und unterschiedlicher Schnittstellen in eine neu zu definierende Prozessor-Plattform auf der Systemebene wird durch die Interface-Synthese automatisiert. Damit wird ein architekturunabhängiger Ansatz verfolgt, der beliebig viele Prozessorkerne und Peripherie-Module in einem SoC unterbringt. N2C nutzt abstrakte Kommunikationsprotokolle auf Systemebene, um auf einer höheren Ebene Entscheidungen hinsichtlich der Systemarchitektur zu forcieren. Die entstandene ausführbare Spezifikation des Systems kann dem Software-Entwickler als virtuelle Plattform für die Implementierung der Software dienen. Die Ausführung der Software im Modell ist auf dieser Ebene ausreichend schnell, und das Verhalten der Hardware des Systems wird zunächst mit einem Detailgrad modelliert, der für eine erste HW/SW-Integration ausreicht. Nach Auswahl einer geeigneten HW/SW-Partitionierung werden bei N2C softwareseitig die erforderlichen Treiber inklusive der im Speicher eingeblendeten Register und ISRs (Interrupt Service Routine) generiert. Hardwareseitig wird die gesamte Schnittstellenlogik synthetisiert, die das Schnittstellenprotokoll erfordert.

Beispiele dafür sind Adressdekoder für die Einbindung in den adressierbaren Speicherbereich, Logik für die Busarbitration und Bus-Bridges, sofern zwei Komponenten über mehrere Bussysteme miteinander gekoppelt sind. Die Kombination eines Kommunikationsprotokolls mit der Quelle und Senke einer Kommunikationstransaktion wird dabei als Szenario bezeichnet.

VCC ist ein Werkzeug für die Systemebene im HW/SW-CoDesign eines SoC und basiert zum Teil auf Verfahren, die auch beim POLIS-System angewendet werden. Dieses Werkzeug implementiert eine strikte Trennung zwischen einem Verhaltensmodell, das das Verhalten des Systems bestimmt, und einem Architekturmodell, das die Implementierung des Systems vorgibt. VCC unterstützt die Kommunikations- und Interface-Synthese zwischen IP-Cores und ist speziell auf die Integration von IPs in einem Top-Down-Entwurfsfluss für SoCs zugeschnitten. Die verwendete Interface-Synthese basiert auf vordefinierten, vollständig implementierten Interfaces und Busarbitrationsmodulen in Bibliotheken (library-based). Darüber hinaus existiert ein spezielles Tool für die Spezifikation und Synthese von Protokollkonvertern zwischen den IPs verschiedener Hersteller.

Die **Interface-Synthese** kann zwar architekturunabhängig durchgeführt werden, sie erzielt jedoch bessere Ergebnisse, wenn die HW/SW-Architektur bereits feststeht. Erfolgt die Interface-Synthese architekturunabhängig, beispielsweise mit Hilfe eines generischen Datenbusses, ist davon auszugehen, dass die synthetisierten Software- und Hardware-Module nicht die Interface-Ressourcen des Prozessors optimal ausnutzen. Vordefinierte, getestete und optimierte Module versprechen dagegen eine perfekte Anpassung an vorgegebene Architekturmerkmale des Zielsystems.

Die Interface-Synthese ist eine der Hauptthematiken dieser Arbeit und wird im Kap. 4.1, Kap. 5.3 und Kap. 6.3 detailliert behandelt. Der Schwerpunkt liegt dabei bei der Generierung von HW/SW-Schnittstellen, die bei der Implementierung einer auf den DSP und das FPGA partitionierten Problemstellung auftreten. Die dort vorgestellte Realisierung stellt eine Kombination von library-based und template-based Interface-Synthese dar.

2.6.3 Hardware/Software-CoVerifikation

Ein weiterer, nicht unwesentlicher Gesichtspunkt im HW/SW-CoDesign stellt die **HW/SW-CoVerfikation** dar. Zunächst werden einige wichtige grundlegende Begriffe im Bereich HW/SW-CoVerifikation definiert: Unter dem Begriff der **Verifikation** wird der Prozess der Evaluierung eines Systems oder einer Systemkomponente verstanden. Dabei wird bestimmt, ob ein Implementierungsschritt fehlerfrei die Funktionalität des Ausgangspunktes erfüllt. Der Begriff der **formalen Verifikation** bezeichnet den formalen Beweis der Korrektheit der Ergebnisse eines Implementierungsschrittes. Bei der **Validierung** wird die Einhaltung von Entwurfsbeschränkungen (Leistungsfähigkeit, Kosten, Leistungsaufnahme etc.) überprüft. Der Begriff **HW/SW-CoVerifikation** kennzeichnet die Überprüfung der Fehlerfreiheit eines Implementierungsschrittes bei der HW/SW-Integration. Hauptaugenmerk liegt auf den HW/SW-Schnittstellen und dem reibungslosen Ablauf eines partitio-

nierten HW/SW-Algorithmus. Diese kann grob umrissen mit Hilfe der folgenden Techniken erfolgen:

- Simulation,
- Emulation,
- Formale Verifikation,
- FPGA-basiertes "Rapid Prototyping".

Darüber hinaus gibt es Abwandlungen und Optimierungen, deren prinzipielle Vorgehensweisen und wesentliche Vor-/Nachteile nachfolgend vorgestellt werden.

Unter einer **Simulation** ist die Ersetzung eines Systems durch ein Modell zu verstehen, dessen kontrollierte Ausführung sich wie das System verhält bzw. arbeitet. Das Ziel ist die Überprüfung der geforderten funktionalen Eigenschaften eines Systems. Dabei stellt sich das Problem, dass durch die Simulation zwar das Vorhandensein eines Fehlers nachgewiesen, jedoch für größere Systeme nie die Fehlerfreiheit bewiesen werden kann.

Unterschieden werden muss zwischen der Simulation der Software-Ausführung, in dieser Arbeit vereinfachend als Software-Simulation bezeichnet, und der Hardware-Simulation, der Simulation eines Hardware-Modells. Für Software-Simulationen gibt es prinzipiell drei Möglichkeiten:

- Der Software-Code kann direkt als Prozess auf dem Simulationsrechner (PC/Workstation) ausgeführt werden. Damit wird er mit der maximal möglichen Geschwindigkeit des Simulationsrechners ausgeführt. Dies ist sicherlich der Idealfall, dafür aber relativ selten anwendbar.
- Der Algorithmus ist in einer maschinenunabhängigen Sprache (z. B. C) definiert, und es gibt einen Cross-Compiler, der den Quelltext für die Simulationsplattform übersetzen kann. Dabei hängt die Ausführungsgeschwindigkeit primär von der Güte des Cross-Compilers ab.
- Kann die Software nicht direkt ausgeführt werden, kommt ein für den Simulationsrechner entwickelter ISS zum Einsatz. Dieser interpretiert den Maschinencode auf Instruktionsebene und simuliert die Effekte der Maschinenbefehle, jeweils für einen Befehl pro Zeitschritt. Viele der ISS speichern Simulationsergebnisse in Dateien, so z. B. Speicher- und Registerinhalte oder auch Ergebnisse des Zeitverhaltens (Anzahl benötigter Taktzyklen).

Die Hardware-Simulation erfolgt heutzutage mit einer HDL-Beschreibung (VHDL oder Verilog) des Modells auf Gatter- bzw. Register-Transfer-Ebene in einem HDL-Simulator. Durch die Erweiterung von VHDL mit analogen Konstrukten (VHDL-AMS [117]) ist der Entwickler heute in der Lage, analoge und digitale Funktionen gemeinsam in einer VHDL-basierten Hardware-Simulation zu simulieren. Ebenfalls möglich ist die gemischte Simulation von VHDL, Verilog und C (heterogene Simulation). Das Erstellen von Modellen und Testbenches in einer HDL für die Simulation spielt immer mehr die dominierende Rolle im zeitlichen Ablauf eines Projekts. Vermieden werden können damit aber lange Prototypenphasen oder aufwendige Re-Designs.

Die Hardware-Simulationen auf Gatter- und Register-Transfer-Ebene laufen ereignisgesteuert ab, wobei die Zahl der Ereignisse etwa proportional zur Anzahl der implementierten Gatter ist und direkt die Simulationszeiten beeinflussen. Eine Reduzierung der Ereignisanzahl wird mit einer zyklusbasierten Simulation (cycle-based) erreicht, bei der aus der Beschreibung auf Register-Transfer-Ebene das Verhalten für jeweils eine Taktperiode abgeleitet wird. Auf die Bestimmung exakter Zeitverzögerungen wird verzichtet, da z. B. bei einem synchronen Entwurf die maximale Verzögerung des längsten Pfades kleiner als die Taktperiode sein muss. Durch die erreichte Simulationsbeschleunigung kann auf eine Emulation oder spezielle Hardware-Beschleunigung der Simulation verzichtet werden. Allerdings ist lediglich eine Untermenge des Synthese-Subsets (siehe Kap. 2.7.1) für die zyklusbasierte Simulation verwendbar.

Probleme der Simulation sind die langen Laufzeiten im Vergleich zum realen System und die Tatsache, dass lediglich das deterministische Verhalten der Schnittstellen eines Systems nachgebildet werden kann.

Das Laufzeitproblem kann abgeschwächt werden, indem der Detailgrad der verwendeten Simulationsmodelle an den notwendigen Verifikationsbedarf ähnlich einem Stufenmodell angepasst wird. Die Aussagekraft der Simulation nimmt jedoch mit jedem weiteren niedrigeren Detailgrad ab, was entsprechend zu berücksichtigen ist. Eine Datenfluss-Simulation z. B. arbeitet auf der Systemebene. Signale sind lediglich ein Strom von Werten ohne jeglichen zeitlichen Bezug. Funktionale Blöcke werden erst ausgeführt, wenn eine genügend große Zahl von Eingangswerten zur Verfügung stehen. Die Eingabe der funktionalen Blöcke und der Verbindungen erfolgt in der Regel schematisch. Der Simulationskern kontrolliert den Datenfluss zwischen den Blöcken mittels FIFOs (First-In First-Out). Die Datenfluss-Simulation stellt einen sehr hohen Grad der Abstraktion und damit einen sehr niedrigen Detailgrad dar, womit die Korrektheit der Algorithmen in einer frühen Entwicklungsphase überprüft werden können. Das deterministische Verhalten der Simulation verhindert die Überprüfung indeterministischer Eingangssignale, z. B. Signale, die einer Bedienungseinheit entstammen (Mensch-Maschine-Schnittstelle). Beide Probleme gemeinsam führen praktisch dazu, dass das Verhalten des Systems auf reale Eingangssignale in der Simulation nicht verifiziert werden kann.

Bei der **hardwarebeschleunigten Simulation** berechnen ein oder mehrere spezielle Co-Prozessoren Teile der Simulation mit einem notwendig messbaren Geschwindigkeitszuwachs, da es einigen Aufwand kostet, die Simulationsteile in spezialisierte Hardware auszulagern. Die Testbench verbleibt dagegen im softwarebasierten Simulator. Die realisierten Systeme implementieren verschiedene Kommunikationsmechanismen zwischen hardwarebeschleunigter Simulation und dem softwarebasierten Simulator, so findet z. B. die Kommunikation auf der Ebene der Signale (bit-level signal) oder auf der Ebene der Transaktionen (transaction-based) statt. Die Wahl des softwarebasierten Simulators ist oft von den Systemschnittstellen und von der Unterstützung der Systemhersteller abhängig. Beispiele für hardwarebeschleunigte Simulationswerkzeuge sind die Palladium-Plattform der Firma Cadence [93] oder NSIM von Mentor Graphics [89].

Unter einer **Emulation** ist die Ausführung einer Funktion auf einer Hardware zu verstehen, die nicht der Ziel-Hardware entspricht und einen hohen Geschwindigkeitsgrad bietet. Mit Hilfe programmierbarer Logikbausteine (FPGA) oder auch spezieller Rechnerhardware werden Systemteile oder das komplette System in eine Emulationsumgebung eingebettet, so dass reelle Eingangssignale eingespeist werden können. Ein emulierter Videochip z. B. kann in einer Emulationsumgebung mit einem Monitor verbunden werden oder ein emulierter Ethernet-Chip kann Signale über ein Ethernet-Netzwerk übertragen. Ein softwarebasierter Simulator unterstützt dies nicht direkt, so dass z. B. eine Nachbildung über File-I/O notwendig ist. Damit die Systembestandteile im FPGA emuliert werden können, ist zunächst eine Synthese mit anschließender Platzierung und Verdrahtung durchzuführen. Steht der Prozessorkern als Soft-IP zur Verfügung, kann er auch in das FPGA übernommen werden. Ist ein Soft-IP nicht vorhanden, gibt es die Möglichkeit, den Prozessorkern als diskrete Komponente in die Emulation einzubeziehen. In der Regel wird die Zieltaktfrequenz in der FPGA-Technologie nicht erreicht, d. h., jeder Bezug zum Zeitverhalten des endgültigen Systems geht verloren. Die Einrichtung eines Emulationssystems ist aufwendig und relativ teuer. Der Vorteil jedoch ist, dass signifikant Simulationszeiten eingespart werden. Außerdem wird die Entwurfssicherheit erhöht, da eine Klasse von Problemen basierend auf indeterministischen Signalen aufgedeckt wird, die in der Simulation nicht aufzufinden ist.

Zu den für Emulationszwecke verfügbaren kommerziellen Systemen gehören die Palladium-Plattform von Cadence [93], VStation und CelaroPro von Mentor Graphics [124] sowie MercuryPlus von Quickturn [80].

Simulation vs. Emulation - In einer (ereignisgesteuerten) Simulation sind zu jedem Zeitpunkt alle Signale verfügbar. Dagegen muss bei Emulatoren zur Übersetzungszeit angegeben werden, welche Signale verfolgbar sein sollen. Wenn das nächste Problem in einer kurzen Simulationszeit (einige Millisekunden) gefunden werden kann, dann sind die eigentlich langsameren Simulatoren aufgrund der kurzen Kompilierungszeiten schneller. Sind die Probleme erst in einigen Sekunden simulierter Zeit zu erwarten, ist der Aufwand für die Einrichtung einer Emulation wegen der besseren Ausführungsgeschwindigkeit zu vernachlässigen, allerdings nur unter der Voraussetzung, dass ein Emulationssystem bereits existiert.

Bei der **formalen Verifikation** handelt es sich um den Nachweis (Beweis) der korrekten Funktionsweise einer Schaltung gegenüber der Spezifikation. Das Ergebnis ist entweder die Aussage, dass die Verifikation erfolgreich war, oder es wird eine Eingangskombination als Gegenbeispiel generiert, bei der die Schaltung nicht das gewünschte Verhalten zeigt. Das große Problem ist, dass eine präzise Spezifikation für einen formalen Vergleich mit der Schaltungsbeschreibung vorliegen muss. Deren Erstellung kann ebenso aufwendig und fehleranfällig sein wie der eigentliche Schaltungsentwurf. Fehler in der Spezifikation können den Nutzen des Äquivalenzbeweises mindern.

In den letzten Jahren gab es in bestimmten Bereichen der formalen Verifikation große Fortschritte, beispielsweise bei Controllern, die durch endliche Automaten modelliert werden (z. B. VIS - Verification Interacting with Synthesis [121]). Die Komplexität des Korrektheitsbeweises limitiert diese Art der Verifikation bisher jedoch auf kleine Schaltungsent-

würfe. Ausgereifte Werkzeuge zur Handhabung komplexer Schaltungen stehen derzeit nicht zur Verfügung.

Methoden der formalen Verifikation können beispielsweise im POLIS-System genutzt werden, indem aus den CFSMs gewöhnliche FSMs generiert werden. Auf FSMs basierende formale Verifikationswerkzeuge, wie VIS, können somit in den Entwurfsfluss von POLIS eingebunden werden.

Für die Verifikation eines Systems in einem realen Umfeld wird das **FPGA-basierte "Rapid Prototyping"** eingesetzt. Mittels Logiksynthese wird die Schaltung zunächst in ein FPGA übersetzt, bevor es in ein ASIC umgesetzt wird. Ein Prototyp-Board enthält außer dem/den FPGAs zusätzliche Schaltungsperipherie, die zusammen annähernd dem später im ASIC zu realisierenden System entsprechen. Die Prototypen bieten eine noch höhere Arbeitsgeschwindigkeit als die Emulation. Für einige Anwendungen lassen sich sogar die Arbeitsfrequenzen des späteren Chip (Echtzeit) erreichen. Da der Prototyp der endgültigen Realisierung sehr nahe kommt, ist eine eingehende Analyse der Leistungsfähigkeit des Systems möglich. Wenn eine zeitlich korrekte Interaktion mit der realen Umgebung gewährleistet werden soll, muss die Abarbeitung des im Prototypen realisierten Modells in Echtzeit erfolgen. Zu den Nachteilen des FPGA-basierten „Rapid Prototyping“ zählen der erhebliche Aufwand für die Entwicklung bzw. die hohen Kosten für den Einkauf eines Prototyp-Boards sowie die gegenüber einem emulierten System eingeschränkten Debug-Fähigkeiten zur Laufzeit des Systems.

Mit den bisher vorgestellten Techniken lassen sich für die **HW/SW-CoVerifikation** von eingebetteten Systemen die folgenden Möglichkeiten ableiten:

- Homogene/Heterogene HW/SW-CoSimulation,
- HW/SW-CoEmulation,
- HW/SW-CoPrototyping.

Die gebräuchlichste Definition für **HW/SW-CoSimulation** ist die, dass simulierte Hardware mit Software manipuliert wird. Im Entwurfsfluss wird eine HW/SW-CoSimulation mit verschiedenen Abstraktionsgraden und auf unterschiedlichen Abstraktionsebenen durchgeführt:

- Auf Systemebene wird die Funktionalität des Systems unabhängig von einer konkreten Implementation und ohne Berücksichtigung des Zeitverhaltens simuliert. Die Kommunikation zwischen den Hardware- und Software-Teilen erfolgt über abstrakte Kommunikationskanäle mittels send()- und receive()-Funktionen, ohne Interrupts oder andere Ereignisse einzubeziehen. Das Ziel der HW/SW-CoSimulation auf dieser Ebene ist die Überprüfung der reinen Systemfunktionalität.
- Auf Architekturebene ist das System partitioniert und auf ein Zielsystem, einer konkreten Implementation, abgebildet. Die Kommunikationsprotokolle sind ausgewählt und die Kommunikation wird einschließlich der Registerzugriffe modelliert. Auf dieser Ebene liegt das Ziel der HW/SW-CoSimulation auf der Validierung der Systempartitionierung und der selektierten Kommunikationsprotokolle.

- Eine zyklengetreue HW/SW-CoSimulation simuliert die Hardware des Systems auf RT-Ebene und nutzt für die Software-Simulation des Systems einen Mechanismus zur zyklengenauen Simulation des Prozessors. Für die Kommunikationsprotokolle sind konkrete Hardware-Schnittstellen ausgewählt und die Kommunikation wird mittels Stimulation von Signalen modelliert. Das Ziel der HW/SW-CoSimulation auf dieser Ebene ist vorrangig die Überprüfung des Zeitverhaltens des Systems.

Die HW/SW-CoSimulation gehört zu den Verfahren, für die der Begriff „Virtual Prototyping“ geprägt wurde. Bei einem HW/SW-CoDesign muss die Software auf der Zielhardware verifiziert werden. Dieser Vorgang kann erst durchgeführt werden, wenn der HW-Entwurf fertiggestellt und die Hardware produziert und verfügbar ist. Fehler, die erst bei Fertigstellung festgestellt werden, erfordern ein aufwendiges Re-Design oder eine Anpassung der Software, bei der Hardwareprobleme zu umgehen sind (Workaround), falls dies überhaupt möglich ist. Noch dringender ist das Problem bei der Entwicklung eines SoC, da die Kosten eines Re-Designs sehr viel höher sind und der Anteil der Software am Gesamtsystem in der Regel größer ist. Die Lösung des Problems kann mittels eines virtuellen Prototypen gefunden werden. Dieser erlaubt dem Entwickler, die Software und die Hardware darin zu integrieren, bevor die Hardware zur Verfügung steht. Eine ausführliche Verifikation, Validierung und Analyse kann in einer „sicheren Umgebung“ durchgeführt werden. Der virtuelle Prototyp enthält für eine SoC-Entwicklung Mechanismen für die Hardware-Simulation (HDL-Simulator) und für die Software-Simulation (Debug-Umgebung des Prozessors). Der volle Zugriff auf die in Hardware realisierte Logik und die Interna des verwendeten Prozessors sind zu jeder Zeit möglich.

Die Ergebnisse der HW/SW-CoSimulation hängen unmittelbar von der Güte der Modelle ab (Abbildungsgrad der Wirklichkeit), mit denen simuliert wird. Die Simulation komplexer eingebetteter Systeme, die Modelle unterschiedlicher IPs umfasst, stellt hohe Anforderungen an die Qualität der Modellierung. Eine Verlagerung der Modellierung auf den Hersteller des IP ist notwendig, da nur dieser die detaillierten Informationen über das IP besitzt und die spezifizierten Eigenschaften garantiert.

Um dem Problem der langen Simulationszeiten zu begegnen, muss der Software-Entwickler eines HW/SW-CoDesigns temporär Funktionen in der Software deaktivieren, die „zu kostbar“ für die Simulation sind. Klassisches Beispiel ist die Sequenz zum Laden der Software in der Simulation. Ist die Bootsequenz komplex, wird sie so lange mitsimuliert, bis sich keine neuen Erkenntnisse für die Verifikation ergeben. Danach wird die Simulation immer mit einem bereits geladenen Speicherabbild der Software gestartet.

Für die HW/SW-CoSimulation auf Systemebene wird im POLIS-System das Werkzeug Ptolemy eingesetzt. Ptolemy unterstützt prinzipiell nicht nur die reine CoSimulation von Hardware und Software, es lassen sich darüber hinaus analoge/digitale Schaltungen und mechanische Spezifikationen gemeinsam in einem Systemmodell simulieren. Im POLIS-System ist es jedoch neben der Nutzung von Ptolemy auch möglich, VHDL-Modellbeschreibungen mit allen Informationen über die CoSimulation zu erzeugen, so dass prinzipiell jeder VHDL-Simulator für die HW/SW-CoSimulation eingesetzt werden kann.

Für die HW/SW-CoSimulation werden die homogene und die heterogene HW/SW-Cosimulation unterschieden:

Bei einer reinen softwarebasierten Simulation wird die Software- und die Hardware-Simulation gemeinsam auf einem PC oder einer Workstation durchgeführt und dies wird als **homogene HW/SW-CoSimulation** bezeichnet. Das simulierte System existiert nur virtuell als ein Prozess auf dem Simulationsrechner. Ein HDL-Simulator führt die Hardware-Simulation aus, indem das gesamte System mit Hilfe von HDL-Modellen auf verschiedenen Abstraktionsebenen für den Prozessor, den Speicher und die Peripherie modelliert wird. Die Software-Simulation wird in die Hardware-Simulation einbezogen, wobei die Software in ein ROM übersetzt wird.

Ist ein eingebettetes System auf Board-Ebene bzw. Chip-Ebene zu simulieren, kann die Board-Level-Simulation [52, 53, 54] für die homogene HW/SW-CoSimulation verwendet werden. Aus Gründen der Simulationsgeschwindigkeit kamen bei der Board-Level-Simulation bisher überwiegend Schnittstellen-Modelle der Prozessorkomponenten zum Einsatz [30, 97]. Die Schnittstellen-Modelle modellieren lediglich die Prozessorschnittstellen im Detail und die Software-Simulation wird nur in Bezug auf die Ein-/Ausgaberoutinen berücksichtigt. Steigende Rechenkapazitäten und die Notwendigkeit einer detaillierten Verifikation an einem virtuellen Prototypen erfordern jedoch eine detaillierte Modellierung der Schnittstellen und des kompletten Rechenkerns des Prozessors inklusive des Zeitverhaltens. Wichtig ist auch, dass der Quelltext und Objektcode für den Prozessor in der Simulation derselbe ist, der später auch im realen System eingesetzt wird, ohne die Notwendigkeit des Einfügens von proprietären Anweisungen, oder ähnlichem. Dies zu gewährleisten, wird im Kap. 3, Kap. 5.4 und Kap. 6.4 ausführlicher behandelt und ist eine weitere Hauptthematik dieser Arbeit.

Ein Problem bei der homogenen HW/SW-CoSimulation ist, dass der Software-Entwickler mit einem für ihn ungewohnten Hardware-Werkzeug, dem HDL-Simulator, arbeiten muss. Dieses Problem kann jedoch mit der Realisierung einer Entwicklungsoberfläche für das Software-Debugging behoben werden und wird im Kap. 6.4.3 vorgestellt. Darüber hinaus wird gezeigt, dass auf allen vom Entwicklungssystem adressierten Entwurfsebenen dieselbe Simulationsumgebung für einen durchgängigen Entwurfsfluss zum Einsatz kommt.

Ein Beispiel eines Werkzeugs speziell für die homogene HW/SW-CoSimulation ist HADES [64], ein Java-basierter Simulator mit schematischer Eingabe und interaktiv animierter Simulationsoberfläche. Adressiert wird die Simulation digitaler Schaltungen, wofür Modelle auf Gatterebene, RT-Ebene bis hin zur Systemebene verfügbar sind. Für die HW/SW-CoSimulation sind Simulationsmodelle einfacher Mikrocontroller, wie PIC1684 oder MIPS R3000, verfügbar. Für eine bessere Unterstützung des Software-Debuggings sind die Simulationsmodelle dieser Mikrocontroller mit zusätzlichen Debug-Pins ausgestattet, die nicht im realen Baustein vorhanden sind, über die jedoch interne Registerzustände während der Simulation mittels Signalverlauf beobachtet werden können. Breakpoints in der Software der Mikrocontroller zur Laufzeit der Simulation werden nicht unterstützt.

Eine **heterogene HW/SW-CoSimulationsumgebung** liegt vor, wenn zwei oder mehr softwarebasierte Simulatoren verschiedenen Typs über definierte Schnittstellen (IPC, TCP/IP etc.) zur Laufzeit verbunden sind. Vorstellbar ist in diesem Zusammenhang, dass die Simulation eines komplexen ASIC auf mehrere Workstations verteilt ist oder dass mehrere ASICs in unterschiedlichen Simulatoren simuliert werden. Zu kontrollieren ist der Datenfluss zwischen den Simulatoren und die Simulationen sind zeitlich zu synchronisieren. Für die heterogene HW/SW-CoSimulation wird in der Regel ein ISS bzw. Debugger des verwendeten Prozessors mit einem HDL-Simulator verknüpft. Der Vorteil ist, dass der HW- und der SW-Entwickler jeweils mit seinem bevorzugten Entwicklungswerkzeug arbeitet.

Ein Geschwindigkeitsvorteil gegenüber einer homogenen HW/SW-CoSimulation lässt sich nur erzielen, wenn der Prozessor(-kern) lediglich „lose“ mit den ihn umgebenden Baugruppen gekoppelt ist. Greift der Prozessor oft auf externe Hardwarebausteine zu, dann ist die Software-Simulation nur so schnell wie die Hardware-Simulation bzw. umgekehrt, je nachdem wie komplex die Hardware- und die Software-Simulation im speziellen Einzelfall ist. Im Gegenteil, der zusätzlich notwendige Kommunikationsbedarf kann die Gesamtsimulation weiter verlangsamen. Der Hauptfaktor für die Geschwindigkeit der heterogenen Simulation ist die Häufigkeit des Zugriffs auf die Hardware innerhalb der Software. Daneben spielen die Ausführungszeiten der verbindenden und kontrollierenden Software ebenfalls eine Rolle. Sind die Simulationen über ein Netzwerk miteinander gekoppelt, kommen der Netzwerkdurchsatz, die -bandbreite und die -verfügbarkeit zu einer Simulationsanalyse hinzu.

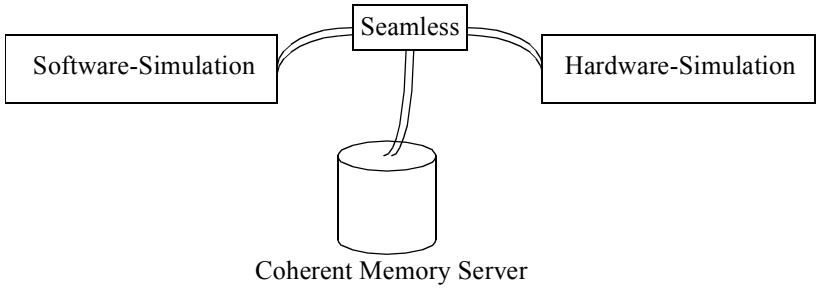


Bild 2-9. Seamless von Mentor Graphics für die HW/SW-CoSimulation.

Kommerzielle Beispiele für heterogene Simulationswerkzeuge sind Seamless von Mentor Graphics [103], N2C von CoWare [39] oder Virtual-CPU von Summit Design [120], ähnliche Programme gibt es auch von anderen Herstellern. Als Besonderheit erwähnenswert ist bei Seamless der Coherent Memory Server (Bild 2-9), der einen dynamischen Wechsel zur Laufzeit der Simulation zwischen einer detaillierten Hardware-Simulation einerseits und einer schnellstmöglichen Ausführung der Software andererseits ermöglicht. In einer Datenbank stehen derzeit mehrere Modelle (PSP - Processor Support Package) für CPUs und DSPs zur Verfügung. Hier stellt sich die Frage, ob und mit welchem Aufwand ein solches Modell für den verwendeten aber nicht in der Datenbank vorhandenen Prozessor erstellt werden kann. Ansonsten gelten die für heterogene HW/SW-CoSimulationen genannten Einschränkungen auch für Seamless.

In der Forschungslandschaft gibt es zum Thema heterogene HW/SW-CoSimulationen mehrere aktuelle Arbeiten, vorgestellt in [6, 68]. Bei der beispielhaft in [6] verwendeten Methode, die Software-Simulation und die Hardware-Simulation über eine Backplane zu verbinden, verschlechtert der zusätzlich notwendige Kommunikationsaufwand die Simulationsgeschwindigkeit. Außerdem erfordert der Einsatz der beschriebenen HW/SW-CoSimulation eine unter Umständen fehlerhafte Anpassung der C-Quelltexte des Prozessors und eine Testbench für den HDL-Code, die die für die Backplane vorgesehenen Kommunikationsmodule instantiiert. Für die Einbindung der Ausführung des C-Codes in die HDL-Simulation werden die C/HDL-Schnittstellen der HDL-Simulatoren verwendet, die derzeit nicht standardisiert und von Simulator zu Simulator verschieden sind. Damit ist die Portabilität der HW/SW-CoSimulation eingeschränkt.

Für die HW/SW-CoVerifikation gibt es folgende Möglichkeiten der Kopplung mit einem Emulationssystem für eine **HW/SW-CoEmulation**:

- ISS ⇔ Kontroll-Software ⇔ Emulator.
- Prototyp-Hardware ⇔ Kontroll-Software ⇔ Emulator.

Im ersten Fall wird die Software-Simulation im ISS ausgeführt und die Hardware im System emuliert. Beim zweiten Punkt erfolgt die Software-Ausführung im Prototyp des Zielsystems, zusätzlich wird die Hardware jedoch weiterhin emuliert. Ansonsten gelten die für die Emulation genannten Vor- und Nachteile.

Die gemeinsame Verifikation von Hardware und Software beim **HW/SW-CoPrototyping** kann sehr viel schneller als bei einer reinen Simulation oder Emulation erfolgen, wenn bereits eine vorgefertigte und leicht auf die eigenen Anforderungen anpassbare Prototyp-Hardware zur Verfügung steht. Der Nachteil ist, dass die Prototyp-Hardware immer auf eine Klasse von Anwendungen zugeschnitten ist und damit die Flexibilität gegenüber einer Emulation bzw. Simulation geringer ist.

Zusammenfassung HW/SW-CoVerifikation - Für den Prozessor eines eingebetteten Systems bzw. für einen Prozessorkern eines SoC gibt es in der Regel eigene Software-Entwicklungssysteme, die einen Simulator (ISS) oder zumindest einen Debugger beinhalten. Sie ermöglichen jedoch keine Kopplung der Software-Simulation mit einer Hardware-Simulation auf Register-Transfer-Ebene, wenn überhaupt, dann nur umständlich z. B. über File-I/O. Deshalb kommt der heterogenen HW/SW-CoSimulation eine besondere Bedeutung zu, es sei denn, es gibt ein Modell des Prozessorkerns in einer HDL für eine homogene HW/SW-CoSimulation. Im letztgenannten Fall kann auf die dedizierten Entwicklungswerkzeuge des Kern-Anbieters bei der Software-Entwicklung teilweise und bei der HW/SW-Integration gänzlich verzichtet werden. Zusammenfassend lässt sich feststellen, dass eine zufriedenstellende Verifikation von eingebetteten Systemen (SoCs) mit einer klassischen Simulation allein nicht möglich ist. Diese liefert jedoch Aufschlüsse über Zusammenhänge und kann insbesondere dann hilfreich sein, wenn ein virtueller Prototyp notwendig ist. Eine Emulation lässt die Beurteilung der Interaktion des Systems mit einer realen Umgebung zu und gewährt einen Einblick in die Interna zur Laufzeit, erreicht aber nicht die Taktraten des Zielsystems. Ein schneller Prototyp für den Anwendungsfall in einer frühen Entwicklungs-

phase in Kombination mit der HW/SW-CoSimulation stellt daher die günstigste Wahl dar und wird in dieser Arbeit innerhalb eines mit *FADES* bezeichneten Systems vorgestellt.

2.7 Hardwarebeschreibungssprache VHDL

VHDL steht für VHSIC Hardware Description Language, wobei VHSIC eine Abkürzung für Very High-Speed Integrated Circuit ist. Die Sprache wurde neben Verilog [112, 113] entwickelt, um die Beschreibung von elektronischen Schaltungen für dokumentarische Zwecke textuell zu ermöglichen. 1987 wurde der erste VHDL-Standard IEEE1076 vom IEEE verabschiedet [114], der zweite 1993 [115]. Konzipiert wurde VHDL als reine Modellierungs- und Simulationssprache. Die derzeit auf dem Markt verfügbaren Verilog/VHDL-Simulatoren sind in Tabelle 5 aufgelistet.

Tabelle 5: Aktuelle Marktübersicht der verfügbaren Verilog/VHDL-Simulatoren.

Simulator	Hersteller	Typ
FineHDL	ACAD	VHDL/Verilog Simulator
Active-HDL	Aldec	VHDL/Verilog Simulator
Affirma	Cadence	VHDL/Verilog event-driven Simulator
Finsim	Fintronics	Verilog event-driven and cycle-based Simulator
Incisive	Cadence	Verilog/VHDL/SystemC Simulator
ModelSim	Mentor Graphics	VHDL/Verilog event-driven Simulator
Riviera	Aldec	VHDL/Verilog Simulator
Scirocco	Synopsys	VHDL event-driven and cycle-based Simulator
Silos	Simucad	Verilog Simulator
SpeedSim	Quickturn	Verilog cycle-based Simulator
VCS	Synopsys	The first alternative Verilog Simulator
NC-VHDL,-Verilog	Cadence	VHDL/Verilog Simulator
VeriLogger Pro	SynaptiCAD	Verilog Simulator
VHDL Simili	Symphony EDA	interpreted VHDL Simulator

Die mit VHDL modellierten Schaltungen bewegen sich im Wesentlichen auf den Abstraktionsebenen Algorithmische Ebene, Register-Transfer-Ebene und Logikebene. Die Systemebene kann ebenso berücksichtigt werden. VHDL unterstützt auf den genannten Abstraktionsebenen eine Beschreibung in der strukturalen Sicht (structural modeling) oder in der Verhaltenssicht (behavioral modeling). Eine Modellierung der geometrischen Sicht eines elektronischen Systems bietet VHDL dagegen nicht (z. B. Layout-Daten im GDSII-Format). Die Grenzen der Abstraktionsebenen sind meistens nicht klar definierbar. Zudem gibt es die Möglichkeit, dass die Beschreibungen für die Komponenten eines komplexen, elektronischen Systems auf mehreren Abstraktionsebenen verfasst sind (multi level modeling).

2.7.1 Synthese-Subset

VHDL dient i. A. der Modellierung und Simulation von Hardware. Es hat sich aber im Laufe der Zeit gezeigt, dass eine Untermenge auch beim Entwurf von Schaltungen (High-Level-Synthese) gut geeignet ist [12]. Nicht alle VHDL-Konstrukte sind prinzipbedingt synthetisierbar. Die Anweisung „wait for 10 ns“ beispielsweise ist ein gebräuchliches VHDL-Konstrukt zur Modellierung von Zeitverzögerungen, führt aber zu keiner sinnvollen Instantiierung einer Hardware-Komponente auf Gatterebene. Darüber hinaus unterstützen die Synthese-Werkzeuge unterschiedliche VHDL-Subsets, so dass der Entwickler gezwungen ist, seine Modellierung an die vorhandenen Werkzeuge anzupassen. Vom IEEE wurde deshalb 1999 der Standard IEEE1076.6 verabschiedet [116], der die Untermenge von VHDL definiert, die für die Synthese auf Register-Transfer-Ebene zulässig ist (Bild 2-10). Dies garantiert die Interoperabilität der VHDL-Beschreibungen zwischen allen Syntheseswerkzeugen, die den Standard unterstützen. Darüber hinaus erleichtert der Standard die Arbeit des Entwicklers, da falsche Syntheseergebnisse aus „unglücklich“ gewählten VHDL-Konstrukten seltener auftreten.

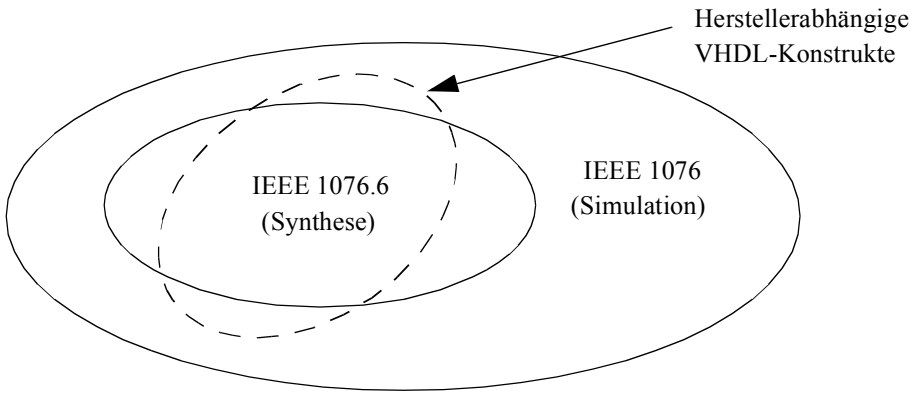


Bild 2-10. VHDL-Subsets; VHDL dient im Allgemeinen zur Modellierung und Simulation von Hardware, nur ein kleiner Teil der Sprache eignet sich für die Synthese.

2.8 XML

2.8.1 Was ist XML?

„XML ist eine textbasierte Meta-Auszeichnungssprache, die die Beschreibung, den Austausch, die Darstellung und die Manipulation von strukturierten Daten erlaubt, so daß diese von einer Vielzahl von Anwendungen genutzt werden können.“ [62]

XML (eXtensible Markup Language) ist eine echte Untermenge von SGML (Standard Generalized Markup Language), einem Standard zur Inhalts- bzw. Strukturbeschreibung von Dokumenten. HTML stellt ebenfalls eine kleine, starre Untermenge von SGML dar. Daher sieht ein XML- einem HTML-Dokument auf den ersten Blick sehr ähnlich. XML wurde 1998 vom W3C (World Wide Web Consortium) als Standard definiert, von dem mittlerweile eine zweite Fassung existiert [125, 126].

XML-Dokumente sind reine Textdateien, wodurch sich die Portabilität erhöht. Es erfolgt eine strikte Trennung der Strukturbeschreibung der Daten und dem Dateninhalt von dem Datenlayout. Im Dokument sind keinerlei Layoutinformationen enthalten, diese werden erst durch Style-Sheets (XSL, CSS) definiert. Der Vorteil ist, dass die Daten eines weitergegebenen XML-Dokuments auch ohne ein vorhandenes Layout sofort weiterverarbeitet oder aufbereitet werden können. Charakteristisch für XML ist auch die Klartextmarkierung mit sogenannten Tags, die durch textatypische Zeichen eingeleitet werden. Mit jedem einfachen ASCII-Texteditor können XML-Dokumente erstellt und gelesen werden.

2.8.2 Aufbau eines XML-Dokuments

Physisch kann ein XML-Dokument über mehrere Dateien verteilt gespeichert sein. Die Einbindung in ein Dokument erfolgt über Entities. Ein Beispiel für ein solches Dokument ist ein Buch, dessen Kapitel in eigenen Dateien abgelegt sind.

Ein XML-Dokument setzt sich logisch aus einem Prolog und einer Dokumenteninstanz zusammen. Im Prolog als XML-Deklaration wird angegeben, wie die Interpretation der Dokumenteninstanz zu erfolgen hat. Sie enthält zum Beispiel die XML-Version, den Kodierungstyp und den Dokumenttyp (Zeile 1 in Listing 2-1). Im Standard wurde die Deklaration der Version eingeführt, um eine automatische Versionserkennung zu erlauben, sollte dies notwendig werden. Die Angabe einer Dokument-Typ-Deklaration (DTD - Document Type Definition) im Prolog ist optional und verhilft einem XML-Dokument bei nachgewiesener Korrektheit zum Status gültig (valid). Fehlt eine DTD, kann ein XML-Dokument maximal den Status wohlgeformt (well-formed) erreichen. Eine DTD definiert weitere Bedingungen für die Struktur einer Klasse von Dokumenten. Sie ist Bestandteil eines XML-Dokuments und kann innerhalb des Dokuments oder separat abgespeichert werden.

```
1.   <?xml version="1.0" encoding="UTF-8"?>
2.
3.   <root>
4.   <tagname1> ... text ... </tagname1>
5.   <tagname2></tagname2>
6.   <tagname3/>
7.   <tagname4 attribut1="0" attribut2="1"/>
8.   <tagname5 attribut1="0"> ... text ... </tagname5>
9.   <![CDATA[ ... nicht interpretierter text ... ]]>
10.  </root>
```

Listing 2-1. Syntax eines wohlgeformten XML-Dokuments.

Die Dokumenteninstanz folgt unmittelbar dem Prolog. Sie umfasst mindestens ein Wurzelelement und fasst die Datenelemente hierarchisch zusammen. Jedes Element ist durch Start- und End-Tags (Zeile 3 und 10 in Listing 2-1) oder, im Falle eines leeren Elements, durch ein Leeres-Element-Tag begrenzt (Zeile 5 oder 6 in Listing 2-1). Der Bezeichner für ein Tag gibt gleichzeitig die Bedeutung des Elements vor. Das Element root stellt das Wurzelelement des Beispiel-XML-Dokuments dar und repräsentiert das Dokument als Ganzes. Jedes weitere Element stellt eine Komponente des Ganzen dar. Das eigentliche Dokument

entspricht einer Baumstruktur, welche die interne Struktur, also die Schachtelung der Tags im Text des Dokuments, wiederspiegelt. Dieser Baum wird durch einen sogenannten XML-Parser (oder XML-Prozessor) während der Verarbeitungsphase aufgebaut und gibt dem Anwender oder der Anwendung die Möglichkeit, über eine definierte Schnittstelle auf einzelne Knoten (Tags), deren Attribute oder Blätter (Daten) des Baumes zuzugreifen.

Die Attribute eines Elements werden innerhalb der spitzen Klammern des Start-Tags in Art einer Aufzählung vermerkt. Zeile 7 in Listing 2-1 enthält ein ansonsten leeres Element, das zwei Attribute besitzt. Ist das Element nicht leer, ist es mit einem End-Tag abzuschließen, das den Namen des Start-Tags wiederholt (Zeile 8 in Listing 2-1). Mit einem speziellen CDATA-Abschnitt werden ganze Textblöcke geschützt, die Zeichen enthalten, die normalerweise als Tag interpretiert würden (Zeile 9 in Listing 2-1). Dies lässt sich für individuelle textuelle Beschreibungen, Programmcode oder ähnliches nutzen.

XML ist erweiterbar. Im Gegensatz zu HTML lassen sich beliebige eigene Tags definieren. Dazu müssen jedoch Richtlinien eingehalten werden, damit beliebige Computerprogramme die XML-Dokumente auch verarbeiten können. Ein XML-Dokument heißt wohlgeformt, wenn alle syntaktischen Regeln für das Hauptdokument eingehalten werden, wenn z. B. jedes Start-Tag mit einem End-Tag versehen und die Reihenfolge der Verschachtelung der Start- und End-Tags korrekt ist. Dies stellt den kleinsten gemeinsamen Nenner für die erfolgreiche Verarbeitung von XML-Dokumenten dar. Darüber hinaus heißt ein XML-Dokument gültig, wenn eine DTD definiert und eingehalten wird. Aufgrund dessen lassen sich XML-Parser in zwei Klassen unterteilen: die nicht validierenden und die validierenden. Der nicht validierende Parser überprüft nur, ob ein XML-Dokument wohlgeformt ist. Im Gegensatz dazu überprüft der validierende Parser auch die Dokumentenstruktur anhand der angegebenen DTD. Weitere Informationen zum Aufbau eines XML-Dokuments sind in der Spezifikation [125, 126] oder in den zahlreichen XML-Büchern nachzulesen.

3 VHDL-Simulation

Ausgehend von den VHDL-Simulationstechniken und -Simulationstypen wird in diesem Kapitel hauptsächlich die VHDL-Board-Level-Simulation vorgestellt, die im Rahmen dieser Arbeit für die homogene HW/SW-CoSimulation eingesetzt wird.

Unter einer **Simulation** ist der Prozess zu verstehen, einem Modell Stimuli zuzuführen und die zugehörigen Antworten des Modells zu dem Zeitpunkt zu generieren, zu dem sie auftreten. Sie dient der Verifikation der Korrektheit von Entwurfsschritten innerhalb des Entwurfsflusses. Bei einer VHDL-Simulation wird die Hardwarebeschreibungssprache VHDL zur Modellierung und Simulation eingesetzt. Es werden überwiegend Verhaltensmodelle auf abstrakten Entwurfsebenen (System-, Algorithmische und Register-Transfer-Ebene) und die entsprechenden strukturalen Modelle verwendet.

3.1 VHDL-Simulationstechniken

Die Kenntnis der verschiedenen Simulationstechniken erleichtert die Auswahl des Simulators, da diese direkt die Leistungseigenschaften beeinflusst. Es werden für die VHDL-Simulation grob zwei Ansätze unterschieden: interpretierende und compilierende Simulationstechniken. Eine dritte Simulationstechnik vereint die Vorteile der beiden vorgenannten Ansätze. Zusammenfassend lassen sich folgende VHDL-Simulationstechniken charakterisieren:

- *Interpretierende Simulationstechnik:* Ist durch eine kurze Simulationsvorbereitung, dafür aber durch eine längere Simulation gekennzeichnet. Der Eingabe-Code wird in einen Zwischen-Code übersetzt, der vom Simulator ausgeführt wird. Diese Technik ist aufgrund der längeren Simulationszeiten eher für kleine Modelle geeignet.
- *Compilierende Simulationstechnik:* Hierbei wird der VHDL-Quelltext komplett in die Programmiersprache C übersetzt und anschließend von einem C-Compiler in Objekt-Code umgesetzt (C-compiled). Die Zeit zur Vorbereitung der Simulation steigt. Die Simulation selbst wird beschleunigt, da direkt Maschinen-Code ausgeführt wird. Damit eignet sich diese Technik auch für größere Modelle.
- *Native-compiled Simulationstechnik:* Vereint die Vorteile der interpretierenden und compilierenden Technik. Die Umsetzung des VHDL-Quelltextes in die Programmiersprache C entfällt, stattdessen wird direkt der Objekt-Code für die Simulation erzeugt (Direct-compiled). Es wird sowohl die Simulationsvorbereitung als auch der Simulationsablauf optimiert.

Zusätzlich zu den Eigenschaften der genannten Simulationstechniken gibt es noch Verbesserungen im Detail. Für die native-compiled Simulationstechnik z. B. wird von einigen Simulatoren der Objekt-Code zur Übersetzungszeit zunächst plattformunabhängig abgelegt. Beim Start der Simulation wird dieser in einen plattformabhängigen Code zur Optimierung

umgewandelt. Moderne kommerzielle VHDL-Simulatoren verwenden in der Regel die native-compiled Simulationstechnik.

3.2 VHDL-Simulationstypen

Unterschieden wird bei VHDL-Simulatoren zwischen der ereignisgesteuerten (event-driven) und der zyklusbasierten (cycle-based) Simulation, mit möglichen hybriden Realisierungen (Bild 3-1).

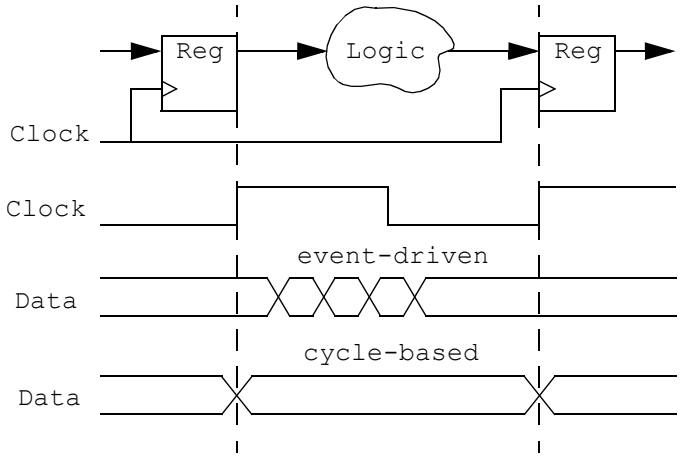


Bild 3-1. Gegenüberstellung von ereignisgesteuerter und zyklusbasierter Simulation.

Bei der ereignisgesteuerten Simulation werden Änderungen der Signale während eines Taktzyklus über Einträge in einer Ereigniswarteschlange verfolgt. Für jedes Signal wird nicht nur der Wert, sondern auch die Zeit berücksichtigt, zu dem sich der Wert für das Signal ändert. In jedem Zeitschritt werden zu Ereignissen in der Warteschlange, die zu diesem Simulationszeitpunkt auftreten, zugehörige Aktionen ausgelöst, deren Ergebnisse wieder neue Ereignisse für die Warteschlange produzieren können. Sind alle Ereignisse des momentanen Zeitpunkts abgearbeitet, wird die Simulationszeit um einen Zeitschritt erhöht bzw. auf den Zeitpunkt des frühest möglichen nächsten Ereignisses gesetzt. Undefinierte Signalzustände und schwache Signalpegel werden als Signalwerte zugelassen. Es werden wenig oder keine Restriktionen hinsichtlich der zu verwendenen VHDL-Konstrukte vorgeschrieben. Diese Simulationsart ist daher sehr flexibel und exzellent geeignet, Probleme im Zeitverhalten der Signale zu finden, aber auch sehr aufwendig und daher langsam.

Bei den zyklusbasierten Simulatoren wird der Entwurf in synchrone und asynchrone Teile unterteilt. Der aktuelle Trend geht immer mehr zu reinen taktsynchronen Entwürfen, von wenigen Ausnahmen abgesehen. Anstatt jede Signaländerung im synchronen Teil zu beliebigen Zeitpunkten zu verfolgen, werden die Signale nur zu den Taktzeitpunkten in der richtigen Reihenfolge berechnet. Dadurch ergibt sich eine schnellere Ausführung der Simulation und es wird nur ein Teil des Speicherbedarfs einer ereignisgesteuerten Simulation benötigt. Nachteilig sind die Restriktionen, die die Verwendung von HDL-Konstrukten, ähnlich der für die Synthese, einschränken.

3.3 VHDL-Board-Level-Simulation

Eine Board-Level-Simulation kann definiert werden als Simulation ein oder mehrerer miteinander verbundener Leiterplatinen, die Standardkomponenten, aber auch ASICs, ASSPs, ASIPs oder FPGAs enthalten können. In diesem Zusammenhang wird oft von „Virtual Prototyping“ und manchmal auch von System-Simulation gesprochen. In diesem Kontext bezeichnet der Begriff System-Simulation nicht die Simulation auf Systemebene, sondern die Simulation des kompletten modellierten Board-Level-Systems. Ähnlich kann das Zusammenspiel der Komponenten eines SoC als einem modernen Systemansatz mit Hilfe einer Board-Level-Simulation simuliert werden.

Das Ziel der Board-Level-Simulation ist die Verifikation des Verhaltens des Board-Entwurfs, so z. B. ob die Komponenten im selektierten Modus korrekt arbeiten oder die Zusammenschaltung der Komponenten stimmt. Mit ihrer Hilfe ist es ebenso möglich, eine Verifikation der HW/SW-Kommunikation durchzuführen, indem z. B. überprüft wird, ob Register der Hardware programmiert werden können oder ob die Software-Treiber ein korrektes Verhalten zeigen (Bild 3-2). Zusätzlich kann die Leistungsfähigkeit des Systems bis ins Detail analysiert werden, denn eine Board-Level-Simulation gibt auch einen Einblick in das zeitliche Verhalten des Systems.

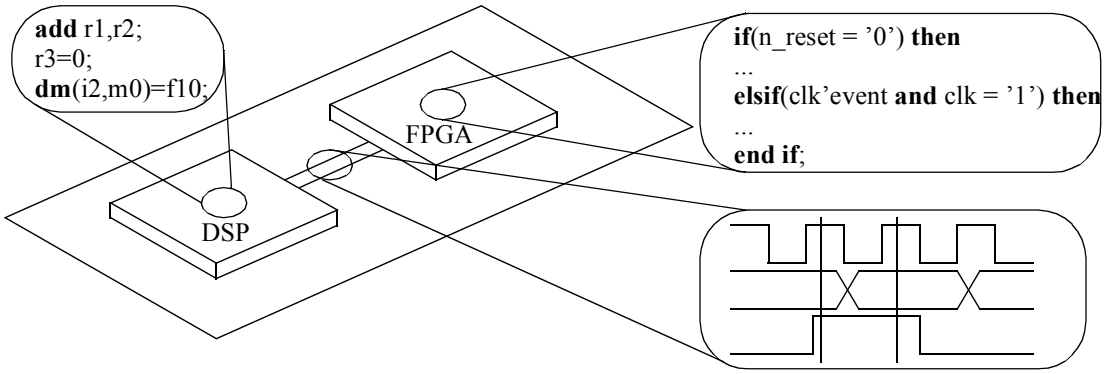


Bild 3-2. Die Sicht auf den Entwurf in einer Board-Level-Simulation, unabhängig von einer Realisierung als Leiterplatine oder als SoC.

Das Gebiet dieser Simulationen wird immer interessanter, je schneller die zur Verfügung stehenden Simulatoren und je größer die Rechenkapazitäten der Computeranlagen werden, denn Zeit ist der limitierende Faktor bei dieser Vorgehensweise. Es ist sicherlich nicht sinnvoll, ein komplettes Betriebssystem und entsprechende Applikationen innerhalb der Simulation zu starten, der Zeitaufwand für die Simulation wäre einfach zu hoch, jedoch können die Firmware, ein RTK und die Hardware-Treiber mit Hilfe der Board-Level-Simulation getestet werden. Die Zeit, die für die Modellierung und die Simulation aufgewendet werden muß, ist dem Nutzen, der daraus zu ziehen ist, gegenüberzustellen.

3.3.1 Anwendungen und Vorteile

Befindet sich die Hardware eines Entwurfs in der Entwicklung, können innerhalb des Top-Down-Entwicklungsprozesses schon relativ frühzeitig die Anforderungen überprüft wer-

den. Es können mehrere Lösungen im Prototyp-Stadium ausgetestet werden. Die Produktion der Hardware wird verzögert, bis die gesamte Spezifikation festgeschrieben ist und alle Schnittstellen etabliert und getestet sind. Die Board-Level-Simulation bringt den Hardware- und den Software-Entwickler in einer frühen Phase des Entwurfsprozesses zusammen und lässt beide gemeinsam Schnittstellenprobleme lösen, bevor die Hardware feststeht.

Steht bereits eine Leiterplatine als Prototyp für die Entwicklung von SoCs zur Verfügung, so bietet die Board-Level-Simulation die Möglichkeit, die Verifikation auf die Theorie auszudehnen. Denn nicht alle Signale der Leiterplatine könnnen mit Test-Equipment versehen werden, z. B. aus Platzgründen oder aufgrund des komplizierten Zeitverhaltens der Signale. Hardware- und Software-Entwickler können bei jeder neuen SoC-Aufgabenstellung gemeinsam die HW/SW-Schnittstellen und das Zusammenspiel zwischen Hardware und Software überprüfen. Der Hardware-Entwickler erhält dabei eine andere Sicht auf die Board-Level-Simulation als der Software-Entwickler. Für ersteren z. B. in Form von Signalverläufen und für letzteren in Form eines Disassemblers.

Sind Prozessoren oder Prozessorkerne vorhanden, bietet es sich an, Analysen der Leistungseigenschaften aus dem zeitlichen Verhalten der Software-Ausführung und ihrer Interaktion mit der Hardware abzuleiten (Profiling). Voraussetzung dafür ist, dass der Detailgrad der Prozessormodellierung entsprechend hoch ist. Essentiell ist dabei die Zyklentreue der Software-Ausführung, die z. B. in Prozessoren mit Cache eine Rolle spielt. Es ergibt sich im Laufe der Simulation ein messbarer Unterschied, ob die Ausführung eines Software-Befehls vereinfachend auf einen Systemtaktschritt abgebildet wird oder mit dem realen Zeitverhalten, zwei oder mehr Systemtaktschritte, modelliert wird.

Kommt VHDL für die Modellierung der Komponenten einer Board-Level-Simulation zum Einsatz, so wird die Wahrscheinlichkeit, die Simulation auf verschiedenen Plattformen und Simulatoren durchführen zu können, erheblich vergrößert, da VHDL-Modelle keine bzw. sehr wenige Anpassungen an einen anderen Simulator erfordern. Die Simulation kann auch Nicht-VHDL Repräsentationen enthalten, so z. B. Netzlisten (Post-Synthese) oder schematische Eingabedateien. Dies ist immer dann sinnvoll, wenn zu einem Baustein kein VHDL-Modell existiert.

3.3.2 Modellierungskonzepte

Modelle stellen eine formale Beschreibung einer Sache dar. Die Beschreibung ist abstrakt, d. h., sie enthält nur das Notwendigste und ist bezüglich ihrer Abstraktion vollständig.

Ein niedriger Abstraktionsgrad ist mit einem hohen Detailgrad, im Gegensatz dazu ist ein hoher Abstraktionsgrad mit einem niedrigen Detailgrad gleichzusetzen. Die verschiedenen Abstraktionsgrade werden in Abstraktionsebenen unterteilt, die spezifisch für ein Problemkreis definiert werden müssen. Im Hardwareentwurf werden die folgenden Abstraktionsebenen unterschieden: System-, Algorithmische-, Register-Transfer-, Logik-/Gatter-, Switch-Level- und Transistorebene. Die Register-Transfer-Ebene spielt für die High-Level-Synthese eine wichtige Rolle, die auf HDL-Beschreibungen aufsetzt.

Die Modelle im Bereich der Hardware-Modellierung werden hinsichtlich des Detailgrades der Modellierung der Zeit, der Daten, der Funktion, der Struktur und der Software unterschieden. Die zeitliche Auflösung beinhaltet die Genauigkeit des zeitlichen Verhaltens der zu modellierenden Komponente. Sie reicht von einer zeitlich geordneten Menge von Ereignissen (Datenfluss) bis hin zu einer (nano-/pico-)Sekunden-genauen Beschreibung des Start-/Endpunktes eines Ereignisses. Der Detailgrad der Daten eines Modells entspricht der Präzision, mit der die Datenwerte dargestellt werden. Präzision ist in diesem Kontext nicht mit Genauigkeit gleichzusetzen. Daten können ohne Informationsgehalt als Token oder aber auch mit der höchsten Präzision auf Bitebene repräsentiert werden. Die Beschreibung der Funktion einer Komponente in einem Modell ist ebenfalls auf unterschiedlichen Detailstufen möglich. Ein von der Komponente berechneter Algorithmus kann als eine Menge von mathematischen Formeln aufgefasst oder durch eine Verknüpfung von Logikgleichungen repräsentiert werden. Der Detailgrad für die Modellierung der Struktur einer Komponente reicht vom Extrem „keine internen Implementierungsinformationen vorhanden“ (Black-Box) bis hin zu „gesamte Implementierung strukturell beschrieben“ (White-Box). Dabei wird nicht zwischen physikalischer und virtueller Implementierung unterschieden. Der letzte Unterscheidungspunkt für die Modelle im Bereich der Hardware-Modellierung ist der Detailgrad der Software-Repräsentation. Sie bestimmt, wie die Software im Modell bei der Ausführung interpretiert wird. Auf der höchsten Abstraktionsebene werden lediglich die für den Datenfluss notwendigen Funktionsblöcke zusammengefasst. Der eigentliche Objekt-Code der Software wird auf der niedrigsten Abstraktionsebene im Modell interpretierend ausgeführt.

Es werden generell primäre und spezielle Modellierungsklassen unterschieden. Modelle einer **primären Modellierungsklasse** können prinzipiell auf allen Abstraktionsebenen existieren. Dazu gehören:

- *Behavioral Model (Function with Timing):* Ein Verhaltensmodell modelliert die Funktion und das Zeitverhalten einer Komponente, ohne eine konkrete Implementation zu beschreiben. Der Abstraktionsgrad hängt davon ab, wie weit das Modell von den Implementierungsdetails losgelöst ist. Ein Prozessor kann z. B. auf der algorithmischen Ebene oder auch auf der Befehlssatzebene mit niedrigerem Abstraktionsgrad modelliert werden.
- *Functional Model (Function without Timing):* Ein funktionales Modell beschreibt die Funktion einer Komponente oder eines Systems, ohne das Zeitverhalten und konkrete Implementierungsdetails zu berücksichtigen. Alle sonstigen Eigenschaften des Verhaltensmodells gelten auch für ein funktionales Modell.
- *Structural Model:* Ein Strukturmodell repräsentiert eine Komponente oder ein System mittels Verbindungen zwischen den konstituierenden Unterkomponenten. Die Beschreibung folgt der Hierarchie einer möglichen Implementation. Die Modelle der Unterkomponenten können struktural, funktional oder verhaltensspezifisch angelegt sein. Damit ein Strukturmodell simuliert werden kann, müssen alle Blätter des Hierarchiebaums als funktionale Modelle oder Verhaltensmodelle vorliegen.
- *Interface Model (bus-functional Model):* Ein Interface-Modell beschreibt eine Komponente bei der Interaktion mit anderen Komponenten ihrer unmittelbaren Umgebung. Modelliert werden die Anordnung, die Funktion und das Zeitverhalten der extern sicht-

baren Schnittstellen der Komponente. Es werden weder Implementierungsdetails noch die Funktionalität der Interna der Komponente betrachtet.

Die **speziellen Modellierungsklassen** sind nicht auf eine Abstraktionsebene festgelegt, sondern sie sind für verschiedene Fälle spezifisch. Dazu gehören im Einzelnen die folgenden Modellierungsklassen:

- *Performance Model (Uninterpreted Model):* Ein Performance-Modell kann prinzipiell auf allen Abstraktionsebenen implementiert werden. Mit ihnen werden Informationen zur Leistungsfähigkeit des modellierten Systems ermittelt. In diesem Kontext umfasst die Leistung das Antwortzeitverhalten, den Durchsatz und die Auslastung der Ressourcen des Systems. Für einen Prozessor kann z. B. die Zeit modelliert sein, die für den Zugriff auf den Speicher benötigt wird.
- *Mixed-Level Model (Hybrid Model):* Ein Mischebenenmodell stellt eine Kombination von Modellen verschiedener Abstraktionsebenen dar. In einem Performance-Modell sind z. B. Verhaltensmodelle für Unterkomponenten eingebettet. An den Ein- und Ausgängen des eingebetteten Verhaltensmodells sind besondere Schnittstellen zur Datenkonvertierung einzurichten.
- *Virtual Prototype:* Als virtuellen Prototyp wird eine Computer-Simulation einer Komponente bzw. eines kompletten Systems bezeichnet. Es handelt sich nicht direkt um ein Modell, sondern um die Rolle, die ein Modell im Entwurfsprozess auf verschiedenen Abstraktionsebenen einnehmen kann. Die Rolle kann beispielhaft sein, auf der Systemebene verschiedene Entwurfsalternativen zu testen oder auf der Register-Transfer-Ebene die Erfüllung der Entwurfsvorgaben und die korrekte Funktionsweise zu überprüfen. Im Gegensatz zu einem physischen Prototyp kann ein virtueller Prototyp früher im Entwurfsprozess eingesetzt werden, außerdem sind Änderungen kosteneffektiver und schneller eingearbeitet. Zusätzlich gewährt die Simulation auf dem Rechner einen tieferen Einblick in das modellierte System als es bei einem Hardware-Prototyp möglich wäre.

Es werden folgende **Modelle zur Hardware-Beschreibung** auf spezifischen Abtraktionsebenen (mit absteigendem Abstraktionsgrad) unterschieden:

- *Abstract Behavioral Model:* Ein Verhaltensmodell, bei dem, im Gegensatz zum Detailed-Behavioral Model, die Schnittstellen der Komponente abstrakt modelliert werden. Das heißt, die Schnittstellen werden nicht auf der Pin-Ebene beschrieben. Der Datenport eines Mikroprozessors kann z. B. als Datentyp Integer repräsentiert werden.
- *Detailed-Behavioral Model (full-functional Model):* Ein Verhaltensmodell, das das Verhalten und die Schnittstellen einer Komponente modelliert. Die Schnittstellen der Komponente werden bis auf Pin-Ebene exakt beschrieben. Die Modelle enthalten alle Informationen über das Zeitverhalten und die Funktionalität, ohne die interne Struktur der Komponente detailgetreu zu beschreiben. Das in Modellen höherer Abstraktionsebenen fehlende Zeitverhalten wird mit Hilfe dieser Modelle dargestellt und erhöht damit die Genauigkeit.
- *Instruction Set Architecture (ISA) Model:* Beschreibt das Verhalten eines programmierbaren Prozessors auf Befehlssatzebene ohne externe Schnittstellen. Das im Modell aus-

geführte Maschinenprogramm liefert die gleichen Ergebnisse wie auf dem realen Prozessor, soweit im Maschinenprogramm keine externen Zugriffe initiiert werden. Die zeitliche Auflösung ist auf einen Instruktionszyklus festgelegt, der in der Realität unter Umständen mehrere Taktzyklen umfassen kann. Die ISA-Modelle repräsentieren keine internen Prozessorstrukturen, aber Register und Speicherstellen werden Bit-genau dargestellt.

- *Register Transfer Level (RTL) Model:* Modelliert eine Komponente aus einer Zusammenschaltung von Registern, Multiplexern und kombinatorischer Logik. Einige interne Strukturmerkmale können ausgedrückt werden, sie sind aber nicht bindend. RTL-Modelle können auch Zustandsautomaten verwenden.
- *Logic Level Model:* Eine Komponente wird mittels äquivalenter boolescher Funktionen und einfacher Speicherelemente, z. B. Flip-Flops, modelliert. Das Modell enthält nicht die exakte Implementation basierend auf logischen Gattern.
- *Gate-Level Model:* Ein Gate-Level-Modell beschreibt die Funktion, das Zeitverhalten und die Struktur einer Komponente mit Hilfe einer strukturierten Zusammenschaltung von logischen Blöcken. Diese logischen Blöcke umfassen einfache boolesche Verknüpfungen wie AND, OR, NOT, NAND, NOR und XOR. Das Modell repräsentiert die aktuelle Struktur der Implementation der modellierten Komponente.
- *Switch-Level Model:* Es werden die Verbindungen der Transistoren einer Schaltung repräsentiert, wobei die Transistoren als einfache Schalter (on/off) modelliert werden.
- *Circuit-Level Model:* Ein Modell das die Strom-/Spannungscharakteristik einer Schaltung, bestehend aus Widerständen, Kapazitäten und Induktivitäten, modelliert.

Auf System-Modelle, Architektur-Modelle und Software-Modelle wird an dieser Stelle nicht weiter eingegangen, da sie für eine VHDL-Board-Level-Simulation nicht verwendet werden. Weiterführende Informationen zu den Modellierungskonzepten sind [123] zu entnehmen.

Für die **VHDL-Board-Level-Simulation** kommt eine Mixtur aus Strukturmodellen, Verhaltensmodellen, funktionalen Modellen und Interface-Modellen auf den Abstraktionsebenen Register-Transfer und Logik/Gatter zum Einsatz. Das Modell für die Leiterplatine ist ein reines Strukturmodell, das die Verbindungen der Komponenten beschreibt. Die einzelnen Komponenten können die gesamte genannte Bandbreite der Modelle für die VHDL-Board-Level-Simulation verwenden. Die Entscheidung, welches Modell auf welcher Abstraktionsebene für eine Komponente eingesetzt wird, ist abhängig von der Verfügbarkeit, den Leistungseigenschaften der Simulation und des notwendigen Detailgrades. In VHDL können über Architekturdeklarationen Modellvarianten verschiedener Abstraktionsebenen in eine Modellbeschreibung integriert werden. Je detaillierter ein Entwurf ausgearbeitet ist, desto detaillierter werden auch die eingesetzten Modellvarianten. Allgemein gilt: Je höher der Abstraktionsgrad, desto größer ist die Simulationsgeschwindigkeit und umgekehrt.

Unterschieden werden muss zwischen Verhaltensmodellen und Interface-Modellen für die Board-Level-Simulation. Bei Verhaltensmodellen werden die Eingangssignale auf unzulässige Werte und die Einhaltung von Timing-Randbedingungen überprüft. Die Ausgangssignale erhalten zu bestimmten Zeitpunkten (Timing) in Abhängigkeit vom Zustand des

Kerns der Komponente ihre entsprechenden Signalwerte. Innerhalb der Simulationsumgebung können Debug-Informationen ausgetauscht werden bzw. über Assertions Meldungen zur Laufzeit der Simulation generiert werden (Bild 3-3). Bei Interface-Modellen fehlt die komplette Modellierung des Kerns der Komponente. Aktionen, die von der zu modellierenden Komponente ausgehen, werden generisch als Stimuli in das Interface-Modell integriert. In einer HW/SW-CoSimulation ist es jedoch unerlässlich den Zustand des Kerns der Softwarekomponente (GPP, DSP, ASIP) in die Simulation einzubeziehen, damit das Verhalten des gesamten Systems verifiziert werden kann.

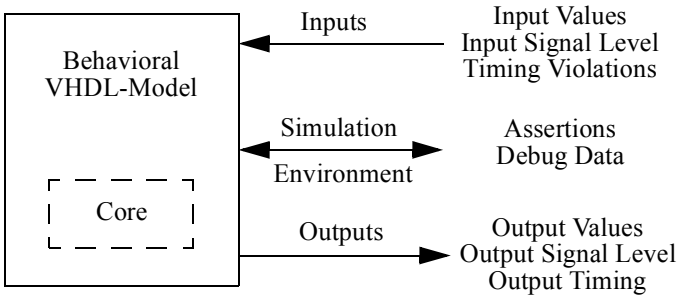


Bild 3-3. Blockbild eines VHDL-Verhaltenmodells (Funktion + Timing).

Es gibt prinzipiell zwei Möglichkeiten der Modellierung für die Board-Level-Simulation. Erstens eine unabhängige Entwicklung des Modells aus einer funktionalen Spezifikation oder eines Datenblattes, und zweitens der Ausbau eines vorhandenen synthetisierbaren Entwurfs auf Gatterebene. Bei der Entwicklung aus einem lediglich vorhandenen Datenblatt kann es vorkommen, dass Details fehlen, Fehler enthalten sind oder auch dass das Datenblatt an einigen Stellen unterschiedlich interpretiert wird. Zumindest eine zusätzliche Quelle als Referenz sollte zur Sicherstellung des Wahrheitsgehaltes der Informationen zur Verfügung stehen. Wenn es Inkonsistenzen zwischen dem Datenblatt und einer weiteren Referenzquelle gibt, ist es unangebracht, die betreffende Funktion aus der Implementierung herauszunehmen, denn dann wären die Ergebnisse der Simulation unbrauchbar. Besser ist es, an dieser Stelle einen Hinweis oder eine Warnung zur Laufzeit der Simulation zu erzeugen, die die unter Umständen fehlerhafte bzw. fehlende Funktionalität meldet. Ein Ausbau eines vorhandenen synthetisierbaren Entwurfs auf eine höhere Abstraktionsebene ist möglich und gut für die Leistungseigenschaften der Simulation. Wie in [54] dargelegt, sollte ein Simulationsmodell mindestens auf Register-Transfer-Ebene oder höher definiert sein.

3.3.3 Anforderungen an die Modelle

Das Verhalten des Modells, so wie es von außen zu sehen ist, sollte dem Verhalten entsprechen, welches die zu modellierende Komponente zeigt und sollte auch die volle Funktionalität umfassen. Ausgenommen davon sind spezifische Testmodi, die nur bei der Produktion der Komponente benötigt werden. Die Schnittstellensignale des Modells müssen denselben Signalverlauf haben, wie er bei der Komponente beobachtet wurde bzw. der im Datenblatt angegeben ist. Das Modell braucht nicht die interne Struktur und den internen Zustand der Komponente widerzuspiegeln. Interne Signale müssen nicht im Detail simuliert werden, weil sie unter Umständen keine genauen Informationen über die wahren internen Zusammenhänge liefern können.

Wichtige Aufgaben eines Modells einer Komponente für die Board-Level-Simulation sind:

- Dynamische und statische Eingangssignale sind auf nicht zulässige Werte zu überprüfen, dazu zählen die Treiberstärken 'X', 'U', '-'.
- Prüfen der Einhaltung von Zeitparametern, wie z. B. Setup- und Hold-Zeiten an den Eingängen.
- Generierung der Verzögerungen für Ausgangssignale.

Die Modelle sollten eine flache Hierarchie aufweisen, damit wird die Anzahl der Signale zwischen den Hierarchie-Ebenen und daraus resultierend der Simulationsaufwand reduziert. Für sehr kleine Modelle reichen ein bzw. zwei Hierarchie-Ebenen aus, wie in Bild 3-4 dargestellt. Größere Modelle sind entsprechend ihrer funktionalen Gliederung zu segmentieren. Zum Beispiel enthält ein Prozessor neben seinem Rechenkern (ALU, Multiplizierer etc.) auch einen I/O-Kern, die beide Bestandteil der Hierarchie des gesamten Prozessormodells sind.

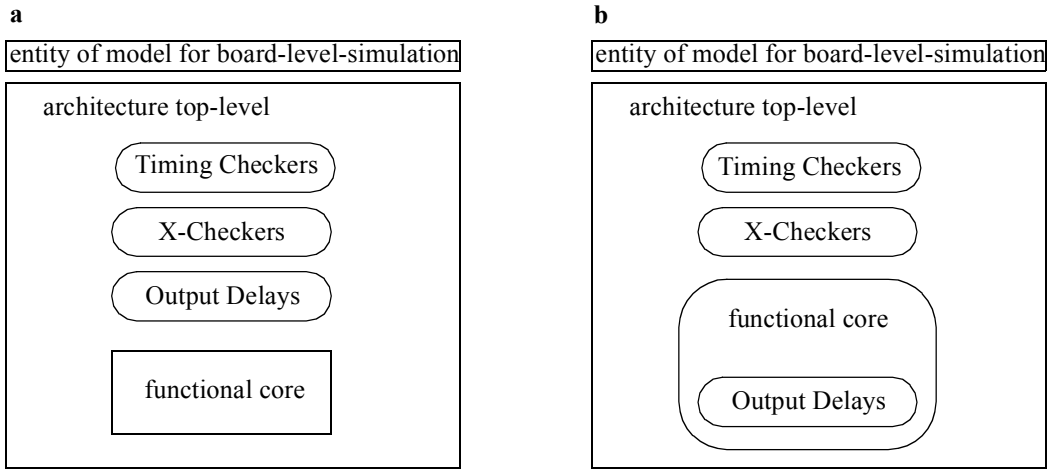


Bild 3-4. Trennung der funktionalen Modellierung von der Modellierung des Zeitverhaltens nach außen **a** in zwei oder **b** einer Hierarchie-Ebene. Rechtecke mit runden Ecken stellen nebenläufige Signalzuweisungen oder Prozesse dar, die „normalen" Rechtecke kennzeichnen Komponenten einer untergeordneten Hierarchie-Ebene.

3.3.4 Aspekte der Leistungseigenschaften der Simulation

Übergreifend lässt sich feststellen, dass die Simulationswerkzeuge und die Rechenanlagen, auf denen die Simulationen laufen, immer leistungsfähiger werden. Werden jedoch einige Grundlagen bereits während der Modellierung einer Komponente für die Board-Level-Simulation beachtet, lässt sich schon hierdurch eine Steigerung der Qualitätseigenschaften der Simulation erreichen. Die im Folgenden angegebenen Strategien wurden vom Autor bei der Modellierung der in dieser Arbeit verwendeten Komponenten verwendet und basieren zum Teil auf Ausführungen in [53, 54].

Prozesse werden zur Laufzeit der Simulation bei Änderungen der Signale in der Sensitivitätsliste aktiviert und bis zur nächsten wait-Anweisung ausgeführt. Jede Aktivierung eines Prozesses hat einen Kostenfaktor in Form von Simulationsrechenzeit, so dass prinzipiell die

Anzahl der Prozesse klein zu halten ist. Auch jede nebenläufige Signalzuweisung kann als Prozess mit einer Sensitivitätsliste ihrer Zuweisungsargumente aufgefasst werden und ist daher, wenn möglich, zu vermeiden. Prozesse sollten generell Sensitivitätslisten verwenden, da diese in der Simulation statisch alloziiert werden und damit potentiell weniger Simulationsrechenzeit beanspruchen. Wait-Anweisungen werden demgegenüber dynamisch erzeugt.

```
1.   xchecker: if(TimingChecksOn) generate
2.   begin
3.        timing: process(n_cs, n_wr)
4.        begin
5.        ...
6.        end process;
7.   end generate;
```

Listing 3-1. TimingChecksOn ist ein Generic-Parameter des Entity vom Typ BOOLEAN, mit dessen Hilfe entschieden wird, ob der Prozess timing in die Simulation einbezogen wird oder nicht. Der Generic-Parameter kann für Simulationsläufe mit unterschiedlichem Detailgrad genutzt werden.

Code-Blöcke, die von der Modellierung der Funktionalität der Komponente unabhängige Funktionen darstellen, z. B. die Überprüfung von Eingangssignalen, sind in einem separaten Prozess zusammenzufassen. Sind diese gesonderten Prozesse über Generic-Parameter von der Simulation ausschließbar, ist dies nicht über eine if-then-else-Bedingung am Anfang des Prozesses zu realisieren, sondern der Prozess an sich ist über die generate-Anweisung ein- oder auszublenden. Ansonsten würde der Prozess unnötigerweise aktiviert werden, wenn es ein Ereignis für ein Signal in der Sensitivitätsliste gibt. Die generate-Anweisung stellt sicher, dass der Prozess von der Simulation ausgeschlossen ist und eliminiert jegliche Kosten für dessen Aktivierung. Listing 3-1 zeigt dafür ein Beispiel.

```
1.   synchron: process(n_reset, clock)
2.   begin
3.        if(n_reset = '0') then
4.             -- asynchrones Reset
5.        elsif(clock'event and clock = '1') then
6.             -- synchrone Zuweisungen
7.        end if;
8.   end process;
```

Listing 3-2. Typischer synchroner Prozess, sensitiv zu den Signalen clock und n_reset.

Bei **taktsynchronen Prozessen**, die typischerweise sensitiv zu einem Taktsignal und einem asynchronen Reset-Signal sind, sind nur diese beiden Signale in der Sensitivitätsliste enthalten, um unnötige Aktivierungen zu vermeiden (siehe Listing 3-2).

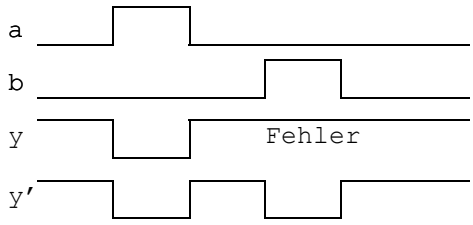
Bei **taktasynchronen Prozessen** stehen nur die Signale in der Sensitivitätsliste, die direkt das Verhalten beeinflussen (Listing 3-3 zeigt dafür ein Beispiel). Diese Vorgehensweise gilt jedoch nur für die Simulation. In einer für die Synthese geeigneten VHDL-Beschreibung dagegen führt dies zu einer fehlerhaften Schaltung, wie das Beispiel in Listing 3-4 zeigt.

Signale sind, wann immer es möglich ist, durch Variablen zu ersetzen. Lediglich die Kommunikation zwischen den Prozessen erfolgt über Signale. Denn jede Signalzuweisung erfor-

```
1.   asynchron: process(n_reset, n_cs, n_rw)
2.   begin
3.        if(n_reset = '0') then
4.             -- asynchronous Reset
5.             DBusEnable <= FALSE;
6.        elsif(n_cs'event and n_cs = '1') then
7.             if(n_rw = '0') then
8.                  Addr <= Addr_In;
9.                  Data <= Data_In;
10.            end if;
11.            DBusEnable <= FALSE;
12.       elsif(Now /= 0 ns) then
13.            DBusEnable <= (n_rw = '1') and (n_cs = '0') and
14.                       (n_reset = '1');
15.       end if;
16.  end process;
```

Listing 3-3. Asynchroner Prozess mit Sensitivitätsliste. Zu beachten ist, dass Addr_In und Data_In nicht in der Sensitivitätsliste stehen, da Änderungen dieser Signale für das Einspeichern keine Rolle spielen. Setup- und Hold-Zeiten müssen in einem separaten Prozess geprüft werden.

```
1.   P1: process(a, b)
2.   begin
3.     if (a='0' and b='0') then
4.       y <= '1';
5.     else
6.       y <= '0';
7.     end if;
8.   end process;
```



Listing 3-4. Das Weglassen eines Signals in der Sensitivitätsliste (in diesem Fall des Signals b) eines Prozesses kann zu einem Fehlverhalten der synthetisierten Schaltung führen - Signalverlauf y in der Simulation vor der Synthese; y' in der Simulation nach der Synthese.

dert die Berechnung der resultierenden Treiberstärke für das Signal, für das es mehrere Treiberquellen geben kann. Für ein Signal vom Typ std_logic sind insgesamt neun Treiberstärken (9-wertige Logik) definiert. Variablen dagegen sind lokal an einen Prozess gebunden und erfordern lediglich eine unter Umständen mehrfach sequentielle, jedoch keine parallele Zuweisung der Treiberstärke. Die 9-wertige Logik ermöglicht die Modellierung des elektrischen Zustands von Leitungen, die innerhalb einer Board-Level-Simulation zwischen den Komponenten auftreten. Mit ihrer Hilfe sind zum Beispiel Pull-Up und Pull-Down-Widerstände zu modellieren und es können fehlerhafte Mehrfach-Zugriffe auf Daten- und Adressleitungen in der Simulation erkannt werden (bus contention checking). Für eine Modellierung auf einer abstrakteren Ebene oder an Stellen, wo die 9-wertige Logik nicht benötigt wird, können Signale oder Variablen vom Typ bit (2-wertige Logik) verwendet werden. Bei der Modellierung von Speicher zum Beispiel lässt sich durch die Verwendung von Variablen vom Typ bit viel Speicherplatz zur Laufzeit der Simulation einsparen.

Die verwendeten **Datentypen** spielen bei den Leistungseigenschaften der Simulation ebenfalls eine wichtige Rolle. Numerische Datentypen, wie z. B. integer, erfordern weniger Rechenzeit und weniger Speicherplatz während der Simulation als std_logic-Vektoren oder bit-Vektoren. Es ist oft schneller, zwei std_logic-Vektoren in Integer-Variablen zu konvertieren, diese zu addieren und das Ergebnis dann wieder in einen std_logic-Vektor zurückzuwandeln, als die Addition direkt auf die beiden std_logic-Vektoren anzuwenden. Ein

Beispiel: Ein 8-Bit-Register braucht 32-Bit Speicherplatz (4 Bytes) für die Repräsentation als std_logic-Vektor, da jedes Bit des Vektors 4-Bit für alle 9 std_logic-Zustände benötigt. Dasselbe Register als Integer (Range 0 to 255) repräsentiert, benötigt nur ein Byte, mit dem Nachteil, dass nicht alle 9 Treiberzustände gespeichert werden. Es ist also zwischen der Genauigkeit der Modellierung interner Speicherstellen einerseits und dem Speicherplatzbedarf während der Simulation andererseits abzuwägen. Die notwendigen Typkonvertierungen sind im funktionalen Teil des Modells durchzuführen, genau dann, wenn der betreffende Wert benutzt wird. Sind die dem Wert zugehörigen Signale Eingänge der zu modellierenden Komponente, kann die Konvertierungsfunktion gleichzeitig die Eingangssignale auf unzulässige Werte (’U’, ’X’, ’W’, ’-’) überprüfen.

Aufzählungstypen sind besser als Array-Typen, besonders bei der Kodierung von Zustandsmaschinen, wo die Verwendung von std_logic-Vektoren zu einer höheren Simulationsrechenzeit führt. Dies liegt daran, dass Zustandsmaschinen in der Regel mit Hilfe von case-Anweisungen implementiert werden. Die Auswahl des entsprechenden case-Zweiges, also des aktuellen Zustandes, erfolgt über einen Vergleich der Zustandsvariablen. Der Simulator mit zugehörigem Compiler hat bei einer Zustandsvariablen in Form eines Aufzählungstyps mehr Möglichkeiten der Optimierung, als wenn Arrays verglichen werden müssen.

Für den Test und die Implementierung eines Modells ist der Datentyp real nicht zu verwenden, da dessen Realisierung simulatorabhängig ist und damit inkonsistente Simulationsergebnisse hervorgerufen werden könnten. Dies würde zudem die Portabilität des Simulationsmodells einschränken.

Es ist nicht sicher, welche **Optimierungen** ein VHDL-Simulator anwendet, so dass es besser ist, schon manuell im Code Optimierungen vorzunehmen, wie z. B. die im Listing 3-5 gezeigte Vorausberechnung gemeinsam genutzter Terme. Es ist einzusehen, dass dadurch Simulationszeit eingespart wird. Andererseits gibt es für temporäre Variablen ebenfalls einen Kostenfaktor, der, wenn es geht, zu vermeiden ist.

```
     -- Original                      -- Optimiert
1.   result0 := a - (b + c);          tmp := b + c;
2.   result1 := d - (b + c);          result0 := a - tmp;
3.                                    result1 := d - tmp;
```

Listing 3-5. Codeoptimierung durch Einfügung temporärer Variablen für gemeinsam genutzte Teilausdrücke.

Listing 3-6 zeigt ein Gegenbeispiel, bei dem die Berechnung temporärer Variablen weggelassen wird, indem die kompletten Ausdrücke zur Berechnung des Ergebnisses herangezogen werden. An dieser Stelle ist die einmalige Verwendung der temporären Variablen das Kriterium hierfür.

```
     -- Original                      -- Optimiert
1.   tmp0 := a + b;                   result := (a + b) / (c mod d)
2.   tmp1 := c mod d;
3.   result := tmp0 / tmp1;
```

Listing 3-6. Zusammenlegung von Teilausdrücken mit Einsparung von Variablenzuweisungen unter der Voraussetzung, dass diese im weiteren Programmverlauf nicht mehr benötigt werden.

Das Prinzip der **Short-Circuit-Evaluation** verhilft ebenfalls zu einer kürzeren Simulationszeit. Hierbei wird ausgenutzt, dass, wenn mehrere Signale in einer bedingten Abfrage ausgewertet werden, jenes Signal zuerst auszuwerten ist, das mit der höchsten Wahrscheinlichkeit die Bedingungsabfrage falsifiziert bzw. diese mit wahr auswertet. Damit ist der Status der anderen Signale unerheblich und deren Prüfung nicht mehr notwendig, wie das Beispiel in Listing 3-7 zeigt. Alle weiteren Signale in der Abfrage sind am günstigsten mit abnehmender Wahrscheinlichkeit zu sortieren.

```
1.   if((n_cs='0') and (wr='1')) then
2.      ...
3.   else
4.      ...
5.   end if;
```

Listing 3-7. Die Reihenfolge der Signale in Bedingungsabfragen ist mit entscheidend für die Leistungseigenschaften der Simulation. Die Bedingung wr='1' muss erst überprüft werden, wenn n_cs='0' gilt (erst wenn die Komponente über das Chip-Select-Signal aktiviert wird, ist es sinnvoll zu prüfen, ob ein schreibender Datentransfer ansteht oder nicht).

Bedingungsabfragen an sich stellen darüber hinaus auch eine Möglichkeit dar, die Simulationsgeschwindigkeit zu verbessern. Die äußeren Abfragen kapseln die inneren, die Zweige mit der höchsten Wahrscheinlichkeit sind zuerst zu prüfen. Verkettete boolesche Ausdrücke in einer Abfrage sind so anzuordnen, dass die Ausdrücke, von denen die meisten anderen in der Hierarchie abhängen, zuerst vorkommen, um den gesamten Term so früh wie möglich zu falsifizieren.

Wird das komplette VHDL-Modell einer Komponente eines Board-Entwurfs nicht zur Verifikation benötigt, kann sie durch eine Minimalversion, auch als **minimales Interface-Modell** bezeichnet, ersetzt werden. Dieses ist darauf abgestimmt, auf allen Ausgängen, eventuell in Abhängigkeit von Konfigurationseingängen, einen logischen Pegel zu treiben. Damit kann in der Simulation geprüft werden, ob es einen Treiberkonflikt auf den entsprechenden Leitungen gibt (contention checking). Außerdem prüft das minimale VHDL-Modell die Eingangssignale darauf, ob es einen lesenden oder schreibenden Zugriff auf die Komponente gibt, z. B. ein aktives Chip-Select-Signal in Kombination mit einem Write-Enable-Signal. Dies stellt eine Verletzung der Bedingung dar, dass die Komponente nicht zur Verifikation benötigt wird, und ist durch eine geeignete Meldung dem Entwickler mitzuteilen. Die Unterscheidung zwischen dem kompletten VHDL-Modell und der Minimalversion kann mittels der Eigenschaft von VHDL genutzt werden, mehrere Architekturen für ein Entity zu definieren. In verschiedenen Konfigurationen ist es dann möglich, zwischen den definierten Architekturen eines Entity auszuwählen (Listing 3-8).

Es gibt oft bei einer Komponente für die Board-Level-Simulation, z. B. bei einem Prozessor oder Prozessorkern, die Situation, dass alle taktsynchronen Prozesse im Laufe der Simulation aktiviert werden, weil ein Ereignis für das Taktsignal clock ansteht, jedoch aufgrund einer besonderen Situation, z. B. eines idle-Zyklus oder das Warten auf ein externes Ereignis, die Berechnung des Prozesses unnötigerweise erfolgt. Dafür sind besondere Signale innerhalb des Modells vorzusehen (halt_condition), die aktiviert werden, wenn eine solche vorgenannte Situation eintritt. Eine Aktivierung hat zur Folge, dass das ursprüngliche Taktsignal abgeschaltet wird (**„gated clock“**), so dass keine Ereignisse für die Warteschlange

```
1.   entity model is
2.     ...
3.   end model;
4.
5.   architecture full_functional of model is
6.     ...
7.   end full_functional;
8.
9.   architecture minimal_functional of model is
10.    ...
11.  end minimal_functional;
12.  ----------------------------------------
13.  entity system is
14.    ...
15.  end system;
16.
17.  architecture behave of system is
18.    component model
19.      ...
20.    end component;
21.  begin
22.    u1: model;
23.    ...
24.  end behave;
25.
26.  configuration system_with_full_functional_model of system is
27.    for behave
28.      for u1: model use entity work.model(full_functional);
29.      end for;
30.    end for;
31.  end;
32.
33.  configuration system_with_minimal_functional_model of system is
34.    for behave
35.      for u1: model use entity work.model(minimal_functional);
36.      end for;
37.    end for;
38.  end;
```

Listing 3-8. Definition und Auswahl eines kompletten VHDL-Modells und wahlweise eines minimalen VHDL-Modells.

der Simulation produziert werden (siehe Listing 3-9). Damit kann effektiv Simulationszeit eingespart werden. Alle Prozesse, die verantwortlich dafür sind, dass die besonderen Situationen unterbrochen werden, z. B. Prozesse zur Interruptverarbeitung, sind mit dem ursprünglichen Taktsignal clock anzusteuern, so dass sie weiterhin aktiv bleiben.

3.3.5 Verifikation der Modelle

Die Aufgabe der Verifikation ist es, die Korrektheit des Modells gegenüber dem funktionalen Verhalten und dem Zeitverhalten der Komponente sicherzustellen. Existiert für die Komponente keine Netzliste auf Gatterebene bzw. keine synthetisierbare Beschreibung auf Register-Transfer-Ebene als Referenz, ist es angebracht, die Verifikation unabhängig von der Entwicklung des eigentlichen Modells durchzuführen, um z. B. Fehlinterpretationen des

```
1.   gated_clock: process(n_reset, clock)
2.   begin
3.     if(n_reset = '0' or (clock'event and clock = '0')) then
4.       gated_clock <= '0';
5.     elsif(clock'event and clock = '1') then
6.       -- halt_condition asserted/deasserted at falling edge of clock
7.       if(halt_condition = '0') then
8.         gated_clock <= '1'; -- running system clock
9.       -- else halt system clock
10.      end if;
11.    end if;
12.  end process;
```

Listing 3-9. Prinzip der „gated clock“.

Datenblattes aufzudecken. Die Verifikation am Ende der Entwicklung des Modells beinhaltet alle funktionalen Test-Stimuli der Unterkomponenten und ihrer Verbindungen untereinander, die im Laufe der Entstehung des Modells zur Prüfung herangezogen wurden. Diese Verifikation ist am Besten vom Entwickler des Modells durchzuführen, da er die Details des Modells kennt. Der Test kann in Abhängigkeit von der zu modellierenden Komponente sehr komplex werden, da er jede gültige und ungültige Kombination von Eingangswerten für die Unterkomponenten enthält und, wenn es möglich ist, jeden Zustand anspricht.

Eine Gesamtverifikation des Modells kann durchgeführt werden, indem es in einen Board-Level-Entwurf eingesetzt wird, um mögliche Systemprobleme aufzudecken. Alle Testbenches sind im Idealfall von jemandem zu entwickeln, der nicht direkt in die Modellierung involviert war, damit mögliche Fehlerquellen nicht überdeckt werden. Auf dieser hohen Ebene ist es aus Komplexitätsgründen nicht möglich, alle denkbaren Zustände eines umfangreichen Modells zu verifizieren.

An dieser Stelle lassen sich die in Bild 3-5 gezeigten Verifikationsmethoden, funktionale und periphere Verifikation, für die Verifikation eines Modells unterscheiden.

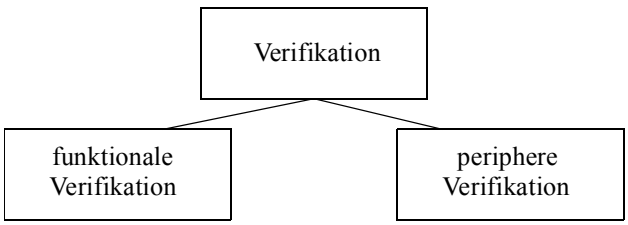


Bild 3-5. Verifikationsarten.

Mit Hilfe der funktionalen Verifikation wird die modellierte Funktionalität überprüft. Für den funktionalen Test ist es zunächst nicht notwendig, die Einhaltung von Setup- und Hold-Zeiten an den Eingängen der Komponente zu überprüfen. Darüber hinaus muss die funktionale Testbench Mechanismen bieten, Ergebnisse mit anderen Referenzquellen, z. B. eines Demonstrationsboards, automatisiert vergleichen zu können.

Die periphere Verifikation der Schnittstellen der Komponente ist getrennt von der funktionalen Verifikation durchzuführen. Hierbei ist oft eine automatische Verifikation nicht möglich, da eine visuelle Inspektion der Resultate notwendig ist. Sie erfolgt, damit die korrekte

Funktionsweise der Routinen zur Überprüfung des Zeitverhaltens nachgewiesen werden kann, jedoch nicht die korrekte Funktionsweise des Modells nach dem Auftreten einer Zeitverletzung (Timing Violation). Jeder Eingang der Komponente umfasst alle Treiberstärken des Typs std_logic, damit nicht zulässige Werte entdeckt werden können. Zur Verifikation gehören Input-to-Output Delay, Clock-to-Output Delay und Setup- und Hold-Zeiten für alle Simulationsrandbedingungen, die über Generic-Parameter des Modells selektierbar sind (BestCase, TypCase, WorstCase). Außerdem wird für alle Ausgangssignale die Korrektheit der Output Delays überprüft.

3.3.6 Testbench

Eine Testbench ist ein VHDL-Entity, welches Stimuli für die Eingangssignale der zu testenden Komponente erzeugt und die von der Komponente generierten Ausgangssignale auf Korrektheit überprüft. Stimmen die Ergebnisse nicht mit den erwarteten Ergebnissen überein, wird eine Fehlermeldung über Assertions ausgegeben. Die in VHDL implementierte Testbench ist simulatorunabhängig im Gegensatz zu mittels Skripten automatisierten Simulationsläufen. Bei einem Wechsel des Simulationsprogramms müssten die Test-Skripte komplett neu erstellt werden.

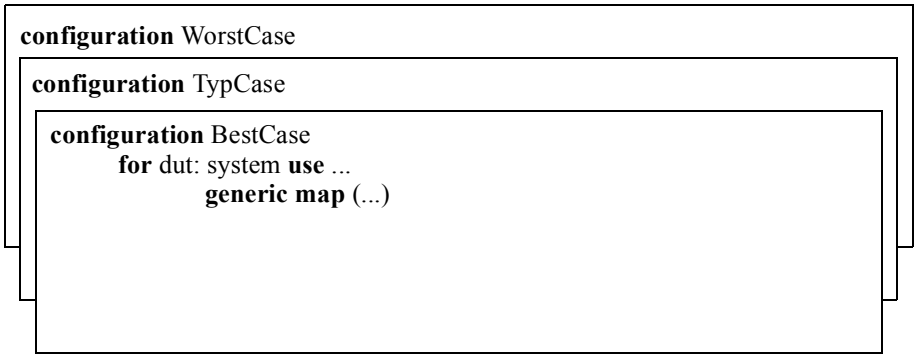


Bild 3-6. Verschiedene Konfigurationsdeklarationen für eine Testbench, die hinsichtlich ihrer Modellierung des Zeitverhaltens verschieden sind.

Die Testbench für den gesamten Board-Level-Entwurf ist das Top-Level Entity. Dieses enthält keine Ports und keine Generics, so dass die Entity-Deklaration leer bleibt. Die Auswahl verschiedener Simulationsrandbedingungen, denen der Board-Level-Entwurf unterworfen werden soll, geschieht über Konfigurationsdeklarationen, z. B. die Auswahl verschiedener Timing-Randbedingungen (BestCase, WorstCase, TypCase) wie in Bild 3-6 dargestellt.

Wenn Daten aus Dateien eingelesen und Ergebnisse in Dateien abgelegt werden können, dann steigt die Flexibilität einer Testbench. Um die Portabilität zu gewährleisten, sind keine Binärdateien zu verwenden. Ein Ansatz ist, Eingangsdaten für das DUT („Device under Test“) aus einer Datei einzulesen und die vom DUT erzeugten Ergebnisdaten in einer Ergebnisdatei zu speichern (z. B. beim Testen von Verarbeitungshardware für Audio und Video). In Bild 3-7 wird dies graphisch dargestellt.

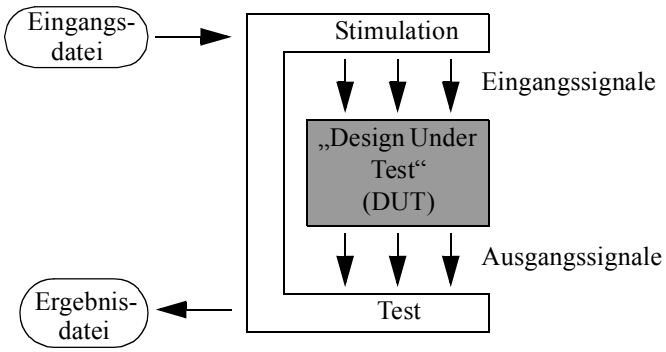


Bild 3-7. Testbench als Simulationsumgebung.

3.3.7 Die assert-Anweisung

Zur Überprüfung und Meldung von Fehlern bei der Verifikation des Modells bzw. beim Einsatz des Modells in einem Board-Level-Entwurf kann die assert-Anweisung in VHDL genutzt werden, die folgendermaßen definiert ist:

assert *Bedingung* report *Meldung* severity *Fehlerklasse*;

Die Anweisung ermöglicht bei einer falsifizierten Bedingung, eine Meldung auf der Konsole des VHDL-Simulators auszugeben und gegebenfalls die Simulation abzubrechen. Ein Kriterium für den Abbruch des Simulationslaufs stellt die Fehlerklasse dar, in die jede assert-Anweisung vom Entwickler einzuordnen ist. Die Fehlerklassen unterteilen sich in die Klassen Note, Warning, Error und Failure. Ab welcher Fehlerklasse der Simulationslauf abgebrochen wird, kann normalerweise im Simulator eingestellt werden. Standardmäßig ist es die Failure-Klasse. Es gibt ebenso die Möglichkeit, die Ausgabe der Meldung auf der Konsole für dedizierte Fehlerklassen zu unterbinden.

4 Kombination FPGA/DSP

Dieses Kapitel erläutert die Eigenschaften einer Kombination von FPGA und DSP. Es werden zunächst die Schnittstellen einer Kombination als Grundlage für die Interface-Synthese erörtert. Anschließend wird beschrieben, wie für einen möglichen Einsatz des FPGA als Co-Prozessor eine DSP-Funktionalität optimiert realisiert werden kann und welche Werkzeuge dafür bereits existieren.

4.1 FPGA/DSP-Schnittstellen

In Anlehnung an das ISO/OSI-Schichtenmodell der Netzwerktechnik, hier allerdings nicht auf die Kommunikation zwischen zwei Rechner bezogen, werden vom Autor für die Kommunikation zwischen FPGA und DSP drei Schichten definiert (Top-Down):

1. Anwendungsschicht,
2. Kommunikationsschicht,
3. Physikalische Schicht.

Die physikalische Schicht und die Kommunikationsschicht der FPGA/DSP-Kommunikation sind relevant für die Interface-Synthese (siehe Kap. 6.3). Es wird zwischen den folgenden zu erzeugenden Interfaces unterschieden, die auf der physikalischen Schicht und der Kommunikationsschicht einzuordnen sind:

- *Hardware-Interface:* Als Hardware-Interfaces werden synchrone und asynchrone Interfaces klassifiziert. Sender und Empfänger benutzen im synchronen Fall ein gemeinsames Taktsignal, im asynchronen Fall kommen Handshake-Verfahren zum Einsatz.
- *Software-Interface:* An der Durchführung einer Kommunikation können verschiedene Software-Schichten beteiligt sein. Auf den unteren Ebenen (low level) greifen Software-Funktionen direkt auf die verfügbaren Ressourcen der I/O-Schnittstellen eines Prozessors zu. Sie sind häufig als Interruptserviceroutine (ISR) implementiert. Kommt ein RTOS oder ein RTK zum Einsatz, können Treiber (device driver) ein Interface zur Verfügung stellen. Diese können prinzipiell zeitgesteuert oder ereignisgesteuert implementiert sein. Zeitgesteuert bedeutet, dass der Treiber periodisch eine Kommunikationsaufgabe ausführt (polling). Die ereignisgesteuerte Variante kann auf externe Ereignisse reagieren und mit einem Interrupt die Ausführung eines Treibers anstoßen.
- *Hardware-/Software-Interface:* Diese Interfaces verbinden Hardware- und Software-Blöcke. Der Prozessor kann über eigene I/O-Schnittstellen verfügen oder die Hardwareblöcke können im Speicherbereich eingeblendet (memory mapped) angebunden werden. Bei der Implementierung ist das Zeitverhalten der Schnittstellen zu beachten.

In den folgenden Ausführungen zu den einzelnen Schichten der Kommunikation zwischen FPGA und DSP werden für den DSP spezielle Merkmale der ADSP-2106x-Familie von

Analog Devices und für das FPGA die der Virtex-Familie von Xilinx verwendet. Eine Verallgemeinerung ist jedoch möglich.

4.1.1 Physikalische Schicht

In der physikalischen Schicht werden die Kommunikationsmedien zusammengefasst, die physisch im DSP vorhanden sind. Die des FPGA sind aufgrund der Programmierbarkeit praktisch nur durch die Verfügbarkeit von I/O-Pins, unterstützter I/O-Standards (z. B. LVTTL, GTL etc.) und benötigter Logikblöcke begrenzt. Grundvoraussetzung ist, dass die physikalischen Verbindungen elektrisch zusammenpassen. Darüber hinaus muss besonders auf die Einhaltung der Zeitvorgaben der Schnittstellen geachtet werden. Es existieren beispielhaft folgende physische Medien im DSP, die im Bild 4-1 zusammengefasst sind:

- *Systembus (asynchron/synchron):* Die synchrone Übertragung auf dem Systembus erfordert einen gemeinsamen Systemtakt für das FPGA und den DSP. Bei der asynchronen Datenübertragung über den Systembus gibt es speziell die Möglichkeit, nur mit externen Wartezyklen (im DSP programmierbar) oder mit einem Handshake-Verfahren zu arbeiten.
- *Link-Ports:* Die Link-Ports bieten in Multiprozessorumgebungen separate höher getaktete Busse zwischen zwei Prozessoren mit lediglich 4-Bit Datenbreite (Nibble) als Punkt-zu-Punkt-Verbindung. Alternativ ist anstelle eines DSP eben ein FPGA als Co-Prozessor in die Multiprozessorumgebung einzubinden und im FPGA sind lediglich Transmitter und Receiver für diese Schnittstelle zu implementieren (siehe auch [137]).
- *Serial-Ports:* Die serielle Verbindung der Serial-Ports kann für den Anschluss externer CODECs bzw. anderer Peripherie, z. B. eines FPGA, genutzt werden. Sie unterstützen eine variable Taktung (extern/intern) und konfigurierbare Formate für unterschiedliche serielle Datenströme in einer Punkt-zu-Punkt-Verbindung.

In der Generierungsphase der Interface-Synthese erfolgt eine Festlegung des von der Anwendung zu verwendenden physikalischen Mediums.

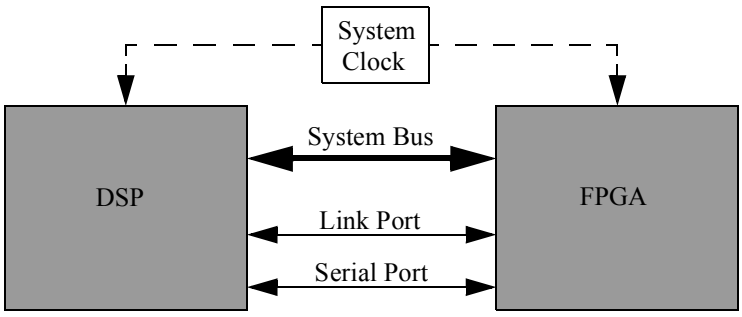


Bild 4-1. Beispiele für Kommunikationsmedien der physikalischen Schicht.

4.1.2 Kommunikationsschicht

Die Kommunikationsschicht enthält die Methoden, die für eine erfolgreiche Kommunikation zwischen FPGA und DSP notwendig sind. Diese sind als Software-Treiber für den DSP und als Hardware-Module (z. B. als VHDL-Entity) für das FPGA zu implementieren.

Bei jeder Art von Kommunikation werden von einem oder mehreren Sendern Daten erzeugt und zu einem oder mehreren Empfängern übertragen. Dabei wird generell zwischen der asynchronen und synchronen Kommunikation unterschieden:

- *asynchrone Kommunikation:* Bei einer asynchronen Kommunikation nehmen der Sender und der Empfänger nicht zeitgleich an der Kommunikation teil. Damit keine Daten verlorengehen, sind Zwischenpuffer mit zwei Ports notwendig, so dass der Zugriff weder den Sender noch den Empfänger blockiert. Für die Protokollierung der Gültigkeit der Daten sind zusätzliche Maßnahmen notwendig, z. B. Zeitstempel.
- *synchrone Kommunikation:* Eine Datenübertragung erfolgt bei der synchronen Kommunikation unter gleichzeitiger Teilnahme beider Kommunikationspartner. Dieses Verfahren ist blockierend, da Sender bzw. Empfänger warten müssen, bis der jeweils andere Kommunikationspartner bereit ist (Rendezvous).

Für die asynchrone Kommunikation können FIFOs als Zwischenspeicher verwendet werden. Dabei müssen zwei Sonderfälle beachtet werden: Schreibt ein Sender auf einen vollen FIFO, wird entweder der letzte Wert überschrieben oder der Sender wird blockiert. Liest ein Empfänger dagegen von einem leeren FIFO, so wird er blockiert, bis wieder Daten im FIFO vorhanden sind oder es gibt die Möglichkeit einer Abfrage im Voraus, wieviele Daten im FIFO enthalten sind, damit der blockierende Lesezugriff vermieden werden kann.

Eine weitere, häufig eingesetzte Kommunikationsart erfolgt über einen gemeinsamen Speicherbereich (shared memory) mit koordinierenden Semaphoren und gegenseitigem Ausschluss (mutual exclusion) zum Schutz gemeinsamer Ressourcen.

Unterschieden werden Kommunikationsprotokolle für folgende Systemkonstellationen der vorliegenden FPGA/DSP-Architektur (vgl. Bild 4-2):

a. Das FPGA arbeitet als Co-Prozessor in einem Multiprozessor-Szenario.
b. Das FPGA ist ein Co-Prozessor, eingeblendet in einen Speicherbereich.
c. Wie b, jedoch mit Handshake-Verfahren.
d. Zusammenfassung von Hardware („Glue Logic“) im FPGA.

Liegt die Systemkonstellation **a** vor, kommen effektiv nur die Kommunikationsmedien Systembus oder Link-Port in Frage. Die Einhaltung des Kommunikationsprotokolls beinhaltet z. B. die Arbitrierung auf dem Systembus oder die Kontrolle des Ablaufs eines DMA-Transfers (Systembus oder Link-Port). Auch der Zugriff auf einen gemeinsamen Speicher (shared memory), aufgrund des exklusiven Zugriffs auf den Systembus mit gegenseitigem Ausschluss, und die Synchronisation mittels Semaphoren kann Bestandteil des Kommunikationsprotokolls sein. Die Kommunikation auf dem Systembus kann sowohl

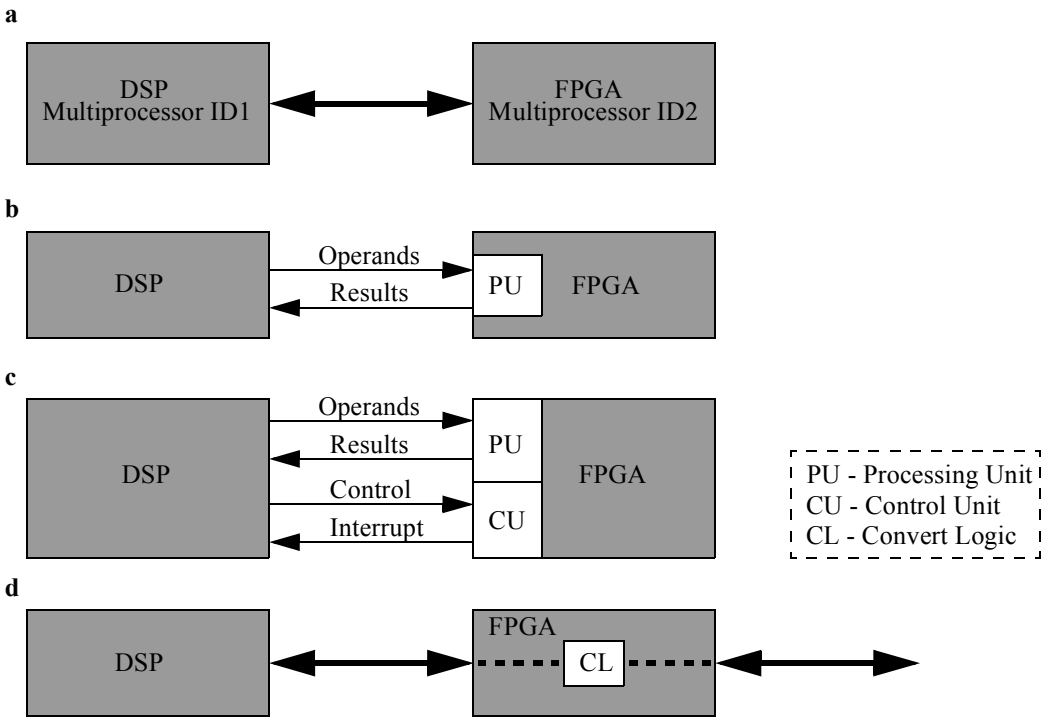


Bild 4-2. Verschiedene mögliche Systemkonstellationen bei der Kombination von DSP und FPGA.

asynchron als auch synchron erfolgen, sie ist also zunächst unabhängig vom verwendeten Protokoll der physikalischen Schicht. Das FPGA ist im Multiprozessor-Speichersegment des Systems eingeblendet (memory mapped), wenn die Kommunikation über den Systembus erfolgt.

Die Punkte **b** und **c** unterscheiden sich nur darin, welche Kontroll-Logik das FPGA für das Kommunikationsprotokoll als Co-Prozessor besitzt. In beiden Fällen initiiert der DSP die Kommunikation und das FPGA reagiert, indem es die ihm übertragene Aufgabe ausführt. Bei **b** gibt es keine Rückmeldung über die Erledigung eines Auftrags durch das FPGA, so dass der DSP dieses Ereignis aktiv überwachen bzw. abwarten muss. Aktiv überwachen heißt z. B., dass ein Statusregister im FPGA in regelmäßigen Abständen abgefragt werden muss (polling). Handelt es sich um eine ausgelagerte Systemfunktion (z. B. CORDIC), steht zumeist das Ergebnis der Berechnungsoperation nach einer festen Zahl von Taktzyklen zur Verfügung. Dieses Wissen kann zur Optimierung des Assembler-Codes für den DSP herangezogen werden. Durch ein Umordnen der Befehlsreihenfolge lassen sich parallel zur Ausführung im FPGA weitere unabhängige Operationen im DSP berechnen. Enthält die FPGA-Schaltung zusätzliche Kontroll-Logik für die ausgelagerte Funktionalität im Punkt **c**, also z. B. die Ansteuerung eines externen Interrupts für den DSP, muss die Bereitstellung der Ergebnisse nicht aktiv vom DSP überprüft werden. Im Assembler-Code des DSP ist eine Routine für die Bearbeitung des Interrupts vorzusehen, deren Aufgabe es ist, die Ergebnisse in den normalen Programmfluss einfließen zu lassen. Gemeinsam ist in den Punkten **b** und **c** die Einblendung des FPGA in das Speichersegment des DSP, der für externen Speicher vorgesehen ist, wenn der Systembus genutzt wird.

Bei der Systemvariation **d** wird das FPGA für das Zusammenfassen ansonsten zusätzlich notwendiger Hardware genutzt. Ein Beispiel ist die Umsetzung der S/PDIF-Signale vom

S/PDIF-Receiver des digitalen Audioeingangs in ein vom seriellen Port des DSP unterstütztes Format. Generell lassen sich mit Hilfe der programmierbaren FPGA-Hardware unterschiedliche Schnittstellen-Probleme lösen. Denn nicht alle möglichen Schnittstellen (PCI, USB etc.) können bereits im DSP monolithisch integriert sein, so dass das FPGA eine zeitgemäße Anpassbarkeit an variierende Schnittstellen-Standards ermöglicht. Von übergreifender Bedeutung sind z. B. Schnittstellen für die Ansteuerung von SDRAM/DDR-Speicher.

Anzumerken ist, dass für die Implementierung einzelner Kommunikationsprotokolle (**a-d**) sowohl im DSP als auch im FPGA Ressourcen notwendig sind, die dann nicht mehr für die eigentliche Applikation zur Verfügung stehen. Dies gilt umso mehr für das FPGA, da im DSP bereits grundlegende I/O-Schnittstellen integriert sind, die im FPGA erst eingerichtet werden müssen.

Die Einblendung des FPGA in ein Speichersegment des Gesamtsystems muss sowohl im Programm des DSP als auch in der FPGA-Schaltung konsistent sein. Dieses sicherzustellen, ist eine wichtige Aufgabe der Interface-Synthese.

4.1.3 Anwendungsschicht

In diesem Abschnitt werden einige Beispiele der Anwendungsschicht speziell für die Audiosignalverarbeitung vorgestellt, die unterschiedliche Kommunikationsprotokolle und Kommunikationsmedien der vorhergehenden Abschnitte nutzen.

Adaptives FIR-Filter (Bild 4-3). Ein FIR-Filter, der an sich bereits eine parallele Struktur besitzt, wird basierend auf Festkomma-Arithmetik in Hardware im FPGA realisiert. Die Adaption der Filterkoeffizienten an die Charakteristik der Audiosignale am Eingang erfolgt mit Fließkomma-Genauigkeit im DSP und die Übertragung der berechneten Koeffizienten erfolgt in regelmäßigen Abständen per DMA-Prozess über den Systembus.

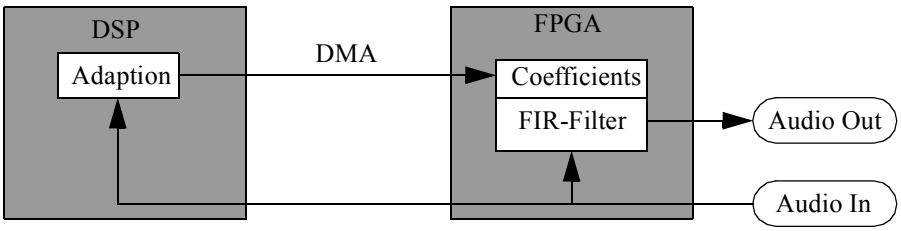


Bild 4-3. Anwendungsbeispiel „Adaptives FIR-Filter“.

Vor-/Nachbearbeitung durch das FPGA (Bild 4-4). Ist das System so ausgelegt, dass das Audiosignal vom analogen oder digitalen Eingang über das FPGA zum DSP und von dort wieder über das FPGA zum Ausgang zurückgeleitet wird, so kann das FPGA generell für eine Vor- und/oder Nachbearbeitung des Audiosignals genutzt werden. Ein konkretes Beispiel für die Nachbearbeitung ist ein im FPGA realisierter Equalizer.

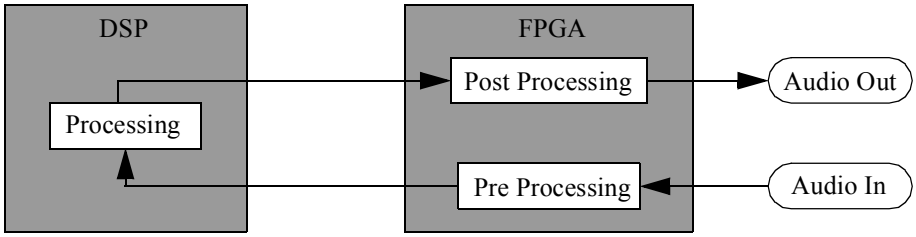


Bild 4-4. Mögliche Vor-/Nachbearbeitung durch das FPGA.

Verschlüsselte Übertragung komprimierter Audiodaten (Bild 4-5). Die Audio-Rohdaten werden mit Hilfe des DSP komprimiert und im FPGA erfolgt eine Verschlüsselung (z. B. Triple-DES) zur sicheren Übertragung auf einem Kommunikationskanal. Die Verschlüsselung ist in Hardware zu realisieren, da es sich hierbei um einen Algorithmus handelt, der eine parallele Struktur aufweist und in der Regel auf Ganzzahl-Arithmetik beschränkt ist. Die Komprimierung der Audiodaten ist für eine Realisierung im DSP geeignet. Mittels Profiling und iterativer Partitionierung ist aber zu klären, ob Teile davon zusätzlich in die Hardware ausgelagert werden müssen.

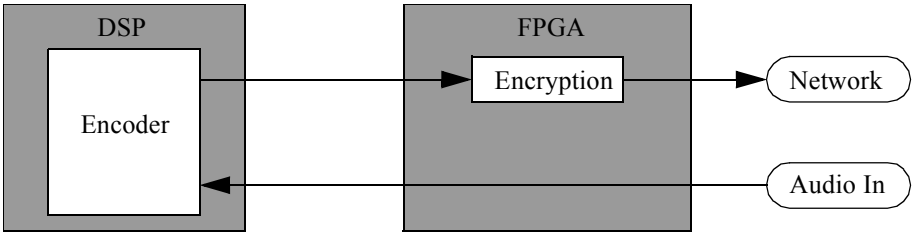


Bild 4-5. Komprimierung und Verschlüsselung von Audiodaten für ein „unsicheres“ Netzwerk.

Portables Abspielgerät für komprimierte Audiodaten (Bild 4-6). Ein portables Audiowiedergabegerät umfasst in aller Regel einen ladbaren Datenspeicher, eine Anzeigeeinheit und wesentliche Bedienelemente. Die komprimierten Audiodaten, die auf einem Speichermedium (Festplatte, Flash-Memory) abgelegt sind, werden mittels Software vom DSP dekomprimiert. Im FPGA implementiert sind neben dem Controller für das Speichermedium die Schaltung für die Ansteuerung eines LCD, die Auswertung der Bedienelemente, ein Equalizer und eine Regelung für die Lautstärke der auszugebenden dekomprimierten Audiodaten. Die Ausgabe auf dem LCD umfasst z. B. den Titel und die Spieldauer des aktuellen Audiostückes. Der Equalizer und die Regelung der Lautstärke kann auch innerhalb der Software des DSP realisiert sein.

Es ließen sich an dieser Stelle beliebig viele Anwendungsbeispiele anführen. Auch Anwendungen aus anderen Einsatzgebieten sind möglich, z. B. aus der Bild- oder der Videosignalverarbeitung.

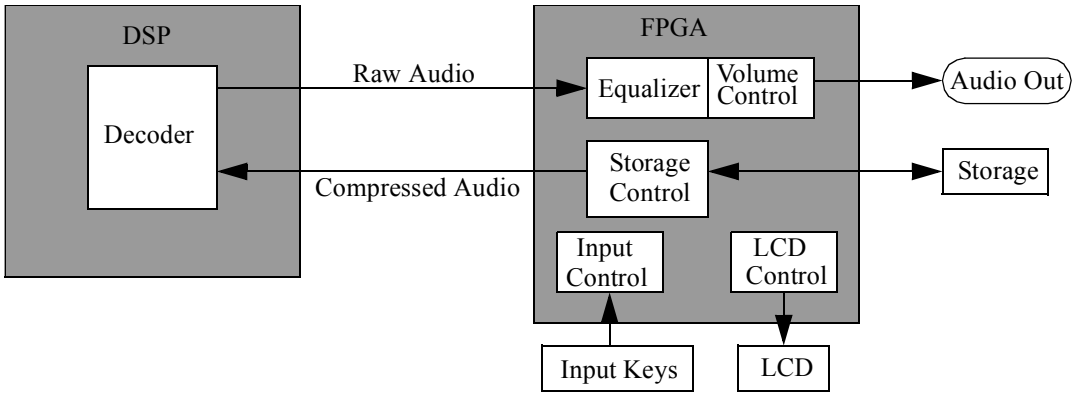


Bild 4-6. Portables Audiowiedergabegerät.

4.2 DSP-Funktionalität im FPGA

In diesem Abschnitt werden Möglichkeiten und Werkzeuge zur optimierenden Realisierung von DSP-Funktionalität im FPGA vorgestellt. Hardware-Techniken einer geeigneten Umsetzung (Distributed Arithmetic, Constant Coefficient Multiplier, CORDIC etc.) mit Erläuterungen des theoretischen Hintergrundes sind beispielsweise in [81] zu finden.

4.2.1 FIR-Filter

Am Beispiel eines FIR-Filters lässt sich am Besten der Vorteil einer Hardware- gegenüber einer Software-Implementierung zeigen, auch wenn die Realisierbarkeit im FPGA bereits seit ca. 1993 veröffentlicht ist [35]. Während in einem DSP die Anzahl der Ausführungseinheiten festgelegt ist, ist die Anzahl und der Typ der Ausführungseinheiten bei FPGAs zur Übersetzungszeit variabel. Dies erlaubt eine Auswahl zwischen Lösungen mit maximaler Parallelisierung einerseits oder minimalem Flächenaufwand andererseits. Die allgemeine Formel eines FIR-Filters lautet:

$$y(n) = \sum_{i=0}^{N-1} x(n-i) \cdot k_i \qquad (4.1)$$

Koeffizient, Summation, Multiplikation

Die zugehörige parallele Struktur des FIR-Filters ist in Bild 4-7 graphisch dargestellt.

Die Abbildung des Algorithmus für den FIR-Filter in einem GP-Prozessor bedingt die N-malige Ausführung einer Multiplikation mit anschließender Aufsummierung in einem Akkumulationsregister pro Ausgabewert eines Zeitschrittes. Der Faktor N ist abhängig von der Anzahl der Filterstufen des FIR-Filters. Eine Realisierung des FIR-Filters in einem DSP erfordert dagegen die N-malige Berechnung der MAC-Operation, die die Multiplikation und Addition in DSPs zusammenfasst, indem sich im Multiplikationspfad ein zusätzliches Akkumulationsregister befindet, das im Idealfall im selben Taktzyklus beschrieben wird. Dies stellt gegenüber der GP-Lösung bereits eine Leistungssteigerung dar, unter der ideali-

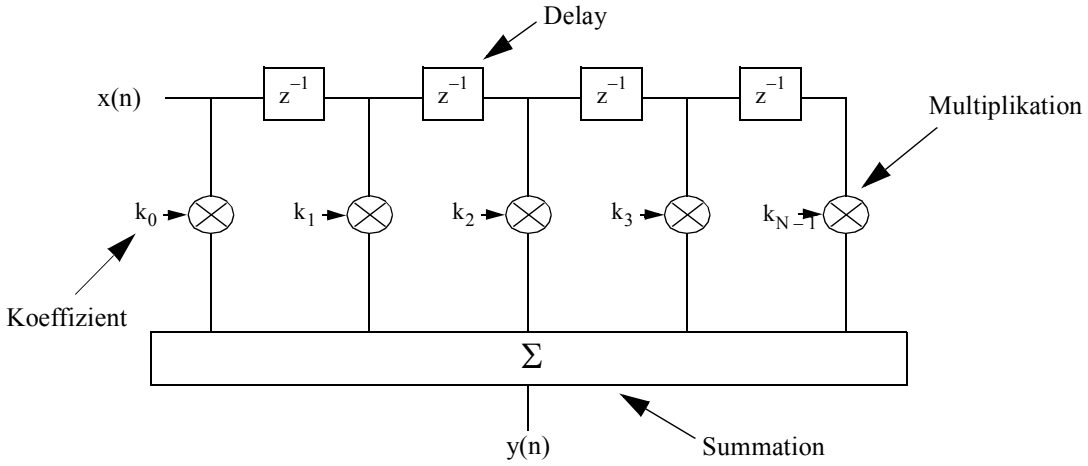


Bild 4-7. Parallele Struktur des FIR-Filters erkennbar im Blockdiagramm.

sierten jedoch nicht realitätsnahen Voraussetzung, dass die Taktfrequenzen und die sonstigen Architekturmerkmale (Registerzugriff, Speicheranbindung etc.) identisch sind. Wird die Schleife des Algorithmus bei einer Hardware-Realisierung parallel implementiert, so ist eine weitere Steigerung der Leistung zu beobachten. Ist die Taktfrequenz identisch mit derjenigen beim DSP, liegt die Steigerung bei einem Faktor gleich N. Zusätzlich notwendige Pipelinestufen in Kombination mit höheren/niedrigeren Taktfrequenzen können das Verhältnis allerdings verschieben. Eine vergleichende Gegenüberstellung der DSP- und Hardware-Realisierung des FIR-Filters ist schematisch in Bild 4-8 dargestellt.

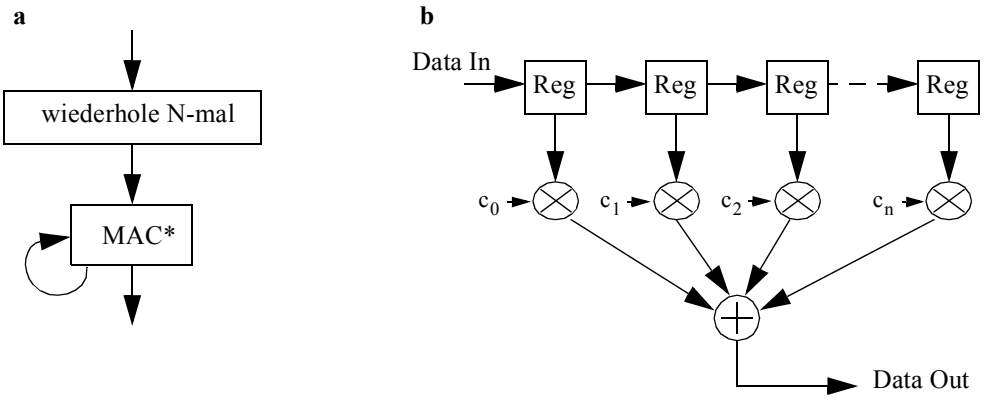


Bild 4-8. Mögliche Realisierungen des FIR-Filters im DSP oder FPGA, **a** sequentielle Berechnung in einem DSP, **b** parallele Berechnung im FPGA.

4.2.2 Fließkomma-Arithmetik

Im Vergleich zur Fließkomma- ist eine Festkomma-Realisierung bei gleicher Mantissenlänge kleiner und schneller, wirft jedoch Probleme bei der Entwicklung von Anwendungen hinsichtlich Skalierung und Rundung auf. Trotzdem sind sie für viele DSP-Applikationen ausreichend. Fließkomma-Arithmetik dagegen bietet einen, oftmals notwendigen, großen Dynamikbereich und umgeht die vorgenannten Probleme.

Tabelle 6 zeigt eine Übersicht wichtiger Unterscheidungsmerkmale zwischen Festkomma- und Fließkomma-Arithmetik, die im Bereich der digitalen Signalverarbeitung Anwendung finden. Die Dynamikbereiche in Tabelle 6 wurden mittels Formel (4.2) für die Festkomma- und mittels Formel (4.3) für die Fließkomma-Darstellung bestimmt. Fließkomma-Arithmetik lässt sich grundsätzlich im FPGA realisieren, aufgrund der schlechten Flächenausnutzung gibt es jedoch Einschränkungen hinsichtlich der Bitbreiten der Operanden und der Fließkomma-Operation. Diese können erst mit wesentlich größeren FPGAs überwunden werden. Aktuelle Veröffentlichungen zum Thema Fließkomma-Arithmetik, implementiert im FPGA, sind in [46, 67, 87] nachzulesen.

Tabelle 6: Vergleich von Fließkomma- und Festkomma-Arithmetik.

	Festkomma	Fließkomma IEEE-754 SP (Single Precision)
Genauigkeit	16 Bit: 64 K 24 Bit: 16 M 32 Bit: 4 G	24 Bit Mantisse: 16 M
Dynamikbereich	16 Bit: 96 dB 24 Bit: 144 dB 32 Bit: 192 dB	8 Bit Exponent: 1540 dB
Handhabbarkeit	umständlich (eine Fließkomma-muss oft erst in eine Festkomma-Darstellung überführt werden)	einfach (keine Rundungsprobleme, hohe Genauigkeit, großer Dynamikbereich)
Kosten	gering	hoch (große Chipfläche, geringere Taktraten)

$$D_{fxp} = 20 \cdot \log(2^{b_f}) = 6,02 \cdot b_f(\text{dB}) \text{ mit} \tag{4.2}$$

D_{fxp} Dynamikbereich einer Festkomma-Repräsentation,
b_f Anzahl der Bits (fraction).

$$D_{flp} = 20 \cdot \log\left(2^{2^{b_e}}\right) = 6,02 \cdot 2^{b_e}(\text{dB}) \text{ mit} \tag{4.3}$$

D_{flp} Dynamikbereich einer Fließkomma-Repräsentation,
b_e Anzahl der Bits im Exponenten.

4.2.3 Werkzeuge

Die Virtex-Familie von Xilinx beinhaltet spezielle Strukturen in den Logikzellen, mit denen schnelle und kompakte Addierer, Multiplizierer und flexible Speicherarchitekturen (FIFO, Shift-Register etc.) realisiert werden können. Diese Ressourcen bleiben ungenutzt, wenn sie nicht in der High-Level-Synthese geeignet berücksichtigt werden. Eine Lösung dieses Problems stellen der CoreGenerator und ebenso der SystemGenerator dar, die für die FPGA-Architektur optimierte Schaltungsteile generieren (z. B. die Implementierung von FIFOs im Block-SelectRAM der Virtex-Bausteine [133]). Die Virtex-II-Familie von Xilinx enthält darüber hinaus fest integrierte 18x18-Bit-Multiplizierer, deren Nutzung von den Entwicklungswerkzeugen unterstützt werden muss.

Ist die Zieltechnologie ein ASIC, dann ist bei einer Verwendung von spezifischen Strukturen der FPGA-Familien zu beachten, dass diese mit Hilfe der Bibliotheken des ASIC-Herstellers ersetzt bzw. nachgebildet werden müssen. Zum Beispiel ist dies mit den Block-SelectRAMs der Virtex-FPGAs von Xilinx nicht immer der Fall, z. B. aus Kostengründen oder aufgrund technologischer Einschränkungen beim ASIC-Hersteller [139].

Ein Beispiel für Werkzeuge auf Systemebene zur Generierung von DSP-Funktionen, optimiert für die Architektur der FPGA-Familien, ist der **SystemGenerator** von Xilinx, der im Jahre 2001 erstmals vorgestellt wurde [69]. Die darin integrierte Möglichkeit der graphischen Eingabe mittels Systemblöcken erleichtert dem Entwickler die Spezifikation und Simulation der Systemeigenschaften. Im Gegensatz zu anderen Tools auf Systemebene, z. B. Ptolemy, Signal Processing Worksystem (SPW) etc., werden hier die Besonderheiten der zugrunde liegenden Hardware, des Xilinx-FPGA, berücksichtigt. Es stellt eine Kombination aus graphischem Systemspezifikationswerkzeug und einem Generator von IP-Cores (der CoreGenerator von Xilinx wird genutzt) dar, das auf Simulink [105] aufsetzt. Simulink wiederum ist in Matlab [79] integriert. Matlab ist eine interpretierte Hochsprache zur Entwicklung von Algorithmen und zur Visualisierung von Daten. Es bietet eine Bibliothek mit Funktionen für mathematische Modellbildungen und Berechnungen, die der Anwender in komplexe Programme einbauen kann. Simulink ist eine auf Blockdiagrammen basierende Entwicklungsumgebung für die Modell-Entwicklung und Simulation. In Matlab fertig entwickelte und optimierte Komponenten können dann in Simulink integriert und als Bestandteil des Entwurfsprozesses auf der Systemebene im Zusammenspiel mit anderen Systemkomponenten simuliert werden. Die Fähigkeiten von Simulink werden durch den SystemGenerator dahingehend erweitert, dass im FPGA zu realisierende Schaltungen bit- und zyklengenau modelliert werden und dass Schaltungsteile ausgehend von einem Simulink-Modell zusammen mit synthetisierbarem VHDL und zugehöriger Testbench generiert werden können. Dafür beinhaltet das Werkzeug Simulink-Bibliotheken für arithmetische und logische Funktionsblöcke, Speicher und DSP-Funktionen (Xilinx Blockset). Der SystemGenerator ist im Gesamtpaket der XtremeDSP-Initiative von Xilinx [141] zusammen mit Matlab/Simulink und zugehöriger Synthese- und Place&Route-Werkzeuge als Systemlösung für FPGA-basierte DSP-Anwendungen erhältlich (Bild 4-9).

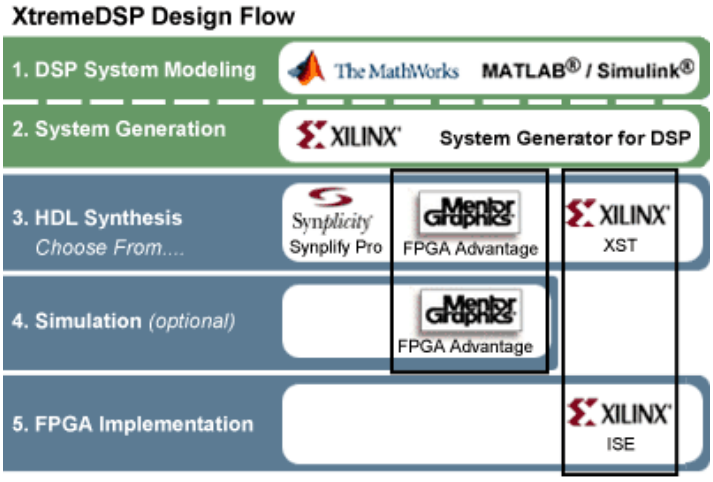


Bild 4-9. Spezifizierter Entwurfsfluss der XtremeDSP-Initiative von Xilinx. (Quelle: [107])

Ebenfalls im Jahre 2001 erschien eine Vorabversion des **DSPBuilder** genannten Werkzeugs von Altera für die DSP-orientierte Entwicklung auf Systemebene [82]. Als Entwicklungsumgebung wird, ebenso wie beim SystemGenerator, Matlab/Simulink verwendet. Der DSPBuilder stellt die Verbindung zwischen dieser Entwurfsumgebung und der Quartus-II-Software oder der MAX+PLUS-II-Software von Altera, jeweils Back-End-Werkzeuge für die Implementierung der FPGA-Schaltung, her. Zusätzliche bit- und zyklengenaue Simulink-Blöcke werden vom DSPBuilder hinzugefügt und stehen für eine Hardware-Realisierung von Teilen des spezifizierten Entwurfs zur Verfügung, so z. B. arithmetische Operationen und Speichereinheiten als Basisfunktionen, aber auch digitale Filter als komplexere DSP-Blöcke. Im Rahmen der MegaCore genannten IP-Bibliothek können IPs weiterer Hersteller über Simulink-Blöcke in den Entwurf eingebunden werden. Aus den Simulink-Modellen werden VHDL-Beschreibungen und Tcl-Skripte mittels Signal-Compiler erzeugt, die für die Back-End-Werkzeuge von Altera geeignet sind.

Die Ziele der vorgestellten Werkzeuge sind:

- Integrierte Verifikation auf Systemebene,
- Möglichkeit für den Systementwickler ohne HDL-Kenntnisse Hardware zu entwickeln,
- Integration in den HDL-Entwurfsfluss,
- Optimierung von Funktionsblöcken für die digitale Signalverarbeitung im FPGA.

Der Nachteil bei diesen Werkzeugen ist, dass bei großen, zeitkritischen Entwürfen die synthetisierte Schaltung nicht funktionieren kann und doch wieder HDL implementiert bzw. in den automatisierten Syntheseablauf eingegriffen werden muss.

5 Systemkonzepte von *FADES*

In diesem Kapitel werden die Konzepte von *FADES* vorgestellt und diskutiert, bevor im nächsten Kapitel die Implementierungsdetails erläutert werden. Nach einer anschließenden Einordnung von *FADES* in den Top-Down-Entwurfsfluss wird der Ablauf des Entwurfs innerhalb von *FADES* und die Konzepte der Interface-Synthese und der integrierten HW/SW-CoSimulation erörtert. Danach wird in diesem Kapitel der Hardware-Prototyp von *FADES* und abschließend die mit *FADES* vergleichbaren Entwicklungssysteme vorgestellt.

5.1 Einordnung

FADES ordnet sich auf den Ebenen nach der System-Repräsentation in den Top-Down-Entwurfsfluss ein. Es geht von einem bereits partitionierten Problem aus, d. h., die Frage der Partitionierung der Aufgabenstellung in einen Hardware- und einen Software-Teil ist bereits, zumindest vorläufig, entschieden. In *FADES* sind die Software-Synthese, die Hardware-Synthese, die Interface-Synthese und die HW/SW-CoVerifikation integriert (Bild 5-1). Die Hardware-, die Software-Synthese und die innerhalb der HW/SW-CoVerifikation durchgeführte HW/SW-CoSimulation wurden zum Teil mittels Einbindung von Fremdsoftware realisiert.

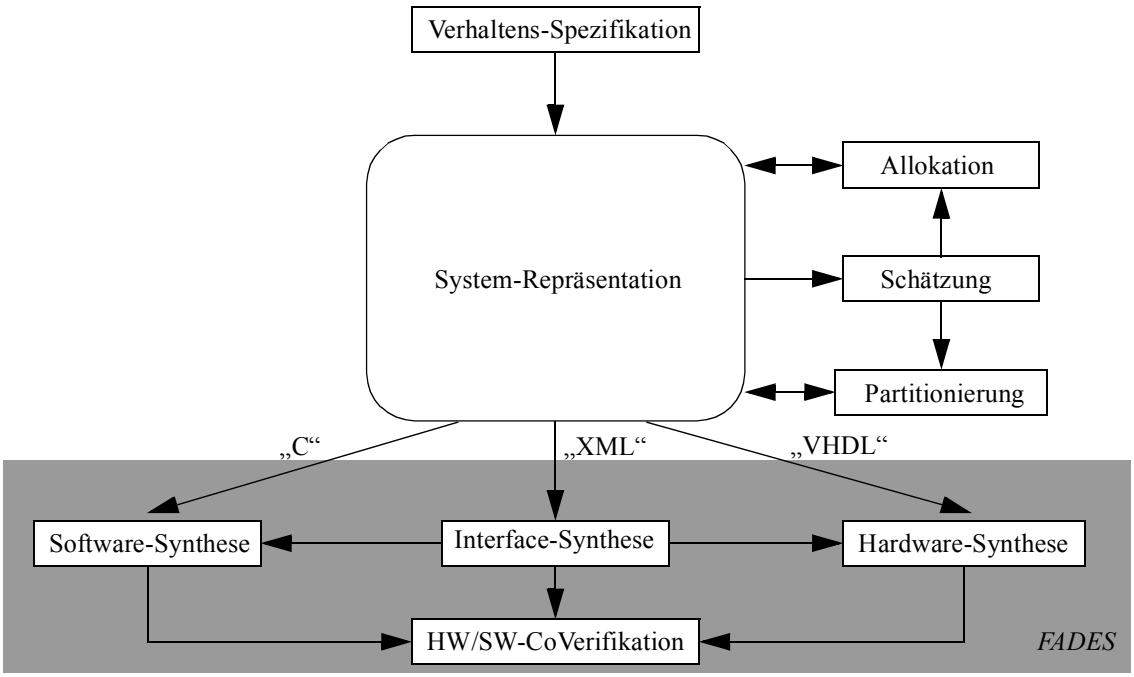


Bild 5-1. Einordnung von *FADES* in den Top-Down-Entwurfsfluss.

Die FPGA-Wahl fiel auf die Virtex-Familie von Xilinx, weil es am Institut bereits Erfahrungen mit Xilinx-FPGAs und die notwendigen Software-Werkzeuge gab. Zudem ließen die technischen Features, z. B. BlockRAM, Anzahl der Logikgatter etc., Freiheitsgrade bei der

Größe der zu implementierenden Schaltungen und bei der vergleichenden Analyse von verschiedenen Implementierungen offen. Die Möglichkeit der partiellen Rekonfigurierbarkeit der Virtex-Bausteine zur Laufzeit des Systems stellt eine Grundlage für weitere Forschungsaktivitäten dar. Xilinx positioniert darüber hinaus die Virtex-Familie innerhalb der XtremeDSP-Initiative [141] für die Realisierung von DSP-Funktionen und wurde deshalb auch aus diesem Grund als applikationsspezifischer Co-Prozessor für den DSP ausgewählt.

Die Wahl eines geeigneten DSP fiel auf die ADSP-2106x-Familie von Analog Devices, da diese Bausteine nicht nur Festkomma-, sondern auch Fließkomma-Arithmetik unterstützen und damit größere Freiheitsgrade bei der Implementierung von Algorithmen gewähren. In diesem Zusammenhang kann die notwendige Konvertierung einer Fließkomma- auf eine günstigere Festkomma-Realisierung eines Algorithmus (Kosten, Geschwindigkeit, Verlustleistung, Platzbedarf) eine wichtige Rolle spielen. Ein weiterer ausschlaggebender Vorteil war die Verfügbarkeit eines C-Compilers für die schnelle Realisierung von in C geschriebenen Anwendungsprogrammen. Das für die Implementierung des VHDL-Modells hilfreiche Evaluations-Board SHARC EZ-KIT LITE mit zugehörigem Simulator war ein weiterer Grund für die Auswahl der ADSP-2106x-Familie.

5.2 Entwurfsfluss

In Bild 5-2 ist der grobe Überblick über den Entwurfsfluss und in Bild 5-3 der detaillierte Entwurfsfluss dargestellt, der in *FADES* eingesetzt wird. Nach einer CoDesign-Phase wird zunächst die Interface-Synthese durchgeführt, die Schnittstellen zwischen Hardware und Software generiert. Ausgabe der Interface-Synthese ist VHDL-Code für das FPGA und Assembler-/C-Code für den DSP. Die restliche Verhaltensbeschreibung für die Schaltung im FPGA ist in VHDL zu implementieren. Sie kann auch mittels Fremdsoftware aus einer schematischen Eingabe automatisch generiert worden sein oder auch IP-Cores beinhalten, die z. B. vom Xilinx-Werkzeug CoreGenerator erzeugt werden. Mit Hilfe des VHDL-Compilers werden die vordefinierten VHDL-Modelle und der VHDL-Code für das FPGA in die Simulationsbibliothek übersetzt und können direkt für eine funktionale Simulation der Hardware im VHDL-Simulator genutzt werden.

Die Beschreibung des restlichen Programms für den DSP erfolgt in Assembler oder ANSI-C. Es bietet sich für die Implementierung von Algorithmen der digitalen Signalverarbeitung an, Werkzeuge auf der Systemebene, z. B. Matlab von The Mathworks oder Incisive-SPW der Firma Cadence [71] für die Spezifikation von Filtereigenschaften, für vordefinierte DSP-Blöcke etc., zu benutzen. Diese Werkzeuge integrieren oft die Möglichkeit, automatisch ANSI-C Code aus Modellbibliotheken zu generieren. Mit Hilfe der SHARC-Tools von Analog Devices, dazu zählen der C-Compiler (modifizierter GNU C-Compiler), der Assembler, der Linker und der Boot-Loader, wird die Software-Synthese durchgeführt. Die Ergebnisse der Software-Synthese werden aufbereitet und in die funktionale Simulation einbezogen, so dass diese zu einer HW/SW-CoSimulation erweitert wird. Dazu wird aus der resultierenden ausführbaren Datei für den DSP ein Speicherabbild für das VHDL-Modell des On-Chip-Speichers des DSP generiert und zu Beginn der Simulation in das Modell über Text-I/O eingelesen. Das vom Boot-Loader in der Software-Synthese erzeugte EPROM-

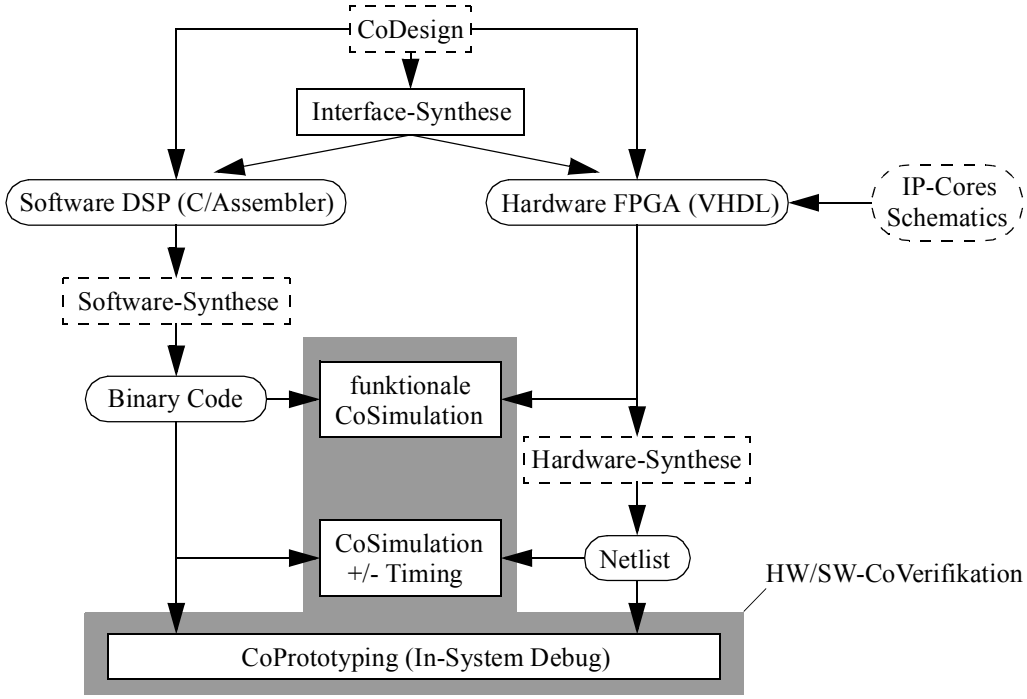


Bild 5-2. Übersicht über den Entwurfsprozess in *FADES*. Die gestrichelten Kästchen kennzeichen die Teile, die mittels Fremdsoftware realisiert sind.

Speicherabbild wird in ein eigenes Format für das VHDL-Modell des Flash-Bausteins, das den Boot-Code für den DSP speichert, konvertiert und ebenfalls beim Start der Simulation über Text-I/O eingelesen.

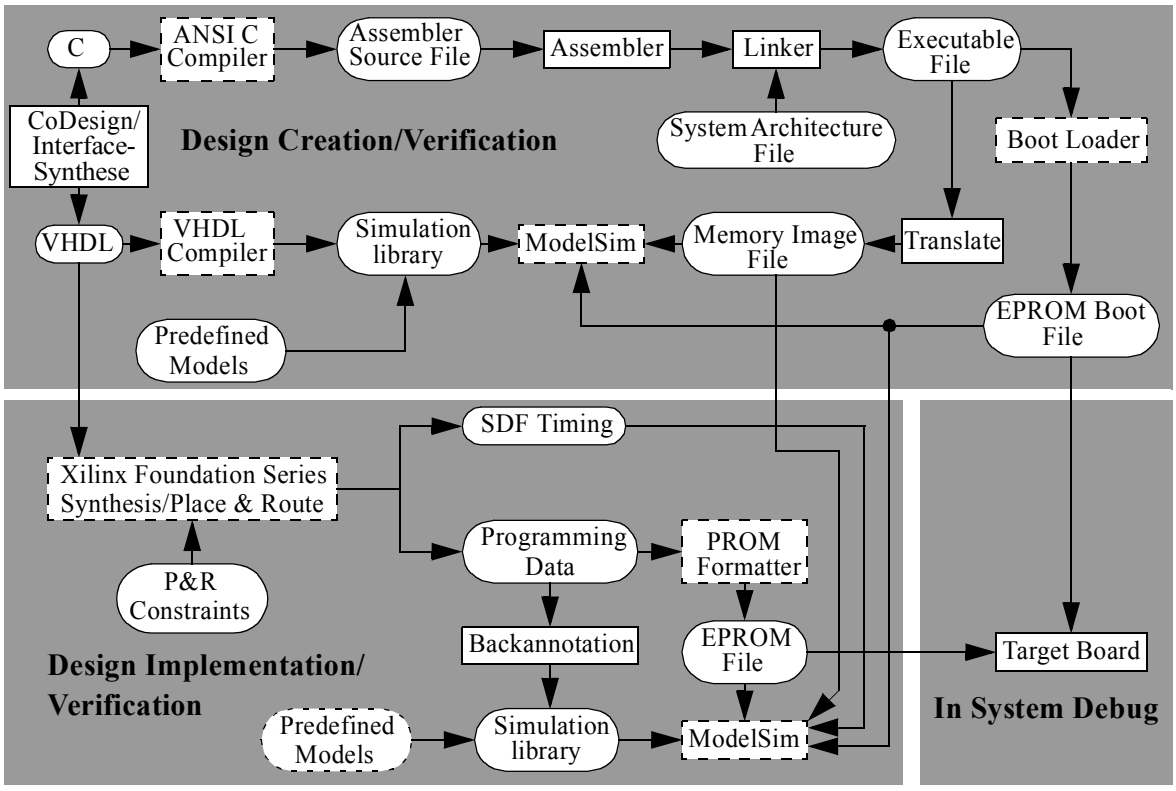


Bild 5-3. Vollständiger Entwurfsprozess in *FADES*. Die gestrichelt gezeichneten Kästchen kennzeichnen Fremdsoftware.

Ist die funktionale HW/SW-CoSimulation erfolgreich abgeschlossen, wird die Synthese für das FPGA mit Hilfe der Foundation-Software von Xilinx durchgeführt. Dazu gehört der FPGA-Compiler von Synopsys und die Place&Route-Werkzeuge von Xilinx. Die resultierende VHDL-Beschreibung (back-annotated VHDL) der platzierten und verdrahteten Schaltung wird wiederum in die Simulationsbibliothek übersetzt, wie auch die VHDL-Modelle, wenn diese nicht schon übersetzt wurden, so dass auch nach der Synthese eine Simulation mit oder ohne die ermittelten Verzögerungszeiten erfolgen kann. Die Verzögerungszeiten stehen nach der Platzierung und Verdrahtung der Schaltung im FPGA innerhalb einer SDF-Datei (Standard Delay Format) zur Verfügung. Bei SDF handelt es sich um einen Standard zur Spezifikation von Zeitparametern in einem definierten Dateiformat und wird häufig für Simulationen verwendet [102]. Auch nach der Hardware-Synthese werden die Daten der Software-Synthese in die Simulation einbezogen, so dass die Software-Simulation in die Hardware-Simulation zur HW/SW-CoSimulation integriert ist. Aus den Programmierdaten für das FPGA wird mit Hilfe des PROM-Formatters von Xilinx ein EPROM-Speicherabbild erzeugt, das für die Simulation der Bootphase des FPGA genutzt wird. Dafür wird das Speicherabbild in ein eigenes Format konvertiert und zu Beginn der Simulation über Text-I/O in das VHDL-Modell des Flash-Bausteins, das die Konfigurationsdaten für das FPGA speichert, eingelesen.

Für die HW/SW-CoSimulation auf den unterschiedlichen Entwurfsebenen kommt dieselbe Simulationsumgebung zum Einsatz, was einen durchgängigen Entwurf ermöglicht.

Bis hierher handelt es sich um einen „Virtual Prototyping“ genannten Entwurfsprozess. Die Hardware muss nicht existent sein. Sie kann parallel entwickelt werden oder auch momentan nicht verfügbar sein. Sind alle Simulationen erfolgreich abgeschlossen, erfolgt abschließend der Test der Anwendung bzw. des Algorithmus auf dem Hardware-Prototyp.

Der Entwurfsfluss verläuft sicherlich nicht immer linear. Sollten z. B. Fehler bei der Simulation nach der FPGA-Synthese auftreten, ist ein Re-Design der VHDL-Quelltexte notwendig, die eine erneute funktionale Simulation und FPGA-Synthese erforderlich werden lassen.

5.3 Interface-Synthese

Ein wichtiger Schritt beim HW/SW-CoDesign ist die Interface-Synthese. In der Regel werden Daten zwischen Hardware und Software ausgetauscht, die Variablen eines partitionierten Algorithmus sein können oder auch Kontroll- oder Statusinformationen repräsentieren. Die manuelle Implementierung der für den Transfer der Daten verantwortlichen Komponenten, sowohl software- als auch hardwareseitig, ist aufwendig und fehleranfällig. Spätere Änderungen ziehen sich durch den gesamten Entwurfsfluss, bei dem Inkonsistenzen zwischen Hardware und Software entstehen können, wenn z. B. die Abbildung der Daten im Adressraum (memory mapped) nicht übereinstimmt. Ziel ist es daher, die notwendige Synchronität bei der Adressierung und der Semantik der Daten an einer zentralen Stelle automatisch änderbar zu gestalten und aus einem Fundus an bereits getesteten Komponenten zur Kommunikation zwischen Hardware und Software schöpfen zu können.

Der Benutzer von *FADES* spezifiziert vom FPGA und DSP innerhalb eines partitionierten Problems zu übertragende Daten in einer eigens definierten XML-Notation. Aus dieser und aus vom Benutzer zu selektierenden Bibliothekselementen für Schnittstellen der physikalischen Schicht und zugehöriger Protokolle generiert die Software Module für den C- oder Assembler-Code des DSP und für die VHDL-Verhaltensbeschreibung des FPGA. Nach Einbindung zusätzlicher VHDL- bzw. C-/Assembler-Module durch den Benutzer wird im weiteren Entwurfsfluss die Software- und die Hardware-Synthese mit Hilfe der zugehörigen Fremdsoftware durchgeführt (Bild 5-4).

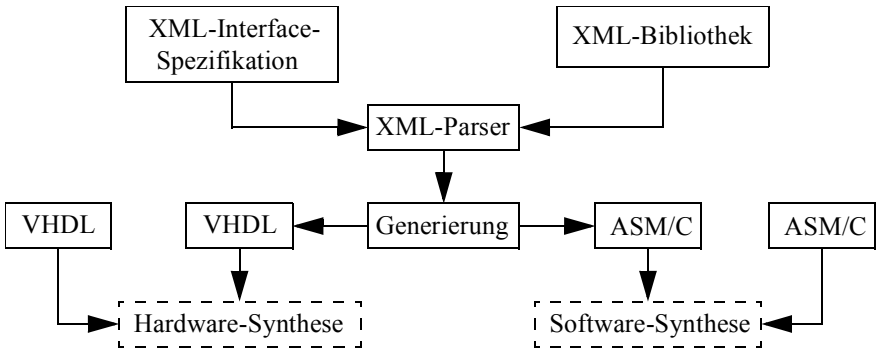


Bild 5-4. Interface-Synthese im HW/SW-CoDesign. Die gestrichelt gezeichneten Kästchen kennzeichnen Teile, die mittels Fremdsoftware realisiert sind.

Da in *FADES* keine HW/SW-Partitionierung auf Systemebene integriert ist, liegen die partitionierten Hardware- und Software-Teile nicht in einer internen Repräsentation vor (z. B. FSMs). Damit können diese nicht für eine direkte Interface-Synthese genutzt werden, die für eine schnelle Exploration verschiedener Interface-Lösungen auf Systemebene geeignet wäre. Aufgrund der begrenzten Zahl von Schnittstellen in der physikalischen Schicht kann darauf jedoch verzichtet werden. Der Benutzer von FADES kann explizit eine geeignete Schnittstelle der physikalischen Schicht und ein zugehöriges mögliches Protokoll der Kommunikationsschicht für die Übertragung der Daten zwischen Hardware und Software wählen und dafür die Software-Treiber und Hardware-Schaltungen mittels der Interface-Synthese generieren. Sollten Leistungsanforderungen aufgrund der gewählten Schnittstelle nicht erreicht werden können, ist die Interface-Synthese erneut durchzuführen.

5.4 Hardware/Software-CoSimulation

In FADES wurde eine homogene HW/SW-CoSimulation für die virtuelle HW/SW-CoVerifikation konzipiert. Dafür wird eine VHDL-Board-Level-Simulation eingesetzt, d. h., alle wesentlichen Komponenten des Zielsystems wurden in VHDL auf unterschiedlichen Abstraktionsebenen modelliert und in ein Simulationsmodell des Systems integriert. Bei der Modellierung der Komponente DSP wurde zusätzlich die Ausführung der Software berücksichtigt (Detailed-Behavioral Model), so dass die reine Hardware-Simulation zu einer homogenen HW/SW-CoSimulation erweitert wird. Die Software-Simulation wird zusammen mit der Hardware-Simulation innerhalb eines Simulatorkerns durchgeführt, was durch das Bild 5-5 verdeutlicht wird. Für die VHDL-Simulation wird der HDL-Simulator ModelSim eingesetzt, der zu den ereignisgesteuerten, native-compiled Simulatoren gehört.

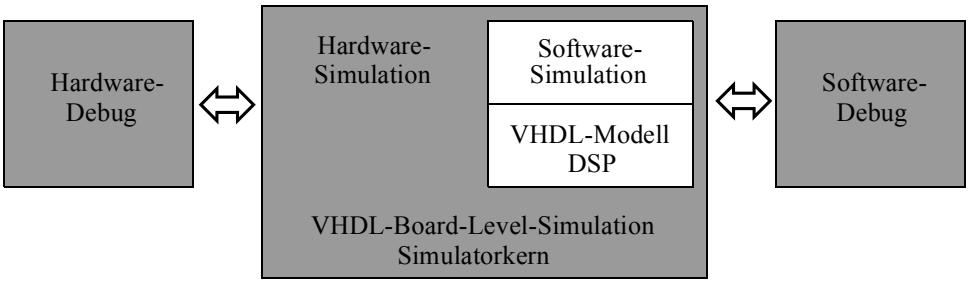


Bild 5-5. Homogene HW/SW-CoSimulation.

Der größte Kostenfaktor für die VHDL-Board-Level-Simulation stellt die Software-Simulation mittels VHDL-Modell für den verwendeten DSP dar. Eine andere Möglichkeit der Implementierung eines Verhaltensmodell für den DSP bestünde, wenn die Programmierschnittstelle VHDL PLI (VHPI) bereits standardisiert [118] und von den VHDL-Simulatoren unterstützt werden würde. Eine PLI (Programming Language Interface) ist prinzipiell dafür geeignet, in der Hochsprache C definierte Modelle in die VHDL-Simulation einzubetten. Bisher ist lediglich eine Verilog PLI standardisiert [112, 113]. Der Standard stellt sicher, dass die Modelle zwischen den Verilog-Simulatoren, die laut Datenblatt PLI unterstützen, austauschbar sind. Wenn jedoch für die FPGA-Synthese VHDL eingesetzt wird, ist der Einsatz des Modells für den DSP, das das Verilog PLI nutzt, später auf die gemischten VHDL/Verilog-Simulatoren eingeschränkt. Ob sich durch die Realisierung des Verhaltensmodells in C darüber hinaus eine Steigerung der Leistungseigenschaften der VHDL-Board-Level-Simulation ergeben würde, ist ebenso ungeklärt.

Zur Unterstützung des Software-Entwicklers bei der Nutzung des VHDL-Simulators bei der HW/SW-CoSimulation ist eine Debug-Oberfläche für die Software-Simulation notwendig. Diese muss neben der Visualisierung der Software-Ausführung auch die Möglichkeit eines Eingriffs zur Laufzeit bieten, z. B. eine Änderung von Variablen im Speicher. Graphische Benutzeroberflächen für das Debuggen der Hardware und der Software setzen beide auf den Kern des VHDL-Simulators auf (Bild 5-5).

Die plattformunabhängige Debug-Oberfläche wurde für die Tcl/Tk-Schnittstelle des HDL-Simulators ModelSim konzipiert und ist eine wichtige Voraussetzung für das Software-Debugging. Bei Tcl/Tk handelt es sich um eine Skript-Sprache, deren Grundlagen in [100] erläutert werden. Über spezielle when-Klauseln im Tcl/Tk-Quelltext kann auf Änderungen von Signalen im Simulationskern reagiert und entsprechende Aktionen ausgelöst werden. Darüber hinaus bietet Tcl/Tk eine Schnittstelle zum Windows-API des darunter liegenden Betriebssystems. Dadurch ist es möglich, aus den erhobenen Daten in der Simulation auf der einen und definierter Platzhalter in einem graphischen Front-End auf der anderen Seite Zustände des Modells im Simulationskern zu visualisieren.

Die Debug-Oberfläche bietet eine abstrakte Sicht auf die Simulation der Software, ohne im Detail das Modell für den Prozessor, hier den DSP, kennen zu müssen. Zudem wird die Kommunikationslücke zwischen dem Hardware- und dem Software-Entwickler verringert, da eine einheitliche Verifikationsumgebung zur Verfügung steht. Andere VHDL-Simulatoren bieten ebenfalls eine Tcl/Tk-Schnittstelle, z. B. NC-Sim von Cadence, Active-HDL und Riviera von Aldec oder VHDL-Studio von Green Mountain Computing Systems, Inc. und

weitere. Im Rahmen dieser Arbeit wurde jedoch nicht geprüft, ob die Tcl/Tk-Quelltexte auf andere VHDL-Simulatoren übertragbar bzw. inwieweit sie portierbar sind. Unter Umständen müssen andere Lösungen gesucht werden: Eine Programmierschnittstelle (PLI) beispielsweise erlaubt extern eingebundendem C-Code, auf interne Zustände und Funktionen des Simulators lesend und schreibend zuzugreifen. Damit wäre ein Austausch von Daten und Kontrollanweisungen zur Laufzeit der Simulation mit einem parallel laufenden Prozess, z. B. einer Debug-Oberfläche, möglich. Der Simulationsablauf und die Visualisierung der Daten würden von der Debug-Oberfläche gesteuert. Es gibt aber bisher keinen einheitlichen Standard bei VHDL PLI. Das im VHDL-Simulator ModelSim genutzte FLI (Foreign Language Interface) ist inkompatibel zum VHPI, welches vom VHDL-Simulator Riviera von Aldec eingesetzt wird. Darauf basierende graphische Erweiterungen würden somit vom eingesetzten VHDL-Simulator abhängig sein.

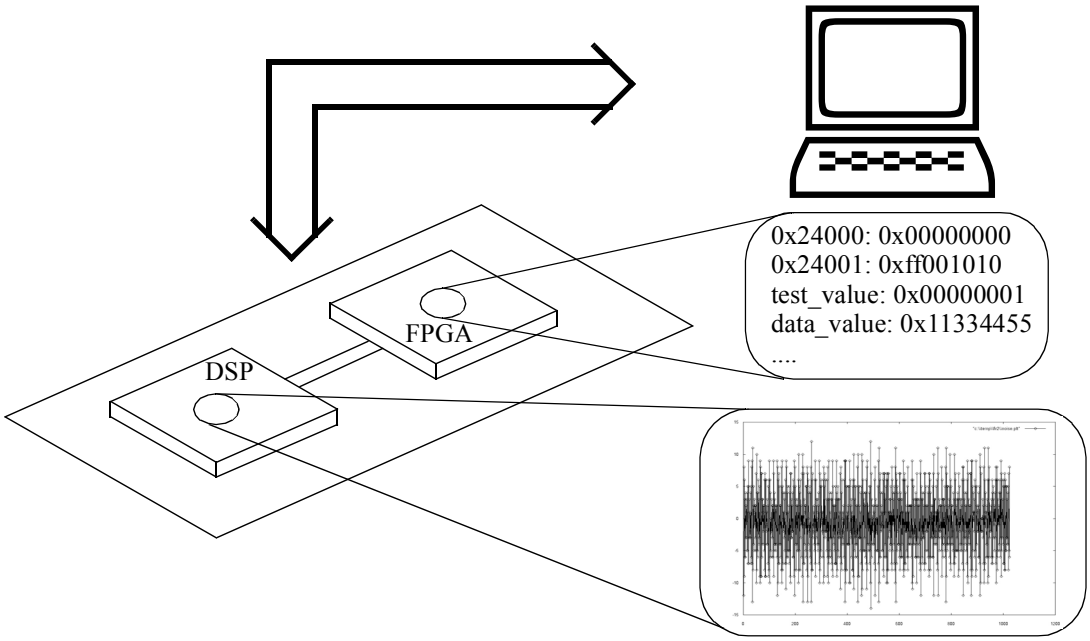


Bild 5-6. Debug-Möglichkeiten ausgehend von der Software des Host-Rechners für den DSP und das FPGA.

5.5 Hardware-Prototyp

Der abschließende Echtzeittest einer entwickelten Anwendung (eines entwickelten eingebetteten Systems) erfolgt auf einer Prototyp-Platine, die eine HW/SW-In-Circuit-CoVerifikation bietet. Alle Komponenten des Zielsystems sind darin in diskreten Ausführungen enthalten. Die Spezialisierung auf die digitale Audiosignalverarbeitung im Hardware-Prototyp des Entwicklungssystems erfolgt durch die Integration digitaler Audioein-/ausgänge. Für das Debuggen einer Anwendung steht ein lesender/schreibender Zugriff zur Laufzeit des Systems vom Host-Rechner mittels Software auf die systeminternen Bestandteile des Hardware-Prototyps zur Verfügung. Zu diesen Bestandteilen gehören sowohl Speicherstellen im internen Speicher des DSP als auch implementierte Register in der FPGA-Schaltung, wenn ein entsprechendes Debug-Modul im FPGA eingebettet ist. Auch eine graphische Darstellung der Daten in einem 2-D-Plot wird unterstützt. Diese Sachverhalte werden in Bild 5-6 graphisch dargestellt.

Der Nebeneffekt der Prototyp-Hardware ist, dass diese die korrekte Funktionsweise der HW/SW-CoSimulation nachweist, zum einen, indem die entwickelte Anwendung auch in Echtzeit in Hardware arbeitet und zum anderen in einer detaillierten Form, z. B. mittels Binärvergleich der produzierten digitalen Audiodatenströme oder mittels Vergleich von Variablenwerten im Speicher des Hardware-Systems und des in der VHDL-Board-Level-Simulation simulierten Speichers.

5.6 Vergleichbare Entwicklungssysteme

Die SignalMaster-Reihe der Firma LYR Signal Processing (LSP) [78] umfasst unterschiedlich bestückte Entwicklungsboards mit Kombinationen von DSPs und FPGAs. Mit dieser Produktreihe wird primär die schnelle Entwicklung von DSP-Applikationen mit optionaler Zusammenfassung der „Glue Logic“ in zusätzlich vorhandener rekonfigurierbarer Hardware in Form von FPGAs unterstützt. Erst in neueren Varianten wurden größere FPGA-Bausteine auf den Entwicklungsplatinen eingesetzt, die darauffolgend einen Einsatz der FPGAs als Co-Prozessoren für den oder die DSPs erlaubten. Es wird angeführt, dass sich die Funktionen der digitalen Signalverarbeitung in zwei Klassen unterteilen: datenintensiven- auf der einen und kontroll-/rechenintensiven Funktionen auf der anderen Seite. Zur letzteren Klasse gehören die adaptive Filterung, Kompressionsalgorithmen und Verschlüsselung und zur erstgenannten Klasse FIR-, IIR-Filter und Sample-Rate-Anpassungen.

Die direkt mit dem in dieser Arbeit vorgestellten FPGA/DSP-Entwicklungssystem vergleichbaren Systeme der SignalMaster-Reihe der Firma LSP mit kurzer Auflistung der wesentlichen Systembestandteile sind:

- *TI c67x-based SignalMaster:* Texas Instruments C67x-DSP (166 MHz), Virtex-FPGA von Xilinx (XCV300-XCV800), CS4228 CODEC von Crystal (96 kHz, 24 Bit), 16 MByte SDRAM, cPCI.
- *ADSP-21160-based SignalMaster:* Xilinx-FPGA XCV300-XCV800; 2 x ADSP-21160 von Analog Devices, CS4228 CODEC von Crystal (96 kHz, 24 Bit), 16 MByte SDRAM, PCI Interface.
- *SHARC-based SignalMaster:* Analog Devices ADSP-21062, National NS486SXL Mikrocontroller, XC4010E oder XC4028E von Xilinx, CS4218 CODEC von Crystal (7-48 kHz, 16 Bit), RS232, Ethernet.

Es handelt sich jeweils um Standalone-Boards mit verschiedenen Schnittstellen (PC/104, cPCI, Analog Stereo In/Out etc.). Externe Zusatzmodule, z. B. für ein AES/EBU-Interface (digitale Audioschnittstelle für den professionellen Bereich) oder ein Video-I/O-Modul, sind ebenso erhältlich.

Für die Entwicklung wird Matlab in Kombination mit Simulink, dem Real-Time Workshop und dem SystemGenerator von Xilinx verwendet, für die die Firma LSP ein Software Add-on mit eigenen, für die Entwicklungsboards notwendigen, Bibliothekselementen mitliefert. Damit kann Software-Code für den oder die DSPs der Systeme aus einer Beschreibung auf Systemebene erzeugt werden. VisualDSP++ von Analog Devices und das Code Composer

Studio C6000 von Texas Instruments werden z. B. für die Übersetzung der generierten Software-Quelltexte verwendet.

Der Real-Time Workshop erzeugt von einem Zielsystem unabhängigen Code aus dem Simulink-Modell, kann aber über Module z. B. für ein „Rapid Prototyping“-System angepasst werden. Dadurch wird eine ausführbare Spezifikation aus der Modellbeschreibung abgeleitet. Außerdem werden optimierte, portable und individuell anpassbare ANSI-C Quelltexte aus Simulink-Modellen generiert. Die erzeugten Programme können in Echtzeit ausgeführt oder als ausführbare Spezifikation in einer Simulation dienen. Für die Entwicklung optimierten Codes für den DSP gibt es eine Bibliothek von Assemblerfunktionen.

Bis zur Integration des SystemGenerators in die Software-Pakete gab es weder die Möglichkeit VHDL-Code in die Simulation auf Systemebene einzubeziehen noch VHDL-Code mittels vordefinierter IP-Blöcke zu generieren. Für die Einbeziehung in die Simulation mussten Black-Boxes in Simulink definiert werden, die das I/O-Verhalten der FPGA-Schaltungsteile modellierten. Der VHDL-Code wurde mit externen Werkzeugen implementiert, optimiert und kompiliert.

Zur Sharc-basierenden SignalMaster-Reihe gehören als „low level“-Software die Werkzeuge von Xilinx für die Hardware-Entwicklung und VisualDSP++ von Analog Devices für die Software-Entwicklung. Für die Systemebene sind Werkzeuge zur HW/SW-CoSimulation vorhanden. Auf den Ebenen unterhalb der Systemebene, bevor der Algorithmus in Echtzeit auf der Entwicklungsplatine ausgeführt und getestet wird, arbeiten die Werkzeuge (VisualDSP++ und Xilinx-Tools) getrennt voneinander. Eine gemeinsame HW/SW-CoSimulation kann lediglich manuell eingerichtet und durchgeführt werden.

In Simulink lässt sich die Simulation des Gesamtmodells mit Teilen des Modells in einem VHDL-Simulator heterogen co-simulieren. So können in VHDL spezifizierte Schaltungsteile für das FPGA in einer Black-Box innerhalb des Gesamtmodells simuliert werden. Die Schnittstelle zum VHDL-Simulator wird hierbei mit Hilfe eines als „Gateway“ bezeichneten Simulink-Blocks hergestellt. Der Nachteil ist, dass keine Timing-Informationen der FPGA-Schaltung auf den unteren Abstraktionsebenen in die heterogene HW/SW-CoSimulation auf Systemebene einbezogen werden können.

Eine gemischte Simulation ist, je nachdem welche Komponenten auf den Boards verfügbar sind, möglich. So lassen sich die Modelle in Simulink simulieren und parallel dazu die Hardware im FPGA-Baustein testen. Umgekehrt kann der Code im DSP in Kombination mit der Simulation der FPGA-Schaltung ausgeführt werden.

6 Implementierung von *FADES*

In diesem Kapitel werden wesentliche Implementierungsdetails von *FADES* vorgestellt. Zunächst wird die Hardware *Faust* und danach die Software *Mephisto* des Systems erläutert. Zwei wesentliche Bestandteile im Entwurfsfluss von *FADES* sind die Interface-Synthese und die HW/SW-CoSimulation, die beide näher vorgestellt werden. Anschließend wird detaillierter auf die Implementierung und die Verifikation der Schlüsselkomponente DSP für die HW/SW-CoSimulation eingegangen.

6.1 Die Hardware *Faust*

Die Hardware *Faust* setzt sich zusammen aus einem FPGA der Virtex-Baureihe von Xilinx (XCV300-4) und einem DSP der ADSP-2106x-Familie von Analog Devices (ADSP-21061), beides Zielarchitekturen des Entwicklungssystems. Darüber hinaus wird ein weiteres FPGA von Xilinx für die Aufgaben eines Controllers im System verwendet, so dass die Logikressourcen des eigentlichen Ziel-FPGAs in vollem Umfang für die zu entwickelnde Anwendung zur Verfügung stehen. Der Aufbau des Hardwaresystems ist in Bild 6-1 graphisch dargestellt.

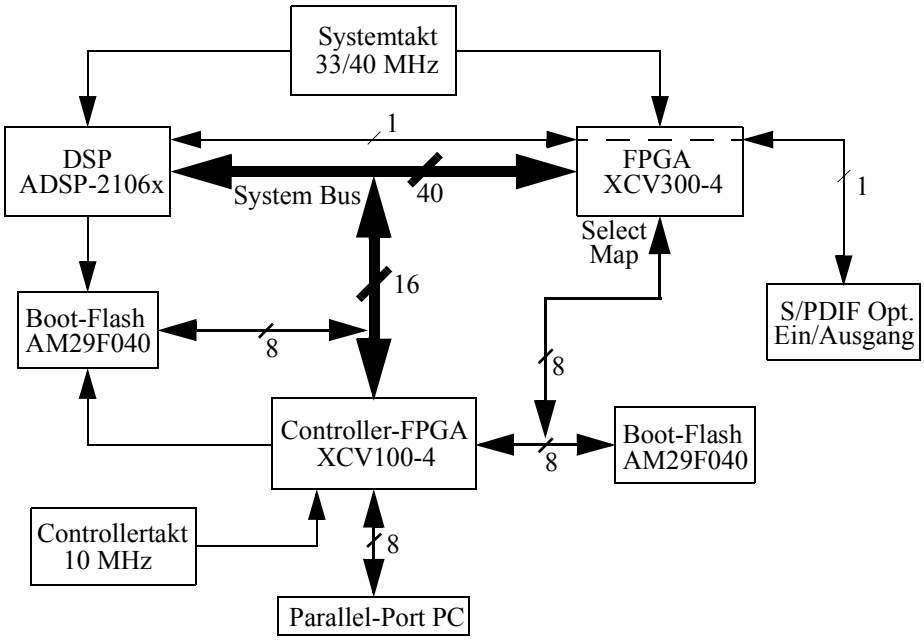


Bild 6-1. Blockschaltbild der Hardware *Faust*.

Das Controller-FPGA vermittelt zwischen dem Parallel-Port des Host-PC (EPP - Enhanced Parallel Port [28, 51]) einerseits und dem 16-Bit breiten Hostbus, der ein Teil des 40-Bit breiten Systembusses ist, andererseits. Zur dauerhaften Speicherung der Konfigurationsdaten einer entwickelten Anwendung stehen dem DSP und dem FPGA je ein Flash-Baustein von AMD mit 4 MBit Speicherkapazität (512k x 8bit) zur Verfügung. Mit Hilfe des Con-

troller-FPGA können vom Host-PC die beiden Flash-Bausteine mit neuen Daten programmiert werden. Weiterhin wird über ihn der Bootvorgang des FPGA gesteuert, da das FPGA nicht von sich aus über den Konfigurationsport (SelectMap) seine Konfigurationsdaten laden kann. Das Controller-FPGA generiert die Adresse und die Steuersignale für den Flash-Baustein und dessen Daten werden synchron zu einem Taktsignal mit maximal 5 MHz nach jedem Systemstart oder System-Reset in das FPGA geladen. Der DSP ist per Konfigurationspins so eingestellt, dass er sein Programm beim Systemstart automatisch aus dem Boot-Flash einliest. DSP und FPGA booten parallel, weshalb besondere Synchronisationsmechanismen notwendig sind, da die Bootphasen unterschiedlich lange dauern (in der Regel dauert die Bootphase des FPGA länger). Auch diese Aufgabe übernimmt das Controller-FPGA, das für das FPGA die Reset-Leitung (die Reset-Leitung der implementierten Schaltung und nicht die des SelectMap-Ports) und für den DSP eine Interrupt-Leitung ansteuert, von denen die ADSP-2106x-Familie insgesamt drei besitzt. Der Zeitpunkt, zu dem die Bootphase des FPGA abgeschlossen ist, ist dem Controller-FPGA bekannt, da dieses selbst die Adresse für das Boot-Flash erzeugt. Beim DSP generiert die startende Software, der RTK, ein externes periodisches Timer-Signal, das das Controller-FPGA auswertet. Sind beide Bootphasen abgeschlossen, wird ein Interrupt für den DSP aktiviert und das Reset-Signal für die Schaltung im FPGA deaktiviert. Eine Interruptserviceroutine (ISR) im DSP verzweigt im Programm zur Funktion main() und startet damit den Softwarealgorithmus. Das deaktivierte Reset-Signal aktiviert die Schaltung im FPGA. Der erläuterte Synchronisationsmechanismus wird in Bild 6-2 graphisch dargestellt.

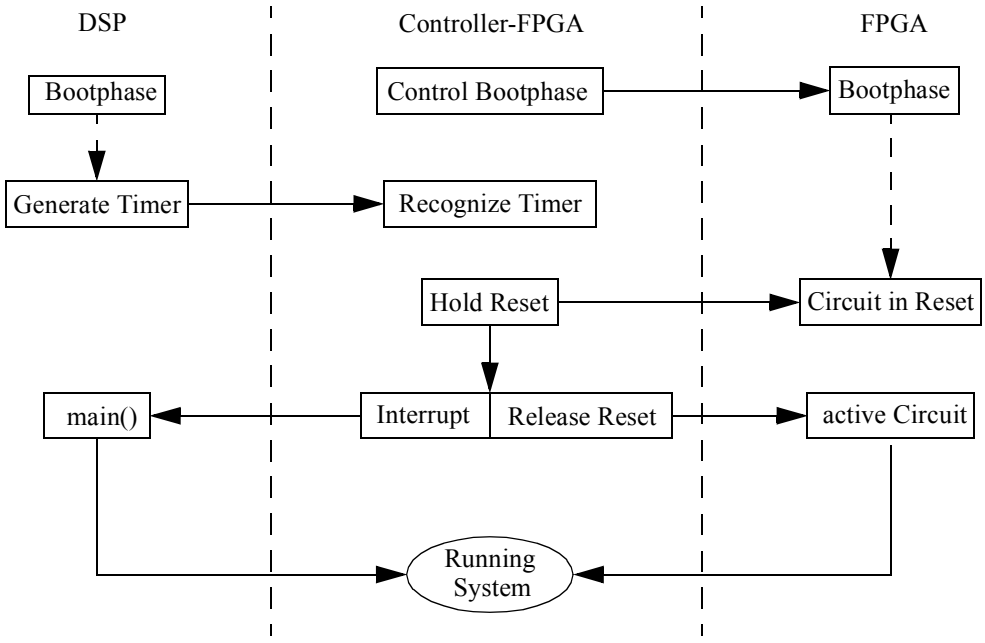


Bild 6-2. Synchronisationsschemata beim System(neu)start.

Das Controller-FPGA kann auf beide Targets als Host sowohl lesend als auch schreibend zugreifen. Der Zugriff vom Host-PC über den Parallel-Port zum SelectMap-Port des FPGA geht nicht über den gemeinsamen Systembus. Damit wird eine dynamische (partielle) Rekonfiguration des FPGA möglich, ohne die Kommunikation auf dem Systembus unterbrechen zu müssen!

Als zusätzliche externe Schnittstelle wurde ein optischer S/PDIF-Ein/Ausgang implementiert, der es ermöglicht, audiosignalverarbeitende Algorithmen zu entwickeln. S/PDIF ist ein von Sony und Philips etablierter Standard, der zwei Übertragungsmedien unterstützt: elektrisch (auch als koaxial bekannt) und optisch. S/PDIF benutzt gewöhnlich ein 16-Bit Datenwort bei einer Sample-Frequenz von 44,1 kHz. Über lediglich eine physische Verbindung werden beide Kanäle des Stereo-Audiosignals übertragen.

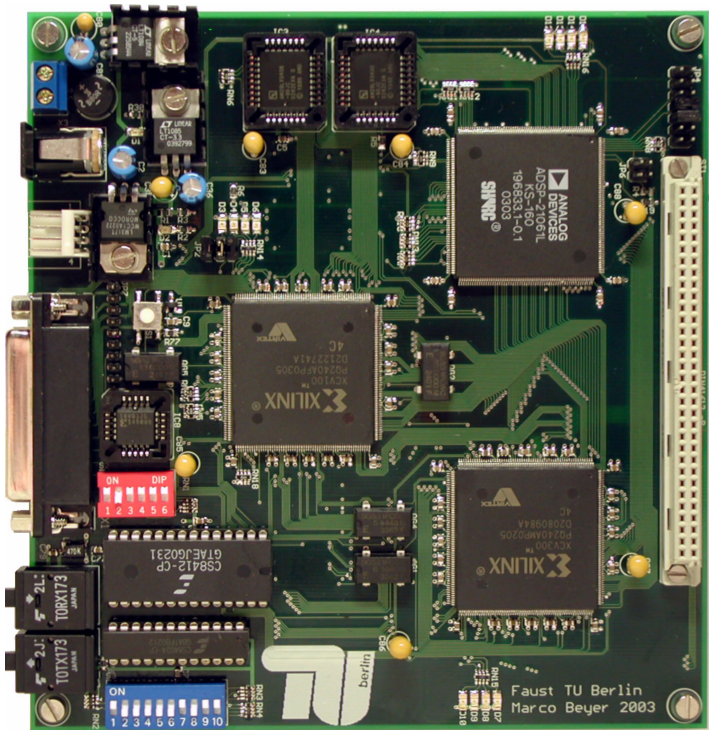


Bild 6-3. Prototyp-Hardware (zweite Revision).

Weiterhin ist ein Steckverbinder für zusätzliche Erweiterungskarten vorgesehen, der für die Anbindung spezieller Verbindungssysteme oder für Buttons, 7-Segment-Anzeigen, LC-Displays etc. genutzt werden kann. Einige I/O-Pins des FPGA und des Controller-FPGA sind mit diesem verbunden, außerdem zwei der im System vorhandenen Versorgungsspannungen (+5V, +3,3V) und einige Masseleitungen (siehe Anhang A.2.2).

In einer zweiten Revision der Prototyp-Hardware ist SDRAM-Speicher (drei ICs organisiert in 8M x 16 = 16 MByte) vorgesehen, der als gemeinsamer Speicher (Shared-Memory) für den DSP und das FPGA genutzt werden kann. Er ist auch für die Erweiterung des begrenzten Programm- und Datenspeichers des DSP konzipiert. Damit die gesamte Datenbreite des Programmspeicherbus bedient werden kann, sind die 16-Bit breiten Datenbusse der drei Speicher-ICs parallel geschaltet. Der notwendige SDRAM-Controller ist im Controller-FPGA realisiert. Bild 6-3 zeigt ein Foto der Prototyp-Platine in der zweiten Revision.

6.1.1 Das Controller-FPGA

Die Aufgaben, die das Controller-FPGA übernimmt, wurden im vorherigen Abschnitt dargelegt. In diesem Kapitel werden einige wesentliche Implementierungsdetails erläutert.

Die Schaltung für das Controller-FPGA wurde mittels synthetisierbarem VHDL-Code implementiert. Die zugehörige Hierarchie der VHDL-Beschreibung ist in Bild 6-4 dargestellt.

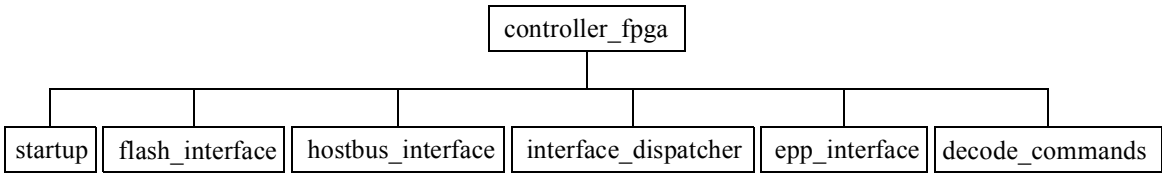


Bild 6-4. Hierarchie des synthetisierbaren VHDL-Modells für das Controller-FPGA.

Die Kommunikation über den Parallel-Port, der im EPP-Modus arbeitet, ist im Entity **epp_interface** implementiert. Die Software auf dem Host_PC kann das Controller-FPGA über diesen Port unter Einhaltung eines spezifizierten Protokolls in unterschiedliche Betriebszustände versetzen. Dazu gehören:

- der lesende/schreibende Zugriff auf die Flash-Bausteine, den DSP oder das FPGA,
- das Zurücksetzen (Reset) des DSP, des FPGA oder des gesamten Boards,
- der exklusive Zugriff auf den SelectMap-Port des FPGA.

Mit dem Entity **interface_dispatcher** sollte es ermöglicht werden, mehrere Schnittstellen mit dem Systembus zu verbinden, also neben dem Parallel-Port u. a. auch eine Ethernet-Schnittstelle. Implementiert wurde als Basisversion die Kommunikation über den Parallel-Port. Der Ablauf des Zugriffs auf den 16-Bit breiten Hostbus wird im Entity **hostbus_interface** kontrolliert. Über den Hostbus können Speicherstellen im DSP oder FPGA geändert oder gelesen werden, im FPGA jedoch nur, wenn dort ein Modul für die Kommunikation über den Hostbus aktiv ist. Die Re-Programmierung der Flash-Bausteine wurde mit dem Entity **flash_interface** realisiert. Die Adressen und die Daten für den Flash-Baustein des DSP werden dabei über den Hostbus übertragen, weshalb die Kommunikation auf dem Systembus unterbrochen werden muss. Im Entity **startup** wird in einem Zustandsautomat die Bootphase des FPGA kontrolliert gesteuert und zeitgleich der Synchronisationsmechanismus, beschrieben im Kap. 6.1, überwacht. Nach erfolgreichem Systemstart befindet sich der Automat im Zustand Running_System.

Das Controller-FPGA liest beim Systemstart seine Konfiguration aus einem seriellen EEPROM. Dieser nichtflüchtige Speicherbaustein kann über JTAG (IEEE 1149.1 Boundary-Scan**)** von einem Host-PC mittels In-System-Programmierung (ISP) mit neuen Konfigurationsdaten beschrieben werden [130].

Im Hardware-Prototyp wurde für das Controller-FPGA ein XCV100-4 von Xilinx eingesetzt. Die Ergebnisse der Foundation-Software von Xilinx für die Platzierung und Verdrahtung der Schaltung sind in Listing 6-1 zu sehen. Lediglich 25% des FPGA-Bausteins sind beim derzeitigen Stand der Implementierung belegt. Die statische Timinganalyse ermittelte eine maximale Verzögerungszeit von 17,69 ns, dies entspricht einer maximalen Taktrate von ca. 56 MHz.

```
1.   Release 4.1.03i - Map E.33
2.   Xilinx Mapping Report File for Design 'synth_controller_fpga'
3.
4.   Design Information
5.   ------------------
6.   Target Device  : xv100
7.   Target Speed   : -4
8.
9.   Design Summary
10.  --------------
11.     Number of errors:      0
12.     Number of warnings:    2
13.     Number of Slices:                309 out of  1,200   25%
14.     Number of Slices containing
15.        unrelated logic:                0 out of    309    0%
16.     Number of Slice Flip Flops:      305 out of  2,400   12%
17.     Total Number 4 input LUTs:       463 out of  2,400   19%
18.        Number used as LUTs:                        443
19.        Number used as a route-thru:                 20
20.     Number of bonded IOBs:           114 out of    166   68%
21.     Number of GCLKs:                   3 out of      4   75%
22.     Number of GCLKIOBs:                1 out of      4   25%
23.  Total equivalent gate count for design:  5,536
24.  Additional JTAG gate count for IOBs:  5,520
25.
26.  ---------------------------------------------------------------------
27.    Constraint                          | Requested   | Actual  | Logic
28.                                        |             |         | Levels
29.  ---------------------------------------------------------------------
30.    NET "C2391/IBUFG" PERIOD = 30 nS  | 30.000ns    |17.690ns | 6
31.    HIGH 50.000000 %                  |             |         |
32.  ---------------------------------------------------------------------
33.
34.  All constraints were met.
```

Listing 6-1. Ausschnitt aus den Place&Route-Ergebnissen für das Controller-FPGA.

6.2 Die Software *Mephisto*

Die Software *Mephisto* wurde für die MS-Windows-Plattform (Win9x/ME/XP) des Host-PC entwickelt, wobei die Back-End-Funktionen plattformunabhängig in ANSI-C implementiert sind. Insgesamt wurden dafür ca. 30.000 C-Codezeilen in Microsoft Visual C++ 6.0 implementiert. Die Software bietet im Front-End eine Projektverwaltung für die zu implementierende Aufgabenstellung und integriert im Front- und Back-End die Interface-Synthese. Weiterhin übernimmt sie die Aufgabe der Koordination des HW/SW-CoDesigns, der HW/SW-CoSimulation und der HW/SW-Synthese. Der HDL-Simulator ModelSim wird z. B. für die HW/SW-CoSimulation mit vorbereiteten Skripten gestartet oder auch die Projektverwaltung der Foundation-Software von Xilinx für die Hardware-Synthese gesteuert.

Zum Testen auf der Prototyp-Hardware stellt die Software die Verbindung zur Hardware über den Parallel-Port des Host-PC zur Verfügung. Hierbei können die Flash-Bausteine mit neuen Konfigurationsdaten programmiert werden. Zudem ist der Zugriff auf den gesamten Adressbereich des Systems als Host sowohl lesend als auch schreibend möglich. Unterstützt

wird dabei die Möglichkeit, größere Adressbereiche in einem Stück zu übertragen. Die aus einer Datei gelesenen Daten werden zum System transferiert (Memory Upload) oder vom System lesend in eine Datei geschrieben (Memory Dump). Es können z. B. die Koeffizienten eines FIR-Filters mit einem Memory Upload zur Laufzeit aktualisiert werden und die Ergebnisse des neu konditionierten Filters sind bei einer Audiobearbeitung direkt hörbar. Zusätzliche graphische Auswertungssoftware ist in die Software integriert (Plot). So lässt sich ein kompletter Speicherbereich in einem eingebundenen gnuplot-Programm automatisch darstellen. Damit können z. B. die im Speicher abgelegten Daten einer Fourier-Transformation oder eines FIR-Filters graphisch überprüft werden. Unterstützt wird die Lokalisierung der Variablen und sonstiger Stellen im Speicher durch Symboltabellen. Außerdem kann der Inhalt des On-Chip-Speichers des DSP in verschiedenen Formaten (disassembliert, hexadezimal, floting-point) inspiziert werden, so wie er in der Bootphase des Systems initialisiert wird. Dazu wird die in der Software-Synthese erzeugte ausführbare Datei für den DSP analysiert. Möglich ist auch, zur Laufzeit des Systems Software-Code im On-Chip-Speicher des DSP zu ändern. Dabei muss jedoch beachtet werden, dass die Abarbeitung der Software im DSP währenddessen weiterläuft. Eine neu übersetzte Softwareversion lässt sich zudem direkt auf den DSP übertragen, ohne den Flash-Baustein neu zu programmieren. Beim Neustart des Systems wird wieder das gespeicherte Programm geladen.

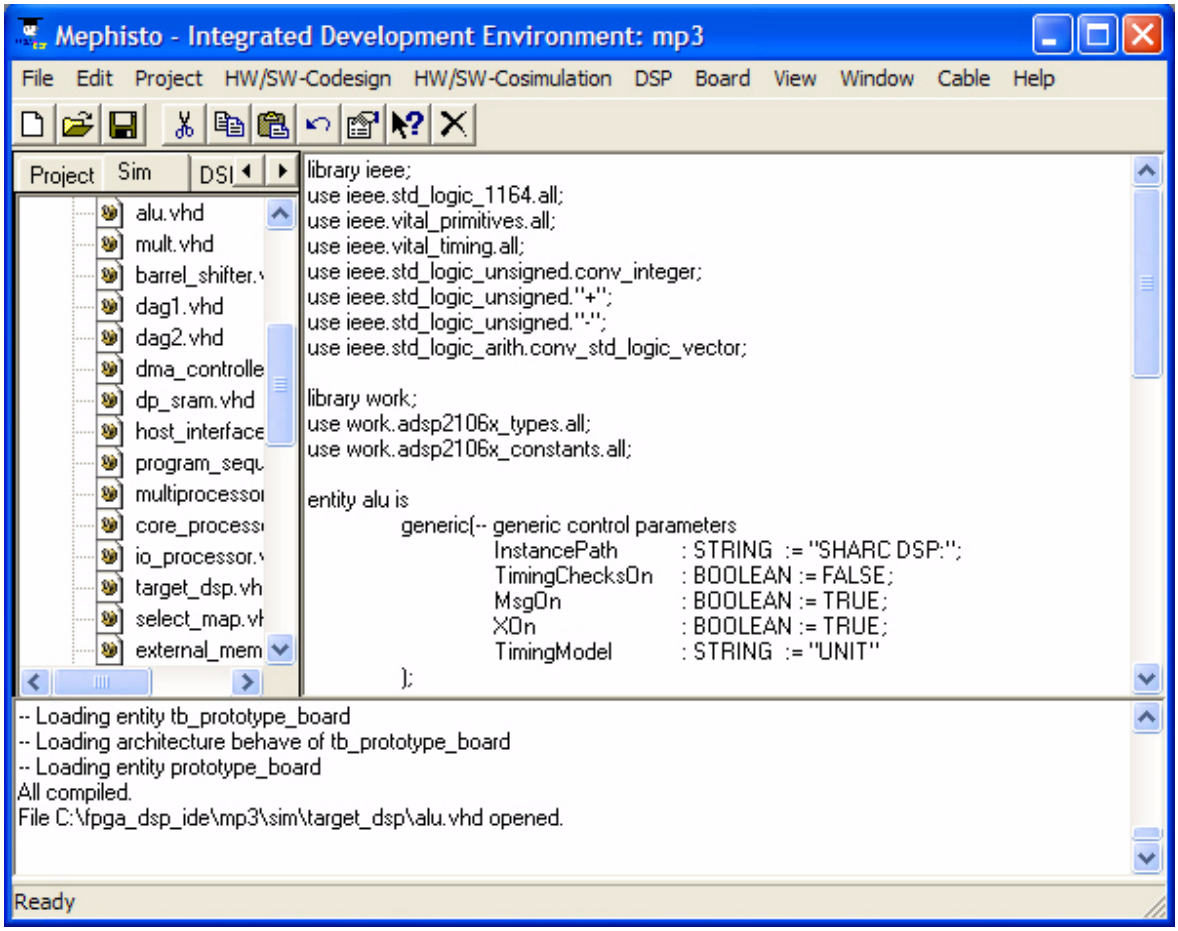


Bild 6-5. MS-Windows-Oberfläche der Software *Mephisto* mit geöffnetem Projekt.

Die integrierte Projektverwaltung (siehe Bild 6-5) erleichtert den Umgang mit verschiedenen gleichzeitig zu bearbeitenden Aufgabenstellungen. Die Dateien für die Software (DSP) und die Hardware (FPGA) werden in anpassbaren Verzeichnissen zusammengefasst. Damit ist eine einfache Versionsverwaltung realisiert. Die Kompilierungs-, Simulations- und Synthese-Schritte sind automatisiert, so dass eine schnelle Entwicklung unterstützt wird.

Für die Simulation der digitalen Audioschnittstelle (S/PDIF) stehen Konvertierungsfunktionen innerhalb der Software bereit, die Dateien im WAV-Format in das vom Modell des S/PDIF-Receivers verwendete Format umwandeln. Umgekehrt lassen sich die in der Simulation erzeugten Daten des S/PDIF-Transmitters in WAV-Dateien konvertieren. Es werden dabei WAV-Dateien mit wählbarer Sample-Rate (48 kHz, 44,1 kHz), Kanalanzahl (Mono oder Stereo) und Auflösung (8- oder 16-Bit) erzeugt.

6.3 Interface-Synthese

Die Interface-Synthese ist in die Software *Mephisto* von *FADES* integriert. Auf die einzelnen Bestandteile des in Kap. 5.3 gezeigten Konzepts wird in den folgenden Abschnitten detailliert eingegangen.

6.3.1 XML-Interface-Spezifikation

In Listing 6-2 ist die Notation der XML-Spezifikation der Variablen aufgelistet. Mit dem Tag **shared_codesign** als Wurzelelement wird der Beginn der Spezifikation der gemeinsamen Variablen eingeleitet. Unterhalb des Wurzelelements in der Baumstruktur sind die Variablen mit zugehörigen Attributen zu spezifizieren. Jede Variable erhält zunächst einen eindeutigen Bezeichner **name**, der frei wählbar ist. Der Typ der Variable **type** kann integer, float oder double sein. Er bestimmt für die Hardwarerealisierung im Wesentlichen die Wortbreite des zu übertragenden Variablenwertes und z. B. die Position im Datenwort auf dem Systembus. Mit dem Attribut **mode** wird bestimmt, wie der Zugriff vom DSP auf die Variable im FPGA erfolgt: nur lesend, nur schreibend oder lesend und schreibend. Auf Ergebniswerte wird lediglich lesend zugegriffen, Operandenwerte werden geschrieben und unter Umständen zur Überprüfung auch zurückgelesen. Die Auswahl der Lese-/Schreib-Modi der gemeinsamen Variablen geschieht aus der Sicht des DSP, da es sich immer um eine Auslagerung in das FPGA handelt. Stellt die Variable ein eindimensionales Speicherelement dar, so kann mit dem optionalen Attribut **size** dessen Elementeanzahl spezifiziert werden.

```
1.   <?xml version="1.0" encoding="UTF-8"?>
2.
3.   <shared_codesign>
4.   <variable name="NAME" type="TYPE" mode="MODE" [size="SIZE"]/>
5.   ...
6.   </shared_codesign>
```

Listing 6-2. Notation der XML-Eingabe-Spezifikation für die Interface-Synthese.

6.3.2 XML-Bibliothek

Die Übertragung der Variablenwerte zwischen DSP und FPGA übernehmen die auswählbaren Software- und Hardware-Module, die in einer XML-kodierten Bibliothek abgelegt sind. Vom Benutzer kann entschieden werden, ob die Übertragung der Variablenwerte synchron oder asynchron erfolgen soll oder welche Kommunikationsmodelle zum Einsatz kommen (siehe Kap. 4.1). Des Weiteren enthält die Bibliothek für eine Wiederverwendung Elemente, die nicht speziell für die Kommunikation zwischen Hardware und Software entwickelt wurden. Dies können Funktionseinheiten sein, die Berechnungen in Hardware und/oder Software durchführen, so z. B. die Berechnung des CORDIC-Algorithmus. Ein weiteres Beispiel ist die Komponente, die die Weiterleitung der Signale der S/PDIF-Schnittstelle im FPGA übernimmt (in Bild 6-1 durch die gestrichelte Linie angedeutet). Ohne diese Komponente könnte die S/PDIF-Schnittstelle von der Software im DSP nicht genutzt werden. Ob dabei eine Änderung der Daten implementiert wird (mögliche Vor- oder Nachbearbeitung durch das FPGA, siehe Kap. 4.1.3) oder nur eine Weiterleitung stattfindet, ist dem Benutzer des Entwicklungssystems überlassen und hängt von der konkret zu bearbeitenden Aufgabe ab.

Die Bibliothek ist in zwei XML-Dateien gespeichert, der eigentlichen Modul-Bibliothek library.xml und der Bibliothek für die Modulbeschreibungen library_desc.xml. Listing 6-3 zeigt den Prolog, das Wurzel- und ein Beispiel-Element der Modul-Bibliothek. Das Beispiel-Element entity_generic_package stellt für die Interface-Synthese ein generisches VHDL-Package für den VHDL-Code des FPGA zur Verfügung. Die CDATA-Abschnitte enthalten den eigentlichen Text des Moduls. Weitere eindeutige Unterelemente sind als Leeres-Element-Tag integriert, auf deren Funktion im Kap. 6.3.4 näher eingegangen wird.

In library_desc.xml sind zusätzlich Beschreibungen für jedes Bibliothekselement gespeichert, die im Dialogfenster der Software für das Durchsuchen der Bibliothek eingeblendet werden, wenn das Modul in der Liste angewählt (markiert) ist. Listing 6-4 zeigt einen Ausschnitt aus diesem Teil der Bibliothek für das Beispiel entity_generic_package. Dabei ist erkennbar, dass die Bezeichner für die Elemente in den beiden Teilbibliotheken übereinstimmen. Mittels CDATA-Abschnitt wird die Beschreibung des Moduls eingegrenzt. Es sind keine weiteren Unterelemente definiert.

Die XML-Bibliothek ist erweiterbar, denn die Realisierung erhebt keinen Anspruch auf Vollständigkeit. Sie ist unkomprimiert als Textdatei gespeichert und damit leicht editierbar. Für die Einsparung an Speicherplatz ist eine Komprimierung vorstellbar. Diese dürfte jedoch kein proprietäres Format erzeugen, um die Schreib- und Lesbarkeit auch außerhalb der Software des Entwicklungssystems zu gewährleisten.

6.3.3 XML-Parser

Für den XML-Parser wurde das Toolkit expat [36] eingesetzt. Expat ist ein nicht validierender, Stream-orientierter Parser für XML 1.0, implementiert in C. Eine im Toolkit verfügbare Exportdatei, xmlparse.lib, wird zur Übersetzungszeit der Software *Mephisto* vom Linker

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <library ver="1.0">
4.
5.  <entity_generic_package>
6.  <![CDATA[
7.  library ieee;
8.  use ieee.std_logic_1164.all;
9.
10. package generic_package is
11.
12. type int_array is array (integer range <>)
13.                                 of std_logic_vector(31 downto 0);
14. type float_array is array (integer range <>)
15.                                 of std_logic_vector(31 downto 0);
16. type double_array is array (integer range <>)
17.                                 of std_logic_vector(39 downto 0);
18.
19. ]]>
20.
21. <generic_package_declarations/>
22.
23. <![CDATA[
24.
25. end;
26. ]]>
27. </entity_generic_package>
28. ...
29. </library>
```

Listing 6-3. Ausschnitt aus der Modul-Bibliothek library.xml.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <library ver="1.0">
4.
5.  <entity_generic_package>
6.  <![CDATA[
7.  (HW-FPGA) Generic Package to specify shared hw/sw variables in VHDL.
8.  ]]>
9.  </entity_generic_package>
10. ...
11. </library>
```

Listing 6-4. Ausschnitt aus der Bibliothek der Modulbeschreibungen library_desc.xml.

eingebunden. Dynamisch geladen werden die Importbibliotheken xmlparse.dll und xmltok.dll beim Start der Software. Die exportierten Funktionen des Parsers können zur Laufzeit der Software entsprechend der Definition genutzt werden.

Da der XML-Parser in verschiedenen Situationen mit variablen Bedingungen eingesetzt werden sollte, wurde eine Funktion XML_Parser_Wrapper() implementiert. Sie ist vollständig im Anhang A.1.1 aufgelistet. Die Aufgabe dieser Funktion besteht darin, eine neue Instanz des XML-Parsers zu initialisieren und einen übergebenen XML-Stream kontrolliert dem Parser zuzuführen. Der erste Parameter der Funktion ist demnach ein zum Lesen geöffneter XML-Stream. Zunächst wird ein neuer Parser mit XML_ParserCreate() erzeugt.

Anschließend werden den einzelnen im XML-Dokument auftretenden Strukturelementen Callback-Funktionen, sogenannte Handler, zugeordnet, die vom Parser immer dann aufgerufen werden, wenn das entsprechende Strukturelement im XML-Stream auftritt. Zur Laufzeit des Parsers kann die Zuordnung auch dynamisch geändert werden. Für die Registrierung der Handler gibt es Zuweisungsfunktionen, so können beispielsweise mit XML_SetElementHandler() dem Start- und End-Tag eines Elements je ein Handler zugeordnet werden. Weitere in XML_Parser_Wrapper() genutzte Zuweisungsfunktionen sind:

- *XML_SetCharacterDataHandler():* Handler für jede Art auftretenden Textes ohne Tag,
- *XML_SetCdataSectionHandler():* Handler für Start-/End-Tag einer CDATA-Sektion.

Die Namen der Handler-Funktionen, die den Strukturelementen zuzuordnen sind, werden als Parameter der Funktion XML_Parser_Wrapper() übergeben.

Um die Kommunikation zwischen mehreren Handler-Funktionen ohne die Definition von globalen Variablen zu ermöglichen, ist eine eigene Datenstruktur für die Variablen zu definieren. Mit XML_SetUserData() wird dem Parser der Zeiger auf diese Datenstruktur mitgeteilt. Bei jedem Aufruf einer Handler-Funktion wird dann der Zeiger als ein Parameter übergeben. XML_Parser_Wrapper() benötigt als zweiten Parameter den Zeiger für die situationsabhängige Datenstruktur, die jedoch auch NULL sein kann.

Nachdem die Instanz des XML-Parsers initialisiert wurde, erfolgt die kontrollierte Übergabe der Eingangsdaten aus dem XML-Stream in einem Zwischenpuffer an die Funktion XML_Parser(), bis das Ende des Streams erreicht wird. Abschließend wird die erzeugte Instanz des Parsers mit XML_ParserFree() freigegeben.

6.3.4 Interface-Generierung

In dieser Phase der Interface-Synthese werden die beim Parsen der XML-Interface-Spezifikation gesammelten Informationen über die Variablen in die ausgewählten Hardware-/Software-Module eingesetzt. Die Informationen sind in einer internen Datenstruktur (dynamisch verkettete Liste mit Hash-Werten) gespeichert.

Innerhalb eines Moduls der XML-Bibliothek, das ein HW/SW-Modul darstellt und damit für die Interface-Synthese bestimmt ist, sind zusätzliche Leere-Element-Tags für nicht-statische Teile eingefügt. Die mit XML_SetElementHandler() zugeordneten Funktionen für das Start- und End-Tag können auf das Vorkommen des Bezeichners des Elements reagieren und die entsprechenden variablen Bestandteile generieren. Die Generierung wird zum Teil also schon während des Parsens des Bibliothekselements vorgenommen.

Die Zuordnung der Funktionen zu den Tags eines Moduls in der Bibliothek erfolgt im C-Quelltext von *Mephisto* in der Header-Datei library.h und in der Datei library.c. Es können Funktionen für folgende Tags assoziiert werden: Init-Tag, Start-Tag, End-Tag und Data-Tag. Das Init-Tag ist kein in XML selbst definiertes Tag, sondern kennzeichnet dasjenige Ereignis, bei dem das Start-Tag des zu generierenden Moduls in der Bibliothek erreicht wird. Die

mit diesem Ereignis verknüpfte Funktion kann zur Initialisierung der Generierung des Moduls genutzt werden. Die mit den Start- und End-Tags assoziierten Funktionen werden immer dann aufgerufen, wenn ein Start-, End- oder ein Leere-Element-Tag innerhalb der Beschreibung des Moduls in der Bibliothek vorkommt. In diesen Funktionen ist die eigentliche Logik der Generierung der variablen Bestandteile des Moduls zu implementieren. Ein Data-Tag tritt auf, wenn eine CDATA-Sektion eines Moduls erreicht wurde. Eine CDATA-Sektion wird in der Regel für die Abgrenzung des statischen vom dynamischen Teil des Moduls in der Bibliothek verwendet. Der statische Teil, wie z. B. VHDL für ein Hardware-Modul, kann mit Hilfe einer Standard-Funktion formatiert im Back-End ausgegeben werden. Die für das Beispiel entity_generic_package aus Listing 6-3 den genannten Tags zugeordneten Funktionen zeigt zusammengefasst Listing 6-5.

```
1.  // library.h
2.  // declare functions for entities in library (init, data, start and
3.  // end element)
4.  void entity_dummy_init(FILE *dest, void *userData, const char *name,
                      const char **atts);
5.  void entity_generic_data_element_handler(FILE *dest, void *userData,
                      const char *data, int len);
6.  void entity_generic_package_start_element_handler(FILE *dest, void
                      *userData, const char *name, const char **atts);
7.  void entity_dummy_end_element_handler(FILE *dest, void *userData,
                      const char *name);
8.
9.  // library.c
10. char *emit_entity_name_decl[] = {"entity_generic_package", NULL};
11.
12. static EmitEntityInit emit_entity_init_decl[] = {
13.           entity_generic_package_init, entity_dummy_init};
14.
15. static EmitEntityDataElementHandler
16.           emit_entity_data_element_handler_decl[] = {
17.                     entity_generic_data_element_handler,
18.                     entity_generic_data_element_handler};
19.
20. static EmitEntityStartElementHandler
21.           emit_entity_start_element_handler_decl[] = {
22.                     entity_generic_package_start_element_handler,
23.                     entity_dummy_start_element_handler};
24.
25. static EmitEntityEndElementHandler emit_entity_end_decl[] = {
26.           entity_generic_package_end_element_handler,
27. entity_dummy_end_element_handler};
```

Listing 6-5. Die in library.c und library.h für die Generierung des Beispiels entity_generic_package mit den Tags assoziierten Funktionen.

Alle Dummy-Funktionen, wie beispielsweise entity_dummy_init() (Zeile 4 in Listing 6-5), beinhalten keine Funktionalität und kehren nach ihrem Aufruf ohne Rückgabewert zurück. Standardmäßig sind alle Tags mit diesen Dummy-Funktionen verknüpft. Eine Ausnahme bildet das Data-Tag, welches mit der Funktion entity_generic_data_element_handler() verknüpft ist. Diese steuert die formatierte Ausgabe des Text-Bereiches, der mit dem Data-Tag in der XML-Bibliothek versehen ist. Dadurch ist es möglich, dass Module für eine spätere

Wiederverwendung zur Bibliothek hinzugefügt werden können, ohne die Software anpassen zu müssen.

Das Ergebnis der Interface-Generierung ist eine VHDL-Beschreibung auf Register-Transfer-Ebene für die Hardware im FPGA als Eingabe für die Hardware-Synthese (RTL-Synthese) und C-/Assembler-Code für den DSP als Eingabe für die Software-Synthese (Compiler/Assembler/Linker). Im generierten Programm-Code für den DSP wird die Abbildung der Variablen aus der XML-Spezifikation im Adressraum (Mapping) gesteuert, da kein direkter Zugriff auf die Software-Synthese möglich ist, weil dafür Software von Analog Devices verwendet wird. Dieselbe konsistente Abbildung der Variablen ist im VHDL-Code für das FPGA sichergestellt.

```
1.   library ieee;
2.   use ieee.std_logic_1164.all;
3.
4.   package generic_package is
5.
6.   type int_array is array (integer range <>)
7.                                   of std_logic_vector(31 downto 0);
8.   type float_array is array (integer range <>)
9.                                   of std_logic_vector(31 downto 0);
10.  type double_array is array (integer range <>)
11.                                  of std_logic_vector(39 downto 0);
12.
13.  signal operand: std_logic_vector(31 downto 0);
14.  signal result: std_logic_vector(31 downto 0);
15.  end;
```

Listing 6-6. Vom Back-End generiertes VHDL für das Beispiel entity_generic_package.

Im Back-End werden die generierten Hardware- und/oder Software-Module in Dateien ausgegeben. Für das Beispiel entity_generic_package aus Listing 6-3 und Beispiel-Variablen ergibt sich die in Listing 6-6 erzeugte VHDL-Datei, deren Name vom Benutzer vor dem Start der Interface-Synthese in einem Auswahldialog frei gewählt werden kann. Für dasselbe Beispiel wurden für den DSP die in Listing 6-7 dargestellten Assembler- und C-Dateien generiert.

```
     a                                b
1.   .segment /dm seg_fpga;           #include <macros.h>
2.
3.   .global operand;                 DEF_PORT(operand, int, seg_fpga, dm);
4.   .global result;                  DEF_PORT(result, int, seg_fpga, dm);
5.
6.   .var operand;
7.   .var result;
8.
9.   .endseg;
```

Listing 6-7. Generierte **a** Assembler-Datei und **b** C-Datei für das Beispiel entity_generic_package.

Die erzeugten Dateien sind vom Benutzer in das aktuelle Projekt einzufügen, indem beispielsweise das in der VHDL-Datei generierte Entity in die Hierarchie der Schaltungsbeschreibung für das FPGA instantiiert wird.

Sollen neu entwickelte Module in die Bibliothek aufgenommen werden, sind sie manuell mit einem Texteditor in die XML-Bibliothek zu integrieren. Enthalten die Module rein statische Bestandteile, muss die Software nicht angepasst werden. Hierbei werden Standard-Funktionen assoziiert, die in der Regel den statischen Inhalt des Moduls ausgeben. Sind dagegen die Module zur Interface-Synthese vorgesehen, so sind die Quelltexte der Software *Mephisto* anzupassen und neu zu übersetzen.

6.4 Hardware/Software-CoSimulation

Auf die Konzepte der HW/SW-CoSimulation, die in *FADES* integriert ist, wurde bereits im Kap. 5.4 eingegangen. Es wird zunächst ein Überblick über die entwickelten VHDL-Modelle der Komponenten der Hardware für die HW/SW-CoSimulation gegeben. Danach werden die möglichen Simulationsarten und die Debug-Oberfläche für die Simulation zusammen mit der Implementierung erläutert.

6.4.1 VHDL-Modelle der Hardware-Komponenten

Bild 6-6 veranschaulicht die Hierarchie der nachfolgend beschriebenen VHDL-Modelle der Hardware-Komponenten von *Faust*.

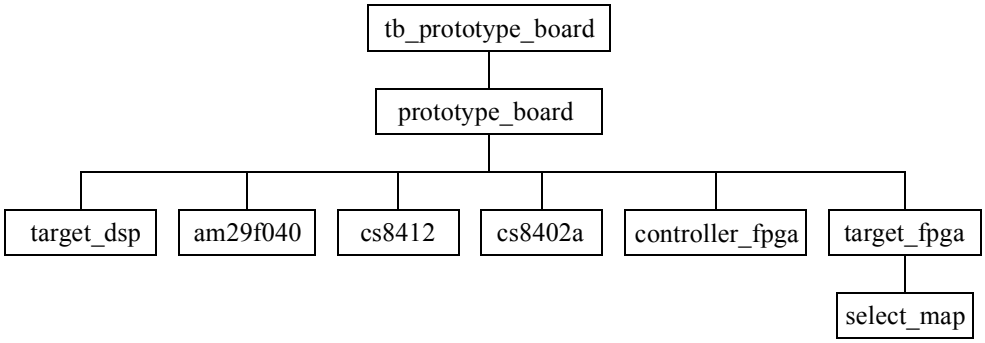


Bild 6-6. Hierarchie der VHDL-Modelle der Hardware-Komponenten.

Das zyklengetreue VHDL-Modell **target_dsp** für die ADSP-2106x-Familie implementiert die Zeitparameter des Datenblattes [7, 8], überprüft das Zeitverhalten der Eingänge und berechnet die Werte der Ausgänge in ihrer zeitlichen Relation und in Abhängigkeit vom Zustand des Rechenkerns (Detailed-Behavioral Model). Der Zustand des Rechenkerns wird in erster Linie durch das auf dem DSP ablaufende Programm bestimmt, kann aber auch über den Host-Port des DSP beeinflusst werden. Das Programm für den DSP wird zu Beginn der Simulation in das Modell mit Hilfe von Text-I/O (VHDL-93) im ASCII-Format eingelesen, was der Portabilität des Modells dient. Der Name der Datei, die das Programm als Speicherabbild enthält, wird mit einem Generic-Parameter dem Modell übergeben. So können mehrere Instanzen des DSP parallel in einer VHDL-Simulation integriert werden. Mit diesem Modell ist es also möglich, die Software, die im System auf dem DSP abgearbeitet wird, in die VHDL-Simulation mit einzubringen. Dies ist eine wichtige Voraussetzung für das effektive Testen eines Algorithmus, der partitioniert in Hardware und Software arbeitet.

Für die Flash-Bausteine AM29F040B von Analog Devices wurde auf ein VHDL-Modell von [58] zurückgegriffen. Im Modell **am29f040** wurde die Möglichkeit hinzugefügt, den Inhalt des Speicherbausteins zu Beginn der Simulation aus einer Textdatei über Text-I/O einzulesen, bzw. nach einer Schreiboperation den neuen Inhalt in dieselbe Datei zu schreiben. Der Name der Datei wird als Generic-Parameter dem Modell übergeben und kann so zur Übersetzungszeit flexibel angepasst werden.

Der digitale S/PDIF-Ein-/Ausgang, bzw. dessen Receiver (**cs8412**) und Transmitter (**cs8402a**) von Crystal, wurden ebenso in VHDL modelliert. Die Daten werden im Modell ebenfalls aus/in Textdateien über Text-I/O gelesen/geschrieben. Die Timing-Parameter sind den entsprechenden Datenblättern entnommen [40, 41].

Das synthetisierbare VHDL-Modell **controller_fpga** des Controller-FPGA wurde bereits in Kap. 6.1.1 erläutert. Die VHDL-Beschreibung der platzierten und verdrahteten Schaltung im Controller-FPGA ist in einem eigenen Entity **synth_controller_fpga** gegeben. Hierbei handelt es sich um eine aus der Netzliste für das FPGA generierte VHDL-Strukturbeschreibung, die mit Hilfe der Werkzeuge von Xilinx (ngd2vhdl) erzeugt wurde.

Das Modell für den Konfigurationsport (SelectMap) des FPGA **select_map** (Entity-Deklaration im Anhang A.3.2) wurde in erster Linie für die Simulation des kompletten Bootvorgangs des FPGA und damit der Überprüfung der Schaltung im Controller-FPGA entwickelt. Eine CRC-Summe wird anhand der über diesen Port in der Simulation eingehenden Konfigurationsdaten berechnet, so wie es in der Dokumentation zum Virtex-FPGA [134, 135] beschrieben ist. Außerdem wird hier auch das korrekte Zeitverhalten der Steuersignale überprüft. Darüber hinaus ist es mit Hilfe dieses Modells auch möglich, eine dynamische (partielle) Rekonfiguration des FPGA in der Simulation zu überwachen. Die Beschreibung für die Schaltung im FPGA ist zunächst nur in der Schnittstellendefinition des VHDL-Entity **target_fpga** bzw. in implementierten Beispielen gegeben. Der Benutzer von *FADES* kann die Schnittstellendefinition mit einem Algorithmus auf Register-Transfer-Ebene befüllen und dann synthetisieren.

Das den vorgenannten Modellen übergeordnete Modell **prototype_board** in der Hierarchie beschreibt schließlich die Verbindungen der Komponenten auf der Leiterplatine und definiert in der Portdeklaration nur eine externe Schnittstelle, den Parallel-Port. Die Testbench **tb_prototype_board** instantiiert prototype_board als einzige Komponente, so dass hiermit die Schnittstelle des Systems zum Host-PC getestet werden kann. Dies ist für den Benutzer von *FADES* nicht besonders interessant, es wurde jedoch bei der Entwicklung der Schaltung für das Controller-FPGA genutzt.

6.4.2 Simulationsarten

Es gibt insgesamt drei wesentliche Simulationsarten für die in *FADES* integrierte HW/SW-CoSimulation:

1. inklusive der kompletten Bootphase,

2. ohne Bootphase, mit den beiden Flash-Bausteinen,
3. ohne Bootphase, ohne die beiden Flash-Bausteine.

Die erste Simulationsart ist immer dann wichtig, wenn auch die Bootphase für die Verifikation des zu entwickelnden Algorithmus im System eine Rolle spielt, wie beispielsweise das Verhalten beim synchronen Start von DSP und FPGA. Hierfür wird insgesamt die meiste Simulationszeit aller drei Simuationsarten benötigt.

Bei der zweiten Simulationsart wird die komplette Bootphase übersprungen und die Simulation startet zu dem Zeitpunkt, zu dem der DSP und das FPGA im System bereits konfiguriert sind. Simulationszeit wird somit eingespart, wenn die Bootphase zur Verifikation nicht benötigt wird. Die Flash-Bausteine sind weiterhin in die Simulation einbezogen und können zur Laufzeit der Simulation mit neuen Daten über den simulierten Parallel-Port des Host-PC programmiert, oder auch vom DSP als zusätzlichen externen Speicher adressiert werden. Die Simulation der Parallel-Port-Schnittstelle kann dafür genutzt werden, die Schaltung im Controller-FPGA auszubauen, falls zukünftig Änderungen vorgenommen werden.

Die dritte Simulationsart enthält aus Performancegründen nicht das komplette VHDL-Modell der Flash-Bausteine, wenn deren Simulation zur Verifikation des Systems nicht benötigt wird. Besonders das Einlesen der Daten aus dem Flash-Speicherabbild zu Beginn der Simulation über Text-I/O benötigt einen nicht vernachlässigbaren Teil der Simulationszeit. Das komplette VHDL-Modell wird durch eine Minimalversion (einfaches Interface-Modell) ersetzt (vgl. Kap. 3.3.4). Aufgrund der Modularisierbarkeit von VHDL lassen sich auch andere Teile des Gesamtsystems in die Simulation einbeziehen oder durch vereinfachte Modelle, die weniger Simulationszeit benötigen, ersetzen. Dies kann zum Beispiel bei der S/PDIF-Schnittstelle von *Faust* angewendet werden, wenn für einen Algorithmus keine digitalen Audiodaten ein- oder auszulesen sind. Für die eigentliche Simulation eines Algorithmus, der auf dem DSP und dem FPGA partitioniert abläuft, wird diese Simulationsart relevant sein.

Die Auswahl der verschiedenen Simulationsarten erfolgt transparent für den Benutzer über einen Auswahldialog, der zugehörige Makros zusammen mit dem Start des VHDL-Simulators lädt. Dafür wurden entsprechende Konfigurationen deklariert, die wiederum Generic-Parameter der Teilmodelle verschiedentlich ansteuern.

6.4.3 Debug-Oberfläche

Die Hardware- und die Software-Simulation laufen, wie bereits in Kap. 5.4 erläutert, innerhalb eines Simulatorkerns, nämlich des Kerns der VHDL-Simulation. Es stehen alle Debug-Möglichkeiten zur Verfügung, die der VHDL-Simulator, hier der ModelSim, anbietet. Dazu zählen Signalverläufe, Signallisten, Variablenüberwachung oder auch die Darstellung des VHDL-Quelltextes und im Besonderen die Möglichkeit, Breakpoints im VHDL-Code zu setzen. Es ist aber auch wichtig, dass für das Software-Debugging eine abstraktere, prozessorspezifische Sicht auf die Simulation der Software möglich ist. Das heißt, der Software-Entwickler für das HW/SW-Entwicklungssystem muss nicht nur in den Signalverläufen der

internen Signale der VHDL-Simulation oder in ihm unbekanntem VHDL-Quelltext des Prozessormodells nach Fehlern suchen. Deshalb wurde eine plattformunabhängige graphische Debug-Oberfläche für die Tcl/Tk-Schnittstelle des HDL-Simulators ModelSim implementiert. Sie arbeitet unabhängig von *Mephisto* innerhalb des VHDL-Simulators. Der Simulator wird jedoch im Entwicklungssystem zusammen mit für die im Kap. 6.4.2 definierten Simulationsarten vorbereiteten Skripten gestartet.

Die Debug-Oberfläche versetzt den Entwickler in die Lage, den Algorithmus, der partitioniert in Hardware und Software implementiert ist, Schritt für Schritt zu verifizieren. Bild 6-7 zeigt die Debug-Oberfläche bei der Simulation des MPEG1-Layer3-Player (siehe Kap. 7.4). Im DSP-Code können damit Breakpoints gesetzt werden (❶) und die aktuelle PC-Adresse erscheint farbig hervorgehoben im Fenster mit dem aktuellen Speicherausschnitt (❷). Der DSP-Code kann neben anderen Formaten (hexadezimal, dezimal, floating-point) auch disassembliert im Speicherfenster dargestellt werden. Alle möglichen internen Register (Systemregister, Registerfile) sind im graphischen Front-End über Menübefehle zugänglich und ihr Inhalt wird in eigenen Fenstern angezeigt (❸). Komplette simulierte Speicherbereiche können in Dateien zur späteren Referenz gesichert werden (❹). Aufgrund der zusätzlich benötigten Rechenzeit wurde die Debug-Oberfläche derart gestaltet, dass sie zur Laufzeit der Simulation an- und abdockbar ist, um die Simulation zu beschleunigen.

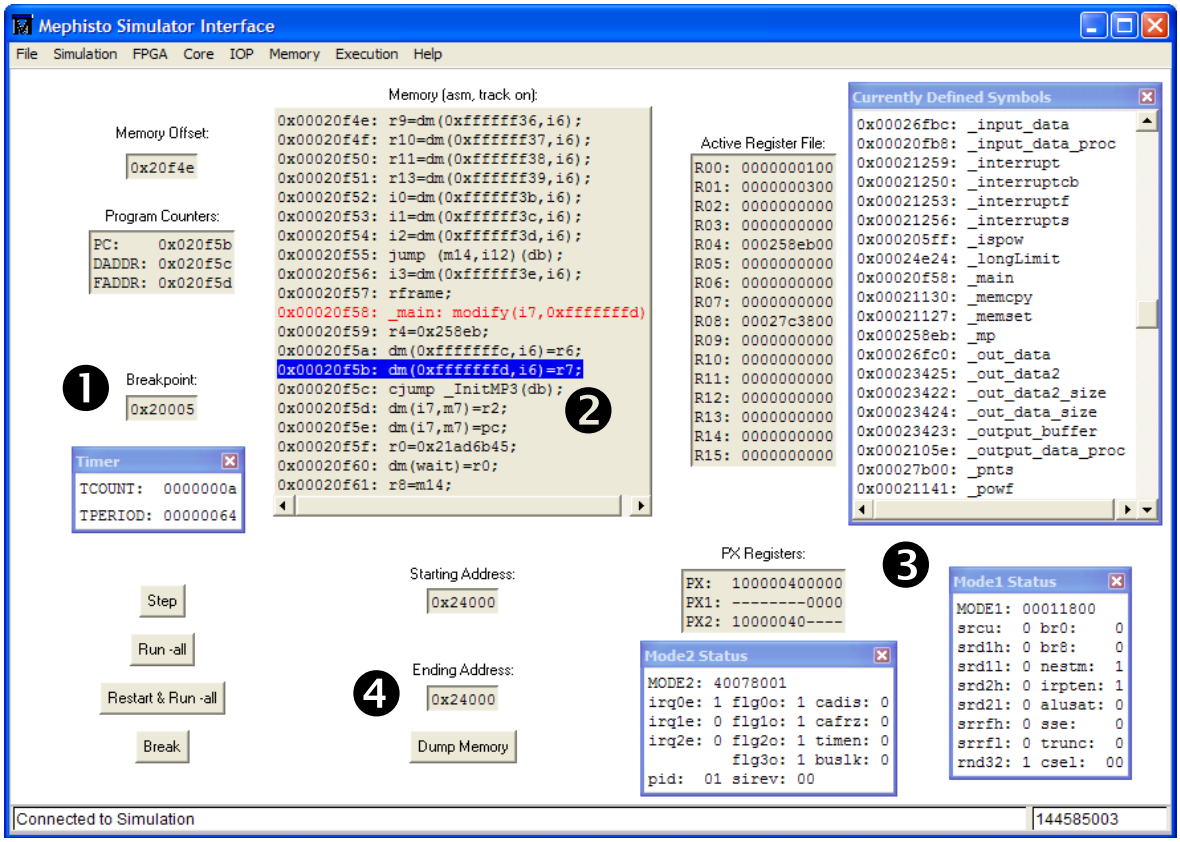


Bild 6-7. Debug-Oberfläche für die HW/SW-CoSimulation.

Zusatzinformationen können mit Hilfe einer (optionalen) linearen statistischen Analyse (Profiling) während der Simulation gesammelt, oder auch die Leistungsfähigkeit des Algorithmus (Rechenzeit, Speicherverbrauch) anhand der zyklengetreuen Simulation evaluiert

werden. Sollten die erreichten Qualitätseigenschaften der Implementierung nicht für die Aufgabenstellung ausreichend sein, müssen die gesammelten Daten der statistischen Analyse ausgewertet und Verbesserungsansätze gefunden werden. Oft ist es so, dass 10% des Codes 90% der Verarbeitungszeit der Applikation beanspruchen [66]. Unter Umständen muss eine Re-Partitionierung des Problems stattfinden.

Nicht alle Funktionen, die für die Debug-Oberfläche notwendig waren, wurden in Tcl/Tk implementiert. In ANSI-C wurden die Disassembler-Funktionalität, die Funktionen der statistischen Analyse und einige Konvertierungsroutinen programmiert. Dies erhöhte zum einen die Ausführungsgeschwindigkeit und zum anderen konnte parallel die Funktionalität auch für *Mephisto* genutzt werden. Der C-Quelltext wurde mit Hilfe der Wrapper-Software SWIG [25] und Microsoft Visual C++ in eine Dynamic Link Library (DLL) für den Zugriff in Tcl/Tk umgesetzt (Bild 6-8).

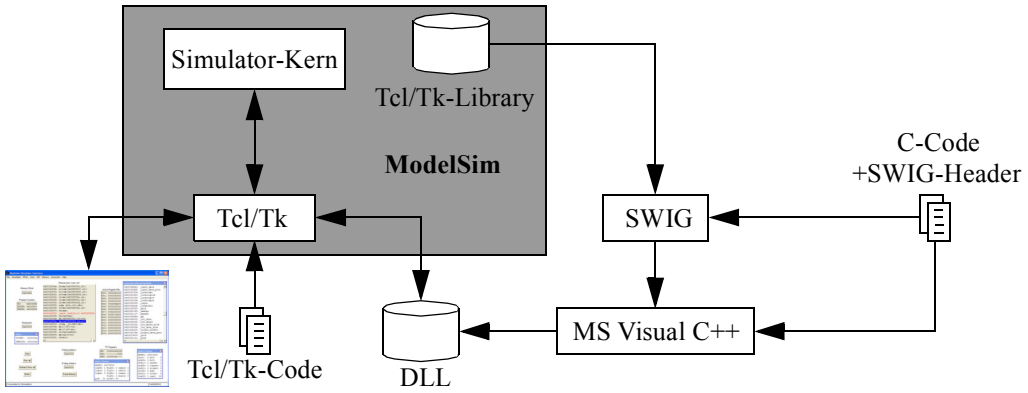


Bild 6-8. Einbindung der mit Hilfe von SWIG erzeugten DLL in die Realisierung der Debug-Oberfläche für die HW/SW-CoSimulation im HDL-Simulator ModelSim.

SWIG bietet für eine Reihe von Skriptsprachen (Perl, Python, Tcl/Tk, ...) die Definition einer Schnittstelle für C/C++-Erweiterungen in der Art einer IDL (Interface Definition Language). SWIG generiert aus dieser Schnittstellenbeschreibung für die jeweilige Skriptsprache eine spezifische C-Header-Datei, die die nötigen Definitionen und Konvertierungen zwischen C/C++ und der Skriptsprache definieren (Wrapper). Diese Header-Datei kann dann in einem C/C++-Programm zur Typdefinition benutzt werden. Mit SWIG wurde es damit mit relativ wenig Aufwand möglich, in Tcl/Tk auf nativen C-Code zuzugreifen. Die generierte DLL wird im Tcl/Tk-Code mit dem Befehl load zur Ausführungszeit dynamisch in den Speicher geladen. Jegliche in der DLL exportierte Funktion steht nun in Tcl/Tk zur Verfügung und kann gemäß der Deklaration ihrer Parameterliste genutzt werden.

6.4.4 Implementierung & Verifikation des ADSP-2106x-Modells

Der Aufwand für die Entwicklung und den Test der Simulationsmodelle ist nicht zu unterschätzen. Im Folgenden werden einige Implementierungsdetails des VHDL-Modells für die ADSP-2106x-Familie erläutert und es wird beschrieben, wie die Verifikation durchgeführt wurde.

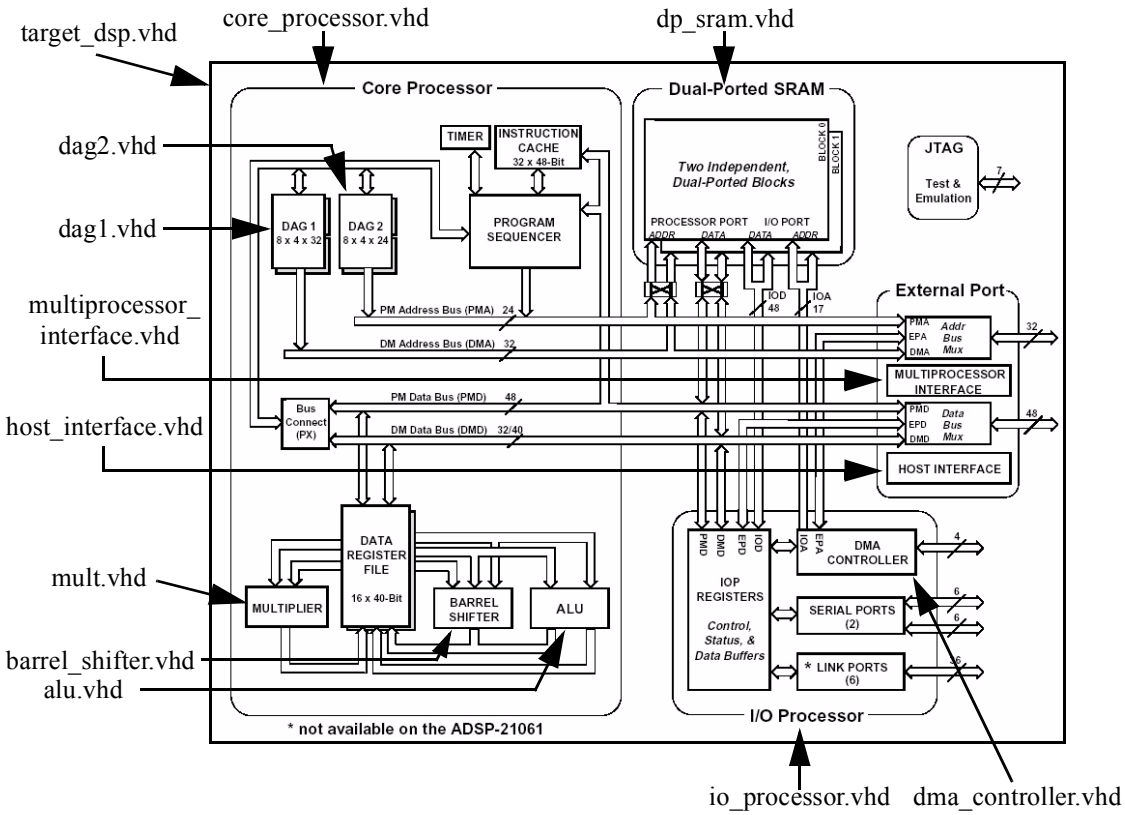


Bild 6-9. Hierarchie des VHDL-Modells für die ADSP-2106x-Familie visualisiert im Blockschaltbild (Quelle Blockschaltbild: [7]).

Die Entity-Deklaration des VHDL-Modells ist im Anhang A.3.1 abgebildet. Die Strukturierung der VHDL-Dateien für das Modell ist dem hierarchischen Aufbau der Funktionsblöcke der ADSP-2106x-Familie entnommen. In Bild 6-9 sind die VHDL-Dateinamen des Modells den Blöcken im Blockschaltbild der ADSP-2106x-Familie zur Übersicht visuell zugeordnet. Die Namen der Dateien entsprechen dabei denen der zugehörigen Entities. In Bild 6-10 sind alle Entities für einen Überblick in einem Hierarchiebaum dargestellt. Für die Leistungseigenschaften der Simulation wäre es günstiger, die gesamte Funktionalität innerhalb eines VHDL-Entity zu beschreiben. Bei dem vorliegenden Umfang der Funktionalität ginge dies zu Lasten der Übersichtlichkeit und wurde deshalb nicht umgesetzt.

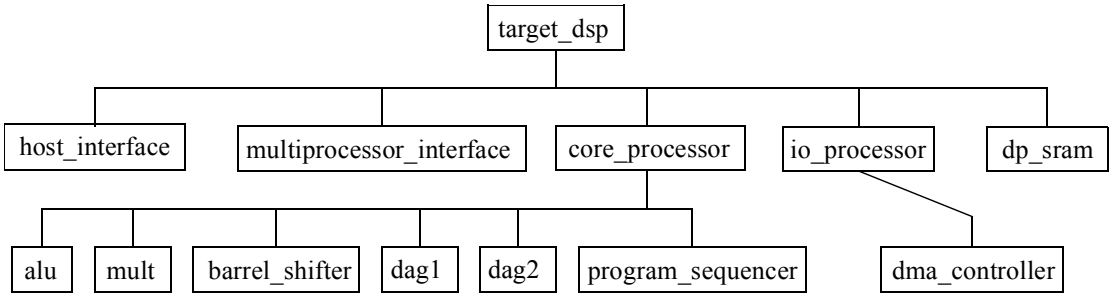


Bild 6-10. Hierarchie des VHDL-Modells für die ADSP-2106x-Familie.

Zunächst wurden die Recheneinheiten der ADSP-2106x-Familie in VHDL modelliert. Dazu zählen die ALU (**alu**), der Barrel-Shifter (**barrel_shifter**) und die Multipliziereinheit (**mult**). Da es sich hierbei um kombinatorische Logik handelt, müssten die Prozesse, die das

Verhalten der Einheiten beschreiben, sensitiv zu allen möglichen Eingangssignalen sein. Damit aber nicht bei jeder Änderung eines Eingangssignals der Prozess aktiviert wird, wurde ein zusätzliches enable-Signal eingeführt. Dieses wird im Entity für den Core-Prozessor nur aktiviert, wenn alle Eingangssignale stabil anliegen und die Einheit wirklich eine Operation ausführen soll. Zusätzlich sind die Prozesse sensitiv zum asynchronen Hardware- und zum synchronen Software-Reset, deren Aktivierung für die Initialisierung der Variablen und Ausgangssignale genutzt wird. Diese Vorgehensweise ist möglich, da es sich nicht um ein synthetisierbares VHDL-Modell handelt. Sie ist sinnvoll, da die Simulationsgeschwindigkeit aufgrund der geringeren Zahl von Prozessaktivierungen verbessert wird (vgl. Kap. 3.3.4).

Die Fließkomma-Funktionen wurden auf einer relativ hardwarenahen Ebene (std_logic) implementiert ohne Verwendung des Datentyps real, um Abhängigkeiten von Bibliotheken der VHDL-Simulatoren zu vermeiden. Rundungsfehler hätten dazu geführt, dass die Simulation des VHDL-Modells in verschiedenen Simulatoren inkonsistente Ergebnisse liefern würde. Darüber hinaus verwenden die DSPs der ADSP-2106x-Familie neben dem IEEE-Single-Precision Fließkomma-Format (24-Bit Mantisse, 8-Bit Exponent, 1 Vorzeichenbit) auch ein Extended-Precision Fließkomma-Format (32-Bit Mantisse, 8-Bit Exponent, 1 Vorzeichenbit), das so nicht direkt im IEEE-Standard 754-1985 [70] verzeichnet ist. Als Single-Extended Fließkomma-Format ist dort festgehalten, dass der Exponent 11-Bit umfassen muss. Daraus folgt, dass dafür auch kein VHDL-Package verfügbar gewesen wäre und damit eine eigene Implementierung notwendig war. In Listing 6-8 ist als Beispielimplementierung für das VHDL-Modell der ALU die Berechnung des Absolutwertes eines Fließkomma-Operanden aufgeführt, anhand derer nachfolgend einige wesentliche Implementierungstechniken erläutert werden.

Jede Operation innerhalb der ALU setzt oder löscht Flags unabhängig oder in Abhängigkeit von den Operanden. Alle zugehörigen Zuweisungen zu den Flag-Signalen werden zu Beginn der Operation standardmäßig mit den entsprechenden Signalwerten durchgeführt (Zeilen 2 bis 12 in Listing 6-8). Sind aufgrund der Operanden andere Signalwerte als die Standardwerte zuzuweisen, wird die Möglichkeit in VHDL ausgenutzt, dass bei sequentiellen Signalzuweisungen die letzte Zuweisung den Signalwert bestimmt. Alle vorherigen Zuweisungen werden überschrieben (vgl. Zeile 38 in Listing 6-8). Das vereinfacht den VHDL-Code und wirkt sich nicht negativ auf die Simulationsgeschwindigkeit aus. Weiterhin ist zu erkennen, dass außer dem Datentyp std_logic auch der Datentyp integer verwendet wird. Dieser ist ebenfalls simulatorunabhängig und kann ohne Einschränkungen verwendet werden.

Im Anschluss an die ALU wurden mit dem Programmsequenzer (**program_sequencer**) und den Adressgeneratoren DAG1 (**dag1**) und DAG2 (**dag2**) das Teilmodell für den Core-Prozessor des DSP (**core_processor**) komplettiert. Der zentrale Ablauf eines Programms wird im Programmsequenzer gesteuert. Im zugehörigen VHDL-Entity wird die 3-stufige Pipeline, die Interruptbehandlung, der Befehls-Cache, der Timer und der Befehlsdekoder modelliert. Mit dem Modell für den I/O-Prozessor (**io_processor**) inklusive dem Modell für den DMA-Controller (**dma_controller**) war es möglich, die Bootphase des DSP zu simulieren. Außerdem werden die seriellen Ports und alles Weitere modelliert, was mit dem

```
1.   when alu_float_abs =>
2.        -- always set
3.        af_flag <= '1';
4.        -- always cleared
5.        av_flag <= '0';
6.        an_flag <= '0';
7.        ac_flag <= '0';
8.        au_flag <= '0';
9.        -- default, must be overwritten below
10.       az_flag <= '0';
11.       as_flag <= rx(39);
12.       ai_flag <= '0';
13.
14.       rx_exponent := conv_integer(rx(38 downto 31));
15.       if(mode1_rnd32 = '1') then
16.            -- truncate input to 32-bit boundary
17.            rx_fraction := rx(30 downto 8) & X"00";
18.       else
19.            rx_fraction := rx(30 downto 0);
20.       end if;
21.       if(rx_exponent = 255) then
22.            if(conv_integer(rx_fraction) = 0) then
23.                 -- +Infinity
24.                 result := '0' & rx(38 downto 31) & rx_fraction;
25.            else
26.                 ai_flag <= '1';    -- input operand is a NAN
27.                 if(mode1_rnd32 = '1') then
28.                      -- round to 32-bit boundary
29.                      result := X"FFFFFFFF00";
30.                 else
31.                      result := (others => '1');
32.                 end if;
33.                 -- overwrite negative flag
34.                 as_flag <= '0';
35.            end if;
36.       elsif(rx_exponent = 0) then      -- denormal or +/-Zero
37.            result := (others => '0');  -- flush to +Zero
38.            az_flag <= '1';
39.       else
40.            result := '0' & rx(38 downto 31) & rx_fraction;
41.       end if;
42.       alu_result1 <= result;
```

Listing 6-8. Berechnung des Absolutwertes eines Fließkomma-Operanden im VHDL-Modell der ALU der ADSP-2106x-Familie.

externen Zugriff auf den Systembus und mit dem internen Zugriff auf den Speicher über den I/O-Bus im Zusammenhang steht. Das Modell für das Host-Interface (**host_interface**) dient der Simulation des Zugriffs eines Host auf den DSP. Das Multiprozessor-Interface (**multiprocessor_interface**) kann für eine Simulation in einem Multiprozessor-Szenario genutzt werden, in der maximal sechs Bausteine miteinander verknüpft sind. Die Multiprozessor-ID der Instanz des DSP wird dem Entity als Generic-Parameter übergeben. Davon abhängig ist das Protokoll, mit dem die Prozessoren die Buszugriffe koordinieren.

Im VHDL-Entity für den On-Chip-Speicher der ADSP-2106x-Familie (**dp_sram**) ist die Möglichkeit eingebaut, zu Beginn der Simulation den Speicher über Text-I/O mit dem

Inhalt zu initialisieren, der nach einer Bootphase des DSP vorhanden wäre. Damit kann die Bootphase bei der Simulation übersprungen werden. Die Textdatei, die dazu einzulesen ist, wird von *Mephisto* aus der ausführbaren Datei für den DSP generiert. Ein Unterscheidungsmerkmal der Bausteine der ADSP-2106x-Familie ist die Größe und Organisation des On-Chip-Speichers. Da das VHDL-Modell gleich für die gesamte Familie gültig sein sollte, wurde im VHDL-Entity dp_sram ein Generic-Parameter implementiert, über den die Auswahl des konkreten Bausteins der Familie erfolgt (siehe Listing 6-9). In Abhängigkeit vom instantiierten Generic-Parameter wird ein zugehöriger sequentieller Prozess im VHDL-Code mit Hilfe der generate-Anweisung generiert. In den jeweiligen Prozessen für die drei möglichen Bausteine der Familie ist schließlich die Organisation, der Zugriff und die Größe des On-Chip-Speichers individuell implementiert. Zur Laufzeit des Systems, bei der ein bestimmter Typ der ADSP-2106x-Familie instantiiert ist, werden damit keine unnötigen sequentiellen Prozesse oder andere Anweisungen ausgeführt. Dies wirkt sich positiv auf die Simulationsgeschwindigkeit aus.

```
1.   entity dp_sram is
2.          generic(  -- 0 -> 21060; 1 -> 21061; 2 -> 21062
3.                    adsp2106x : integer := 1;
4.                    ....);
5.   end dp_sram;
6.   ....
7.   adsp21060: if (adsp2106x = 0) generate
8.   begin
9.          sram: process(n_reset, soft_reset, clkin)
10.         ....
11.  end generate;
12.  ....
13.  adsp21061: if (adsp2106x = 1) generate
14.  begin
15.         sram: process(n_reset, soft_reset, clkin)
16.         ....
17.  end generate;
18.  ....
```

Listing 6-9. Generierung von Prozessen im Entity für den On-Chip-Speicher der ADSP-2106x-Familie in Abhängigkeit von der Auswahl eines konkreten Bausteins.

Der Speicher selbst wird als lokale Variable modelliert, die ein Array des Typs bit_vector darstellt. Die Verwendung des Typs bit_vector spart Speicher zur Laufzeit der Simulation ein, da nicht die gesamte 9-wertige Logik des Typs std_logic_vector gespeichert werden muss. Eine dynamische Verwaltung des Speichers in der Simulation ist an dieser Stelle zu aufwendig. Die Idee hierbei wäre gewesen, nur die von Null verschiedenen Speicherstellen in dynamisch alloziierten Speicherblöcken einstellbarer Größe zusammen mit der Adresse und einem Zeiger auf den nächsten gültigen Speicherblock abzuspeichern. Problem dabei ist der Verwaltungsaufwand, der die Simulationsgeschwindigkeit verringert hätte. Diese Methodik eignet sich eher für die Simulation sehr großen Speichers (z. B. 4M x 32) mit wenigen lesenden/schreibenden Zugriffen. Da der On-Chip-Speicher der ADSP-2106x-Familie maximal 256k x 16 (ADSP-21060) beträgt, ist eine Simulation des Speichers in einem Block sinnvoll.

Mit der Systemintegration wurden alle Entities im Top-Level-Entity **target_dsp** zusammengefasst. Ab diesem Zeitpunkt war es möglich, komplette Testprogramme im VHDL-Modell ablaufen zu lassen. Im Anhang A.3.1 ist die Deklaration vom Entity target_dsp abgebildet. Zu erwähnen sei an dieser Stelle die Deklaration einiger wichtiger Generic-Parameter (Listing 6-10). Mit dem Parameter **booting** kann die komplette Bootphase des DSP übersprungen werden, wenn dieser den Wert FALSE zugewiesen bekommt. **Generic_Id** bestimmt die ID in einer Multiprozessorumgebung, in der sich der DSP befinden kann. Zur Unterscheidung der Bausteine der ADSP-2106x-Familie im VHDL-Modell ist der Generic-Parameter **adsp2106x** vorgesehen, der z. B. zum Entity dp_sram weitergleitet wird. Mit **InstancePath** wird dem Modell der Pfad zur Instanz mitgeteilt, damit Meldungen (assertion reports) für mehrere Instanzen des Modells unterschieden werden können. Alle Operationen zur Überprüfung des Zeitverhaltens der Eingänge sind mit Hilfe des Parameters **TimingChecksOn** für die Simulation an- bzw. abschaltbar. Mit **MsgOn** und **XOn** können gezielt für einzelne Instanzen die Generierung von Meldungen und/oder Warnungen über undefinierte Eingangssignale abgestellt werden. **SimCondition** schließlich bestimmt die anzuwendenden Timing-Randbedingungen, dazu zählen BestCase, WorstCase und Typ-Case.

```
1.   entity target_dsp is
2.     generic( booting: BOOLEAN := TRUE;
3.              -- Multiprocessing ID as generic
4.              generic_id: integer := 1;
5.              -- 0 -> 21060; 1 -> 21061; 2 -> 21062
6.              adsp2106x: integer := 1;
7.              ...
8.              -- generic control parameters
9.              InstancePath: STRING  := "SHARC DSP:";
10.             TimingChecksOn: BOOLEAN := FALSE;
11.             MsgOn: BOOLEAN := TRUE;
12.             XOn: BOOLEAN := TRUE;
13.             TimingModel: STRING  := "UNIT";
14.             SimCondition: SimConditionType := WorstCase);
15.             ...
16.  end entity;
```

Listing 6-10. Ausschnitt aus der Generic-Deklaration des Entity target_dsp.

Für die Definition der Verzögerungszeiten wurde der VITAL-Standard (VHDL Initiative Towards ASIC Libraries) gewählt [122]. Die VHDL-Packages des Standards sind speziell für eine geringe Simulationsrechenzeit optimiert und deshalb für diese Aufgabe prädestiniert. Zusätzlich werden sie von den meisten VHDL-Simulatoren in einer beschleunigten Version (pre-compiled) in die Simulation eingebunden. Für das Entity target_dsp wurden aus den VITAL-Bibliotheken z. B. die Funktionen VitalPeriodPulseCheck, VitalSetupHoldCheck und VitalSignalDelay benutzt. Mit diesen werden die Eingangssignale des Modells auf die Einhaltung des Zeitverhaltens überprüft.

Die **Verifikation** des Modells für die ADSP-2106x-Familie nahm etwa 75% der Entwicklungszeit in Anspruch. Zum Beispiel wurden für die Implementierung der IEEE Fließkomma-Addition/Subtraktion (Single/Extended Precision) innerhalb des Modells der ALU des SHARC-DSP 290 VHDL-Codezeilen benötigt. Zur Verifikation der Funktionalität der

genannten Funktionen wurden 8755 VHDL-Codezeilen innerhalb der selbsttestenden Testbench für die ALU implementiert, also rund 30 Mal mehr als für die eigentlichen ALU-Operationen. Weitere Zahlenangaben zum Umfang einiger Funktionen der ALU des ADSP-2106x-Modells, die das Verhältnis zwischen Implementierungs- und Testaufwand zeigen, sind in Tabelle 7 aufgeführt. Die Simulation der Testbench für die Fließkomma-Addition/Subtraktion benötigte auf einer wenig ausgelasteten SunU10-300 Workstation 7 Tage (CPU Time 582.832 s).

Tabelle 7: Gegenüberstellung der Anzahl der Codezeilen für das Modell einerseits und für die selbsttestende Testbench andererseits.

Funktion/Operation	Implementierung [Anzahl Zeilen]	Testbench [Anzahl Zeilen]	Beschreibung
fixed-point add/sub	48	226	Fixed-point Addition oder Subtraktion
fixed-point dual add/sub	87	313	Fixed-Point Addition und Subtraktion parallel
floating-point add/sub	290	8755	Floating-Point Addition oder Subtraktion
floating-point dual add/sub	445	7479	Floating-Point Addition und Subtraktion parallel
floating-point compare	181	1405	Floating-Point Vergleich
floating-point min/max	152	2971	Floating-Point Minimum oder Maximum

Für die drei Recheneinheiten der ADSP-2106x-Familie wurden separate Testbenches in VHDL implementiert. Insgesamt umfassen die Testbench für die ALU ca. 40.000, für den Multiplizierer ca. 15.000 und für den Barrel-Shifter ca. 1000 Codezeilen. Da jede Funktion der drei Recheneinheiten prinzipiell von drei Control-Flags abhängt (ALUSAT, TRUNC und RND32) und zusätzlich jede Fließkomma-Operation mit Single- oder Extended-Precision arbeiten kann, ist der Testpattern-Raum entsprechend groß. Darüber hinaus generieren die Funktionen unterschiedliche Flags (u. a. Overflow, Zero), die in der Testbench ebenso abgeprüft werden mussten. Daher wurden Äquivalenzklassen im Lösungsraum der Funktionen abgegrenzt, deren Flags und Ergebnisse identisch bzw. in Abhängigkeit von den Operanden berechenbar sind, so dass eine Partitionierung der Eingabedaten erreicht wurde. Die partitionierten Eingabedaten wurden dann in Schleifenkonstrukte innerhalb der Testbench gepackt und nur deren Anfang und Ende (Grenzbereiche der Äquivalenzklassen) in der Simulation überprüft. Listing 6-11 zeigt eine solche Schleife innerhalb der Testbench für die Fließkomma-Addition der ALU. In diesem Fall sind die Beträge der beiden Operanden gleich, jedoch die Vorzeichen entgegengesetzt zueinander. Das Ergebnis der Fließkomma-Operation ist Null, im gesamten zulässigen Wertebereich der Operanden (weder Denormal noch +/-Unendlich).

Zusätzlich zur Simulation der Grenzbereiche der Äquivalenzklassen wurden Zufallszahlen für die Operanden generiert und die Ergebnisse der Operationen auf dem EZ-KIT LITE Board in die VHDL-Testbench über konstante Tabelleneinträge eingebracht, um möglicher-

```
1.   -- rx positive and ry negative; rx = ry; no underflow; result = +Zero
2.   for i in 1 to 254 loop -- rx & ry exponent loop
3.        -- rx positive
4.        rx <= '0' & conv_std_logic_vector(i,8) & "000" & X"0000000";
5.        -- ry negative
6.        ry <= '1' & conv_std_logic_vector(i,8) & "000" & X"0000000";
7.        for j in 0 to 300 loop -- rx & ry mantissa loop; max 0x7fffffff
8.             check_pos_zero_in_all_cases("Floating-Point Add");
9.             rx <= rx + 1;
10.            ry <= ry + 1;
11.       end loop;
12.
13.       -- switch to end of intervall
14.       rx(30 downto 0) <= (others => '1');
15.       ry(30 downto 0) <= (others => '1');
16.       for j in 0 to 300 loop -- rx & ry mantissa loop; max 0x7fffffff
17.            check_pos_zero_in_all_cases("Floating-Point Add");
18.            rx <= rx - 1;
19.            ry <= ry - 1;
20.       end loop;
21.  end loop;
```

Listing 6-11. Schleifenkonstrukt der ALU-Testbench für einen abgegrenzten Lösungsraum der Fließkomma-Operation.

weise nicht abgedeckte Testfälle aufzudecken. Das im HDL-Simulator ModelSim eingebaute Werkzeug Code-Coverage wurde dabei zur Überprüfung der Testabdeckung des VHDL-Quelltextes des Modells genutzt. Ein vollständiges Testen schon nur einer Fließkomma-Operation zweier Operanden hätte bei einer angenommenen Rechenzeit von 30 ns (ein Taktzyklus bei 33 MHz) pro Operanden/Ergebnis-Paar maximal zwischen 2^{41} und 2^{42} Tage benötigt, siehe Formel (6.1). Der Berechnung zugrunde gelegt sind zwei 40-Bit breite Operanden (Extended Precision) mit zusätzlich drei Control-Flags, ohne die Semantik einer Operation, wie beispielsweise bei einer Fließkomma-Addition das Kommutativgesetz, oder unzulässige Kombinationen der Control-Flags zu berücksichtigen.

$$t = (2^{40})^2 \cdot 2^3 \cdot 30\text{ns} = 2^{83} \cdot 30\text{ns} = 2^{53} \cdot 30\text{s} = (2^{41} \cdot 1,5)\text{Tage} \qquad (6.1)$$

Das übergreifende Vorgehen der Verifikation des VHDL-Modells sah so aus, dass für die einzelnen Funktionen der Recheneinheiten unter Zuhilfenahme des Datenblattes Testprogramme geschrieben, und deren Operanden und Ergebnisse im internen Speicher des DSP abgelegt wurden. Der Vergleich der Ergebnisse vom entwickelten Modell und denen des EZ-KIT LITE Boards und Simulators erfolgte nach einem Memory Dump der jeweiligen Speicherbereiche (Bild 6-11). Dabei wurden des öfteren Lücken im Datenblatt oder auch Fehler im EZ-KIT LITE Simulator entdeckt. Ohne die drei Referenzen (Datenblatt, EZ-KIT LITE Board und Simulator) wäre ein Zustandekommen des Modells sehr schwierig gewesen.

Die Zyklentreue des VHDL-Modells wurde mit Hilfe des programmierbaren Zählers (Timer) des DSP verifiziert, der in den Testprogrammen an den entsprechenden Stellen zum Einsatz kam. Eine mögliche Befehlszeilenfolge dafür wird in Listing 6-12 gezeigt, bei der die Zyklentreue eines Speicherzugriffs über den Programmspeicherbus überprüft wird.

Über den Bus wird normalerweise die nächste Instruktion in die Pipeline eingelesen (Fetch). Wird dieser jedoch für einen normalen Speicherzugriff verwendet und die nächste Instruktion ist nicht im Cache gespeichert, dann wird zunächst der Speicherzugriff ausgeführt und im darauffolgenden Takt die Instruktion, mit gleichzeitiger Speicherung im Cache, eingelesen. Diese Verzögerung um einen Takt muss bei einem zyklengetreuen Modell mitsimuliert werden!

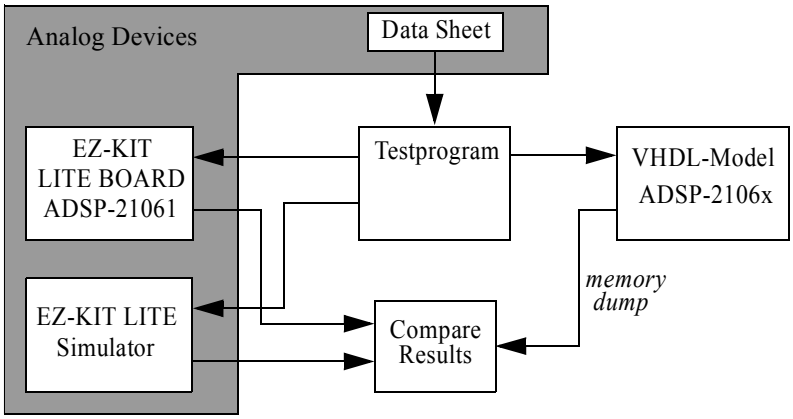


Bild 6-11. Verifikation des VHDL-Modells für die ADSP-2106x-Familie.

```
     a                              b
1.   tcount=0x64;                   tcount=0x64          // initial tcount
2.   bit set mode2 TIMEN;           bit set mode2 TIMEN
3.   nop;                           nop;                 // timer enabled
4.   r1=pm(test);  // PM            r1=dm(test);  // DM
5.   bit clr mode2 TIMEN;           bit clr mode2 TIMEN;
6.   nop;                           nop;                 // timer disabled
7.   r1=tcount;                     r1=tcount;           // read tcount
8.   // check: tcount==0x60         // check: tcount==0x61
```

Listing 6-12. Verifikation der Zyklentreue des VHDL-Modells bei einem Speicherzugriff **a** über den Programmspeicherbus (PM) mit zusätzlicher Verzögerung um eine Taktperiode und **b** über den Datenspeicherbus (DM) (tcount ist um eins größer als bei **a** und damit wurde ein Takt weniger benötigt).

7 Beispiel-Implementierungen

In diesem Kapitel werden einige der im und mit *FADES* implementierten Benchmarks und Beispiele vorgestellt und zugehörige Ergebnisse aufgezeigt. Zunächst wird erläutert, mit welchen Hilfsmitteln die Messungen durchgeführt wurden und wie der Messaufbau für die Benchmarks allgemein gestaltet war. Anschließend wird beginnend mit kleineren Benchmarks im weiteren Verlauf des Kapitels gezeigt, wie mit Hilfe des CORDIC-Algorithmus das FPGA als Co-Prozessor genutzt wird und wie beim Beispiel MPEG1-Layer3-Player zusätzliche extern notwendige Hardware im FPGA flexibel, sprich programmierbar, zusammengefasst werden kann.

7.1 Aufbau der Messumgebung

Die Messungen wurden durchgeführt um zu zeigen, wieviel Zeit für die VHDL-Board-Level-Simulation zur HW/SW-CoVerifikation für verschiedene Testfälle benötigt wird. Dies ist ein wichtiges Kriterium für die gesamte Entwicklungszeit eines Algorithmus bzw. einer Anwendung, die maßgeblich die Time-to-Market eines Produkts mitbestimmt.

Für die VHDL-Board-Level-Simulation wurde der kommerzielle HDL-Simulator ModelSim der Elite Edition in der Version 5.6d von Mentor Graphics (ehemals Model Technology) eingesetzt. Für alle Messungen kam der Befehl simstats im HDL-Simulator unter Linux und Unix zum Einsatz. Da dieser Befehl Routinen des darunter liegenden Betriebssystems zur Messung der benötigten CPU-Zeit verwendet, funktioniert er nicht unter Windows, so dass hier keine vergleichbaren Messungen möglich waren [86]. Experimentell wurde eine Messungenauigkeit von ca. ±3 % ermittelt (Tabelle 8).

Tabelle 8: Experimentelle Ermittlung der Messungenauigkeit des im VHDL-Simulator ModelSim integrierten Befehls simstats.

Messung/Auswertung	Simulationszeit [s] bei 1 ms simulierter Zeit		Simulationszeit [s] bei 100 ms simulierter Zeit	
1. Messung	36,86		5260,62	
2. Messung	38,3		5429,69	
3. Messung	36,69		5106,61	
4. Messung	36,09		5147,2	
Arithmetisches Mittel	36,985		5236,03	
max. Messabweichungen +- [%]	+3,55	-2,42	+3,7	-2,47

7.2 Benchmarks

Da keine frei verfügbaren Benchmarks für solche Systeme existieren, wurden beispielhaft ein 127-Tap FIR-Filter, die diskrete Fourier Transformation (DFT) und die Fast Fourier Transformation (FFT) untersucht. Alle Beispiele laufen auf dem DSP und nutzen in einigen Fällen das FPGA als Quelle und Senke der Audiodaten, die von der digitalen S/PDIF-Schnittstelle kommen. Tabelle 9 zeigt die gemessenen Simulationszeiten für die implementierten Beispiele. Simuliert wurde das Gesamtsystem ohne Bootphase und ohne die beiden Flash-Bausteine (Simulationsart 3 - Siehe Kap. 6.4.2). Es werden die Zeiten angegeben, die die CPU mit der Berechnung der Simulation des jeweiligen Beispiels ausgelastet war, ohne Berücksichtigung von Initialisierungsphasen, die jedoch nicht mehr als 10 s CPU-Zeit in Anspruch nahmen.

Tabelle 9: Simulationszeiten der VHDL-Board-Level-Simulation für einige Benchmark-Beispiele.

Benchmark	Simulierte Zeit [ns]	Simulationszeit **ohne** Debug-Oberfläche [s]		Simulationszeit **mit** Debug-Oberfläche [s]		Anzahl Eingangswerte	Anmerkung
		1 GHz-Athlon	SunU10-300	1 GHz-Athlon	SunU10-300		
127-Tap FIR-Filter	42.485.745	1.231,65	3.303,98	16.142,4	54.272,15	1000 in Memory	C
127-Tap FIR-Filter	5.500.800	163,45	516,77	2.631,23	11.595,55	1000 in Memory	fir()*
127-Tap FIR-Filter	13.491.420	381,82	1.133,5	4.348,69	15.338,1	500 von S/PDIF	fir()*
256-Point Radix-2 DIT complex FFT	230.160	7,36	20,53	79,83	318,74	256 in Memory	Asm
256-Point complex FFT	443.040	11,13	34,26	146,74	589,11	256 in Memory	cfft()*
256-Point real FFT	388.620	9,67	29,96	122,45	469,63	256 in Memory	rfft()*
64-Point DFT	261.210	9,93	27,72	165,33	718,30	64 in Memory	Asm

* aus C-Bilbliothek von Analog Devices

Deutlich erkennbar ist ein überproportionaler Anstieg der Simulationszeiten (Skalierungsfaktor 13), wenn die Debug-Oberfläche für den ModelSim während der Simulation aktiv ist. Grund dafür sind viele Signale, deren Änderungen im Simulationskern ein Aufruf der implementierten Funktionen in den Tcl/Tk-Quelltexten bedingen und deren Ausführung durch die Interpretation der Quelltexte länger dauert.

Weiterhin ist der Skalierungsfaktor 3 zwischen der Simulation auf der Sun-Workstation und der Simulation auf dem Athlon-PC zu erkennen, der in etwa dem Verhältnis der Taktraten der beiden Systeme entspricht (300 MHz ⇔ 1 GHz). Mit diesem Skalierungsfaktor lässt sich ungefähr der Zeitgewinn einer Simulation auf einem schnelleren Rechner ableiten, da mittlerweile selbst 1 GHz-Prozessoren nicht mehr dem aktuellen Stand der Technik entsprechen.

Aus Tabelle 9 ergibt sich darüber hinaus für die Anzahl der simulierten Instruktionen pro Sekunde in der verwendeten HW/SW-CoSimulation ein ungefährer Wert von 1.300, unter der idealen Voraussetzung, dass eine Instruktion innerhalb einer Taktperiode von 30 ns (33 MHz) abgearbeitet wird.

Die „straightforward"-Methode der Implementierung des 127-Tap FIR-Filters in Zeile 1 von Tabelle 9 zeigt, dass die Nutzung einer optimierten Bibliotheksfunktion in Zeile 2 eine erhebliche Verbesserung der Laufzeit des FIR-Filters bedeuten kann. Der verwendete C-Compiler hat hierbei den resultierenden Maschinencode nicht geeignet optimieren können. Gibt es zeitkritische Abschnitte im DSP-Code, sind diese mittels manuell optimiertem Assembler-Code zu implementieren. Zu den zeitkritischen Abschnitten zählen oft wenige Zeilen C-Code, die in datenstromorientierten Applikationen sehr häufig ausgeführt werden und deren Ineffizienz sich deshalb sehr stark bemerkbar macht. Die notwendigen Assembler-Optimierungen galten und gelten heute immer noch generell für DSP-spezifische Applikationen, da in der Regel wenige praxistaugliche C-Compiler für die speziellen DSP-Architekturen existieren [55]. Die zeitkritischen Stellen in der Software zu finden, ist eine Aufgabe der linearen statistischen Analyse (Profiling), die innerhalb der Debug-Oberfläche der HW/SW-CoSimulation implementiert ist.

7.3 CORDIC

Für dieses Beispiel wurde das FPGA als Co-Prozessor für den DSP eingesetzt. Mit Hilfe des CORDIC-Algorithmus [10] wurde die Berechnung der Sinus-/Cosinuswerte eines Winkels mit einer Genauigkeit von 32-Bit in das FPGA, basierend auf einem Core von Richard Herveille [91], ausgelagert. Der Operand und die Ergebnisse wurden mit Hilfe eines Handshake-Verfahrens asynchron zwischen DSP und FPGA übertragen, mit einem Taktzyklus als zusätzlicher Verzögerung. Der Transfer eines Datenwertes benötigte somit zwei Taktzyklen. Für die Berechnung eines Sinus/Cosinus-Paares waren zunächst der Winkel als Operand und danach die beiden Ergebniswerte zu übertragen. Mit der in Listing 7-1 gezeigten XML-Spezifikation wurde die Interface-Synthese für das CORDIC-Beispiel durchgeführt.

```
1.   <?xml version="1.0"?>
2.   <shared_codesign>
3.   <variable name="operand" type="integer" mode="rw"/>
4.   <variable name="sin_result" type="integer" mode="r" />
5.   <variable name="cos_result" type="integer" mode="r" />
6.   </shared_codesign>
```

Listing 7-1. XML-Eingabe-Spezifikation für die Interface-Synthese im Fall des CORDIC-Beispiels.

Der schreib- und lesbare Operand (Modus rw = Read/Write) und die lediglich lesbaren Ergebniswerte für den Sinus und den Cosinus, alle vom Typ integer, stellen hier die zu übertragenden Werte dar.

Da beide Hauptkomponenten der Architektur von *Faust* mit demselben Systemtakt getaktet werden, macht es keinen Sinn elementare Rechenoperationen, z. B. die IEEE Fließkomma-Addition, des DSP in das FPGA auszulagern (feinkörnige HW/SW-Partitionierung vgl. Kap. 2.6.1). Der zusätzliche zeitliche Aufwand, der notwendig ist, um die Operanden und Ergebnisse einer Operation zu transferieren, würde den zeitlichen Gewinn durch die parallele Ausführung im FPGA eliminieren. Deshalb ist es nur sinnvoll, größere zusammenhängende Blöcke, wie zum Beispiel die Software-Berechnung der Sinus-/Cosinuswerte eines Winkels, in das FPGA auszulagern (grobkörnige HW/SW-Partitionierung). Die C-Bibliotheksfunktionen, wenn es diese gibt, sind unter Umständen suboptimal, eine Realisierung in Hardware an dieser Stelle eventuell günstiger. Parallel dazu kann das Programm auf dem DSP normal weiterlaufen, bis das Ergebnis des ausgelagerten Blocks berechnet ist.

```
1.   Release 4.1.03i - Map E.33
2.   Xilinx Mapping Report File for Design 'target_fpga'
3.
4.   Design Information
5.   ------------------
6.   Target Device  : xv300
7.   Target Speed   : -4
8.
9.   Design Summary
10.  --------------
11.     Number of errors:      0
12.     Number of warnings:    1
13.     Number of Slices:              1,681 out of  3,072   54%
14.     Number of Slices containing
15.        unrelated logic:                0 out of  1,681    0%
16.     Number of Slice Flip Flops:    3,102 out of  6,144   50%
17.     Total Number 4 input LUTs:     3,153 out of  6,144   51%
18.        Number used as LUTs:                     3,140
19.        Number used as a route-thru:                13
20.     Number of bonded IOBs:            94 out of    166   56%
21.     Number of GCLKs:                   2 out of      4   50%
22.     Number of GCLKIOBs:                1 out of      4   25%
23.  Total equivalent gate count for design:  61,029
24.  Additional JTAG gate count for IOBs:  4,560
25.
26.  ---------------------------------------------------------------------
27.    Constraint                       | Requested   | Actual  | Logic
28.                                     |             |         | Levels
29.  ---------------------------------------------------------------------
30.    NET "C2391/IBUFG" PERIOD = 30 nS | 30.000ns    |27.818ns | 6
31.    HIGH 50.000000 %                 |             |         |
32.  ---------------------------------------------------------------------
33.
34.  All constraints were met.
```

Listing 7-2. Ausschnitt aus den Place&Route-Ergebnissen des CORDIC-Beispiels für das FPGA.

Die Option der digitalen PLL (DLL), die die Virtex-Architektur bietet, kann für eine Verdopplung des Systemtaktes im FPGA genutzt werden und somit eine Berechnung bis zu einem Faktor zwei, auch gegenüber einer Ein-Takt-Realisierung im DSP, beschleunigen. Diese Möglichkeit wurde jedoch bei der Implementierung der CORDIC-Funktionalität im XCV300-4 nicht berücksichtigt, da der längste Pfad laut statischer Timinganalyse bereits 27,818 ns von 30 ns eines Taktzyklus bei 33 MHz benötigt (Listing 7-2).

```
1.   0x00024000: _sin_result_fpga: 0x3f3504f4
2.   0x00024001: _cos_result_fpga: 0x3f3504f4
3.   0x00024002: _sin_result_dsp: 0x3f3504f3
4.   0x00024003: _cos_result_dsp: 0x3f3504f3
5.   0x00024004: _tcount_fpga: 0x0000003d
6.   0x00024005: _tcount_dsp: 0x00000075
```

Listing 7-3. Memory Dump der Ergebnisse des Moduls cordic.c, unabhängig davon, ob diese nach der HW/SW-CoSimulation oder nach der Berechnung auf der Prototyp-Hardware durchgeführt wurde.

Die Ergebnisse der Berechnung im FPGA wurden mit den Werten der C-Bibliotheksfunktionen sin() und cos() auf dem DSP verglichen. Der Quelltext für das gesamte Modul cordic.c mit den Funktionen main() und sin_cos_fpga() ist im Anhang A.4.1 aufgelistet und ein Memory Dump eines Bereiches des On-Chip-Speichers des DSP ergab die in Listing 7-3 gezeigten Ergebnisse sowohl nach der HW/SW-Cosimulation als auch nach der Berechnung auf der Prototyp-Hardware. Zunächst ist zu erkennen, dass die im DSP berechneten Werte minimal von denen im FPGA berechneten Werten abweichen. Die Abweichung beträgt lediglich 6×10^{-8}. Weiterhin wurden die Taktzyklen der Berechnung im DSP (_tcount_dsp) und derjenigen im FPGA (_tcount_fpga) in der Software mit Hilfe des im DSP vorhandenen Timers gemessen. Dabei wurden für die reine Softwareausführung 117 (0x75) Taktzyklen und für die Hardwareberechnung einschließlich der Zeit für den Austausch der Operanden- und Ergebniswerte 61 (0x3d) Taktzyklen bestimmt. Die 32 Taktzyklen, die die eigentliche Hardwareberechnung benötigt (ein Takt pro einem Bit Genauigkeit), bleiben in dieser Beispielimplementierung ungenutzt, da der DSP 32 NOPs innerhalb der Funktion sin_cos_fpga() ausführt. Die Auslagerung belegte 54 % der Slices des FPGA, das in etwa 61.029 Gatteräquivalenten entspricht (Listing 7-2).

Tabelle 10: Benchmark-Ergebnisse des CORDIC-Beispiels.

Simulationsart	Simulierte Zeit [ns]	Pre Synthese [s]	Post Synthese [s]	Post Synthese + Timing [s]
komplette Bootphase	88.655.595	7.918,89	11.285,2	12.311,5
ohne Bootphase, mit Flash-Bausteine	1.083.705	407,41	2.469,32	2.291,75
ohne Bootphase, ohne Flash-Bausteine	1.083.705	385,75	2.435,76	2.264,64

Auch hier wurde der Befehl simstats im ModelSim für die Messung der Simulationszeiten eingesetzt, jedoch lief die Simulation nur auf dem 1 GHz-Athlon PC unter Linux. In Tabelle 10 auf Seite 119 werden die Simulationszeiten für die drei Simulationsarten auf den Entwurfsstufen „Pre Synthese“, „Post Synthese“ und „Post Synthese with Timing“ jeweils mit aktivierter Debug-Oberfläche aufgezeigt.

Gemessen wurden die Simulationszeiten für die VHDL-Board-Level-Simulation des CORDIC-Beispiels (siehe Kap. 6.4.2):

1. inklusive der kompletten Bootphase,
2. ohne Bootphase, mit den beiden Flash-Bausteinen,
3. ohne Bootphase, ohne Flash-Bausteine.

In Tabelle 10 ist der Skalierungsfaktor 6 zwischen der Simulation vor und nach der FPGA-Synthese zu erkennen. Der zusätzliche Zeitaufwand sollte aber nicht vor der Durchführung der Simulation nach der Synthese abhalten. Oft passiert es dem Entwickler, Konstrukte im VHDL-Code zu verwenden, die zwar der Überprüfung einer funktionalen Simulation standhalten, jedoch in der Synthese zu nicht korrekten Ergebnissen führen. Ein Beispiel dafür ist das Fehlen eines Signals in der Sensitivitätsliste eines Prozesses, das dazu führen kann, dass die synthetisierte Schaltung ein anderes Verhalten zeigt als in der funktionalen Simulation (vgl. Kap. 3.3.4, Listing 3-4).

Die Simulation nach der Synthese mit Timing ist nicht überflüssig, auch wenn es die statische Timinganalyse gibt. Bei der statischen Timinganalyse wird der komplette Entwurf im FPGA in rein kombinatorische Pfade zerlegt. Die Pfade beginnen an FPGA-Eingängen oder an den Ausgängen der Flipflops und enden an FPGA-Ausgängen oder an den Eingängen der Flipflops. Die statische Timinganalyse berechnet das Zeitverhalten aller Pfade. Der Pfad mit der größten Verzögerung wird kritischer Pfad genannt und wird direkt für die Angabe des maximal möglichen Systemtaktes verwendet. Handelt es sich um eine taktsynchrone Schaltung, die lediglich innerhalb des FPGA arbeitet, reicht die statische Timinganalyse für die Angabe des maximal erreichbaren Taktes der platzierten und verdrahteten Schaltung aus. Werden jedoch externe Signale über die I/O-Pads des FPGA angesteuert oder gibt es ein externes asynchrones Handshake-Verfahren, ist die Simulation mit Timing nach der Synthese unerlässlich. Es wurde beim implementierten CORDIC-Beispiel bei der Simulation nach der Synthese mit Timing ein Fehler im Zeitverhalten der Steuersignale zwischen FPGA und DSP aufgedeckt, der daraufhin behoben werden konnte. Dafür wurde die FPGA-Synthese mit angepassten Vorgaben (Constraints) für die kritischen Signalpfade wiederholt. Der Fehler wurde aufgrund der Möglichkeit der Überprüfung der Ergebnisse im Speicher sowohl in der Debug-Oberfläche der Simulation als auch in Echtzeit auf der Prototyp-Hardware nachgewiesen.

7.4 MPEG1-Layer3-Player

Als ein etwas größeres und praxisrelevanteres Beispiel wurde ein MPEG1-Layer3-Player (MPEG1-Layer3: nachfolgend kurz MP3) implementiert. MP3 ist eine Kompressions-

technik für Audiodaten mit einem ungefähren Kompressionsfaktor 10, die von der MPEG (Moving Picture Experts Group) als ISO-MPEG Audio Layer-3 unter ISO/IEC-11172-3 standardisiert wurde. Diese Gruppe setzt sich zusammen aus der International Standards Organisation (ISO) und der International Electro-Technical Commission (IEC). Sie beschäftigt sich mit der internationalen Standardisierung von Komprimierungstechniken von Audio-, Video- und kombinierten Multimediadaten. Weitere Informationen zu den Standards sind auf der offiziellen Webseite [85] zu finden.

Die Entwicklung des MP3-Player gliederte sich in vier Phasen:

I. Referenzimplementation des MPEG1-Dekoders mpg123 in Microsoft Visual C++,
II. Portierung des MP3-Dekoders für den ADSP-21061 mit Ein-/Ausgabe über File-I/O,
III. Hinzufügen der Ein-/Ausgabe über S/PDIF und erster Hardwaretest des MP3-Dekoders auf der Prototyp-Hardware,
IV. Implementierung der ATAPI/IDE-Schnittstelle im FPGA und Erweiterung zu einem autarken MP3-Player.

7.4.1 Phase I - Referenzimplementation

Ausgehend von den Sourcen des frei verfügbaren und in der Programmiersprache C geschriebenen MPEG1-Dekoders mpg123 [83] und dessen Abwandlung mpg123lib [84], eine für die Windows-Plattform angepasste Version von mpg123, wurde zunächst mit Hilfe von Microsoft Visual C++ eine Testumgebung unter Windows implementiert. Die eigentlichen Dekodierfunktionen wurden gekapselt und eine Schnittstelle zu diesem Kern definiert, die so später auf dem DSP verwendet werden konnte.

Das für Streaming geeignete MP3-Format besteht aus sukzessiven Datenblöcken, den MP3-Frames. Damit ist es möglich, von beliebigen Stellen des Datenstroms beginnend korrekt zu dekodieren. Jeder MP3-Frame besitzt einen vier Byte umfassenden Header. Die Idee war, jeweils genau vier Bytes an MP3-Daten einzulesen und zunächst der Schnittstelle zur MP3-Dekodierung zu übergeben. Sollte in der Schnittstellenfunktion bis zu diesem Zeitpunkt kein gültiger Header eines MP3-Frame gefunden oder der vorherige MP3-Frame bereits dekodiert worden sein, wird zunächst das Vorhandensein eines bzw. des nächsten gültigen Headers überprüft. Ist er gültig, werden Werte aus dem Header für die Initialisierung der Parameter des nächsten Aufrufs der Dekodierfunktion verwendet. Außerdem enthält er auch kodiert die Länge des Frame, womit bestimmbar ist, wieviel Daten einzulesen sind, bis die eigentliche Dekodierfunktion aufgerufen werden kann. Ist der Header nicht gültig, unerheblich ob der erste oder der folgende, wird dies der aufrufenden Funktion mit einem entsprechenden Rückgabewert (MP3_ERR) signalisiert. Demgegenüber werden alle einem gültigen Header folgenden Daten in einem Eingangspuffer gespeichert. Mittels MP3_NEED_MORE wird dabei signalisiert, dass der MP3-Frame noch nicht komplett ist. Dies wiederholt sich, bis alle Daten eines MP3-Frame eingelesen sind. Danach wird die MP3-Dekodierung gestartet. Nach erfolgter Dekodierung signalisiert der zurückgegebene Wert MP3_OK, dass der Ausgabepuffer gültige PCM-Samples enthält und ein weiterer

Rückgabewert bestimmt deren Anzahl. Bild 7-1 zeigt dafür den prinzipiellen Ablauf. Der zugehörige Quelltext ist im Anhang A.5.2 aufgelistet.

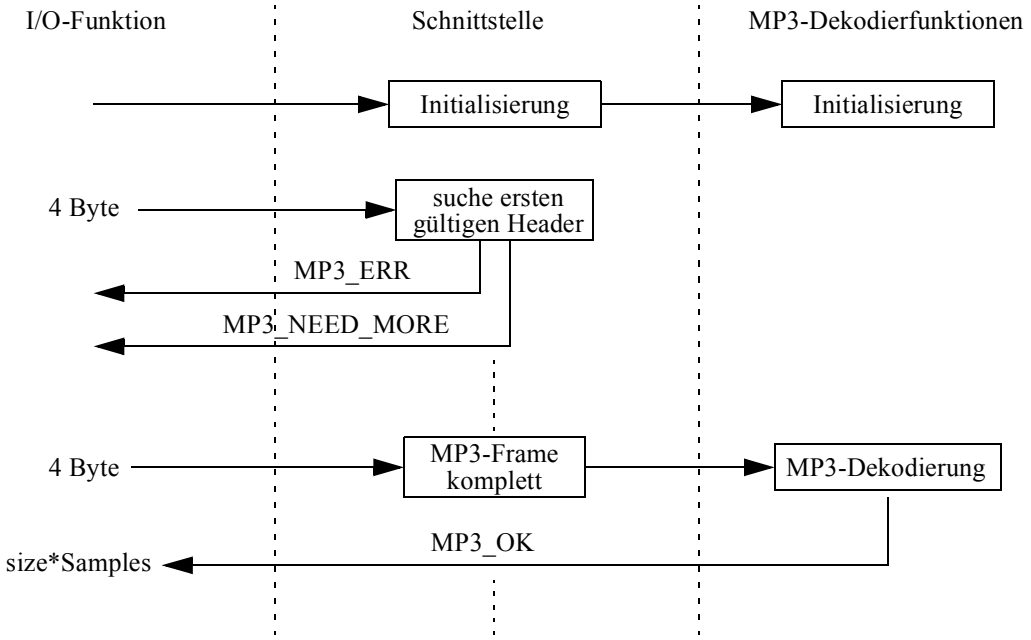


Bild 7-1. Schnittstelle für die Kernfunktionen der MP3-Dekodierung.

Auf die beschriebene Schnittstelle aufsetzend, wurde für die Windows-Plattform eine I/O-Funktion implementiert, die die MP3-Daten aus Binärdateien einliest und die dekodierten PCM-Samples in eine Ausgabe-Binärdatei schreibt. Für verschiedene Testdateien wurden die Ausgabedateien zur späteren Referenz in den nachfolgenden Phasen gespeichert.

7.4.2 Phase II - Portierung auf den ADSP21061

In Phase II der Entwicklung des MP3-Player wurde die I/O-Funktion aus Phase I an den DSP und die Gegebenheiten auf der Prototyp-Hardware angepasst. Zum Einlesen der MP3-Daten wird auf einen definierten Port im Adressraum des FPGA lesend und zur Ausgabe der dekodierten Werte auf einen zweiten Port schreibend zugegriffen. Beide Ports wurden mit Hilfe der Interface-Synthese aus Kap. 6.3 implementiert. An der MP3-Dekodierung und an dessen Schnittstelle hatte sich gegenüber Phase I nichts geändert.

Mit Hilfe des GNU C-Compilers, der im EZ-KIT Lite Entwicklungsset enthalten ist, wurden die Quelltexte für den ADSP-21061 Prozessor übersetzt. Dabei kam es vor allem darauf an, den MP3-Dekoder im eher begrenzten Programm- und Datenspeicher des ADSP-21061 unterzubringen. Es standen lediglich 8K Programmworte und 20K Datenworte zur Verfügung. Dafür mussten einige Änderungen und Optimierungen vorgenommen werden. Tabellen wurden z. B. mit konstanten Koeffizienteneinträgen in Initialisierungsfunktionen berechnet, welches unnötig Programmspeicher kostete. Eine Vorausberechnung der Werte brachte den gewünschten Speichergewinn. Ansonsten wurde der Dekoder konsequent auf MPEG1-Layer3 beschränkt.

Nachdem der MP3-Dekoder erfolgreich übersetzt war, wurde das VHDL-Modell des Gesamtsystems um zusätzliche generische, nicht synthetisierbare Schnittstellen im FPGA erweitert (Bild 7-2). Dort wird der MP3-Datenstrom aus einer Eingabedatei gelesen und der PCM-Datenstrom in eine Ausgabedatei geschrieben. Damit konnten die Ausgaben des MP3-Dekoders während der VHDL-Simulation ebenfalls in Dateien geschrieben werden. Diese wurden dann mit den in Phase I erzeugten Referenzdateien verglichen und deren Übereinstimmung überprüft. Weiterhin wurde ausgewertet, in wieweit die Abweichung bei der Genauigkeit der Fließkomma-Operationen Auswirkungen auf die Ausgabewerte hatten. Im mpg123-Dekoder werden alle Zwischenergebnisse mit der doppelten Genauigkeit (IEEE Double-Precision) berechnet. Im ADSP-2106x dagegen werden alle Fließkomma-Operationen mit einfacher Genauigkeit (IEEE Single-Precision) ausgeführt, eine doppelte Genauigkeit wird lediglich emuliert, was aber zusätzliche Ausführungszeit bedeutet hätte. Es konnte jedoch festgestellt werden, dass die Abweichungen vernachlässigbar waren.

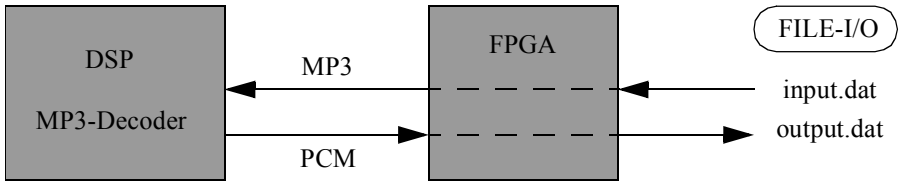


Bild 7-2. Simulation des MP3-Dekoders im VHDL-Modell der ADSP-2106x-Familie zusammen mit den Hardware-Modellen der Prototyp-Hardware mit Ein-/Ausgabe über generische Schnittstellen im FPGA. Die Flash-Bausteine und die S/PDIF-Schnittstelle wurden mit den minimalen VHDL-Modellen simuliert, da sie nicht zur Verifikation benötigt wurden.

7.4.3 Phase III - Erster Hardwaretest

In Phase III wurde die erste Hardwarerealisierung des MP3-Player implementiert. Auch hier hatte sich bei der MP3-Dekodierung und an dessen Schnittstelle gegenüber Phase I und Phase II prinzipiell nichts geändert. Die I/O-Funktion wurde aber so geändert, dass die MP3-Daten über die S/PDIF-Schnittstelle eingelesen und die dekodierten Audiodaten über dieselbe ausgegeben werden konnten. Im FPGA wurde hierfür ein FIFO als Zwischenpuffer für die Eingangsseite implementiert, ein solches für die Ausgangsseite war nicht notwendig (Bild 7-3).

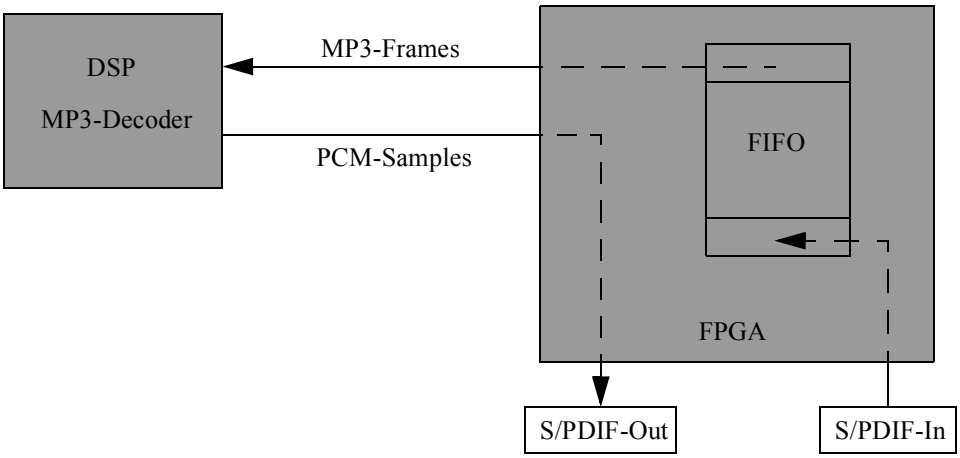


Bild 7-3. Blockschaltbild des MP3-Player in Phase III, der ersten Hardwarerealisierung.

Als Quelle der S/PDIF-Signale wurde ein CD-Player mit optischem S/PDIF-Ausgang eingesetzt. Der CD-Player gibt gemäß dem RedBook-Standard [96] alle Audiodaten (PCM, 16-Bit, Stereo) mit 44,1 kHz Sample-Frequenz aus. Die MP3-Frames mussten daher in ein zum RedBook-Standard passendes Format umgewandelt werden, da sie kontinuierlich über den S/PDIF-Eingang eingespeist werden (Streaming) und nicht nur bei Bedarf vom DSP eingelesen werden konnten. Während der notwendigen Konvertierung werden Füll-Informationen in den Bitstrom eingefügt, um die Eingangsdatenrate der Ausgangsdatenrate anzupassen. Würde das nicht passieren, kämen die MP3-Daten mit zu hoher Datenrate in den DSP und irgendwann würde der Eingangspuffer überlaufen, da er nicht beliebig groß gewählt werden kann. Jeder MP3-Frame wird unabhängig von seiner Länge auf eine feste Audio-Framelänge von 1.152 Stereo-Samples (9.216 Bytes = 2 * 1.152 * 32 Bit) gestreckt. In Abhängigkeit von der jeweiligen MP3-Framelänge werden ensprechend Füll-Informationen eingefügt. Damit wird der eigentliche Vorteil von MP3 eliminiert, nämlich die Audiodaten in einer komprimierten Form abzuspeichern. Aber diese Eigenschaft ist irrelevant, da Phase III nur als Testszenario dient. Die konvertierten MP3-Dateien wurden dann als normale Tracks auf eine Audio-CD gebrannt und vom CD-Player abgespielt.

Die eingefügten Füll-Informationen müssen auf der Empfängerseite, auf der Übertragungsstrecke nach dem S/PDIF-Eingang, gefiltert werden, um die ursprünglichen MP3-Daten zu erhalten. Dies ist Aufgabe der Schaltung im FPGA. Eine weitere Aufgabe bestand darin, den seriellen Eingangsbitstrom zu parallelisieren und die Daten im FIFO abzulegen. Wichtig hierbei war es, richtig auf das Framesync-Signal des S/PDIF-Receivers zu synchronisieren, so dass die Daten beginnend mit den Informationen des linken Stereo-Kanals im FIFO gespeichert werden konnten. Der FIFO wurde mit Hilfe des CoreGenerators von Xilinx für die Block-SelectRAMs des Virtex-Bausteins generiert. Die erzeugte VHDL-Deklaration des Entity wurde in den Entwurf der FPGA-Schaltung integriert und konnte mittels eines zugehörigen Modells aus der Simulationsbibliothek von Xilinx im Gesamtsystem simuliert werden. Die ebenfalls generierte Gatternetzliste wurde während des Platzierungs- und Verdrahtungsvorgangs innerhalb der Werkzeuge von Xilinx in den endgültigen Konfigurationsdatenstrom für das FPGA eingebunden. Die Ergebnisse der Platzierung und Verdrahtung der Schaltung für den XCV300-Baustein in Phase III sind in Listing 7-4 aufgeführt.

Erkennbar ist eine Auslastung der Logikblöcke des Bausteins von lediglich 5 %, so dass genügend Platz für die in Phase IV zu realisierenden Schaltungsteile zur Verfügung steht. Für den FIFO wurden präventiv 100 % der Block-SelectRAMs reserviert, dies entspricht 2048 Worten á 32-Bit.

Da jede Übertragung des Taktes über eine optische Verbindung Jitter produziert, ist auf der Empfängerseite eine Rekonstruktion des Taktes notwendig. Dies wird vom S/PDIF-Receiver (Crystal-Baustein CS8412) durchgeführt, in den für diesen Zweck eine analoge PLL-Synchronisationsschaltung integriert ist. Um eine Asynchronität zwischen Empfangs- und Sendetakt zu vermeiden, wird der Takt für den optischen S/PDIF-Ausgang direkt aus dem Takt des S/PDIF-Eingangs generiert, indem das FPGA das Taktsignal vom S/PDIF-Receiver zum S/PDIF-Transmitter weiterleitet. Da die Sample-Frequenz am S/PDIF-Eingang aufgrund des CD-Player 44,1 kHz beträgt, ist es mit dieser Methodik nur möglich, MP3-Stücke mit genau dieser Frequenz, aber beliebigen Bitraten, abzuspielen. Da die Phase III

```
1.   Release 4.1.03i - Map E.33
2.   Xilinx Mapping Report File for Design 'target_fpga'
3.
4.   Design Information
5.   ------------------
6.   Target Device  : xv300
7.   Target Speed   : -4
8.
9.   Design Summary
10.  --------------
11.     Number of errors:      0
12.     Number of warnings:    0
13.     Number of Slices:                179 out of  3,072    5%
14.     Number of Slices containing
15.        unrelated logic:                0 out of    179    0%
16.     Number of Slice Flip Flops:      192 out of  6,144    3%
17.     Total Number 4 input LUTs:       254 out of  6,144    4%
18.        Number used as LUTs:                        242
19.        Number used as a route-thru:                 12
20.     Number of bonded IOBs:            94 out of    166   56%
21.     Number of Block RAMs:             16 out of     16  100%
22.     Number of GCLKs:                   1 out of      4   25%
23.     Number of GCLKIOBs:                1 out of      4   25%
24.  Total equivalent gate count for design:  265,483
25.  Additional JTAG gate count for IOBs:  4,560
26.
27.  ---------------------------------------------------------------------
28.    Constraint                        | Requested   | Actual  | Logic
29.                                      |             |         | Levels
30.  ---------------------------------------------------------------------
31.    NET "C2391/IBUFG" PERIOD = 30 nS  | 30.000ns    |18.668ns | 2
32.    HIGH 50.000000 %                  |             |         |
33.  ---------------------------------------------------------------------
34.
35.  All constraints were met.
```

Listing 7-4. Ausschnitt aus den Place&Route-Ergebnissen für die Schaltung im FPGA für den MP3-Player in Phase III.

jedoch nur zum Testen des MP3-Dekoders im System implementiert wurde, spielt dieser Aspekt keine entscheidende Rolle.

Geklärt werden musste das Zusammenspiel zwischen dem Einlesen und der Dekodierung eines MP3-Frame einerseits und der Ausgabe der dekodierten Audiodaten andererseits. Es musste ein kontinuierlicher gültiger Ausgangsdatenstrom gewährleistet werden, damit das System in Echtzeit arbeitet. Es handelt sich hierbei um eine weiche Echtzeitanforderung, da keine sicherheitskritischen Rahmenbedingungen einzuhalten waren.

Der Ablauf innerhalb des I/O-Moduls in Phase III wird nachfolgend erläutert. Der zugehörige Quelltext ist in Anhang A.5.1 aufgelistet. Aus Platzmangel im Datenspeicher standen für die MP3-Dekodierung lediglich zwei Ausgabepuffer á 26 ms Audiodaten zur Verfügung. In einer Initialisierungsphase wird zunächst die Kommunikation mit dem FPGA getestet und in einem definierten Kontrollregister ein Bit gesetzt, das den FIFO im FPGA aktiviert. Danach wird eine Funktion gestartet, die sich selbst nicht terminiert. Sie hat die

Aufgabe, kontinuierlich die Daten aus dem FIFO im FPGA auszulesen und der Schnittstelle zur MP3-Dekodierung zu übergeben. Der Schnittstelle wird über einen Zeiger auf einen Speicherbereich mitgeteilt, wo die Ausgabedaten der MP3-Dekodierung abzuspeichern sind. Da zwei Ausgabepuffer zur Verfügung stehen, wird nach jeder erfolgreichen Dekodierung eines MP3-Frame zum jeweils anderen Puffer umgeschaltet. Sind zu Beginn oder nach einem Fehlerfall beide Puffer das erste Mal gefüllt, kann die Ausgabe der Audiodaten in den Ausgabepuffern gestartet werden. Dazu wird eine Initialisierungsfunktion für den ersten der beiden seriellen Ports aufgerufen, der über das FPGA mit dem S/PDIF-Receiver verbunden ist. Die Funktion initialisiert den DMA-Transfer im Chaining-Mode (verkettete DMA-Übertragungen) für die serielle Schnittstelle. DMA-Chaining erlaubt dem DMA-Controller, sich zwischen aufeinander folgenden DMA-Transfers neu zu initialisieren, ohne dass der Core-Prozessor unterbrochen werden muss. Der zum DMA-Kanal gehörige Interrupt wurde ebenfalls aktiviert. Dieser Interrupt wird daraufhin immer ausgelöst, wenn die Ausgabe eines Puffers abgeschlossen wurde und die Autoinitialisierung durchgeführt wird. Für die Behandlung der auftretenden Interrupts wurde eine ISR in die Interrupttabelle eingesetzt, die dazu dient, den momentan per DMA auszugebenden Puffer in einer globalen Variable festzuhalten. Da nur zwei Ausgabepuffer zur Verfügung stehen, wird die globale Variable auf den jeweils anderen gültigen Wert (Puffer 1 oder Puffer 2) gesetzt. Dies ist ein wichtiger Punkt, der verhindert, dass das Schreiben der dekodierten Werte in einen Puffer erfolgt, der gerade ausgegeben wird. Die verketteten DMA-Transfers wechseln automatisch zwischen den beiden für die Ausgabe bestimmten Speicherbereiche. Sollte es bei der Dekodierung oder an einer anderen Stelle zu einem Fehler kommen, werden die beiden Ausgabepuffer mit Nullen überschrieben und damit der Audioausgang praktisch stumm geschaltet.

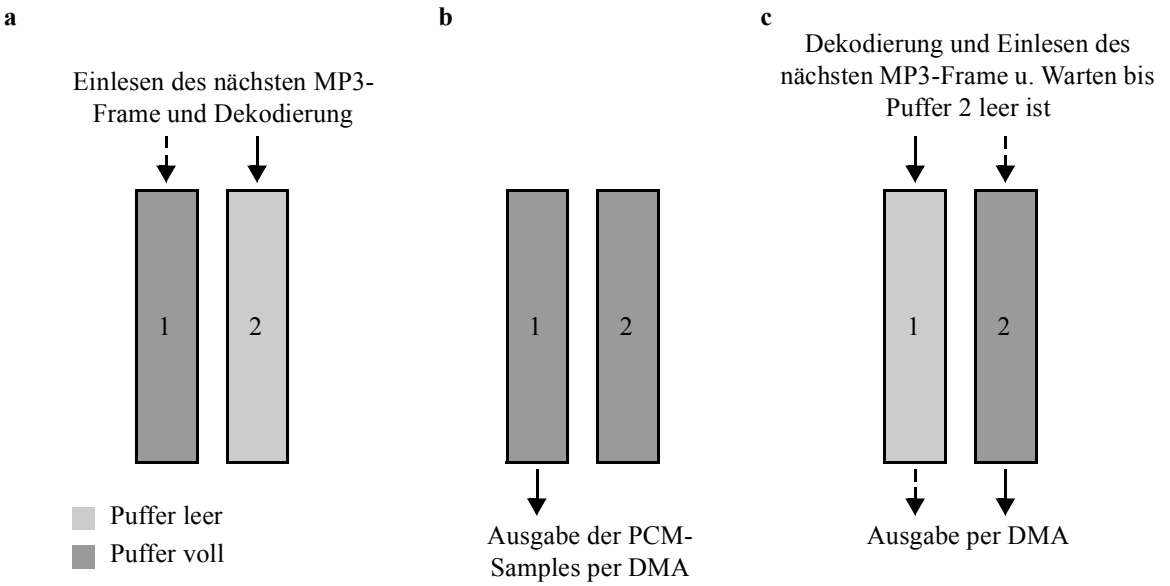


Bild 7-4. Ablauf des Zusammenspiels von Ein- und Ausgabe in Phase III in der I/O-Funktion des MP3-Player **a** Füllen der beiden Ausgabepuffer mit dekodierten Audiodaten in einer Initialisierungsphase **b** Start der Ausgabe per DMA **c** Füllen des aktuell geleerten Puffers mit neuen dekodierten Audiodaten, Einlesen des nächsten MP3-Frame und warten bis der jeweils andere Puffer ausgegeben ist.

Ist die Ausgabe der Audiodaten zweier vollständig gefüllter Ausgabepuffer gestartet, werden sofort die nächsten MP3-Daten aus dem FIFO im FPGA eingelesen. Die Daten werden wieder im Eingangspuffer gespeichert. Wenn der nächste MP3-Frame vollständig ist, muss

überprüft werden, ob der Puffer, in den die dekodierten Werte geschrieben werden sollen, derselbe ist, der noch vom DMA-Prozess ausgegeben wird. Ist dies der Fall, wird der Programmablauf mit der Funktion idle() in einen Ruhezustand überführt. Der DMA-Prozess läuft im Hintergrund weiter und generiert einen Interrupt, wenn der Speicherbereich transferiert wurde. Die weiter oben beschriebene Interruptbehandlung wird ausgeführt und es gibt keinen Konflikt mehr beim Zugriff auf den Ausgabepuffer. Die Interruptfunktion kehrt genau einen Befehl nach der idle()-Anweisung in die Schnittstelle zur MP3-Dekodierung zurück, so dass die Dekodierung des bereits länger vollständigen MP3-Frame starten kann. Diese Änderung wurde am Interface der MP3-Dekodierung durchgeführt, denn nur hier wird der Übergang vom Einlesen zur Dekodierung des MP3-Frame vollzogen. Bild 7-4 zeigt den Ablauf für das Zusammenspiel zwischen Ein- und Ausgabe.

Ausgehend von der Beschreibung des implementierten Algorithmus musste die folgende Echtzeitanforderung für das HW/SW-Echtzeitsystem in Phase III mit Hilfe der VHDL-Board-Level-Simulation verifiziert werden:

$$t_{input} + t_{decode} < t_{output} \text{ mit} \tag{7.1}$$

t_{input} Zeit zum Einlesen eines MP3-Frame,
t_{decode} Zeit zur MP3-Dekodierung des MP3-Frame,
t_{output} Ausgabezeit eines Puffers per DMA.

Die Zeit für das Einlesen eines MP3-Frame plus der Zeit für die Dekodierung desselben durfte nicht die Zeit für die Ausgabe eines Puffers übersteigen, damit ein kontinuierlicher gültiger Ausgangsdatenstrom gewährleistet werden kann.

Eine weitere Anforderung betraf die Kommunikation zwischen FPGA und DSP zum Einlesen der Datenwerte aus dem FIFO im FPGA. Um die Datenintegrität der MP3-Daten sicherzustellen, musste die Kommunikation zu 100 % funktionieren, auch bei Simulation der Verzögerungszeiten der platzierten und verdrahteten Schaltung im FPGA. Des Weiteren musste die Umsetzung der seriellen S/PDIF-Daten auf das FIFO-Format verifiziert werden. Auszuschließen waren dabei, dass Bitfelder verdreht wurden bzw. es musste geprüft werden, dass die Füll-Informationen korrekt von den eigentlichen Daten getrennt werden.

Der dritte Punkt umfasste die Überprüfung des gesamten Ablaufs, angefangen von der Interruptsteuerung über die Initialisierung und den Ablauf des DMA-Prozesses, bis hin zum Zusammenspiel von Ein- und Ausgabe innerhalb der I/O-Funktion. Die eigentlichen Kernfunktionen der MP3-Dekodierung wurden bereits in Phase I und Phase II verifiziert.

Diese Anforderungen wurden erfolgreich mit Hilfe der VHDL-Board-Level-Simulation des Gesamtsystems verifiziert. Für die Dekodierung eines mit 192 KBit/s kodierten MP3-Frames wurden die Simulationszeiten auf der Workstation SunU10-300 unter Solaris und auf einem 1 GHz-Athlon PC unter Linux gemessen (Tabelle 11 auf Seite 128).

Tabelle 11: Simulationszeiten für die Dekodierung eines mit 192 KBit/s kodierten MP3-Frame.

Simulierte Zeit [ns]	Simulationszeit **ohne** Debug-Oberfläche [s]		Simulationszeit **mit** Debug-Oberfläche [s]	
	1 GHz-Athlon	SunU10-300	1 GHz-Athlon	SunU10-300
26.120.370	460,76	1.665,13	7.542,27	24.344,73

Die Framelänge des MP3-Frame betrug 624 Bytes. Es mussten also 156 32-Bit-Werte zwischen DSP und FPGA pro MP3-Frame transferiert werden. Die Werte wurden mit einem Handshake-Verfahren asynchron zwischen DSP und FPGA übertragen, mit einem Taktzyklus als zusätzlicher Verzögerung. Damit ergibt sich, dass zwei Taktzyklen für einen Transfer eines Datenwertes und somit insgesamt für 156 Werte 312 Taktzyklen bei 33 MHz notwendig sind, was äquivalent zu einem Zeitraum von 9,36 µs ist.

Ein MP3-Frame entspricht 26 ms dekodiertem Audiomaterial, so dass sich aus den gemessenen Simulationszeiten weitere Werte extrapolieren lassen. Auf dem 1 GHz-Athlon PC werden für 26 ms Audio ca. 461 s, bzw. 7,7 min Simulationszeit benötigt. Hochgerechnet auf ein 1-minütiges MP3-Stück müssten bereits laut Formel (7.2) ca. 12,3 Tage für dessen Dekodierung simuliert werden.

$$t_{1\min} = \frac{461\mathrm{s} \cdot 60\mathrm{s}}{26\mathrm{ms}} = 1063846,15\mathrm{s} = 12,31\mathrm{Tage} \tag{7.2}$$

Daraus schlussfolgernd wird die allgemeine Vorgehensweise die sein, nur für gezielte Fälle die Simulation über längere Zeiträume durchzuführen. Längere Tests werden dann auf der Prototyp-Hardware in Echtzeit durchgeführt. Sollte sich dabei ein Problem ergeben, lassen sich für die Simulation entsprechende Testdateien generieren und damit das Problem und dessen Ursache virtuell eingrenzen. Dies ist auch wichtig, wenn es keine Möglichkeit gibt, über Messungen an die Ursachen eines Problems zu gelangen.

In dieser Phase der Entwicklung des MP3-Player wurde die Verringerung der Simulationszeiten durch das in Kap. 3.3.4 beschriebene Prinzip der „gated clock“ geprüft. Die Zeiten wurden hierfür nur auf der Workstation SunU10-300 unter Solaris ohne aktivierte Debug-Oberfläche gemessen. Die Werte, die in Tabelle 12 enthalten sind, wurden über vier Messungen arithmetisch gemittelt und davon wurde der prozentuale Gewinn an Simulationszeit laut der Formel (7.3) berechnet.

Tabelle 12: Gemessene Simulationszeiten für die Anwendung des Prinzips der "gated clock".

Simulierte Zeit [ms]	Simulationszeit ohne gated clock [s]	Simulationszeit mit gated clock [s]
100	5207,15	4591,9

Anzumerken ist, dass in der Simulation die Bedingung für die Unterbindung der Flanken des Taktsignals zu 40 % der simulierten Zeit gegeben war. Daraus resultiert eine um ca. 12 % kürzere Simulationszeit.

$$x = \frac{100\% \cdot 4591,9\mathrm{s}}{5207,15\mathrm{s}} - 100\% = -(11,82)\% \tag{7.3}$$

Auf die Formel (7.2) bezogen hätte dies zur Folge gehabt, dass nicht 12,3 Tage für die Simulation der Dekodierung eines 1-minütigen MP3-Stückes notwendig wären, sondern rund 13,8 Tage. Daraus folgt, bereits dieser leicht zu implementierende Mechanismus kann die Simulationsgeschwindigkeit messbar steigern.

Der erste Echtzeittest auf der Prototyp-Hardware lief hervorragend. Kleinere Optimierungen waren rasch implementiert und getestet. Die Erkennung des MP3-Header und die Behandlung eines fehlerhaften MP3-Header wurden z. B. verbessert, da sich beim Test im System herausstellte, dass sich der Wechsel von einem MP3-Stück zum nächsten mit einem Knacksen bemerkbar machte (Drücken der Vorwärts/Rückwärts-Taste auf dem CD-Player). Dies ist ein Beispiel für das Auftreten eines Fehlers, der in der Simulation wahrscheinlich erst nach einer langen Zeit gefunden worden wäre. Nachdem das Problem aber erkannt wurde, war es möglich, eine entsprechend präparierte Eingabedatei für die Simulation aufzubereiten und die Ursache virtuell zu ergründen, um eine Lösung zu finden.

7.4.4 Phase IV - Erweiterung zum autarken MP3-Player

Die abschließende Phase IV beinhaltet die Aufgabe, im FPGA einen ATAPI/IDE-Controller zu entwickeln. Es sollte ermöglicht werden, die MP3-Daten wahlweise von einem CD-ROM-Laufwerk oder von einer Festplatte einzulesen. Andere Schnittstellen sind denkbar, beispielhaft SmartCard, Flash oder auch Ethernet. Zusätzlich sind für einen autarken (portablen) MP3-Player ein LC-Display und Bedienelemente sinnvoll (siehe Kap. 4.1.3). Die MP3-Dekodierung, dessen Interface und die Ausgabe der PCM-Samples über den S/PDIF-Ausgang sind identisch mit denen in Phase III, wie das Bild 7-5 verdeutlicht.

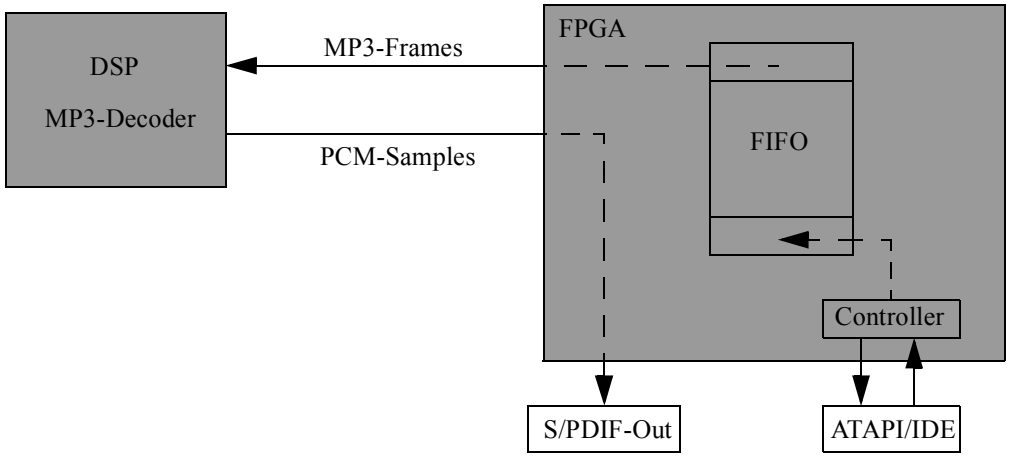


Bild 7-5. Blockschaltbild des MP3-Player in Phase IV.

Auf eine weitere Implementierung in Phase IV wurde verzichtet, da keine neuen wissenschaftlichen Erkenntnisse zu erwarten wären. Extrapolieren lassen sich Werte für die Belegung der Logikblöcke im FPGA aus einem am Institut im Rahmen einer Diplomarbeit entwickelten MP3-Player [76], bei dem in einem FPGA vom Typ XCV150 von Xilinx ein ATAPI-Controller nebst Bedienelementen und Ansteuerung eines LC-Displays implementiert wurde. Zur MP3-Dekodierung kam ein vorgefertigter ASIC-Baustein von Micronas Intermetall, der MAS3507D, zum Einsatz. Schwierig war, die komplette Schaltung in den

```
1.   Xilinx Mapping Report File for Design 'main'
2.   Copyright (c) 1995-1999 Xilinx, Inc.  All rights reserved.
3.
4.   Design Information
5.   ------------------
6.   Target Device  : xv150
7.   Target Speed   : -4
8.
9.   Design Summary
10.  --------------
11.     Number of errors:      0
12.     Number of warnings:    1
13.     Number of Slices:            1,728 out of  1,728  100%
14.        Slice Flip Flops:   1,441
15.        4 input LUTs:       2,483 (38 used as a route-thru)
16.     Number of Slices containing
17.        unrelated logic:           119 out of  1,728    6%
18.     Number of bonded IOBs:         89 out of    166   53%
19.     Number of Tbufs:              160 out of  1,824    8%
20.     Number of GCLKs:                1 out of      4   25%
21.     Number of GCLKIOBs:             1 out of      4   25%
22.  Total equivalent gate count for design:  30,371
23.  Additional JTAG gate count for IOBs:  4,320
24.
25.  ---------------------------------------------------------------------
26.    Constraint                          | Requested   | Actual  | Logic
27.                                        |             |         | Levels
28.  ---------------------------------------------------------------------
29.    NET "C2391/IBUFG" PERIOD = 30 nS  | 100.000ns   |31.131ns | 19
30.    HIGH 50.000000 %                  |             |         |
31.  ---------------------------------------------------------------------
32.
33.  All constraints were met.
```

Listing 7-5. Ausschnitt aus den Place&Route-Ergebnissen für den FPGA-basierten Controller aus [76] realisiert in einem XCV150-4.

damals neu auf dem Markt erschienenen FPGA-Baustein unterzubringen, dies zeigen auch die Ergebnisse der Platzierung und Verdrahtung in Listing 7-5.

Ohne Änderungen wurde derselbe Entwurf als Ausgangsbasis für eine Realisierung im XCV300-Baustein von Xilinx herangezogen, wobei die Synthese-Randbedingungen an den Systemtakt von 33 MHz angepasst wurden. Die Ergebnisse sind in Listing 7-6 aufgeführt. Wie zu erwarten war, ist jetzt im größeren FPGA genug Platz für Verbesserungen und Erweiterungen der damaligen Schaltung. Anzumerken ist, dass der Design Compiler als Synthese-Werkzeug und die Xilinx-Werkzeuge für die Abbildung der Netzliste im FPGA in neueren Versionen benutzt wurden.

```
1.   Xilinx Mapping Report File for Design 'main'
2.   Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.
3.
4.   Design Information
5.   ------------------
6.   Target Device  : xv300
7.   Target Speed   : -4
8.
9.   Design Summary
10.  --------------
11.     Number of errors:      0
12.     Number of warnings:   32
13.     Number of Slices:              1,731 out of  3,072   56%
14.     Number of Slices containing
15.        unrelated logic:                0 out of  1,731    0%
16.     Number of Slice Flip Flops:    1,441 out of  6,144   23%
17.     Total Number 4 input LUTs:     2,476 out of  6,144   40%
18.        Number used as LUTs:                       2,423
19.        Number used as a route-thru:                  53
20.     Number of bonded IOBs:            89 out of    166   53%
21.     Number of Tbufs:                 160 out of  3,200    5%
22.     Number of GCLKs:                   1 out of      4   25%
23.     Number of GCLKIOBs:                1 out of      4   25%
24.  Total equivalent gate count for design:  30,119
25.  Additional JTAG gate count for IOBs:  4,320
26.
27.  ------------------------------------------------------------------------
28.    Constraint                          | Requested   | Actual  | Logic
29.                                        |             |         | Levels
30.  ------------------------------------------------------------------------
31.    NET "C2391/IBUFG" PERIOD = 30 nS  | 30.000ns    |27.321ns | 20
32.    HIGH 50.000000 %                  |             |         |
33.  ------------------------------------------------------------------------
34.
35.  All constraints were met.
```

Listing 7-6. Ausschnitt aus den Plac&Route-Ergebnissen für den FPGA-basierten Controller aus [76] realisiert in einem XCV300-4.

8 Zusammenfassung

Die FPGA-Technologie und damit realisierte Hardware erweist sich einmal mehr für unterschiedliche Einsatzgebiete als sehr leistungsfähig. Trotz der Spezialisierung auf eine Hardware-Technik ist die Flexibilität durch die praktisch unbegrenzte Reprogrammierbarkeit gegeben. Die weitere technische Entwicklung im Bereich rekonfigurierbarer Logik wird den Ansatz, das FPGA als Co-Prozessor in einem dedizierten Zielsystem einzusetzen, für weitergehende Anwendungen sinnvoll umsetzen lassen. Im Bereich der digitalen Signalverarbeitung spricht die Fähigkeit der neuesten FPGA-Generation, auch Fließkomma-Arithmetik abbilden zu können, ebenso dafür.

Für Anwendungen, die sich mit *FADES* entwickeln lassen, erstreckt sich ein nicht überschaubares Feld von Möglichkeiten. In erster Linie unterstützt das System die Entwicklung von Anwendungen für die Unterhaltungselektronik (z. B. MP3-Player, Ogg-Vorbis-Player), da es primär auf die Verarbeitung von Audiosignalen zugeschnitten ist. Andere Schnittstellen sind jedoch über den Erweiterungssteckverbinder möglich, soweit die Daten vom DSP und vom FPGA hinsichtlich der Datenrate verarbeitet werden können. Für die Entwicklung von Anwendungen mit *FADES* gibt es unterschiedliche Sichten:

1. als Entwicklungsplattform für eingebettete audiosignalverarbeitende Echtzeit-SoCs,
2. für die schnelle Prototyp-Entwicklung,
3. als Emulationssystem.

Über zusätzliche Steckverbinder können weitere Tochterplatinen mit der vorhandenen Prototyp-Hardware verbunden werden, so dass schnell ein angepasster Prototyp für die eigene Anwendung implementiert werden kann (Punkt 2). Darüber hinaus lässt sich damit auch die Prototyp-Hardware weiterentwickeln, da I/O-Leitungen des Controller-FPGA ebenfalls über den Steckverbinder geführt sind und eine Rekonfiguration des Controller-FPGA möglich ist. Bei Punkt 3 wird im Zielsystem der DSP von einem ASIC unterstützt und das FPGA wird zur Emulation der zu implementierenden ASIC-Schaltung genutzt.

Eine Entwicklung einer Anwendung unter Verwendung von *FADES* beschränkt sich im Wesentlichen auf den Entwurf der applikationsspezifischen Logik im FPGA und der Software für den DSP. In vielen Fällen kann sofort mit der Verwirklichung der eigenen Anwendung auf einer bereits erprobten und verfügbaren Hardware begonnen werden.

Der fortsetzende Pfad zu einem ASIC wurde in der Arbeit nicht weiter verfolgt, ist aber prinzipiell möglich, auch wenn es den verwendeten DSP von Analog Devices nicht als Prozessorkern in Form eines IP gibt. Es sind jedoch andere DSP-Kerne verfügbar (siehe Kap. 2.4.1). Zu beachten ist, dass die Zeitwerte einer diskreten Komponente nicht mit denen eines ansonsten äquivalenten Prozessorkerns übereinstimmen. Daher ist das VHDL-Verhaltensmodell für die Komponente mit Generic-Parametern für die Zeitwerte ausgestattet bzw. generell auszustatten, so dass diese einfach zu ändern sind (siehe auch VHDL-Modell der

ADSP-2106x-Familie im Anhang A.3.1). Für den applikationsspezifischen Teil der Hardware sind die VHDL-Quelltexte unter Zuhilfenahme von ASIC-Bibliotheken zu synthetisieren. Danach erfolgt ein größtenteils automatisierter Layout-Prozess. Für die layoutete Schaltung ergibt sich ein anderes Zeitverhalten als für die FPGA-synthetisierte Schaltung, welches in der Simulation mit Timing berücksichtigt werden muss.

Zusammenfassend lassen sich die folgenden Vor- und Nachteile für die in *FADES* verwendete VHDL-Board-Level-Simulation für die homogene HW/SW-CoSimulation aufzeigen:

Vorteile - Prinzipiell ist auf allen Stufen des Entwurfsprozesses eine VHDL-Simulation des gesamten Systems möglich. Eine aufwendige externe Kommunikationsschnittstelle, die die zu transferierenden Werte zwischen Hardware und Software und den zeitlichen Bezug speichert, ist nicht notwendig. Daraus ergibt sich eine hohe Flexibilität hinsichtlich der Evaluierung geeigneter Kommunikationsmodelle im HW/SW-CoDesign eines Algorithmus. Ein weiterer Vorteil ist die Portabilität des Modells, da hier auf eine plattformunabhängige Beschreibungssprache zurückgegriffen wurde. Dies gilt, so lange ein VHDL-Compiler zusammen mit einem Simulator für die anvisierte Plattform zur Verfügung stehen. Als weiterer positiver Aspekt ist die Wiederverwendbarkeit der Modelle für die einzelnen Komponenten der Hardware zu nennen. Das zugehörige Entity ist denkbar einfach in die neue Hierarchie über die Schnittstellenbeschreibung zu instantiieren, die Signale zu verbinden und gegebenenfalls die Generic-Parameter anzupassen.

Nachteile - Der Hauptnachteil der Simulation des Gesamtsystems innerhalb einer VHDL-Board-Level-Simulation ist der, dass die Laufzeit gegenüber einer getrennten Simulation von Hardware und Software höher sein kann. Gerade wenn Signale mit eigentlich niedriger Datenrate in einem System mit sehr viel höherer Taktrate bearbeitet werden sollen, durchläuft die Simulation zu viele „kleine“ Schritte, in denen sich eigentlich innerhalb des Systems wenig oder gar nichts ändert. Hier sind geeignete Maßnahmen innerhalb des Modells zu ergreifen, die die Simulation beschleunigen. (z. B. "gated clock"). Ein weiterer Nachteil besteht darin, dass Modelle für die verwendeten Komponenten zur Verfügung stehen bzw. erst entwickelt werden müssen. Dies ist unter Umständen mit einem nicht unerheblichen Aufwand verbunden. Der Idealfall träte ein, wenn es vom jeweiligen Hersteller der Komponente ein (geeignet geschütztes [90]) VHDL-Verhaltensmodell geben würde, wie es heutzutage auch schon verbreitet IBIS-Modelle oder seltener Interface-Modelle gibt.

Die Zeiten für die Generierung von Modulen der HW/SW-Schnittstellen innerhalb der Interface-Synthese liegen im nicht nachweisbaren Promillebereich. Der Zeitverlust durch fehlerhaft manuell implementierte HW/SW-Schnittstellen ist nicht messbar, im Gegensatz zur Generierungszeit sind diese jedoch signifikant.

8.1 Ausblick

In diesem Abschnitt werden Erweiterungen vorgeschlagen, die den momentanen Stand der Implementierung von *FADES* verbessern. Dazu gehören die folgenden Punkte:

- Störend für den schnellen Entwurfsfluss ist die fehlende Möglichkeit, die VHDL-Board-Level-Simulation bei einem Breakpoint zwei, drei oder mehr Taktschritte zurückzuverfolgen (backtrace). Der Breakpoint ist schon im Vorfeld der interessierenden Programmpassage zu setzen, um einen notwendigen Neustart der Simulation und damit lange redundante Simulationszeiten zu vermeiden. Leider fand sich dafür im HDL-Simulator ModelSim keine brauchbare Funktion. Die Software-Simulation hätte vielleicht mit einem Trace-Buffer für die systemimmanenten Register versehen werden können, aber ein solcher Mechanismus ist u. a. hinsichtlich des Speicherbedarfs für die Hardware-Simulation im HDL-Simulator zu aufwendig.
- Die plattformunabhängige Debug-Oberfäche für die VHDL-Board-Level-Simulation ist speziell auf das VHDL-Modell der ADSP-2106x-Familie abgestimmt. Da in SoC-Projekten andere DSPs bzw. Prozessoren eingesetzt werden können, wäre es sinnvoll, eine prozessorunabhängige und konfigurierbare Oberfläche mit definierten Schittstellen zu einem VHDL-Modell zu implementieren.
- Die Möglichkeit, die Software für den DSP in der Hochsprache C zu implementieren, wird mit dem derzeitigen Stand der Implementierung der Debug-Oberfläche für die HW/SW-CoSimulation noch nicht berücksichtigt. Der Objekt-Code wird zwar disassembliert und entspricht damit direkt dem manuell geschriebenen oder automatisch vom C-Compiler generierten Assembler-Code, der Bezug zu den C-Quelltexten wird aber nicht hergestellt. Dafür ist in einer zukünftigen Version ein C-Debugger in die Debug-Oberfläche zu integrieren.
- Eine weitere Ergänzung stellt die Integration der dynamischen partiellen Rekonfiguration über den SelectMap-Port des FPGA in die HW/SW-CoSimulation dar. Durch zusätzliche Kontroll-Logik muss es ermöglicht werden, die alternativen Schaltungsteile in der Simulation zu aktivieren, wenn die entsprechenden Konfigurationsdaten korrekt über den SelectMap-Port eingegangen sind. Bei der VHDL-Simulation nach der Hardware-Synthese ist diese Integration noch schwieriger zu realisieren.
- Die momentan auf der Prototyp-Platine vorhandene Konfigurations- und Debug-Schnittstelle über den Parallel-Port des Entwicklungs-PC ließe sich in einer neuen Version, aufgrund der Programmierbarkeit des Controller-FPGA, durch eine Netzwerkschnittstelle (TCP/IP über Ethernet) zur Erhöhung der Komfortabilität ersetzen bzw. ergänzen.
- Die im Hardware-Prototyp eingebauten digitalen Audioschnittstellen (S/PDIF) könnten in einer neuen Version durch analoge Audioein- und -ausgänge ergänzt werden. Weiter vorausschauend sind in einem System mit einem leistungsfähigeren DSP und FPGA Schnittstellen (analog/digital) für die Videosignalbearbeitung (z. B. MPEG4) denkbar.
- Der Zugriff zum Debuggen zur Laufzeit des Systems auf die Prototyp-Platine erfolgt derzeit über den Parallel-Port und nutzt nicht die Möglichkeiten der Emulation über die JTAG-Schnittstelle der ADSP-2106x-Familie. Leider sind die dafür nötigen Informationen von Analog Devices nicht veröffentlicht. Lediglich der offizielle Teil der Boundary-Scan-Struktur nach IEEE 1149.1 [74, 94] ist dokumentiert. Stünden diese Informationen zur Verfügung, könnte im Controller-FPGA eine Schnittstelle realisiert werden, die die Verbindung zwischen dem Parallel-Port und JTAG herstellt. Zusätzlich müsste die Software des Entwicklungssystems eine geeignete Emulationsoberfläche bieten und Treiber für das Protokoll einbinden. Die Emulation würde einen kontrollierteren Zugriff auf die Software im DSP zur Laufzeit des Systems bieten. So könnten

Breakpoints gesetzt und die Software im Single-Step-Betrieb ausgeführt werden. Auch die Register im Prozessorkern, die in aller Regel die lokalen Variablen speichern, könnten manipuliert werden. Mit der implementierten Variante funktioniert dies nur für die Systemregister, die in den Speicherbereich des Gesamtsystems eingeblendet sind (vgl. Kap. 6.2).

FADES wird derzeit (Sommersemester 2003) erfolgreich in der Lehrveranstaltung „Entwurf komplexer digitaler Systeme (FPGA-Projekt)“ an der Technischen Universität Berlin für die Entwicklung von Anwendungen sowie für die Implementierung von Erweiterungen eingesetzt. Das System kann darüber hinaus für weitere Forschungsaktivitäten genutzt werden; dazu zählen:

- Eingehende Analyse einer dynamischen partiellen Rekonfiguration des FPGA und deren sinnvolle Nutzung zur Verbesserung des Ausnutzungsgrades des FPGA. Die Möglichkeit der Rekonfiguration ist bereits physisch im Hardware-Prototyp vorgesehen, wurde jedoch mangels unterstützender Software nicht eingesetzt und nicht weiter untersucht.
- Tiefergehende Untersuchung der Nützlichkeit eines applikationsspezifischen Co-Prozessors in Hardware für den DSP in weiteren Anwendungsfeldern. Dazu gehört auch die approximative Skalierbarkeit einer stetig wachsenden Anzahl von Logikgattern in FPGAs und steigender Leistungsfähigkeiten bei DSPs.
- Die Untersuchung weiterer möglicher Realisierungen für die Debug-Oberfläche der Software-Simulation, die zusammen mit der Hardware-Simulation zum Zwecke der homogenen HW/SW-CoSimulation im HDL-Simulator läuft. Dazu gehört, wie bereits erwähnt, die Integration eines C-Debuggers, wofür beispielhaft der GNU-Debugger gdb eingesetzt werden könnte.
- Konzeption und Implementierung eines in die Software *Mephisto* integrierten Werkzeugs für die Spezifikation und Partitionierung auf Systemebene.
- Integration einer eigenen Software in das Entwicklungssystem für die Software-, Hardware-Synthese und die VHDL-Simulation, um unabhängig von der bisher eingesetzten Fremdsoftware zu sein.

Für die Zukunft interessant könnte eine Integration eines DSP-Kerns als Hard-IP in einen FPGA-Baustein sein, wie beispielsweise Mikrocontroller-Kerne in den FPGAs von Xilinx, Altra etc. integriert sind. Damit ist die für die digitale Signalverarbeitung notwendige Fließkomma-Arithmetik fest verdrahtet und hinsichtlich Verlustleistung und Taktfrequenz optimiert. Zu diesem festverdrahteten Teil kommt die Flexibilität der applikationsspezifisch programmierbaren Hardware hinzu. Dies könnte unter dem Akronym DSoPC (Digital Signal Processing System-on-Programmable-Chip), also einer Bezeichnung der Vereinigung von DSP und SoPC, zusammengefasst werden.

Anhang

Der Anhang enthält einige wichtige Programmauszüge der Software *Mephisto* (vgl. Kap. 6.2), sowie die schematische Schaltungsbeschreibung und das Board-Layout mit Bauteileliste für den Hardware-Prototyp *Faust* von *FADES* (vgl. Kap. 6.1). Außerdem sind hier Teile des VHDL-Quelltextes der Modelle für die Board-Level-Simulation und die C-Quelltexte für die Beispielimplementierungen (CORDIC - Kap. 7.3, MP3-Player - Kap. 7.4) enthalten.

A.1 Software *Mephisto*

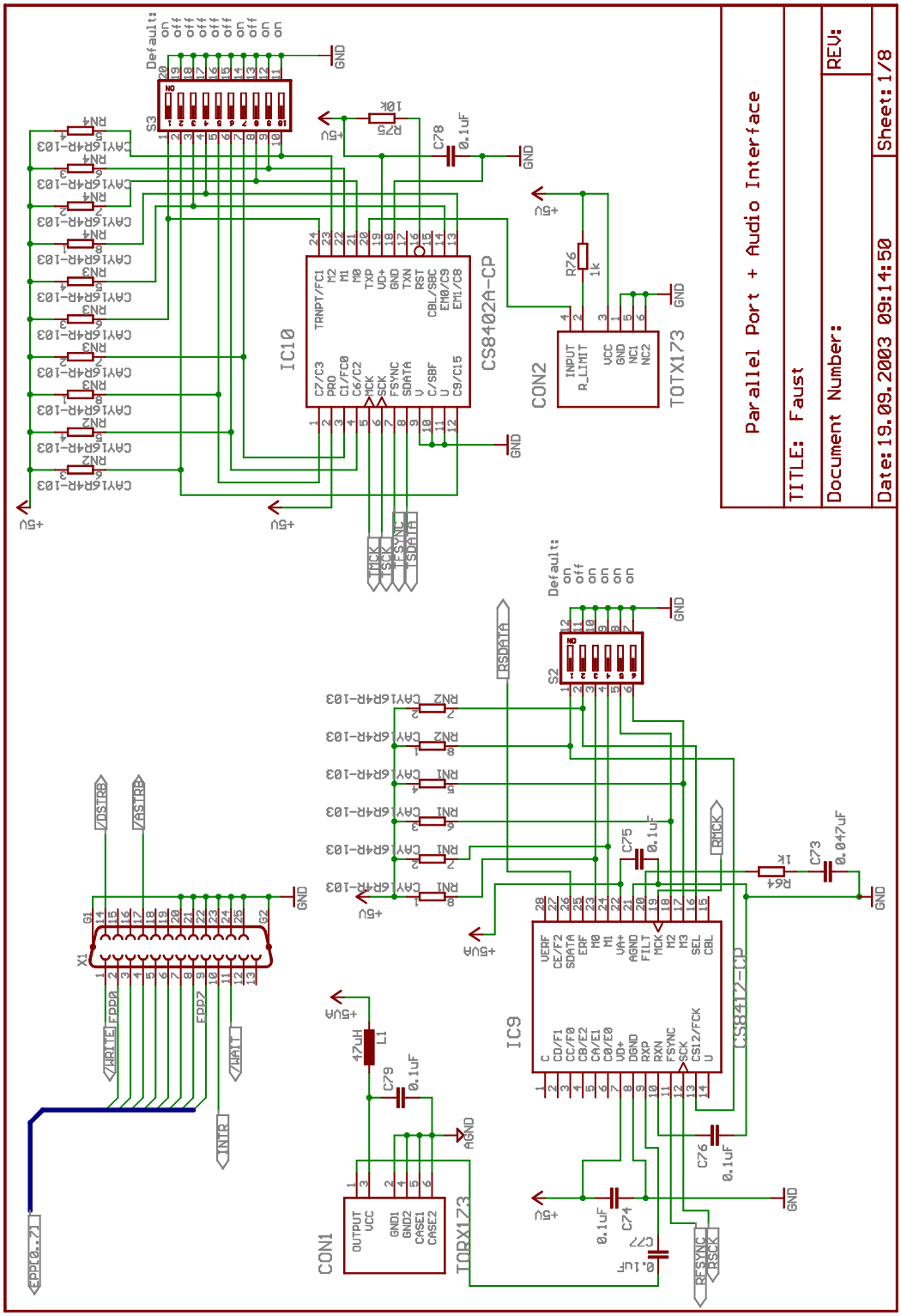
A.1.1 XML_Parser_Wrapper

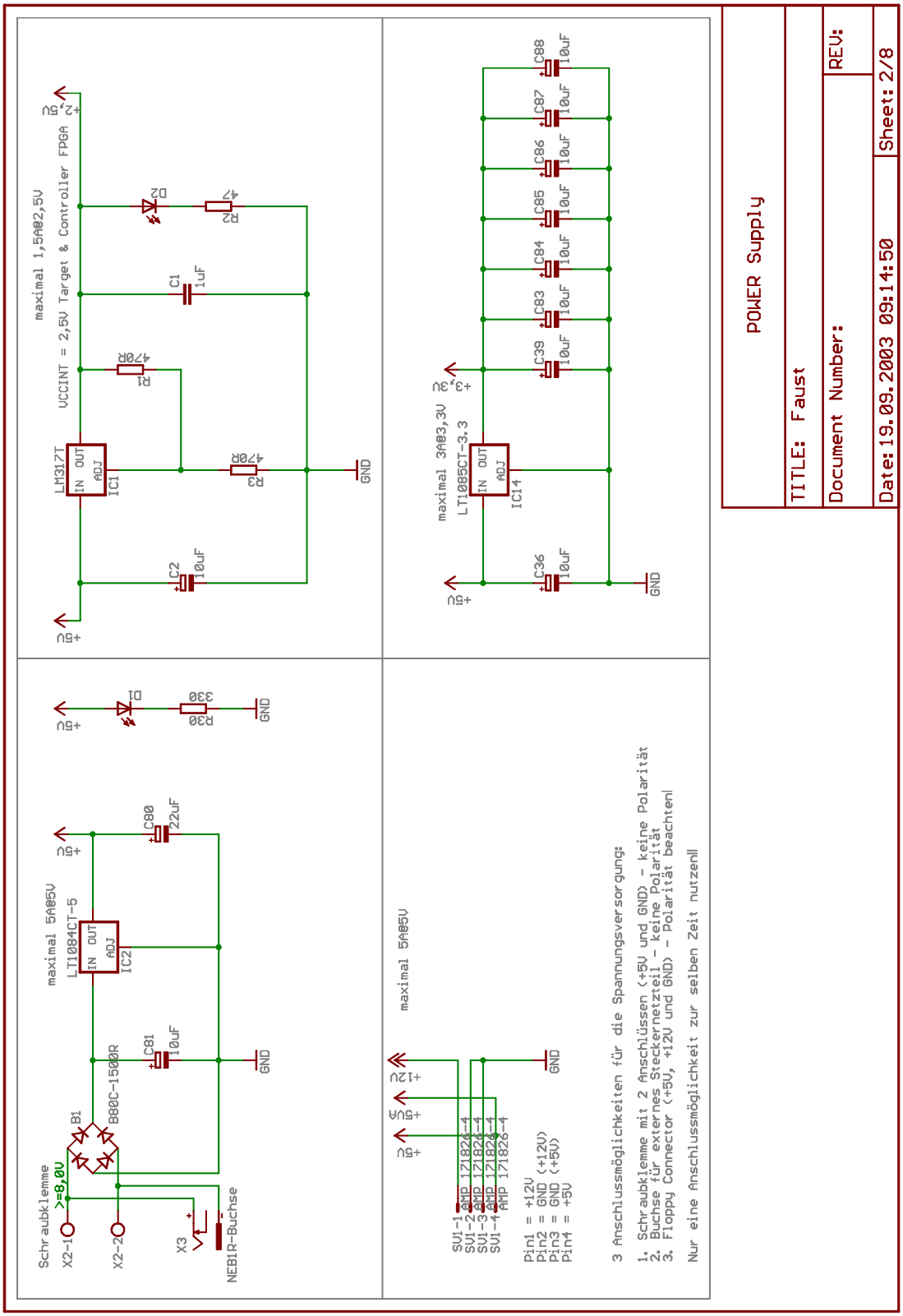
```
////////////////////////////////////////////////////////////////////////
// Function: XML_Parser_Wrapper()
// Parameter: src           - XML stream
//            userPtr       - pointer tu user data elements
//            startElement  - handler function for start tag
//            endElement    - handler function for end tag
//            dataElement   - handler function for all character data
//            startCData    - handler function for CDATA start tag
//            endCData      - handler function for CDATA end tag
// Return:    TRUE if successfull; FALSE else
// Note:      userPtr can be used to address user specified data,
//            NULL if not needed
////////////////////////////////////////////////////////////////////////
BOOL XML_Parser_Wrapper(FILE *src, void *userPtr,
                  XML_StartElementHandler startElement,
                  XML_EndElementHandler endElement,
                  XML_CharacterDataHandler dataElement,
                  XML_StartCdataSectionHandler startCData,
                  XML_StartCdataSectionHandler endCData) {
     char buf[BUFSIZ+1];
     char *last_lf, *buf_end;
     int done;
     XML_Parser parser = XML_ParserCreate(NULL);

     XML_SetUserData(parser, userPtr);
     if(startElement != NULL && endElement != NULL)
          XML_SetElementHandler(parser, startElement, endElement);
     if(dataElement != NULL)
          XML_SetCharacterDataHandler(parser, dataElement);
     if(startCData != NULL && endCData != NULL)
          XML_SetCdataSectionHandler(parser, startCData, endCData);

     // null terminate buffer for strrchr
     buf[sizeof(buf)-1] = '\0';
     // set pointer to last byte of buffer
     buf_end = &buf[sizeof(buf)-1];

     do { size_t len = fread(buf, 1, sizeof(buf)-1, src);
          done = len < (sizeof(buf)-1);
          if(!done) {
               // search last occurence of line-feed character in buffer
               last_lf = strrchr(buf, '\n');
               if(last_lf != NULL) {
                    // set len to the last line-feed character in buffer
                    len = last_lf-buf;
                    // rewind and read all after last line-feed
                    // character in next loop
                    fseek(src, (last_lf+1)-buf_end, SEEK_CUR);
               }
          }
          if(!XML_Parse(parser, buf, len, done)) {
               sprintf(buf, "%s at line %d\n",
                    XML_ErrorString(XML_GetErrorCode(parser)),
                    XML_GetCurrentLineNumber(parser));
               error(buf, 0);
               return FALSE;
          }
     } while(!done);
     XML_ParserFree(parser);
     return TRUE;
}
```

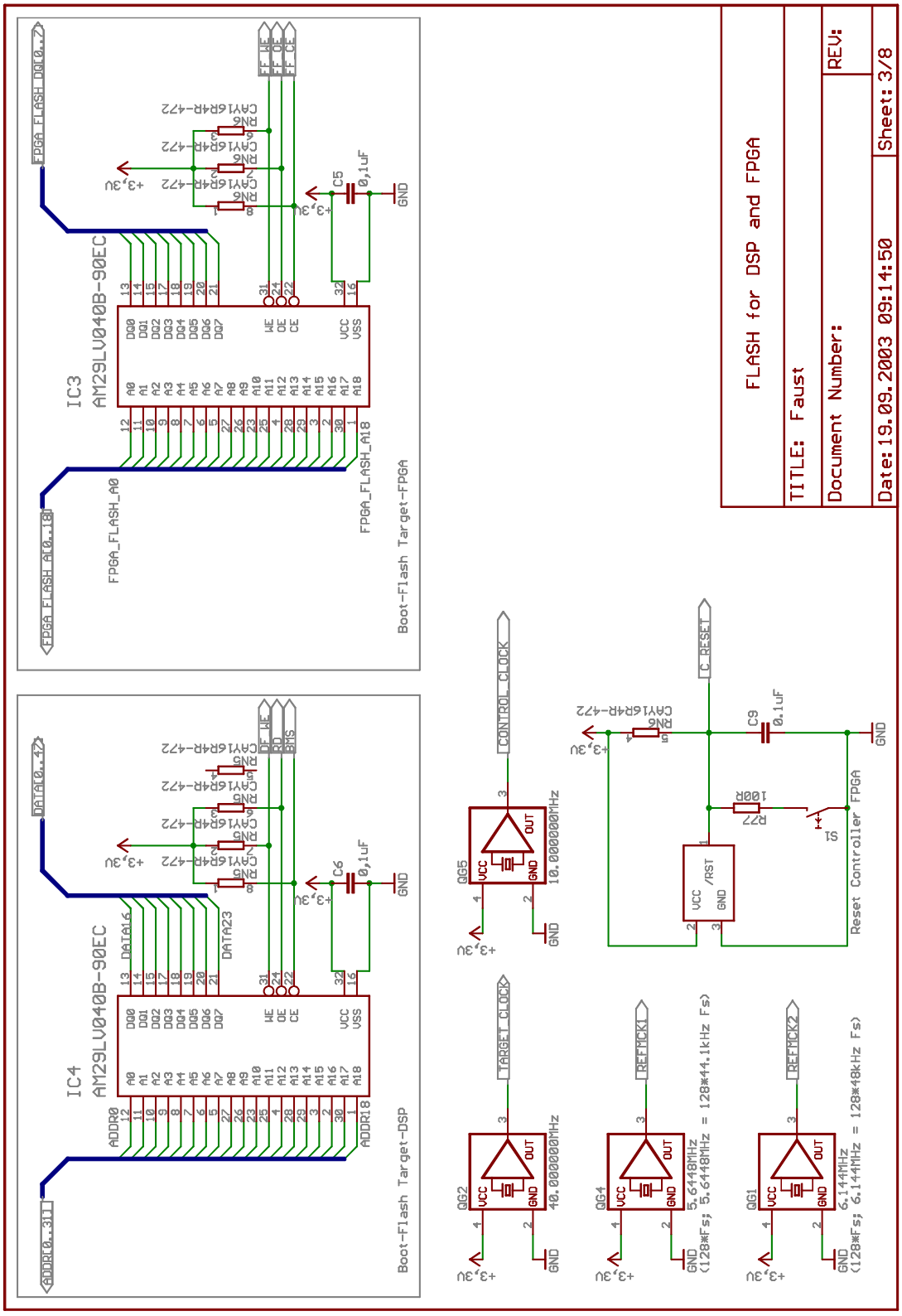
A.2 Hardware *Faust*

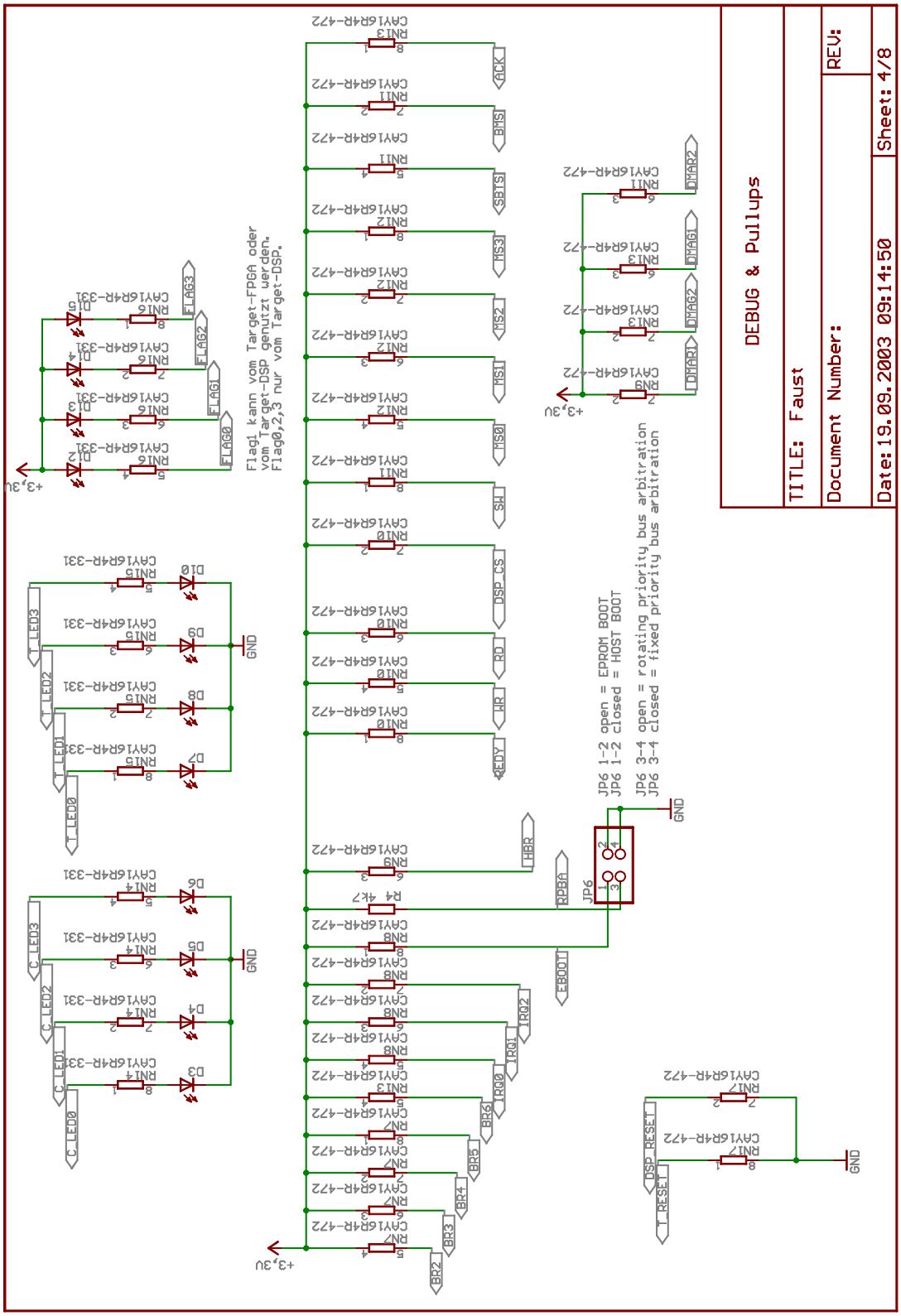
A.2.1 Schematic

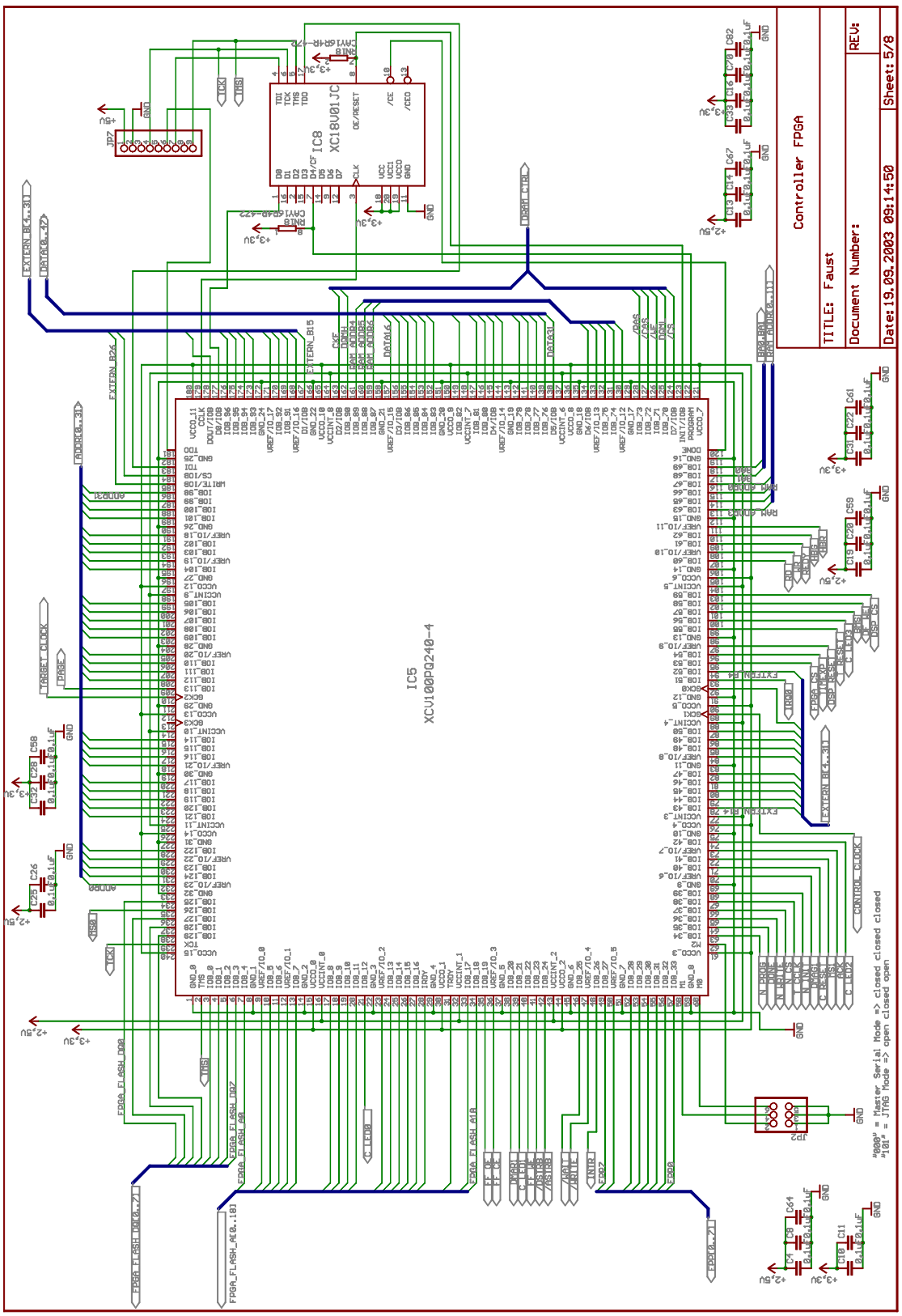
Der Schaltplan und das Platinenlayout wurden mit der Software Eagle der Firma CadSoft entwickelt.

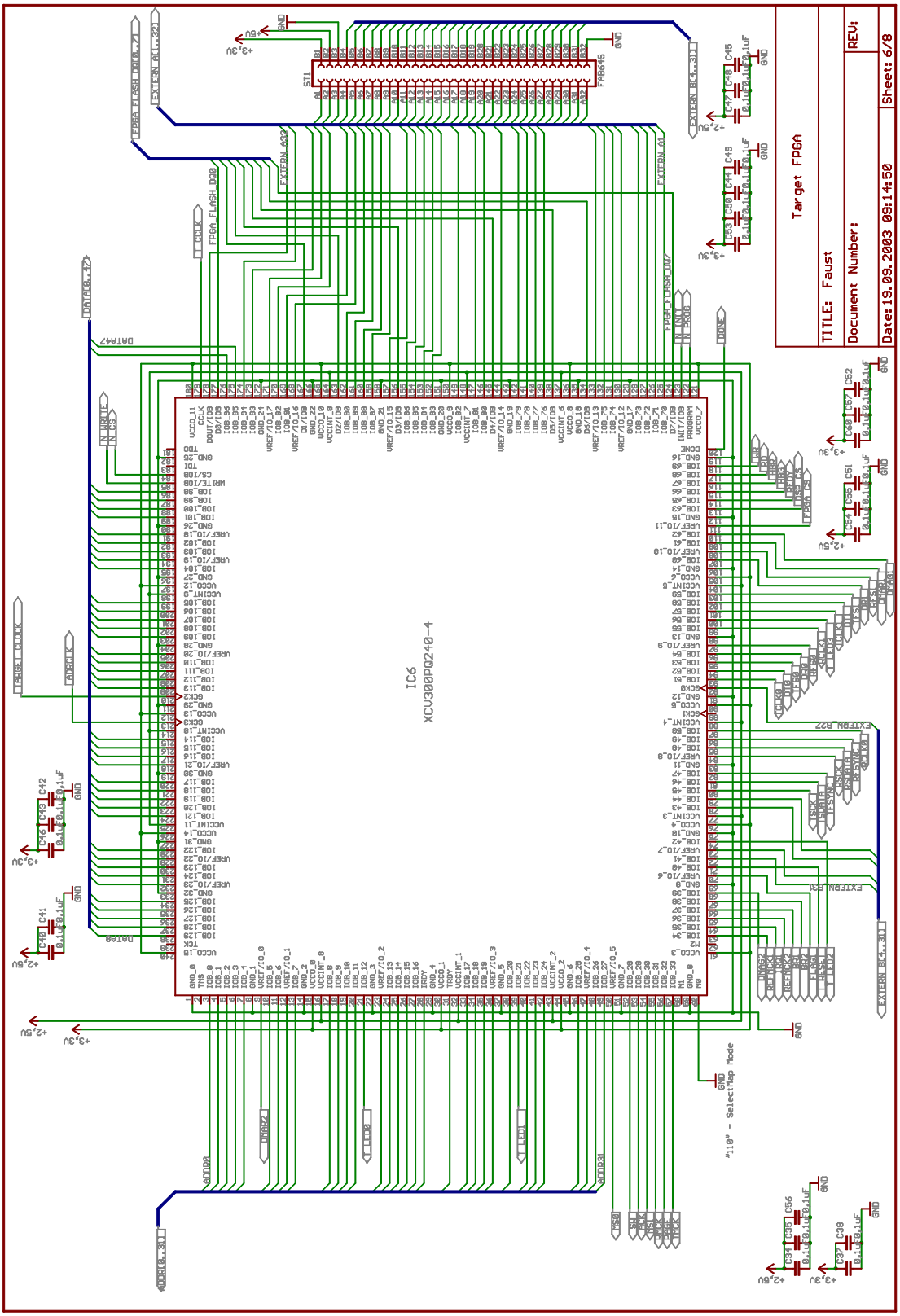


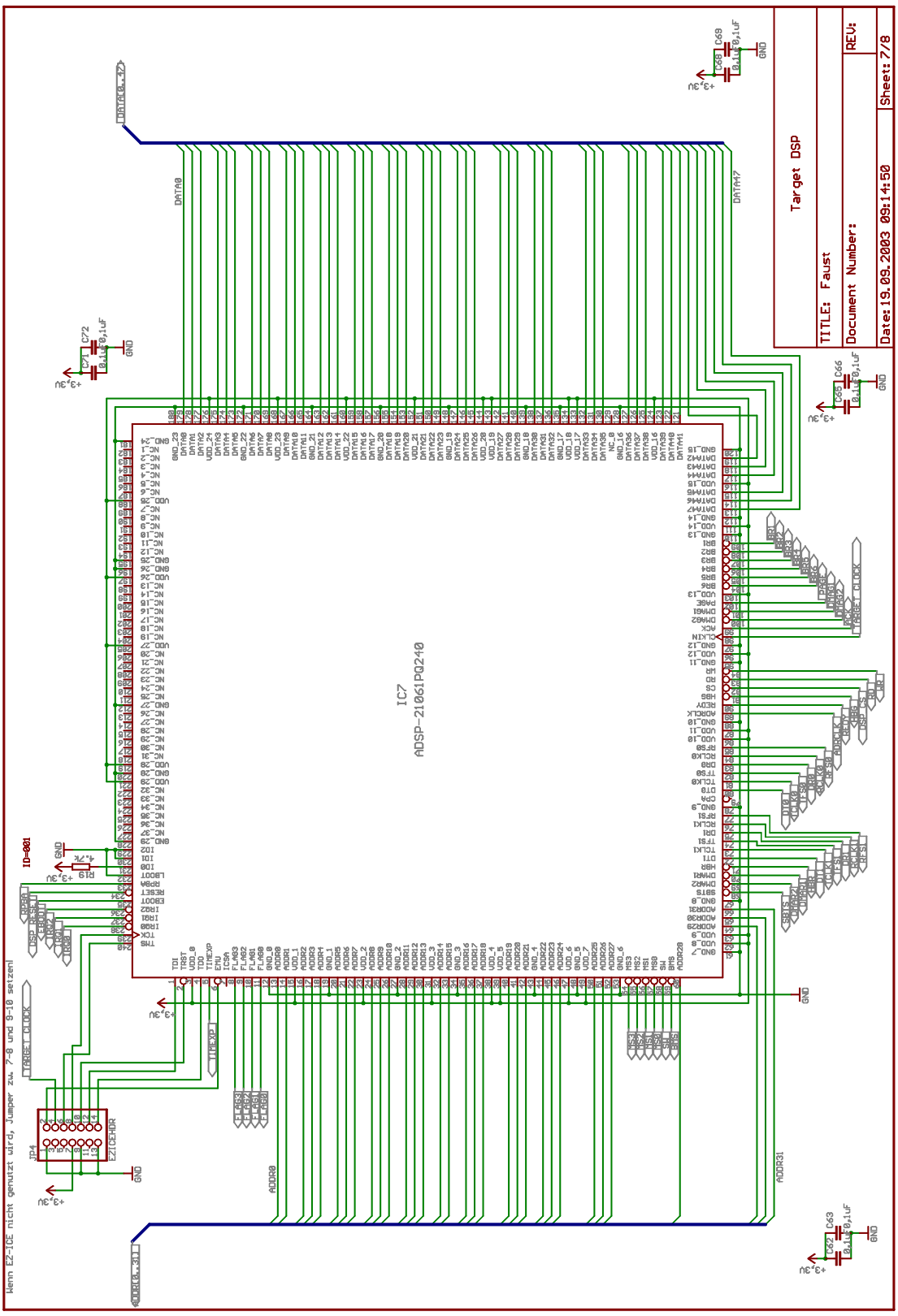

maximal 1,5A@2,5V
VCCINT = 2,5V Target & Controller FPGA
LM317T
IC1
maximal 3A@3,3V
LT1085CT-3.3
IC14
maximal 5A@5V
LT1084CT-5
IC2
Schraubklemme
>=8,0V
X2-1
X2-2
X3
NEB1R-Buchse
B80C-1500R
B1
D1
D2
R1
R2
R3
R30
C1
C2
C36
C39
C80
C81
C83
C84
C85
C86
C87
C88
1uF
10uF
22uF
330
470R
GND
+5V
+3,3V
+12V
SV1-1
SV1-2
SV1-3
SV1-4
AMP 171826-4
Pin1 = +12V
Pin2 = GND (+12V)
Pin3 = GND (+5V)
Pin4 = +5V
3 Anschlussmöglichkeiten für die Spannungsversorgung:
1. Schraubklemme mit 2 Anschlüssen (+5V und GND) - keine Polarität
2. Buchse für externes Steckernetzteil - keine Polarität
3. Floppy Connector (+5V, +12V und GND) - Polarität beachten!
Nur eine Anschlussmöglichkeit zur selben Zeit nutzen!!
POWER Supply
TITLE: Faust
Document Number:
REV:
Date: 19.09.2003 09:14:50
Sheet: 2/8

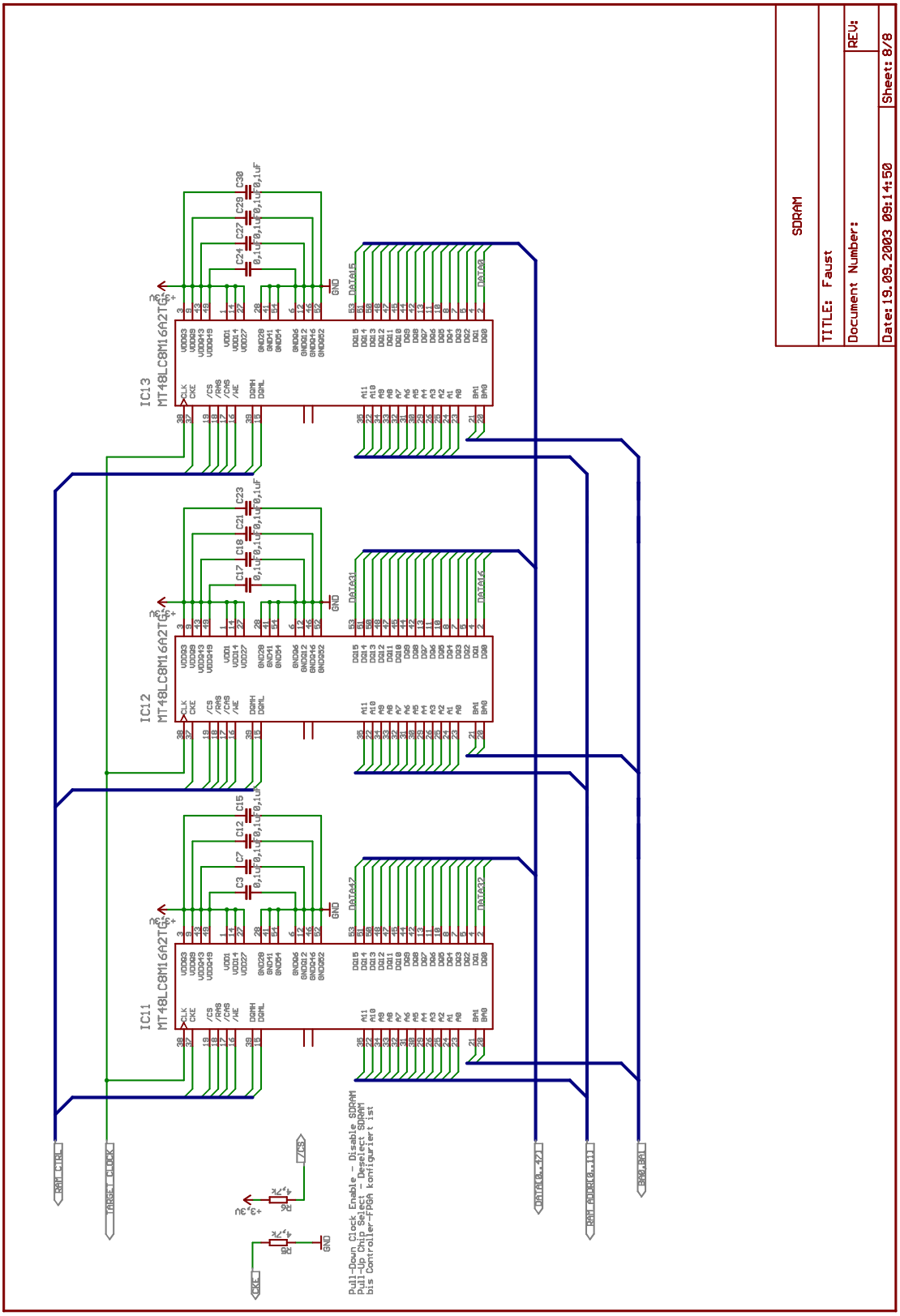

FLASH for DSP and FPGA
TITLE: Faust
Document Number:
REV:
Date: 19.09.2003 09:14:50
Sheet: 3/8
Boot-Flash Target-FPGA
Boot-Flash Target-DSP
IC3
IC4
AM29LV040B-90EC
FPGA_FLASH_A0
FPGA_FLASH_A18
ADDR0
ADDR18
DATA16
DATA23
C5
C6
C9
0,1uF
R27
100R
+3,3V
GND
CONTROL_CLOCK
TARGET_CLOCK
REFMCK1
REFMCK2
C_RESET
Reset Controller FPGA
10.000000MHz
40.000000MHz
5.6448MHz
6.144MHz
(128*Fs; 5.6448MHz = 128*44.1kHz Fs)
(128*Fs; 6.144MHz = 128*48kHz Fs)


DEBUG & Pullups
TITLE: Faust
Document Number:
REV:
Date: 19.09.2003 09:14:50
Sheet: 4/8
Flag1 kann vom Target-FPGA oder vom Target-DSP genutzt werden. Flag0,2,3 nur vom Target-DSP.
JP6 1-2 open = EPROM BOOT
JP6 1-2 closed = HOST BOOT
JP6 3-4 open = rotating priority bus arbitration
JP6 3-4 closed = fixed priority bus arbitration
+3,3V
GND
ACK
BMS
SBTS
MS3
MS2
MS1
MS0
SWI
DSP_CS
RD
WR
RDY
HBR
RPBA
EBOOT
IRQ2
IRQ1
IRQ0
BR6
BR5
BR4
BR3
BR2
DMAR2
DMAG1
DMAG2
DMAR1
FLAG3
FLAG2
FLAG1
FLAG0
T_LED3
T_LED2
T_LED1
T_LED0
C_LED3
C_LED2
C_LED1
C_LED0
DSP_RESET
T_RESET
JP6
R4 4K7
CAY16R4R-472
CAY16R4R-331


Controller FPGA
TITLE: Faust
Document Number:
REV:
Date:19.09.2003 09:14:50
Sheet: 5/8
IC5
XCV100PQ240-4
IC8
XC18V01JC
"000" = Master Serial Mode => closed closed closed
"101" = JTAG Mode => open closed open

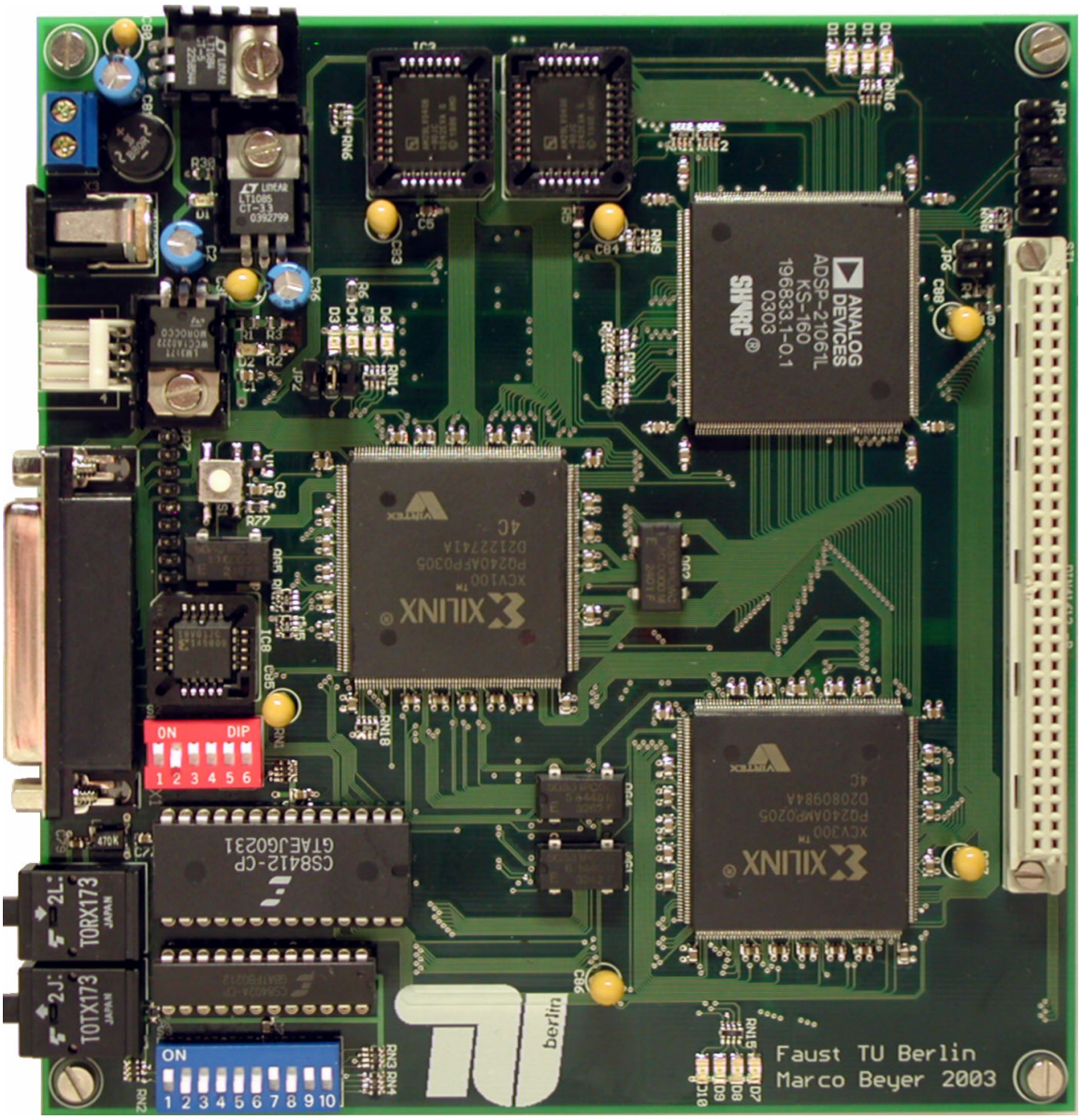

Target FPGA
TITLE: Faust
Document Number:
REV:
Date: 19.09.2003 09:14:50
Sheet: 6/8
IC6
XCV300PQ240-4
"110" - SelectMap Mode


Target DSP
TITLE: Faust
Document Number:
REV:
Date:19.09.2003 09:14:50
Sheet: 7/8
IC7
ADSP-21061PQ240
Wenn EZ-ICE nicht genutzt wird, Jumper zu 7-8 und 9-10 setzen!
EZICEHDR
TARGET_CLOCK
DSP_RST
ID-001
GND
+3,3V


SDRAM
TITLE: Faust
Document Number:
REV:
Date: 19.09.2003 09:14:50
Sheet: 8/8
IC13
IC12
IC11
MT48LC8M16A2TG-75
RAM CTRL
TARGET CLOCK
DATA[0..32]
RAM ADDR[0..11]
BA[0..1]
/CS
/CKE
+3,3V
GND
Pull-Down Clock Enable - Disable SDRAM
Pull-Up Chip Select - Deselect SDRAM
bis Controller-Fpga konfiguriert ist

A.2.2 Board-Layout

Abbildung der Prototyp-Platine



Belegung Erweiterungsstecker ST1

Tabelle 13: Steckerbelegung ST1 (DIN 41612 Bauform B)

I/O-Pin FPGA	Pinreihe A	Pinreihe B	Funktion/ I/O-Pin Controller-FPGA (CP)/ I/O-Pin FPGA (P)
P125	1	1	+3,3V
P126	2	2	+5V
P127	3	3	GND
P128	4	4	CP94
P130	5	5	CP92 (Clockbuf)
P131	6	6	CP87
P132	7	7	CP86
P133	8	8	CP85
P139	9	9	CP84
P140	10	10	CP82
P141	11	11	CP81
P142	12	12	CP80
P144	13	13	CP79
P146	14	14	CP78
P147	15	15	CP167
P149	16	16	CP168
P152	17	17	CP169
P153	18	18	CP170
P154	19	19	CP171
P155	20	20	CP173
P157	21	21	CP174
P159	22	22	CP175
P160	23	23	CP176
P161	24	24	CP178
P162	25	25	CP184
P168	26	26	CP185
P169	27	27	P92 (Clockbuf)
P170	28	28	P79
P171	29	29	P78
P173	30	30	P73
P174	31	31	P72
P178	32	32	GND

Stückliste

Tabelle 14: Liste der Bauteile mit Namen, Typenbezeichnung, erforderlicher Anzahl und der Distributoren.

Qty	Value	Device	Parts	Hersteller/ Distributor
1	-	ADSP-21061 PQFP-240	IC7	Analog Devices/Spoerle
1	-	ARK2	X2	-/Segor
2	-	AM29LV040B-90EC PLCC32	IC3, IC4	AMD/Spoerle
1	-	AMP 171826-4 L04P	SV1	AMP/Spoerle
1	-	B3S-1000SMD	S1	Omron/Segor
1	-	B80C-1500R RB1A	B1	-/Segor
75	0,1uF	C-EUC0805	C3-C35,C37,C38,C40-C72,C74-C79,C82	-/Segor
1	0.047uF	C-EUC0805	C73	-/Segor
1	1uF	C-EUC0805	C1	-/Segor
4	10k	CAY16R4R-103	RN1-RN4	Bourns/Segor
3	330R	CAY16R4R-331	RN14-16	Bourns/Segor
11	4,7k	CAY16R4R-472	RN5-RN13,RN17,RN18	Bourns/Segor
10	10uF	CPOL-EUE2.5-5	C2,C36,C39,C81,C83-C88	-/Segor
1	22uF	CPOL-EUE2.5-6	C80	-/Segor
1	-	CS8402A-CP	IC10	Crystal/Segor
1	-	CS8412-CP	IC9	Crystal/Segor
1	-	DS06	S2	-/Segor
1	-	DS10	S3	-/Segor
1	-	DS1818SOT23	U1	Maxim/Maxim
1	-	F25HP	X1	-/Segor
1	-	FAB64S (DIN41612)	ST1	-/Segor
1	47uH	L-EUL1812	L1	-/Segor
14	-	LED-SMD	D1-D15	-/Segor
1	-	LM317TL	IC1	National Semiconductor/Segor
1	-	LT1084CT-5	IC2	Linear Technology/ Segor
1	-	LT1085CT-3.3	IC14	Linear Technology/ Segor
3	-	MT48LC8M16A2TG-6A	IC11-IC13	Micron/Spoerle
1	-	NEB1R-Buchse	X3	Lumberg/Segor
2	-	PINHD-2X3	JP2, JP3	-/Segor
1	EZICEHDR	PINHD-2X7	JP4	-/Segor
1	-	PINHD-2X9	JP7	-/Segor
2	-	PLCC32-Sockel	IC3,IC4	-/Segor

Tabelle 14: Liste der Bauteile mit Namen, Typenbezeichnung, erforderlicher Anzahl und der Distributoren.

Qty	Value	Device	Parts	Hersteller/ Distributor
2	1k	R-EU_M0805	R64,R76	-/Segor
4	4,7k	R-EU_M0805	R4,R5,R6,R19	-/Segor
1	10k	R-EU_M0805	R75	-/Segor
1	47R	R-EU_M0805	R2	-/Segor
1	100R	R-EU_M0805	R77	-/Segor
1	330R	R-EU_M0805	R30	-/Segor
2	470R	R-EU_M0805	R1,R3	-/Segor
1	6.144MHz	SG531	QG1	Epson/Spezial
1	10.0000MHz	SG531	QG5	Epson/Spezial
1	40.00000MHz	SG531	QG2	Epson/Spezial
1	5.6448MHz	SG531	QG4	Epson/Spezial
1	-	TORX173-T	CON1	Toshiba/Segor
1	-	TOTX173-T	CON2	Toshiba/Segor
1	-	XC18V01JC	IC8	Xilinx/Insight
1	-	XCV100PQ240-4	IC5	Xilinx/Insight
1	-	XCV300PQ240-4	IC6	Xilinx/Insight

Liste der Distributoren:

- Segor Electronics - http://www.segor.de
- Spoerle Electronic - http://www.spoerle.de
- SE Spezial-Electronic AG - http://www.spezial.de
- Insight/Memec - http://www.insight.de.memec.com

A.3 VHDL-Modelle

A.3.1 Entity-Deklaration für das ADSP2106x-Modell

```
entity target_dsp is
generic(
   booting : BOOLEAN := TRUE;
   generic_id: integer := 1;                -- Multiprocessing ID as generic
   adsp2106x: integer := 1;                 -- 0 -> 21060; 1 -> 21061; 2 -> 21062

   -- all timing specifications for 40 MHZ

   -- Clock Input
   tCKmin  : VitalDelayType := 25 ns;       -- clkin period min
   tCKmax  : VitalDelayType := 100 ns;      -- clkin period max
   -- pulse widths
   tCKL    : VitalDelayType := 7 ns;        -- clkin width low min
   tCKH    : VitalDelayType := 5 ns;        -- clkin width high min

   -- Interrupts
   tSIR    : VitalDelayType := 18 ns;       -- n_irqx setup before clkin high
   tHIR    : VitalDelayType := 12 ns;       -- n_irqx hold before clkin high

   -- Timer
   tDTEX   : VitalDelayType := 15 ns;       -- clkin high to timexp delay

   -- Flags
   -- setup times
   tSFI    : VitalDelayType := 8 ns;        -- flagx input setup before clkin
high
   -- hold times
   tHFI    : VitalDelayType := VitalZeroDelay;-- flagx input hold after clkin high
   tHFIWR  : VitalDelayType := VitalZeroDelay;-- flagx input hold after n_rd/n_wr
deasserted
   tHFO    : VitalDelayType := 4 ns;        -- flagx output hold after clkin high
   -- delay times
   tDWRFI  : VitalDelayType := 5 ns;        -- flagx input delay after n_rd/n_wr
low
   tDFO    : VitalDelayType := 16 ns;       -- flagx output delay after clkin
high
   tDFOE   : VitalDelayType := 3 ns;        -- clkin high to flagx output enable
   tDFOD   : VitalDelayType := 14 ns;       -- clkin high to flagx output disable

   -- Memory Read - Bus Master
   -- setup times
   tSADADC : VitalDelayType := VitalZeroDelay;-- address, selects setup before
adrclk high
   -- hold times
   tHDA    : VitalDelayType := 500 ps;      -- data hold from address, selects
   tHDRH   : VitalDelayType := 2 ns;        -- data hold from n_rd high
   tDRHA   : VitalDelayType := VitalZeroDelay;-- address, selects hold after n_rd
high
   -- delay times
   tDAD    : VitalDelayType := 18 ns;       -- address, selects dalay to data
valid
   tDRLD   : VitalDelayType := 12 ns;       -- n_rd low to data valid
   tDAAK   : VitalDelayType := 14 ns;       -- ack delay from address, selects
   tDSAK   : VitalDelayType := 8 ns;        -- ack delay from n_rd/n_wr low
   tDARL   : VitalDelayType := 2 ns;        -- address, selects to n_rd low
   -- pulse widths
   tRW     : VitalDelayType := 12500 ps;    -- n_rd pulse width
   tRWR    : VitalDelayType := 8 ns;        -- n_rd hight to n_wr, n_rd, n_dmagx
low

   -- Memory Write - Bus Master
   -- setup times
   tDDWH   : VitalDelayType := 7 ns;        -- data setup before n_wr high
   -- hold times
   tDWHA   : VitalDelayType := 500 ps;      -- address hold after n_wr deasserted
   -- delay times
   tDAWH   : VitalDelayType := 17 ns;       -- address, selects to n_wr deasser-
ted
   tDAWL   : VitalDelayType := 3 ns;        -- address, selects to n_wr low
```

```
   tDATRWH : TimeArray := (6 ns, 6 ns, 1 ns);-- data disable after n_wr deasserted
   tWWR    : VitalDelayType := 8 ns;         -- n_wr high to n_wr, n_rd, n_dmagx
low
   tDDWR   : VitalDelayType := 5 ns;         -- data disable before n_wr or n_rd
low
   tWDE    : VitalDelayType := -1 ns;        -- n_wr low to data enabled
   -- pulse widths
   tWW     : VitalDelayType := 12 ns;        -- n_wr pulse width

   -- Synchronous Read/Write - Bus Master
   -- setup times
   tSSDATI : VitalDelayType := 3 ns;         -- data setup before clkin
   tSACKC  : VitalDelayType := 6500 ps;      -- ack setup before clkin
   -- hold times
   tHACKC  : VitalDelayType := -1 ns;        -- ack hold after clkin
   tHSDATI : VitalDelayType := 3500 ps;      -- data hold after clkin
   tHADRO  : VitalDelayType := -1 ns;        -- address, n_ms, n_bms, n_sw hold
after clkin
   -- delay times
   tDADRO  : VitalDelayType := 7 ns;         -- address, n_ms, n_bms, n_sw delay
after clkin
   tDPGC   : TimeArray := (16 ns, 16 ns, 9 ns);-- page delay after clkin
   tDRDO   : TimeArray := (4 ns, 4 ns, -2 ns);-- n_rd high delay after clkin
   tDWRO   : TimeArray := (4 ns, 4 ns, -3 ns);-- n_wr high delay after clkin
   tDRWL   : TimeArray := (12500 ps, 12500 ps, 8 ns);-- n_rd/n_wr low delay after
clkin
   tSDDATO : VitalDelayType := 19 ns;        -- data delay after clkin
   tDADCCK : TimeArray := (10 ns, 10 ns, 4 ns);-- adrclk delay after clkin

   -- Synchronous Read/Write - Bus Slave
   -- setup times
   tSADRI  : VitalDelayType := 15 ns;        -- address, n_sw setup before clkin
   tSRWLI  : VitalDelayType := 9500 ps;      -- n_rd/n_wr low setup before clkin
   -- hold times
   tHADRI  : VitalDelayType := 5 ns;         -- address, n_sw hold before clkin
   tHRWLI  : TimeArray := (8 ns, 8 ns, -4 ns);-- n_rd/n_wr low hold after clkin
   -- delay times
   tDACKAD : VitalDelayType := 9 ns;         -- ack delay after address, n_sw
   -- pulse widths (Min)
   tRWHPI  : VitalDelayType := 3 ns;         -- n_rd, n_wr pulse high

   -- Multiprocessor Bus Request and Host Bus Request
   -- setup times (Min)
   tSHBRI  : VitalDelayType := 20 ns;        -- n_hbr setup before clkin
   tSHBGI  : VitalDelayType := 13 ns;        -- n_hbg setup before clkin
   tSBRI   : VitalDelayType := 13 ns;        -- n_br, n_cpa setup before clkin
   tSRPBAI : VitalDelayType := 20 ns;        -- rpba setup before clkin
   -- hold times (Max)
   tHBGRCSV: VitalDelayType := 20 ns;        -- n_hbg low to n_rd/n_wr/n_cs valid
   tHHBRI  : VitalDelayType := 14 ns;        -- n_hbr hold before clkin
   tHHBGI  : VitalDelayType := 6 ns;         -- n_hbg hold before clkin high
   tHBRI   : VitalDelayType := 6 ns;         -- n_br, n_cpa setup before clkin
high
   tHRPBAI : VitalDelayType := 12 ns;        -- rpba hold before clkin
   tHHBGO  : VitalDelayType := -2 ns;        -- n_nbg hold after clkin
   tHBRO   : VitalDelayType := -2 ns;        -- n_br hold after clkin
   -- delay times
   tDHBGO  : VitalDelayType := 7 ns;         -- n_hbg delay after clkin
   tDBRO   : VitalDelayType := 7 ns;         -- n_br delay after clkin
   tDCPAO  : VitalDelayType := 8 ns;         -- n_cpa low delay after clkin
   tDRDYCS : VitalDelayType := 8500 ps;      -- redy low from n_cs and n_hbr low
   tTRDYHG : VitalDelayType := 44 ns;        -- redy disable or redy high from
n_hbg
   tARDYTR : VitalDelayType := 10 ns;        -- redy disable from n_cs or n_nbr
high
   tTRCPA  : TimeArray := (4500 ps, 4500 ps, -2 ns);-- n_cpa disable after clkin

   -- Asynchronous Read/Write - Host to ADSP-2106x
   -- setup times (Min)
   tSADRDL : VitalDelayType := VitalZeroDelay;-- address setup/n_cs low before
n_rd low
   tSCSWRL : VitalDelayType := VitalZeroDelay;-- n_cs low setup before n_wr low
   tSADWRH : VitalDelayType := 5 ns;         -- address setup before n_wr high
   tSDATWH : VitalDelayType := 5 ns;         -- data setup before n_wr high
   -- hold times
   tHADRDH : VitalDelayType := VitalZeroDelay;-- address hold/n_cs hold low after
n_rd
```

```
   tHCSWRH : VitalDelayType := VitalZeroDelay;-- n_cs low hold after n_wr high
   tHADWRH : VitalDelayType := 2 ns;          -- address hold after n_wr high
   tHDATWH : VitalDelayType := 1 ns;          -- data hold after n_wr high
   -- delay times
   tDRDHRDY: VitalDelayType := VitalZeroDelay;-- n_rd high delay after redy
disable
   tSDATRDY: VitalDelayType := 2 ns;          -- data valid before redy disable
from low
   tDRDYRDL: VitalDelayType := 10 ns;         -- redy low after n_rd low
   tHDARWH : VitalDelayType := 2 ns;          -- data disable after n_rd high
   tDWRHRDY: VitalDelayType := VitalZeroDelay;-- n_wr high delay after redy
disable
   tDRDYWRL: VitalDelayType := 10 ns;         -- redy low delay after n_wr/n_cs low
   tSRDYCK : TimeArray := (1 ns, 1 ns, 8 ns);-- redy disable to clkin
   -- pulse widths (Min)
   tWRWH   : VitalDelayType := 6 ns;          -- n_rd/n_wr high width
   tRDYPRD : VitalDelayType := 45 ns;         -- redy low pulse width for read
   tWWRL   : VitalDelayType := 7 ns;          -- n_wr low width
   tRDYPWR : VitalDelayType := 15 ns;         -- redy low pulse width for write

   -- Three-State Timing -- Bus Master, Bus Slave, n_hbr, n_sbts
   -- setup times (Min)
   tSTSCK  : VitalDelayType := 12 ns;         -- n_sbts setup before clkin
   -- hold times
   tHTSCK  : VitalDelayType := 6 ns;          -- n_sbts hold before clkin
   -- delay times
   tMIENA  : VitalDelayType := -1 ns;         -- address/select enable after clkin
   tMIENS  : VitalDelayType := -2 ns;         -- strobes enable after clkin
   tMIENHG : VitalDelayType := -2 ns;         -- n_hbg enable after clkin
   tMITRA  : VitalDelayType := VitalZeroDelay;-- address/select disable after
clkin
   tMITRS  : VitalDelayType := 1500 ps;       -- strobes disable after clkin
   tMITRHG : VitalDelayType := 2 ns;          -- n_hbg disable after clkin
   tDATEN  : VitalDelayType := 9 ns;          -- data enable after clkin
   tDATTR  : TimeArray := (7 ns, 7 ns, 0 ns);-- data disable after clkin
   tACKEN  : VitalDelayType := 7500 ps;       -- ack enable after clkin
   tACKTR  : TimeArray := (6 ns, 6 ns, -1 ns);-- ack disable after clkin
   tADCEN  : VitalDelayType := -2 ns;         -- adrclk enable after clkin
   tADCTR  : VitalDelayType := 8 ns;          -- adrclk disable after clkin
   tMENHBG : VitalDelayType := 19 ns;         -- memory interface enable after
n_hbg high
   tMTRHBG : VitalDelayType := VitalZeroDelay;-- memory interface disable before
n_hbg low

   -- DMA Handshake
   -- setup times
   tSDRLC  : VitalDelayType := 5 ns;          -- n_dmarx low setup before clkin
   tSDRHC  : VitalDelayType := 5 ns;          -- n_dmarx high detup before clkin
   tSDATDGL: VitalDelayType := 10 ns;         -- data setup after n_dmagx low
   -- hold times
   tHDATIDG: VitalDelayType := 2 ns;          -- data hold after n_dmagx high
   tDDGHA  : VitalDelayType := -500 ps;       -- address/select hold after n_dmagx
high
   -- delay times
   tDATDRH : VitalDelayType := 16 ns;         -- data valid after n_dmarx high
   tDMARLL : VitalDelayType := 23 ns;         -- n_dmarx low edge to low edge
   tDDGL   : TimeArray := (15 ns, 15 ns, 9 ns);-- n_dmagx low delay after clkin
   tHDGC   : TimeArray := (6 ns, 6 ns, -2 ns);-- n_dmagx high delay after clkin
   tVDATDGH: VitalDelayType := 8 ns;          -- data valid before n_dmagx high
   tDATRDGH: TimeArray := (7 ns, 7 ns, 0 ns);-- data disable after n_dmagx high
   tDGWRL  : TimeArray := (2 ns, 2 ns, 0 ns);-- n_wr low before n_dmagx low
   tDGWRH  : VitalDelayType := 10 ns;         -- n_dmagx low before n_wr high
   tDGWRR  : TimeArray := (3 ns, 3 ns, 1 ns);-- n_wr high before n_dmagx high
   tDGRDL  : TimeArray := (2 ns, 2 ns, 0 ns);-- n_rd low before n_dmagx low
   tDRDGH  : VitalDelayType := 11 ns;         -- n_rd low before n_dmagx high
   tDGRDR  : TimeArray := (3 ns, 3 ns, 0 ns);-- n_rd high before n_dmagx high
   tDGWR   : VitalDelayType := 5 ns;          -- n_dmagx high to n_wr, n_rd,
n_dmagx low
   tDADGH  : VitalDelayType := 17 ns;         -- address/select valid to n_dmagx
high
   -- pulse widths (Min)
   tWDR    : VitalDelayType := 6 ns;          -- n_dmarx width low (Nonsynchronous)
   tDMARH  : VitalDelayType := 6 ns;          -- n_dmarx width high
   tWDGH   : VitalDelayType := 6 ns;          -- n_dmagx high width
   tWDGL   : VitalDelayType := 12 ns;         -- n_dmagx low width

   -- Link Ports: 1 x CLK Speed Operation
```

```
   -- Receive
   -- setup times
   tSLDCL  : VitalDelayType := 3 ns;         -- data setup before lclk low
   -- hold times
   tHLDCL  : VitalDelayType := 3 ns;         -- data hold after lclk low
   -- delay times
   tDLAHC  : TimeArray := (28500 ps, 28500 ps, 18 ns);-- lack high delay after
clkin high
   tDLALC  : TimeArray := (13 ns, 13 ns, -2 ns);-- lack low delay after lclk high
   -- pulse widths
   tLCLKRWL: VitalDelayType := 6 ns;         -- lclk width low
   tLCLKRWH: VitalDelayType := 5 ns;         -- lclk width high
   -- Transmit
   -- setup times
   tSLACH  : VitalDelayType := 18 ns;        -- lack setup before lclk high
   -- hold times
   tHLACH  : VitalDelayType := -7 ns;        -- lack hold after lclk high
   tHLDCH  : VitalDelayType := -3 ns;        -- data hold after lclk high
   -- delay times
   tDLCLK  : VitalDelayType := 15500 ps;     -- lclk delay after clkin
   tDLDCH  : VitalDelayType := 2500 ps;      -- data delay after lclk high
   -- Receive/Transmit
   -- delay times
   tENDLK  : VitalDelayType := 5 ns;         -- lack, ldat, lclk enable after
clkin
   tTDLK   : VitalDelayType := 20 ns;        -- lack, ldat, lclk disable after
clkin
   -- link port service request interrupts: 1 x and 2 x Speed Operation
   -- setup times
   tSLCK   : VitalDelayType := 10 ns;        -- lack/lclk setup before clkin low
   -- hold times
   tHLCK   : VitalDelayType := 2 ns;         -- lack/lclk hold after clkin low

   -- Link Ports: 2 x CLK Speed Operation
   -- Receive
   -- setup times
   tSLDCL2x: VitalDelayType := 2250 ps;      -- data setup before lclk low
   -- hold times
   tHLDCL2x: VitalDelayType := 2250 ps;      -- data hold after lclk low
   -- delay times
   tDLAHC2x: TimeArray := (28500 ps, 28500 ps, 18 ns);-- lack high delay after
clkin high
   tDLALC2x: TimeArray := (16 ns, 16 ns, 6 ns);-- lack low delay after lclk high
   -- pulse widths
   tLCLKRWL2x: VitalDelayType := 4500 ps;    -- lclk width low
   tLCLKRWH2x: VitalDelayType := 4 ns;       -- lclk width high
   -- Transmit
   -- setup times
   tSLACH2x: VitalDelayType := 8 ns;         -- lack setup before lclk high
   -- hold times
   tHLACH2x: VitalDelayType := -7 ns;        -- lack hold after lclk high
   tHLDCH2x: VitalDelayType := -2 ns;        -- data hold after lclk high
   -- delay times
   tDLCLK2x: VitalDelayType := 8 ns;         -- lclk delay after clkin
   tDLDCH2x: VitalDelayType := 2250 ps;      -- data delay after lclk high

   -- Serial Ports
   -- External Clock
   tSFSE   : VitalDelayType := 3500 ps;      -- tfs/rfs setup before tclk/rclk
   tHFSE   : VitalDelayType := 4 ns;         -- tfs/rfs hold after tclk/rclk
   tSDRE   : VitalDelayType := 1500 ps;      -- receive data setup before rclk
   tHDRE   : VitalDelayType := 4 ns;         -- receive data hold after rclk
   tSCLKW  : VitalDelayType := 9 ns;         -- tclk/rclk width
   tDDTE   : VitalDelayType := 16 ns;        -- transmit data delay after tclk
   tHODTE  : VitalDelayType := 5 ns;         -- transmit data hold after tclk
   -- Internal Clock
   tSFSI   : VitalDelayType := 8 ns;         -- tfs setup before tclk; rfs setup
before rclk
   tHFSI   : VitalDelayType := 1 ns;         -- tfs/rfs hold after tclk/rclk
   tSDRI   : VitalDelayType := 3 ns;         -- receive data setup before rclk
   tHDRI   : VitalDelayType := 3 ns;         -- receive data hold after rclk
   tDFSI   : VitalDelayType := 4500 ps;      -- tfs delay after tclk (internally
generated)
   tHOFSI  : VitalDelayType := -1500 ps;     -- tfs hold after tclk (internally
generated)
   tDDTI   : VitalDelayType := 7500 ps;      -- transmit data delay after tclk
   tHDTI   : VitalDelayType := 0 ns;         -- transmit data hold after tclk
```

```
   -- External or Internal Clock
   tDFSE   : VitalDelayType := 13 ns;        -- tfs/rfs delay after rclk (inter-
nally generated)
   tHOFSE  : VitalDelayType := 3 ns;         -- tfs/rfs hold after rclk (inter-
nally generated)
   -- Enable & Three-State
   tDDTEN  : VitalDelayType := 4500 ps;      -- data enable from external tclk
   tDDTTE  : VitalDelayType := 10500 ps;     -- data disable from external tclk
   tDDTIN  : VitalDelayType := 0 ns;         -- data enable from internal tclk
   tDDTTI  : VitalDelayType := 3 ns;         -- data disable from internal tclk
   tDCLK   : VitalDelayType := 22 ns;        -- tclk/rclk delay from clkin
   tDPTR   : VitalDelayType := 17 ns;        -- sport disable after clkin
   -- External Late Frame Sync
   tDDTLFSE: VitalDelayType := 12 ns;        -- data delay from late external tfs
   tDDTENFS: VitalDelayType := 3500 ps;      -- data enable from late fs

   -- JTAG Test Access Port & Emulation
   tSTAP   : VitalDelayType := 5 ns;         -- tdi, tms setup before tck high
   tHTAP   : VitalDelayType := 6 ns;         -- tdi, tms hold after tck high
   tSSYS   : VitalDelayType := 7 ns;         -- system inputs setup before tck low
   tHSYS   : VitalDelayType := 18 ns;        -- system inputs hold after tck low
   tDTDO   : VitalDelayType := 13 ns;        -- tdo delay from tck low
   tDSYS   : VitalDelayType := 18500 ps;     -- system outputs delay after tck low

   -- generic control parameters
   InstancePath: STRING  := "SHARC DSP:";
   TimingChecksOn: BOOLEAN := FALSE;
   MsgOn   : BOOLEAN := TRUE;
   XOn     : BOOLEAN := TRUE;
   TimingModel: STRING  := "UNIT";
   SimCondition: SimConditionType := WorstCase);

   port(   n_reset                           : in std_logic;
   clkin   : in std_logic;
   n_sbts  : in std_logic;                   -- Suspend Bus Tristate

   -- config pins
   id      : in std_logic_vector(2 downto 0);-- Multiprocessing ID
   rpba    : in std_logic;                   -- Rotating Priority Bus Arbitration
Select
   eboot   : in std_logic;                   -- EPROM Boot Select
   lboot   : in std_logic;                   -- Link Boot - Host Boot Select

   -- address/data i/o
   address_bus: inout std_logic_vector(31 downto 0);
   data_bus: inout std_logic_vector(47 downto 0);

   -- memory control
   n_rd    : inout std_logic;                -- Read Strobe
   n_wr    : inout std_logic;                -- Write Strobe
   ack     : inout std_logic;                -- Acknowledge
   n_sw    : inout std_logic;                -- Synchronous Write Select
   n_bms   : inout std_logic;                -- Boot Memory Select
   n_ms    : out std_logic_vector(3 downto 0);-- Select Lines
   page    : out std_logic;                  -- DRAM Page Boundary
   adrclk  : out std_logic;                  -- Clock Output Reference

   -- DMA
   n_dmar1 : in std_logic;                   -- DMA Request 1
   n_dmar2 : in std_logic;                   -- DMA Request 2
   n_dmag1 : out std_logic;                  -- DMA Grant 1
   n_dmag2 : out std_logic;                  -- DMA Grant 2

   -- multiprocessing
   n_br    : inout std_logic_vector(6 downto 1);-- Multiprocessing Bus Request
   n_cpa   : inout std_logic;                -- Core Priority Access

   -- host interface
   n_cs    : in std_logic;
   n_hbr   : in std_logic;
   n_hbg   : inout std_logic;
   redy    : out std_logic;

   -- serial ports 0, 1
   drx     : in std_logic_vector(1 downto 0);-- Data Receive
   tclkx   : inout std_logic_vector(1 downto 0);-- Transmit Clock
   rclkx   : inout std_logic_vector(1 downto 0);-- Receive Clock
```

```
   tfsx    : inout std_logic_vector(1 downto 0);-- Transmit Frame Sync
   rfsx    : inout std_logic_vector(1 downto 0);-- Receive Frame Sync
   dtx     : out std_logic_vector(1 downto 0);-- Data Transmit

   -- link ports 0-5
   lxdat0  : inout std_logic_vector(5 downto 0);-- Link Port Data 0
   lxdat1  : inout std_logic_vector(5 downto 0);-- Link Port Data 1
   lxdat2  : inout std_logic_vector(5 downto 0);-- Link Port Data 2
   lxdat3  : inout std_logic_vector(5 downto 0);-- Link Port Data 3
   lxclk   : inout std_logic_vector(5 downto 0);-- Link Port Clock
   lxack   : inout std_logic_vector(5 downto 0);-- Link Port Acknowledge

   -- jtag
   tck     : in std_logic;
   tms     : in std_logic;
   tdi     : in std_logic;
   n_trst  : in std_logic;
   tdo     : out std_logic;

   -- other
   n_irq   : in std_logic_vector(2 downto 0);   -- Interrupt Request Lines
   flag    : inout std_logic_vector(3 downto 0);-- Flag Pins
   timexp  : out std_logic;                     -- Timer Expired
   n_emu   : out std_logic;                     -- Emulation Status
   icsa    : out std_logic                      -- reserved, leave unconnected
   );
end target_dsp;
```

A.3.2 Entity-Deklaration für den SelectMap-Port des FPGA

```
entity select_map is
generic(-- Virtex Series Device specificXAPP151 p.6 table 3
   -- all values for XCV50
   row                 : integer := 16;
   column              : integer := 24;
   ram_column          : integer := 2;-- VIRTEX: always 2; VIRTEX-E: >0
   ram_space           : integer := 0;-- >0: VIRTEX-E Family; 0: VIRTEX
   virtex_e            : BOOLEAN := FALSE;-- VIRTEX-E Family?

   memory_clear_delay : time := 100 ns;

   -- D0-7 Setup/Hold
   tSMDCC              : VitalDelayType := 5.0 ns;-- min setup
   tSMCCD              : VitalDelayType := 0 ns;-- min hold

   -- CS Setup/Hold
   tSMCSCC             : VitalDelayType := 7.0 ns;-- min setup
   tSMCCCS             : VitalDelayType := 0 ns;-- min hold

   -- WRITE Setup/Hold
   tSMCCW              : VitalDelayType := 7.0 ns;-- min setup
   tSMWCC              : VitalDelayType := 0 ns;-- min hold

   tSMCKBY             : VitalDelayType := 12.0 ns;-- BUSY Prop Delay

   tCC                 : VitalDelayType := 15 ns;-- CCLK min cycle time (66
MHz)
   tCCNH               : VitalDelayType := 20 ns;-- CCLK min cycle time
without handshake (50 Mhz)

   -- generic control parameters
   InstancePath        : STRING  := "XILINX Virtex:";
   TimingChecksOn      : BOOLEAN := FALSE;
   MsgOn               : BOOLEAN := TRUE;
   XOn                 : BOOLEAN := TRUE;
   TimingModel         : STRING  := "UNIT"
);

port(m2                : in std_logic;
```

```
  m1                : in std_logic;
  m0                : in std_logic;-- mode bits
  select_map_data   : inout std_logic_vector(7 downto 0);
  n_cs              : in std_logic;
  n_write           : in std_logic;
  n_prog            : in std_logic;
  cclk              : inout std_logic;
  n_init            : inout std_logic;-- open drain
  done              : inout std_logic;-- open drain or active drive high
(select via COR Register)
  busy              : out std_logic;-- only needed if cclk is greater
than 50 MHz (open drain)

  gts               : out std_logic;-- de/activating all I/Os in top
level of FPGA
  error_mode        : out BOOLEAN -- indicate fail safe
);
end select_map;
```

A.3.3 Entity-Deklaration für das Modell des Controller-FPGA

```
entity controller_fpga is
generic(
  -- synopsys translate_off
  booting                       : BOOLEAN := TRUE;
  -- synopsys translate_on

  fpga_eeprom_address_width: integer := 19;
  dsp_eeprom_address_width: integer := 19;
  select_map_cclk_divider: integer := 2);
port (  -- local board signals
  clock             : in std_logic;
  n_reset           : in std_logic;

  -- EPP signals
  epp_data          : inout std_logic_vector(7 downto 0);
  n_address_strobe  : in std_logic;
  n_data_strobe     : in std_logic;
  n_write           : in std_logic;
  n_wait            : out  std_logic;
  intr              : out  std_logic;

  -- system bus signals
  address_system_bus : inout std_logic_vector(31 downto 0);
  data_system_bus    : inout std_logic_vector(31 downto 16);-- 16-bit
host bus

  -- hostbus control
  redy              : in std_logic;
  n_hbg             : in std_logic;
  n_hbr             : out std_logic;
  n_rd              : out std_logic;
  n_wr              : out std_logic;

  -- hostbus read/write control (dsp)
  dsp_n_cs                      : out std_logic;

  -- hostbus read/write control (fpga)
  fpga_n_cs         : out std_logic;

  -- signals to local fpga flash eeprom
  fpga_eeprom_data   : inout std_logic_vector(7 downto 0);
  fpga_eeprom_address: out std_logic_vector(fpga_eeprom_address_width-1
downto 0);
  n_fpga_eeprom_cs   : out std_logic;
  n_fpga_eeprom_we   : out std_logic;
```

```
  n_fpga_eeprom_oe   : out std_logic;

  -- signals to local dsp flash eeprom
  n_dsp_eeprom_cs    : out std_logic;
  n_dsp_eeprom_we    : out std_logic;

  -- signals from/to FPGA (Select Map)
  n_init             : in std_logic;
  done               : in std_logic;
  cclk               : out std_logic;
  n_prog             : out std_logic;
  n_fpga_cs          : out std_logic;
  n_fpga_write       : out std_logic;
  n_fpga_reset       : out std_logic;

  -- signals from/to DSP
  timexp             : in std_logic;-- must be set by target dsp after
startup
  n_irq              : out std_logic;
  n_dsp_reset        : out std_logic;

  -- only debug
  Leds               : out std_logic_vector(3 downto 0));
end controller_fpga;
```

A.4 CORDIC

A.4.1 Modul cordic.c

```
#include <stdlib.h>
#include <math.h>
#include <21060.h>
#include <macros.h>
#include <signal.h>

#include "shared_var.h"

#define PI 3.14159265358979323846264338327795

// prototypes
void sin_cos_fpga(int angle, double *sin, double *cos);
void process_timer();

// global variables
double sin_result_fpga;
double cos_result_fpga;
double sin_result_dsp;
double cos_result_dsp;
int tcount_fpga;
int tcount_dsp;
int save_reg1, save_reg2;

int main() {
   int grad = 0;
   double grad_dbl =  0;

   interrupt(SIG_TMZ, SIG_IGN);// ignore timer interrupt

   // init start values
   //////////////
   // 45° = pi/4
   //////////////
   grad = 0x20000000;// 45° -> ((45° * pow(2,32)) / 360) = 0x20000000 !!
   grad_dbl = PI/4;

   // write new value into WAIT System Register
   asm("ustat1 = dm(0x02);
      ustat2 = 0x21ad6b45;// set external wait states to 1; internal wait
states only
      dm(0x02) = ustat2;");

   if(timer_set(0x100,0x100) != 1)
      timer_on();

   // function under test
   sin_cos_fpga(grad, &sin_result_fpga, &cos_result_fpga);

   timer_off();

   asm("dm(_save_reg1)=r1;
      dm(_save_reg2)=r2;
      r1=0x100;
      r2=tcount;
      r2=r1-r2;
      dm(_tcount_fpga)=r2;
      r1=dm(_save_reg1);
      r2=dm(_save_reg2);");

   asm("dm(0x02) = ustat1;");// restore wait register

   if(timer_set(0x100,0x100) != 1)
```

```
    timer_on();

  // same but on dsp
  sin_result_dsp = sin(grad_dbl);
  cos_result_dsp = cos(grad_dbl);

  timer_off();

  asm("dm(_save_reg1)=r1;
    dm(_save_reg2)=r2;
    r1=0x100;
    r2=tcount;
    r2=r1-r2;
    dm(_tcount_dsp)=r2;
    r1=dm(_save_reg1);
    r2=dm(_save_reg2);");

  set_flag(SET_FLAG2, SET_FLAG);
  set_flag(SET_FLAG3, CLR_FLAG);
  if(timer_set(10000000,10000000) != 1)
    timer_on();
  interrupt(SIG_TMZ, process_timer);// blink leds (optional)
  asm("bit set imask 0x100;");

  return;
}

void sin_cos_fpga(int angle, double *sin, double *cos) {
  int i;

  operand = angle;

  for(i=0;i<31;i++) {
    asm("nop;");// wait for 32 cycles; do nothing
  }

  // get result from fpga
  *sin = FLOATBY(sin_result, -31);
  *cos = FLOATBY(cos_result, -31);
  return;
}

void process_timer() {
  set_flag(SET_FLAG2, TGL_FLAG);
  set_flag(SET_FLAG3, TGL_FLAG);
}
```

A.5 MPEG1-Layer3-Player

A.5.1 I/O-Modul mp3_sharc.c in Phase III

```
#include "mpg123_sharc.h"
#include "mpglib_sharc.h"

struct mpstr mp;

#ifdef WIN32
unsigned int out[2304];
unsigned char out_char[4608];
#else
#include <21060.h>

char input_data[4];
extern unsigned int out_data[2304];
extern unsigned int out_data2[2304];// double buffer for output
extern int xmit_tcb1[8];
extern int xmit_tcb2[8];
extern int control;
extern int status;

pm int transfer_started=0;// flag if transfer started (1) or not (0)
pm int decode_buffer=1;// currently output_buffer for decode (1 =
out_data; 2 = out_data2)
pm int output_buffer=2;// currently DMAed output_buffer (1 = out_data; 2 =
out_data2)
pm int out_data_size=sizeof(out_data), out_data2_size=sizeof(out_data2);/
/ size of valid data in out_data and out_data2

// prototype
void input_data_proc();
void output_data_proc();
void process_timer();
void dummy();
#endif

extern void InitMP3(struct mpstr *mp);
extern int decodeMP3(struct mpstr *mp, char *in, int isize, unsigned int
*out, int osize, int *done);

#ifndef WIN32

int main(void)
{
   int fpga_control;

   InitMP3(&mp);

   asm("#include <def21060.h>");
   asm("r0=0x21ad6b45;");// set external wait states to 1; internal wait
states only
   asm("dm(WAIT)=r0;");

   control = 0x1;// enable fifo in target-fpga

   set_flag(SET_FLAG2, CLR_FLAG);
   set_flag(SET_FLAG3, SET_FLAG);

   // 2048 Samples = 1550000 Zyklen = 46,5 ms
   // setup timer and wait for a period and then check for data
   interrupt(SIG_TMZ, dummy);
   if(timer_set(1550000, 1550000) != 1)
     timer_on();
```

```
  // wait for a little time
  idle();

  timer_off();

  set_flag(SET_FLAG2, SET_FLAG);
  set_flag(SET_FLAG3, CLR_FLAG);

  // check if fifo is enabled
  fpga_control = control;
  if((fpga_control & 0x1) == 0x1) {// yes, fifo enabled
    input_data_proc();
  }

  control = 0x0;// disable fifo in target-fpga

  asm("r0=0x21ad6b5a;");// reset WAIT register to default value
  asm("dm(WAIT)=r0;");

  interrupt(SIG_TMZ, process_timer);// blink leds
  if(timer_set(10000000,10000000) != 1)
    timer_on();
  asm("bit set imask 0x100;");
  return 0;
}

void input_data_proc()
{
  int ret, size;
  unsigned int in_data;
  int fpga_status;
  int buffer_cleared = 0;

  for(;;) {
    for(;;) {
      in_data = read_fpga();
      if(in_data == 0x11ee22dd) {
        // got data that is reserved for an empty fifo
        fpga_status = status;
        if(((fpga_status >> 3) & 0x1) == 0x1) {// yes, fifo is blocked
          control = 0x3;// clear fifo block situation

          // wait for a period before continue (period of xx cycles; xx
= 800*samples)
          if(timer_set(160000, 160000) != 1)
            timer_on();

          idle();

          timer_off();
        } else {
          break;// must be valid data -> continue
        }
      } else {
        break;
      }
    }

    input_data[0] = (in_data >> 24) & 0xff;
    input_data[1] = (in_data >> 16) & 0xff;
    input_data[2] = (in_data >> 8) & 0xff;
    input_data[3] = (in_data) & 0xff;

    if(decode_buffer == 1)
      ret = decodeMP3(&mp, input_data, 4, out_data, sizeof(out_data),
&size);
    else
      ret = decodeMP3(&mp, input_data, 4, out_data2, sizeof(out_data2),
&size);
```

```
     if(ret == MP3_OK) {
        if(size == sizeof(out_data)) {
          buffer_cleared = 0;// output_buffer has valid data, reset
buffer_cleared
          control = 0x3;// clear block fifo status
          if(decode_buffer == 1) {
            decode_buffer = 2;
            out_data_size = size;
          } else {
            decode_buffer = 1;
            out_data2_size = size;

            // enable output process, if not already done, after both out-
put buffer filled
            if(transfer_started == 0) {
              init_sport0(&out_data, &out_data2);
              output_buffer = 1;
              interrupt(SIG_SPT0I, output_data_proc);
              transfer_started = 1;
            }
          }
        }
     } else if(ret == MP3_ERR) {
        if(transfer_started == 1 && buffer_cleared == 0) {// clear output
buffer only once
          memset(out_data, 0, sizeof(out_data));
          memset(out_data2, 0, sizeof(out_data2));

          buffer_cleared = 1;
        }
        InitMP3(&mp);// initialize new
     }
   }
//return ;
}

void output_data_proc()
{
   // current DMA transfer complete or first time
   // switch for next transfer to other buffer
   if(output_buffer == 2) {
     output_buffer = 1;
   }
   else if(output_buffer == 1) {
     output_buffer = 2;
   }
   return;
}

void process_timer() {
   set_flag(SET_FLAG2, TGL_FLAG);
   set_flag(SET_FLAG3, TGL_FLAG);

   return;
}

void dummy() {
   return;
}

#else
int main(void)
{
   int size, i;
   int len,ret;
   char buf[4];
   FILE *mp3_file, *wav_file;
```

```
  InitMP3(&mp);

  if(mp3_file == NULL || wav_file == NULL)
    return(-1);

  while(1)
  {
    len = fread(&buf, sizeof(char), sizeof(buf), mp3_file);
    if(len <= 0)
      break;

    ret = decodeMP3(&mp, buf, len, out, sizeof(out), &size);
    while(ret == MP3_OK)
    {
      for(i=0;i<size && i<2304;i++) {
        // lsb first on intel
        out_char[i*2] = (unsigned char) (out[i] >> 16);
        out_char[(i*2)+1] = (unsigned char) (out[i] >> 24);
      }
      fwrite(&out_char, sizeof(char), size*2, wav_file);
      ret = decodeMP3(&mp, NULL, 0, out, sizeof(out), &size);
    }
  }
  fprintf(stdout, "Successfull decoded!\n");
  return 0;
}
#endif
```

A.5.2 Schnittstelle interface_sharc.c

```
#include <stdlib.h>
#include <stdio.h>

#include "mpg123_sharc.h"
#include "mpglib_sharc.h"

/* Global mp */
struct mpstr *gmp;

void InitMP3(struct mpstr *mp)
{
  memset(mp,0,sizeof(struct mpstr));
  mp->fsizeold = -1;
  mp->fr.single = -1;
  mp->synth_bo = 1;
  return;
}

int decodeMP3(struct mpstr *mp, char *in, int isize, unsigned int *out,
int osize, int *done)
{
  unsigned long head;
  int i;
  static unsigned char head_tmp[4];

  gmp = mp;

  /* First decode header */
  if(mp->framesize == 0) {
    if(isize < 4) {
#ifndef BE_QUIET
      fprintf(stderr,"To less input data\n");
#endif
      return MP3_ERR;
    }
    if(mp->bsize > 0) {
```

```
        for(i=0;i<mp->bsize;i++) {
          head <<= 8;
          head |= (0x000000ff & head_tmp[i]);
        }
      }
      for(i=0;i<(4-mp->bsize);i++) {
        head <<= 8;
        head |= (0x000000ff & *(in+i));
      }

      mp->header = head;
      if(decode_header(&mp->fr,mp->header) <= 0)
        return MP3_ERR;

      mp->framesize = mp->fr.framesize;

      if(mp->fr.framesize == 0 || mp->fr.framesize > MAXFRAMESIZE)
        return MP3_ERR;

      wordpointer = mp->bsspace[mp->bsnum] + 512;
      if(mp->bsize > 0) {
        // add incoming data to frame buffer
        memcpy(wordpointer,in+(4-mp->bsize),mp->bsize);
      }
      return MP3_NEED_MORE;
    }

    // add incoming data to frame buffer
    memcpy(wordpointer+mp->bsize, in, isize);
    mp->bsize += isize;
    if(mp->fr.framesize > mp->bsize) {
      return MP3_NEED_MORE;
    }
    if(mp->bsize > mp->fr.framesize) {
      memcpy(head_tmp,wordpointer+mp->fr.framesize,mp->bsize-mp->fr.frame-
size);
    }
    mp->bsnum = (mp->bsnum + 1) & 0x1;
    bitindex = 0;
    if(mp->fr.error_protection)
      getbits(16);

#ifndef WIN32
    {
      // defined in mp3_sharc.c
      extern int transfer_started;
      extern int decode_buffer;
      extern int output_buffer;

      if(transfer_started == 1 && decode_buffer == output_buffer) {
        // stop here until output_buffer switched (spt0 interrupt and ISR
output_data_proc finished)
        idle();
      }
    }
#endif

    *done = 0;
    // see if error and return it
    if ((&mp->fr)->do_layer(&mp->fr, out, done) < 0)
      return MP3_ERR;

    mp->bsize -= mp->framesize;
    mp->fsizeold = mp->framesize;
    mp->framesize = 0;
    return MP3_OK;
}
```

Literaturverzeichnis

[1] *Actel:* FPGA and Package Selector Guide. http://www.actel.com/products/selguide/selguide.pdf, März 2003

[2] *Aldec:* 8051 VHDL Core. http://www.aldec.com/IP_Services/Datasheets/8051.pdf

[3] *AllianceCORE:* Partnerships for Complete Xilinx Programmable Logic Solutions. http://www.xilinx.com/products/logicore/alliance/tblpart.htm

[4] *Altera:* Component Selector Guide. http://www.altera.com/literature/sg/csg.pdf, Februar 2003

[5] *AMBA:* A Standard for System-on-Chip Bus. ARM Ltd., http://www.arm.com/armtech/AMBA?OpenDocument

[6] *Amory, Alexandre; Oliveira, Leandro; Moraes, Fernando:* A Heterogeneous and Distributed Co-Simulation Environment. SBCCI'2002, Porto Alegre, XV Symposium On Integrated Circuits And System Design, 2002

[7] *Analog Devices:* ADSP-2106x SHARC User's Manual. Second Edition, Juli 1996

[8] *Analog Devices:* ADSP-21061 SHARC Preliminary Data Sheet. Oktober 1996

[9] *Anant, A.:* Presentation to EDAC Meeting. Sun Microsystems, 11. April 2000

[10] *Andraka, Ray:* A survey of CORDIC algorithms for FPGAs. FPGA '98, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, 22.-24. Feb. 1998, Monterey, CA. S.191-200

[11] *ARC:* 32-Bit RISC Processor Soft-Core for FPGA. http://www.xilinx.com/ipcenter/catalog/search/alliancecore/arc_32_bit_configurable_risc_processor.htm

[12] *Ashenden, P.J.:* The Designer's Guide to VHDL Second Edition. Morgan Kaufmann Publishers, 2002

[13] *Atmel:* CD-ROM Data Book. Atmel, November 2002

[14] *Atmel, IP Cores:* IP Cores. Atmel, http://www.atmel.com/products/IPCores/

[15] *Auguin, M.; Capella, L.; Cuesta, F.; Gresset, E.:* CODEF: A system level design space exploration tool. IEEE International Conference on Acoustics, Speech, and Signal Processing, 2001. Proceedings, (ICASSP '01), Vol. 2, S. 1145ff., Mai 2001

[16] *Balarin, F.; , Sentovich, E.; Chiodo, M.; Giusto, P.; Hsieh, H.; Tabbara, B.; Jurecska, A.; Lavagno, L.; Passerone, C.; Suzuki, K.; Sangiovanni-Vincentelli, A.:* Hardware-Software Co-design of Embedded Systems -- The POLIS approach. Kluwer Academic Publishers, 1997

[17] *Baleani, M.; Gennari, F.; Jiang, Y.; Patel, Y.; Brayton, R.K.; Sangiovanni-Vincentelli, A.:* HW/SW Partitioning and Code Generation of Embedded Control Applica-

tions on a Reconfigurable Architecture Platform. Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES), Estes Park, Colorado, USA, May, 2002

[18] *BDTi:* Benchmark Scores: BDTimark 2000. Berkeley Design Technology, Inc., http://www.bdti.com/bdtimark/BDTImark2000graphic.pdf, 2000

[19] *BDTi:* Buyer's Guide to DSP Processors, 2001 Edition. Berkeley Design Technology, Inc., 2001

[20] *BDTi:* Choosing a DSP Processor. Berkeley Design Technology, Inc., http://www.bdti.com/articles/choose_2000.pdf, 2000

[21] *BDTi:* Pocket Guide to Processors for DSP. Berkeley Design Technology, Inc., http://www.bdti.com/pocket/pocket2002.pdf, September 2002

[22] *BDTi:* Processor Overviews. Berkeley Design Technology, Inc., http://www.bdti.com/procsum/index.htm, Mai 2002

[23] *BDTi:* The BDTimark 2000: A Summary Measure of DSP Speed. Berkeley Design Technology, Inc., http://www.bdti.com/bdtimark/BDTImark2000.pdf, Februar 2003

[24] *BDTi:* VLIW Architectures for DSP. Berkeley Design Technology, Inc., http://www.bdti.com/articles/vliw_icspat00.pdf, Oktober 2000

[25] *Beazley, Ray:* Simplified Wrapper and Interface Generator (SWIG). http://www.swig.org

[26] *Bechtolsheim, A.; Raza, A.:* Addressing Complex System and IC Verification Bottlenecks. Cisco Systems, September 2001

[27] *Berry, G.; Gonthier, G.:* The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming, Vol. 19, No. 2, S. 87ff., 1992

[28] *Beyer, M.:* Entwicklung eines Videosignal-Framegrabbers für die EPP-Schnittstelle eines PCs. TU Berlin, Diplomarbeit am Institut für Technische Informatik, September 1997

[29] *Beyer, M.; Post, H.-U.:* VHDL-Hardware/Software-Board-Level-Simulation innerhalb eines FPGA/DSP-Entwicklungssystems. In GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Bremen, Shaker Verlag, S. 154, 2003

[30] *Bolotin, G.:* A processor interface model for fast system simulations. ASIC Conference and Exhibit, 1993, Proceedings, Sixth Annual IEEE International, S. 519ff., September 1993

[31] *Bolsens, I. et al.:* Hardware/Software Co-Design of Digital Telecommunication Systems. Proceedings of the IEEE, Vol. 85, No. 3, S. 391ff., März 1997

[32] *Bradbury, Robert J.:* Robert's SIA Roadmap. http://www.aeiveos.com/~bradbury/petaflops/siardmap.html, 2002

[33] *Callahan, Timothy J.; Wawrzynek, John:* Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia, Vol 1482 of Lecture Notes in Computer Science, Springer-Verlag, September 1998

[34] *Cast:* C8051 Legacy Microcontroller Core. http://www.cast-inc.com/cores/c8051/index.shtml

[35] *Chou, C.; Mohanakrishnan, S.; Evans, J.B.:* FPGA Implementation of Digital Filters. Proc. Int. Conf. Signal Proc. Appl. & Tech. (ICSPAT'93), 1993

[36] *Clark, James:* expat - XML Parser Toolkit. http://www.jclark.com/xml/expat.html

[37] *CoreConnect:* IBM CoreConnect bus architecture for system-on-chip designs. IBM Corp., http://www.chips.ibm.com/products/coreconnect/

[38] *COSYMA:* COSYnthesis for eMbedded Architectures. TU Braunschweig, http://www.ida.ing.tu-bs.de/projects/cosyma/home.g.shtml

[39] *CoWare N2C:* Flexible Platform-Based Design With the CoWare N2C Design System. White Paper, CoWare, Inc., Oktober 2000

[40] *Crystal:* Digital Audio Interface Receiver CS8411/CS8412. Oktober 1998

[41] *Crystal:* Digital Audio Interface Transmitter CS8401A/CS8402A. November 1993

[42] *Cygnal:* http://www.cygnal.com/products/productguide.asp

[43] *Cypress:* Cypress Data Book CD-ROM. Cypress, http://www.cypress.com/, 2002

[44] *Dalton Project:* Intel 8051 Model. University of California, http://www.cs.ucr.edu/~dalton/i8051/

[45] *Design and Reuse:* Design and Reuse Silicon IP Catalog. http://www.us.design-reuse.com/sip/

[46] *Dido, J.; Géraudie, N.; Loiseau, L.; Payeur, O.; Savaria, Y.; Poirier, D.:* A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA-based DSPs. FPGA, Monterey, CA, Proceedings, S. 50ff., http://www.sigda.org/Archives/ProceedingArchives/Fpga/Fpga2002/papers/2002/fpga02/pdffiles/2_3.pdf, 2002

[47] *Dipert, Brian:* Counting on gate counts? Don't count on it. EDN, S. 52ff., 3. August 1998

[48] *Elektronik:* Der DSP-Report 2001. Elektronik Heft 23/2001, S. 62ff., 2001

[49] *Elektronik:* Der ASIC-Report. Teil 1, Elektronik Heft 15/2001, S. 62ff., 2001

[50] *Elektronik:* Der PLD-Report. Elektronik Heft 5/2001, S. 84ff., 2001

[51] *EPP:* IEEE Std 1284-2000, IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers. IEEE, New York, USA, 2000

[52] *European Space Agency, Sandi Habinc, Peter Sinander:* Using VHDL for Board Level Simulation. IEEE Design & Test of Computers, Vol. 13, Issue: 3, S. 66ff., Fall 1996

[53] *European Space Agency, Peter Sinander:* VHDL Modelling Guidelines. ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.pdf, September 1994

[54] *European Space Agency, Sandi Habinc, Peter Sinander:* VHDL Models for Board-Level Simulation. ftp://ftp.estec.esa.nl/pub/vhdl/doc/BoardLevel.pdf, Februar 1996

[55] *Eyre, J.:* The Digital Signal Processor Derby. IEEE Spectrum, Juni 2001, S.62-68

[56] *Eyre, J.; Bier, J.:* The Evolution of DSP Processors. IEEE Signal Processing Magazine, S. 46ff., März 2000

[57] *Free-IP:* The Free-IP Project. http://www.free-ip.com/cores.htm

[58] *Free Model Foundry:* http://www.eda.org/fmf

[59] *FPGACPU:* FPGA CPU Links. http://www.fpgacpu.org/links.html

[60] *Garp:* Automatic C Compilation to SW + Reconfigurable HW. BRASS Research Group, http://brass.cs.berkeley.edu/compile.html

[61] *Givargis, T.; Vahid, F.:* Platune: A Tuning Framework for System-on-a-Chip Platforms. IEEE Transactions on Computer Aided Design (TCAD), Vol. 21, No. 11, November 2002

[62] *Goldfarb, C.F.; Prescod, P.:* XML-Handbuch. Prentice Hall 1999

[63] *Gupta, R.; De Micheli, G.:* System-level synthesis using re-programmable components. In Proc. of the European Conference on Design Automation (EDAC), S. 2ff., 1992

[64] *Hendrich, N.:* HADES - A Java-based visual simulation environment, Fachbereich Informatik, Universität Hamburg, http://tams-www.informatik.uni-hamburg.de/applets/hades/html/hades.html, 1997

[65] *Henkel, J.; , Ernst, R.; Trawny, W. Ye, M.; Benner, Th.:* COSYMA: Ein System zur Hardware/Software Co-Synthese. GME Fachbericht Nr. 15 Mikroelektronik, S. 167ff., 1995

[66] *Hennessy, John L.;Patterson, David A.:* Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA, 1996

[67] *Ho, C.H.; Leong, M.P.; Leong, P.H.W.; Becker, J.; Glesner, M.:* Rapid Prototyping of FPGA based Floating Point DSP Systems, Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping (RSP'02), Darmstadt, S. 19ff., http://www.cse.cuhk.edu.hk/~phwl/papers/float_rsp02.pdf, 2002

[68] *Hoffmann, A.; Kogel, T.; Meyr,H.:* A framework for fast hardware-software co-simulation. Design, Automation and Test in Europe, 2001, Conference and Exhibition 2001, Proceedings, S. 760ff., März 2001

[69] *Hwang, J.; , Milne, B.; Shirazi, N.; Stroomer, J.:* System Level Tools for DSP in FPGAs. FPL 2001, Lecture Notes in Computer Science, S.534ff., 2001

[70] *IEEE-Standards:* http://ieeexplore.ieee.org

[71] *Incisive-SPW:* Signal Processing Worksystem. Cadence, http://www.cadence.com/products/incisive_spw.html

[72] *ITRS:* International Technology Roadmap for Semiconductors. Semiconductor Industry Association (SIA), http://public.itrs.net, 2001

[73] *Jansen, Dirk (Hrsg.):* Handbuch der Electronic Design Automation. Carl Hanser Verlag, Februar 2001

[74] *JTAG:* IEEE Std 1149.1-1990, IEEE standard test access port and boundary-scan architecture. IEEE, New York, USA, 1990

[75] *Lattice:* Product Selector Guide. http://www.latticesemi.com/lit/docs/generalinfo/select.pdf, April 2002

[76] *Leitner, C.:* Entwurf eines MPEG1-Layer3-Player mit FPGA-basiertem Controller. TU Berlin, Diplomarbeit am Institut für Technische Informatik, Dezember 1999

[77] *LEON2:* A synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 Architecture. J. Gaisler, http://www.gaisler.com/leon.html

[78] *LSP:* LYR Signal Processing. http://www.signal-lsp.com, 2001

[79] *Matlab:* Matlab, Getting Started with Matlab. The Mathworks, Natick, Massachusetts, 1999

[80] *MercuryPlus:* Emulation System. Quickturn, http://www.quickturn.com/products/mercuryplus.htm

[81] *Meyer-Baese, U.:* Digital Signal Processing with Field Programmable Gate Arrays. Springer-Verlag, 2001

[82] *Molson, P.:* Accelerating intellectual property design flow using Simulink for system on a programmable chip. Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers, 2001, Vol. 1, S. 454ff., November 2001

[83] *MPG123:* http://www.mpg123.de

[84] *MPG123lib 0.92:* http://www.rz.uni-frankfurt.de/~pesch/

[85] *MPEG, Webseite:* Moving Picture Experts Group. http://www.cselt.it/mpeg/

[86] *Model Technology Product Support:* ModelSim SE 5.6b Performance Guidelines. http://www.model.com/resources/pdf/optimizing_perf_56b.pdf, Juli 2002

[87] *Nichols, K.; Moussa, M.; Areibi, S.:* Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks CAINE. San Diego California, http://www.uoguelph.ca/~knichols/research/CAINE_02_paper.pdf, November 2002

[88] *Nios:* Embedded Processor Soft Core. Altera, http://www.altera.com/products/devices/nios/nio-index.html

[89] *NSIM:* Simulation Acceleration Plug-In. Mentor Graphics, http://www.mentor.com/simulation/

[90] *O'Brien, K.; Maginot, S.:* Non-Reversible VHDL Source-Source Encryption. IEEE Proc. EURO-DAC'94, S. 480ff., 1994

[91] *Opencores.org:* Opencores.org Projects. http://www.opencores.org/projects/

[92] *Oregano Systems:* Intel 8051 IP Core. Oregano Systems, http://oregano.at/ip/8

[93] *Palladium:* Simulation Accelerator/Emulator. Cadence, http://www.cadence.com/products/palladium_new.html

[94] *Parker, Kenneth P.:* The Boundary-Scan Handbook. Kluwer Academic Publishers, 1992

[95] *Pino, J. L.; Ha, S.; Lee, E. A.; Buck, J. T.:* Software Synthesis for DSP Using Ptolemy. Journal of VLSI Signal Processing, Nr. 9, 1995

[96] *Pohlmann, Ken C.:* The Compact Disc Handbook - The Computer music and digital audio series. 2nd Ed. Madison, WI: A-R Editions, Inc. 1992

[97] *Pottinger, D.H.J.; Williams, G.R.,III; Kelly, J.S.; Tamboli, S.:* Development of board level simulation models of complex standard components. Circuits and Systems, Vol. 1, 1996

[98] *Ptolemy:* The Ptolemy Project - Heterogeneous Modeling and Design. UC Berkeley, EECS, http://ptolemy.eecs.berkeley.edu/

[99] *Quicklogic:* Quicklogic Databook 2001. http://www.quicklogic.com/images/2001_databookbook.zip, 2001

[100] *Raines, P.:* Tcl/Tk kurz und gut. O'Reilleys Taschenbibliothek, September 1998

[101] *Rowson, J.A.:* Hardware/Software Co-Simulation. Proc 31th ACM/IEEE DAC, S. 439ff., 1994

[102] *SDF:* IEEE Std 1476-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process. IEEE, New York, USA, http://www.eda.org/sdf/, 2001

[103] *Seamless:* Seamless Hardware/Software-CoVerification. Mentor Graphics, http://www.mentor.com/seamless/, 2003

[104] *SIA:* Semiconductor Industry Association. http://www.semichips.org/home.cfm

[105] *Simulink:* Simulink, Dynamic System Simulation for Matlab, Using Simulink. The Mathworks, Natick, Massachusetts, 1999

[106] *SystemC:* VA Software Corporation and Open SystemC Initiative. http://www.systemc.org/

[107] *System Generator:* Design Flow using System Generator for DSP. XtremeDSP, http://www.xilinx.com/products/software/sysgen/design_flow.htm

[108] *Triscend:* http://www.triscend.com/products/index.htm

[109] *Usselmann, R.:* OpenCores SOC Bus Review. Rev. 1.0, http://www.opencores.org/wishbone/doc/soc_bus_comparison.pdf, Januar 2001

[110] *VariCore:* EPGA Technology. Actel, http://varicore.actel.com

[111] *VCC:* Cadence Virtual Component Co-design Environment. Cadence, http://www.cadence.com/articles/vcc.html

[112] *Verilog:* IEEE Std 1364-1995, IEEE standard hardware description language based on the Verilog(R) hardware description language. IEEE, New York, USA, 1996

[113] *Verilog:* IEEE Std 1364-2001, IEEE standard Verilog hardware description language. IEEE, New York, USA, 2001

[114] *VHDL-87:* IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual. IEEE, New York, USA, 1988

[115] *VHDL-93:* IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual. IEEE, New York, USA, 1994

[116] *VHDL:* IEEE Std 1076.6-1999, IEEE standard for VHDL Register Transfer Level (RTL) synthesis. IEEE, New York, USA, 2000

[117] *VHDL-AMS:* IEEE Std 1076.1-1999, IEEE standard VHDL analog and mixed-signals extensions. IEEE, New York, USA, 1999

[118] *VHDL PLI:* IEEE DASC VHDL PLI Task Force, http://www.eda.org/vhdlpli/

[119] *Virtex:* DS003 - Virtex 2.5V Field Programmable Gate Arrays. v2.5, 2001

[120] *Virtual-CPU:* Virtual-CPU Co-Verification Environment. Summit Design, http://www.summit-design.com/Virtual-CPU.htm

[121] *VIS:* A system for Verification and Synthesis. The VIS Group, In the Proceedings of the 8th International Conference on Computer Aided Verification, S. 428ff., Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996

[122] *VITAL:* IEEE Std 1076.4-2000, IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification. IEEE, New York, USA, http://www.vhdl.org/vital, 2000

[123] *VSI Alliance, Inc.:* VSI Alliance Model Taxonomy Version 2.1. System-Level Design Development Working Group, http://www.vsi.org/library/specs/sld220.pdf, 16. Juli 2001

[124] *VStation:* Emulation System. Mentor Graphics, http://www.mentor.com/emulation

[125] *W3C:* Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, 6. Oktober 2000

[126] *W3C:* Extensible Markup Language (XML) 1.0 (Zweite Auflage) - Deutsche Übersetzung. http://edition-w3c.de/TR/2000/REC-xml-20001006, 20. Januar 2002

[127] *Williamson, M. C.:* Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications. Dissertation, UC Berkeley, Electrical Engineering and Computer Sciences, 1998

[128] *Wishbone:* The Wishbone „System-on-Chip“ Architecture. Silicore Corp., http://www.silicore.net/wishbone.htm

[129] *XBlue:* IBM Blue Logic Xilinx FPGA Core License Agreement, Execute Briefing. http://www.xilinx.com/company/press/kits/xil_ibm/xblue_editor.pdf, Juni 2002

[130] *Xilinx:* DS026 - XC18V00 Series of In-System Programmable Configuration PROMs. v3.7, September 2002

[131] *Xilinx:* Data Source CD-ROM. 2002

[132] *Xilinx:* Partial Reconfigurability Frequently Asked Questions (FAQ). http://www.xilinx.com/ise/advanced/partial_reconf_faq.htm

[133] *Xilinx:* XAPP131 - 170 MHz FIFOs using the Virtex Block SelectRAM+ Feature. v1.7, März 2003

[134] *Xilinx:* XAPP138 - Virtex FPGA Series Configuration and Readback. http://www.xilinx.com/xapp/xapp138.pdf, v2.3, Oktober 2000

[135] *Xilinx:* XAPP151 - Virtex Series Configuration Architecture User Guide. http://www.xilinx.com/xapp/xapp151.pdf, v1.5, September 2000

[136] *Xilinx:* XAPP290 - An Implementation Flow for Active Partial Reconfiguration. http://www.xilinx.com/xapp/xapp290.pdf, Mai 2002

[137] *Xilinx:* XAPP634 - Analog Devices TigerSHARC Link. http://www.xilinx.com/xapp/xapp634.pdf, v1.0, Juni 2002

[138] *Xilinx:* Design Reuse Strategy for FPGAs. Xcell Journal 37, S. 40ff., Q3 2000

[139] *Xilinx:* The Rapidly Changing ASIC Conversion Market. Xcell Journal 30, S. 8f., Q4 1998

[140] *Xilinx:* Virtex Series FPGAs Product Matrix. http://www.xilinx.com/publications/matrix/virtex_color.pdf

[141] *XtremeDSP:* Xilinx XtremeDSP-Initiative. Xilinx, http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Xilinx+DSP