TECHNISCHE UNIVERSITÄT
ILMENAU

Modellierung und automatische Generierung von FPGA-basierten Testinstrumenten für den strukturellen Leiterplattentest

Dissertation

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

vorgelegt in der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Herrn Dipl.-Ing. Steffen Ostendorff
geboren am 13.06.1978 in Bremen, Deutschland

Datum der Einreichung:	29.06.2016
Datum der Verteidigung:	28.03.2017
Gutachter:	1. Prof. Dr.-Ing. habil. A. Mitschele-Thiel
	2. Prof. Dr. R. Ubar
	3. Prof. Dr. H. T. Vierhaus

Lilienthal, 27.03.2017

urn:nbn:de:gbv:ilm1-2017000103

Für meine Familie

DANKSAGUNG

Ich möchte mich bei Andreas Mitschele-Thiel und Heinz-Dietrich Wuttke herzlich bedanken, die mir die Möglichkeit gegeben haben am Fachgebiet Integrierte Kommunikationssysteme zu arbeiten, zu lehren und zu lernen. Weiterhin dafür, mir die Möglichkeit zu geben diese Dissertation zu verfassen und mir während dieser Zeit mit Rat und Tat zur Seite zu stehen.

Ich danke den Kollegen der Forschungsprojekte ERADOS und ROBSY Heinz-Dietrich Wuttke, Jörg Sachße und Jorge Hernán Meza Escobar für die gute Zusammenarbeit sowie für die interessanten und fruchtbaren Diskussionen. Besonders Jorge Hernán Meza Escobar danke ich für die Unterstützung bei der Generierung eingebetteter Testinstrumente. Ohne seinen Testprozessor wäre ein wesentliches Stück Flexibilität des Testkonzepts verloren gegangen. Weiterhin danke ich insbesondere folgenden Studenten für ihre Arbeit im Rahmen der Projekte: Christian Hergenröder, Sebastian Richter und Thomas Sasse. Ich danke Thomas Wenzel von GÖPEL electronic und Artur Jutman von Testonica Lab für ihre Diskussionen, Ideen und Fragen.

Ich danke dem gesamten Fachgebiet Integrierte Kommunikationssysteme für den netten und konstruktiven Umgang miteinander und die anregenden Diskussionen zum Thema meiner Dissertation. Explizit möchte ich Markus Brückner für die Bereitstellung seines Parsers danken, der als Grundlage für den im Rahmen dieser Arbeit entworfenen RTDL Compiler dient. Weiterhin für seine Geduld, mir die Programmiersprache Python näher zu bringen. Ein weiterer Dank geht an die Kollegen Karsten Henke und Tobias Vietzke, die mit interessanten Themen außerhalb meiner Dissertation für eine abwechslungsreiche Zeit am Fachgebiet gesorgt haben.

Ein ganz besonderer Dank jedoch geht an meine Familie, besonders an meine Frau Claudia, die immer für mich da war und mich auf diesem langen Weg begleitet hat. Vielen Dank!

Steffen Ostendorff

ABSTRAKT

Die stetige Miniaturisierung im Bereich der Schaltkreis- und Leiterplattenherstellung stellt Testsysteme aufgrund des fortwährend sinkenden Testzugriffs in zunehmendem Maße vor Probleme bei der Testbarkeit von Verbindungen zwischen Schaltkreisen auf bestückten Leiterplatten und verursacht eine reduzierte Testabdeckung und steigende Testkosten. Besonders für FPGAs, die auf der zu testenden Leiterplatte vorhanden sind, fehlen geeignete Methoden, bei denen sich das Testsystem automatisch den Gegebenheiten der zu testenden Leiterplatte anpasst und die es ermöglichen alle Verbindungen zwischen dem FPGA und anderen Schaltkreisen auf der Leiterplatte unter realen Einsatzbedingungen zu testen.

Diese Arbeit beschäftigt sich daher mit dem Problem des FPGA-basierten Testens. Dies setzt immer das Vorhandensein eines FPGAs auf dem Prüfling voraus. Das vorgestellte Konzept nutzt ausschließlich vorhandene Ressourcen des FPGAs, um Testalgorithmen statt auf externen Testgeräten in die Logik des FPGAs zu implementieren und erhöht somit die Herstellungskosten der Leiterplatte nicht. Die Ressourcen des FPGAs stehen während der Testphase exklusiv für das Testen zur Verfügung, werden jedoch nur temporär während dieser Zeit belegt, so dass keine Ressourcen für die eigentliche Anwendung nach Abschluss des Testens verloren gehen.

Ausgehend vom Stand der Technik im Bereich der nicht-invasiven elektrischen Testverfahren für Leiterplattentests mit FPGAs werden im Rahmen dieser Arbeit aktuelle Ansätze miteinander verglichen und analysiert. Aus den Stärken und Schwächen der betrachteten Methoden wird eine detaillierte Zielstellung dessen, was mit dieser Arbeit verbessert werden soll, erarbeitet. Es wird eine Methode zur Generierung von sogenannten Testinstrumenten für das FPGA-basierte Testen vorgestellt, die die Ausführung von Testalgorithmen in den auf der Leiterplatte vorhandenen FPGA verlagern und eine vergleichbare oder bessere Testabdeckung sowie Testgeschwindigkeit im Vergleich zu den etablierten Verfahren liefern ohne dafür auf manuelle Eingriffe bei der Generierung angewiesen zu sein.

Im Rahmen eines Lösungsansatzes wird neben der gewählten Testsystemarchitektur eine Modellierung für alle an den Verbindungstests beteiligten Schaltkreise vorgestellt. Hierbei ermöglicht die Testsystemarchitektur die Ausführung von Testalgorithmen im FPGA auf verschiedene Arten. Entweder in Software auf einem softcore-basierten Prozessor oder direkt in Hardware als diskrete Logik in einem sogenannten Co-Prozessor. Im Falle limitierter

Ressourcen im FPGA erlaubt es das Konzept auch, die Testausführung in Teilen auf einem Test-PC und somit außerhalb des FPGAs auszulagern.

Ziel der Modellierung ist es jeden Schaltkreis getrennt und unabhängig von der Art seiner späteren Implementierung zu modellieren und auch nicht an eine feste Geschwindigkeit, mit der der Test ausgeführt werden soll, gebunden zu sein.

Die unterschiedlichen Möglichkeiten die Testalgorithmen auszuführen erlauben es, ein Testinstrument auf verschieden Weise zu partitionieren und so den konkreten Gegebenheiten eines Prüflings, wie den vorhandenen Ressourcen im FPGA und den Anforderungen an die Testgeschwindigkeit anzupassen. Die Generierung aller nötigen Bestandteile in Software und Hardware, wie auch die Integration dieser Bestandteile zu einem Testinstrument erfolgen dabei vollständig automatisch.

Kern der Arbeit ist die Modellierung, Partitionierung und Generierung für eingebettete Testinstrumente, die auf der Testsystemarchitektur basieren. Der Fokus wird dabei auf die Modellierung und Generierung von zeitlich korrekten Ansteuerungen der an den Verbindungstests beteiligten Schaltkreise gelegt ohne jedoch eine konkrete Implementierung vorzugeben. Hierbei geht es insbesondere darum, alle Teile eines Testsystems modell-basiert und unabhängig voneinander, sowie ohne Wissen des genauen Einsatzumfeldes zu beschreiben, um daraus zu einem späteren Zeitpunkt automatisch eine Adaption an die verwendete Leiterplatte durchzuführen.

In Untersuchungen wird die Generierung von Testinstrumenten für unterschiedliche Schaltkreise betrachtet. Weiterhin wird ein Vergleich verschiedener Partitionierungen für ein ausgewähltes Testinstrument durchgeführt. Diese Untersuchungen belegen die Leistungsfähigkeit der vorgestellten Methode zur automatischen Generierung von FPGA-basierten Testinstrumenten bestehend aus Prozessor und Co-Prozessor und zeigen eine signifikante Beschleunigung des FPGA-basierten Verbindungstests für Leiterplatten, deren genaue Werte jedoch von vielen Faktoren abhängen. In den durchgeführten Untersuchungen ist eine um das 800-fache beschleunigte Testausführung im Vergleich zu Boundary-Scan erreicht worden.

ABSTRACT

The ongoing miniaturization of integrated circuits and printed circuit boards causes problems for test systems due to decreasing test access. This reduces the testability of connections between circuits on populated printed circuit boards and results in decreasing test coverage and increasing test costs. Especially for FPGAs on the printed circuit board, there are missing some appropriate methods that automatically adapt the test system to the conditions of the printed circuit board used and where the test system is also able to test all connections between the FPGA and other integrated circuits under real-life working conditions.

This thesis is about the problems of FPGA-based testing. This always requires the existence of such an FPGA on the printed circuit board. The presented concept solely uses available resources of the FPGA to transfer test algorithms from external test equipment into the programmable logic of the FPGA and therefore does not increase the production costs of the printed circuit board. The resources of the FPGA are exclusively used for testing during the test phase, but only programmed temporarily. Therefore no resources are taken from the target application of the printed circuit board.

Based on state-of-the-art non-invasive electrical test methods for printed circuit boards with FPGAs current approaches are compared and analyzed. From the strengths and weaknesses of the considered methods a detailed description of the goals that should be achieved with this thesis are discussed. A method for the generation of so called test instruments for FPGA-based testing is presented. This method transfers the execution of test algorithms into the FPGA available on the printed circuit board and has a similar or better test coverage as well as test speed compared to the well-established techniques without the need for any manually actions when generating such systems.

Besides the chosen test system architecture the modelling of integrated circuits that are part of the connection test is presented. The test system architecture allows the execution of test algorithms in different ways. They can either be executed in software on a soft-core processor or directly in dedicated logic in so called co-processors. In case of limited resources in the FPGA the concept also allows to outsource parts of the algorithms to a test PC and therefore execute them outside the FPGA.

Goal of the modelling is to model each integrated circuit independent of each other and also independent of the type of implementation (software or hardware) and a fixed execution speed.

The different types of implementation in software and hardware allow a flexible test instrument partitioning. This makes an adaption of the test instruments to the constraints of a specific test object possible, like the available resources or required test speed. The generation of all software and hardware parts of the test system, as well as the integration of all parts into a single test instrument is done fully automatically.

Central element of this thesis is the modelling, partitioning and generation of embedded test instruments, based on the presented test system architecture. The focus is put on the modelling and generation of timing-correct control routines of the integrated circuits that are part of the connection test, without specifying one solution in particular. All parts of the test system should be modeled independent of each other and without knowledge about the use case.

In experiments the generation of test instruments for different integrated circuits is carried out. There will also be a comparison of different partitions for one selected test instrument. These experiments prove the performance of the proposed methods for the automatic generation of FPGA-based test instruments consisting of a processor and co-processor. They show a significant speed-up for FPGA-based tests of printed circuit boards, which varies depending on many different properties of the test system. In the conducted experiments the time needed for testing was reduced to less than $1/800^{th}$ compared to Boundary-Scan.

INHALTSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

ALAP ..... As Late As Possible

ALU ..... Arithmetic Logical Unit

ANSI-C ..... American National Standards Institute Programming Language C

AOI..... Automated Optical Inspection

ASAP ..... As Soon As Possible

ASIC..... Application Specific Integrated Circuit

ATE..... Automated Test Equipment

AXI ..... Automated X-Ray Inspection

BBATG ..... Behavior-based Automatic Test Generation

BGA ..... Ball Grid Array

BIAT ..... Built-In-Assisted-Test

BIST ..... Built-In-Self-Test

BScan ..... Boundary-Scan

BSDL ..... Boundary-Scan Description Language

CASCON ..... Computer Aided Scan-based Observation and Node-control

CASLAN..... CASCON Language

CPLD ..... Complex Programmable Logic Device

CTL ..... Core Test Language

DDR ..... Double Data Rate

DRAM..... Dynamic Random Access Memory

DTD ..... Document Type Definition

DUT ..... Device under Test

DUT-M..... Device under Test Model

EBNF ..... Erweiterte Backus-Naur-Form

EDA ..... Electronic Design Automation

EEPROM ..... Electrically Erasable Programmable Read-Only Memory

ELESIs ..... European Library-based flow of Embedded Silicon test Instruments

ERADOS..... Experimental Research for Adaptive failure Diagnostics based On Structural multi-core emulation test.

FAT ..... FPGA-assisted Test

FBT ..... FPGA-based Test

FCT ..... FPGA-controlled Test

FPGA ..... Field Programmable Gate Array

FPT..... Flying Probe Test

FSM..... Finite State Machine

GDDR ..... Graphics Double Data Rate

HW ..... Hardware

I/O ..... Input / Output

I^2C ..... Inter-Integrated Circuit

IC..... Integrated Circuit

ICL ..... Instrument Connectivity Language

ICT ..... In-Circuit Test

ID ..... Identifier

IEEE..... Institute of Electrical and Electronics Engineering

IJTAG..... Internal JTAG

IP ..... Intellectual Property

ITU-T ..... International Telecommunication Union – Telecommunication Standardization Sector

JTAG ..... Joint Test Action Group

OI ..... Optical Inspection

PaR ..... Place and Route

PBIAT ..... Programmable Built-In Assisted Test

PBIST ..... Programmable Built-In Self Test

PCB ..... Printed Circuit Board

PDL ..... Procedure Description Language

PLD ..... Programmable Logic Device

PLL ..... Phase Locked Loop

PSL ..... Property Specification Language

RISC ..... Reduced Instruction Set Computer

ROBSY ..... Rekonfigurierbares On-Board selbst-test-System.

RTDL ..... ROBSY Test Description Language

RTL ..... Register Transfer Level

SBST ..... Software-basierter-selbst-Test

SDL-RT ..... Specification and Description Language – Real Time

SDRAM ..... Synchronous Dynamic Random Access Memory

SerDes ..... Serialisierer/Deserialisierer

SMD ..... Surface Mounted Device

SoC ..... System-on-Chip

SPI ..... Serial Peripheral Interface

SRAM ..... Static Random Access Memory

STAPL ..... Standard Test and Programming Language

STIL ..... Standard Test Interface Language

SUT ..... System-Under-Test

SVF ..... Serial Vector Format

SW..... Software

TAP ..... Test Access Port

TCK..... Test Clock

TDI ..... Test Data In

TDO ..... Test Data Out

THT ..... Through Hole Technology

TMS ..... Test Mode Select

TRST..... Test Reset

TSC ..... Timed Statecharts

UML..... Unified Modeling Language

USB..... Universal Serial Bus

VHDL..... VHSIC Hardware Description Language

VHSIC..... Very High Speed Integrated Circuit

VLSI..... Very Large Scale Integration

VLSR ..... Variable Length Shift Register

XML..... Extensible Markup Language

BEGRIFFSVERZEICHNIS / GLOSSAR

At-speed	Einen Ablauf so zu steuern, dass dieser in einer Geschwindigkeit erfolgt, die dem späteren Einsatzfall entspricht.
Board-Level Test	Struktureller Test von Verbindungen auf Leiterplatten.
Boundary-Scan-Kette	Eine Reihe von Registerzellen, die in Form einer Kette angeordnet sind und sich innerhalb eines Schaltkreises befinden und mit dessen Pins verbunden sind. Werden mehrere solcher Schaltkreise in Reihe geschaltet, spricht man ebenfalls von Boundary-Scan-Kette.
CASCON	Computer Aided Scan-based Observation and Node-control ist die Bezeichnung einer Software der Firma GÖPEL electronic, die das strukturelle Testen von Leiterplatten ermöglicht.
Co-Prozessor	Beschreibt Hardwareblöcke, die Testfunktionen innerhalb eines Testinstruments ausführen.
Device under test (DUT)	Der Schaltkreis dessen Verbindungen zum FPGA getestet werden sollen.
ERADOS	Das Projekt beschäftigte sich von März 2009 bis März 2011 mit dem Ziel neue Verfahren für das Testen komplexer Systeme zu erforschen. Es sollten dabei klassische Ansätze des Testens unter neuen Aspekten kombiniert werden, um damit den Anforderungen an künftige Diagnose- und Testverfahren Rechnung zu tragen. Das vom Freistaat Thüringen unterstützte Projekt wurde durch Mittel der Europäischen Union im Rahmen des Europäischen Fonds für regionale Entwicklung (EFRE) kofinanziert.

Externes Testsystem ..... Beschreibt den Teil des Testsystems, der nicht auf der zu testenden Leiterplatte vorhanden ist, sich also in Relation dazu extern befindet. Üblicherweise wird dieser Teil durch einen PC realisiert.

Funktionstest ..... Hierbei wird der durchgeführte Test genutzt, um Aussagen über die Funktion des Testobjekts zu erhalten.

Hardwaretest ..... Ein Test, bei dem es das Ziel ist eine gewisse Aussage über die Funktion der Hardware zu erhalten.

Negativtest ..... Ein Test, der das Verhalten der Funktion oder Struktur auf eine falsche (negative) Eingabe prüft.

Positivtest ..... Ein Test, der das Verhalten der Funktion oder Struktur auf eine richtige (positive) Eingabe prüft.

ROBSY ..... Das Projekt beschäftigte sich von April 2011 bis März 2013 mit dem Ziel eines kompletten flexiblen Testsystems für Board-Level-Tests auf FPGA (Field Programmable Gate Array). Gegenstand der Forschung waren Untersuchungen zu re-konfigurierbaren elektronischen Testsystemen mit FPGAs sowie deren automatische Generierung. Das vom Freistaat Thüringen unterstützte Projekt wurde durch Mittel der Europäischen Union im Rahmen des Europäischen Fonds für regionale Entwicklung (EFRE) kofinanziert.

SerDes ..... Serializer-Deserializer Einheit, die das serielle Übertragen (Senden und Empfangen) mit sehr hohen Datenraten unterstützt.

Softwaretest ..... Ein Test, bei dem es das Ziel ist eine gewisse Aussage über die Funktion der Software zu erhalten.

Strukturtest ..... Hierbei wird der durchgeführte Test genutzt, um Aussagen über die Struktur des Testobjekts zu erhalten. Im Rahmen dieser Arbeit wird der Begriff immer in Zusammenhang mit dem Testen von Verbindungen zwischen Schaltkreisen auf Leiterplatten genutzt.

Testalgorithmus ..... Eine eindeutige Handlungsvorschrift für die Ausführung einer bestimmten Testaufgabe, z.B. um zu prüfen, ob die Verbindung zwischen Punkt A und Punkt B fehlerfrei ist.

Testfunktion ..... Teilaufgabe eines Testalgorithmus, z.B. das Anlegen eines bestimmten Testvektors.

Testingenieur	Die Person, die einen Test entwickelt bzw. durchführt.
Testkosten	Die gesamten Kosten für die Durchführung bestimmter Tests bzw. das Erreichen einer bestimmten Testabdeckung.
Testobjekt	Der zu testende Teil eines Prüflings. Dies können je nach Anwendungsfall die Verbindungen zwischen Schaltkreisen auf einem Stück Leiterplatte sein oder aber auch bestimmte Funktionen eines Schaltkreises.
Testprozessor	Beschreibt einen Prozessor, der für die Ausführung von Testalgorithmen verwendet wird.
Testpunkte	Sind kleine Kontaktflächen, die auf einer Leiterplatte platziert werden und mit der Schaltung verbunden sind, um darüber beim Testen eine physikalische Verbindung zwischen dem Testsystem und dem Prüfling herzustellen.
Testsystem	Das System, das einen Test ausführt. Dies kann sich auf der zu testenden Leiterplatte befinden oder extern hierzu angeordnet sein. Auch eine verteilte Anordnung auf mehrere Leiterplatten oder externe Komponenten ist möglich.
Waveformdiagramm	Darstellung von zeitlichen Signalverläufen.
Wrapper	Eine Modul, das mehrere andere Module umschließt und nach außen hin kapselt.
Zeitliche Parameter	Die Randbedingungen, die eine Zugriffsfunktion eines Schaltkreises in Bezug auf die zeitliche Abfolge von Signalen definiert.

1. EINLEITUNG

Das folgende Kapitel liefert eine Hinführung zum Thema der vorliegenden Dissertation. Hierbei wird, ausgehend vom Herstellungsprozess von Leiterplatten, auf mögliche Herstellungsfehler hingewiesen und es werden Möglichkeiten und Methoden zu deren Erkennung dargestellt. Die vorliegende Arbeit konzentriert sich dabei auf Leiterplatten, auf denen sich programmierbare Logik in Form wenigstens eines FPGAs (Field Programmable Gate Arrays) befindet. Aufbauend auf aktuellen Herausforderungen und ungelösten Problemen bei der Verwendung dieser Schaltkreise beim strukturellen Leiterplattentest, werden die allgemeinen Ziele dieser Arbeit definiert.

1.1. MOTIVATION

Im Bereich der programmierbaren Logik, insbesondere der FPGAs, sind in den letzten Jahren zwei deutliche Entwicklungstrends festzustellen. Dies betrifft sowohl die gestiegene Leistungsfähigkeit der FPGAs als auch die ständig steigende Komplexität der Systeme, in denen FPGAs eingesetzt werden. Diese Faktoren haben zu einem deutlichen Anstieg der Möglichkeiten geführt, die sich mit FPGAs ergeben und erheblich zu deren Verbreitung beigetragen.

Laut eines *Investor Factsheet* des mit ca. 50% Marktanteil größten FPGA-Herstellers Xilinx für das 2. Quartal 2016 belaufen sich die Nettoeinnahmen des Herstellers auf $2,38 Milliarden für das Jahr 2014/2015[1]. Dies entspricht einer Steigerung von ca. 50% in den letzten 10 Jahren bzw. 133% in den letzten 15 Jahren.

Eine 2015 durchgeführte Studie [3], gibt für das Jahr 2014 einen Gesamtumsatz des FPGA Markts von $3,92 Milliarden an und schätzt, dass sich der Markt bis zum Jahre 2022 auf $7,23 Milliarden vergrößern wird. Dies entspricht einer Umsatzsteigerung von ca. 84% in den nächsten 8 Jahren.

[1] Stichtag ist der 31.3.2015, Financial Statements [1], [2]

Diese Zahlen lassen auch zukünftig einen anhaltenden positiven Trend und verbreiteten Einsatzes von FPGAs in den unterschiedlichsten Einsatzgebieten erwarten.

Durch die steigende Verbreitung von FPGAs finden diese mehr und mehr Einzug in das tägliche Leben und sind nicht mehr nur in Entwicklungsprototypen oder Nischenprodukten zu finden. Dadurch wird es zwingend notwendig solche Systeme ausgiebig zu testen, um einen hohen Qualitätsstandard zu gewährleisten.

Der Herstellungsprozess von elektronischen Produkten (siehe Abbildung 1) hat dabei einen entscheidenden Einfluss auf deren Qualität und somit Ausfallwahrscheinlichkeit. Jeder Schritt im Herstellungsprozess einer bestückten Leiterplatte unterscheidet sich in der Art der möglichen Fehler, die auftreten können und somit auch in der Art der geeigneten Testverfahren, die sich an jeden Schritt anschließen sollten. Diese Dissertation konzentriert sich auf den letzten Schritt der *Integration auf der Leiterplatte* und auf mögliche Fehler die nach dem Löten entstehen können.

Ziel ist es, insbesondere das Testen von Fehlern auf FPGA-basierten Systemen zu verbessern und damit dem enormen Potential, das der FPGA-Markt bietet, und der aktuell bereits stark zunehmende Verbreitung dieser Technologie Rechnung zu tragen. Diese Arbeit beschäftigt sich daher mit den Möglichkeiten des strukturellen Testens von Verbindungen auf Leiterplatten FPGA-basierter Systeme. Hierbei wird davon ausgegangen, dass ein solcher FPGA bereits auf der zu testenden Leiterplatte vorhanden ist und lediglich für die Dauer des Testens umkonfiguriert wird.

1.2. PRODUKTIONSPROZESS VON LEITERPLATTEN

Der Herstellungsprozess von elektronischen Produkten kann in unterschiedliche Phasen unterteilt werden.

Diese Phasen sind[2]:

- **Konzept**
  Es werden die Funktionen und Anforderungen an das spätere Gesamtsystem sowie notwendige Randbedingungen und Vorgaben definiert.
- **Design**
  Auf Basis des Konzepts wird eine technische Realisierung erarbeitet.
- **Fertigung der Komponenten**
  Die notwendigen Komponenten, wie Leiterplatte, Schaltkreise, Stecker, Eingabe- bzw. Ausgabegeräte und andere elektrische Komponenten werden hergestellt.
- **Integration auf der Leiterplatte**
  Alle Komponenten werden auf einer Leiterplatte miteinander verbunden.

[2] Zur besseren Übersichtlichkeit wird nur die technische Sicht der Elektronikfertigung betrachtet, nicht jedoch andere Aspekte, wie die Sicht auf die Softwareentwicklung oder die Wirtschaftlichkeit.

- **Zusammenbau des Gesamtsystems**
  Alle Leiterplatten werden zusammengebaut und es erfolgt ggf. die Integration in ein Gehäuse.

Hierbei kann die *Integration auf der Leiterplatte*, also die Bestückung der Leiterplatte mit Bauelementen, weiter in folgende Teilschritte untergliedert werden:

- **Bepastung**
  Hierbei werden die metallischen Kontaktflächen der Leiterplatten, auf die später die Bauelemente platziert werden, mit einer Lötpaste bedruckt. Diese Paste verflüssigt sich beim späteren Lötvorgang und bildet dann die elektrische Verbindung zwischen Bauelement und Leiterplatte.
- **Bestückung**
  Die Bauelemente werden auf die bepastete Leiterplatte gesetzt. Hierbei ist zu beachten, dass in diesem Prozessschritt nur SMD-Bauelemente (Surface-mounted Devices) bestückt werden. THT-Bauelemente (Through Hole Technology), werden in einem späteren Arbeitsschritt bestückt und gelötet. Für die späteren Betrachtungen im Rahmen der Arbeit ist es jedoch irrelevant, ob es sich um SMD- oder THT-Bestückung handelt.
- **Löten**
  Im letzten Schritt wird die Leiterplatte mit der Lötpaste und den Bauelementen in einem Ofen gelötet.

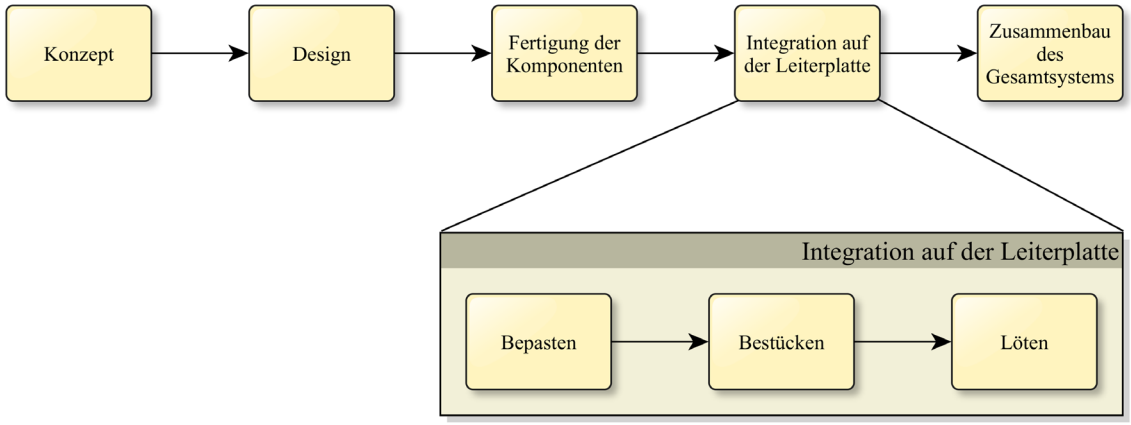


Abbildung 1: Herstellungsprozess von elektronischen Produkten

1.3. WAS IST TESTEN?

An dieser Stelle soll das sehr umfangreiche Thema *Test* noch einmal genauer beschrieben und eingegrenzt werden. Es soll insbesondere geklärt werden, was genau unter Testen zu verstehen ist und welche Bereiche für diese Arbeit relevant sind. Eine kurze Erläuterung zu den einzelnen verwendeten Begriffen ist auch im Begriffsverzeichnis / Glossary zu finden.

Die Fertigung von Leiterplatten unterliegt prozessbedingten Schwankungen, die sich auf die Ausbeute (engl.: yield). auswirken, d.h. nicht alle gefertigten Leiterplatten sind einsetzbar. Nach der Produktion muss somit festgestellt werden, ob eine Leiterplatte den vorher definierten Anforderungen entspricht oder ob sie fehlerbehaftet ist und ausgetauscht werden muss[3]. Diesen Vorgang bezeichnet man als Test [4]. Der Hersteller hat jedoch ein Interesse daran, einen Fehler nicht nur zu erkennen, sondern wenn möglich auch zu diagnostizieren. Dies erlaubt entweder eine gezielte Reparatur der Leiterplatte und/oder es ermöglicht eine Optimierung des Herstellungsprozesses, um in Zukunft die Anzahl der defekten Leiterplatten zu reduzieren und so die Ausbeute zu erhöhen. Der prinzipielle Ablauf ist sowohl bei der Schaltkreisherstellung [4] als auch der Leiterplattenherstellung [5] gleich. Die Verschaltung von Schaltkreisen auf einer Leiterplatte ist also vergleichbar mit der Verschaltung einzelner Gatter innerhalb eines Schaltkreises. Zwar sind die Fehlerursachen andere (u.a. Löt- oder Bestückungsfehler vs. Unreinheiten im Silizium bei der Schaltkreisherstellung), jedoch sind die Fehlermodelle vergleichbar. Laut [6] sieht man bereits, dass die Probleme der Chip- und Boardtests anfangen zu korrelieren. Somit können Erkenntnisse aus dem Bereich des Schaltkreistests teilweise auch auf den Test von Leiterplatten übertragen werden. Entsprechend werden beide Themenbereiche im Laufe der Dissertation an den passenden Stellen berücksichtigt.

Diese Arbeit beschäftigt sich mit strukturellen Leiterplattentests und gehört somit wie auch Coretests und Schaltkreistests zur Kategorie der *Hardwaretests*. D.h. es soll die Korrektheit der getesteten Hardware nachgewiesen werden. Im Gegensatz hierzu soll bei *Softwaretests* eine Aussage über die Funktion einer Software getroffen werden. Beide Verfahren geben anhand ihres Namens an *was* getestet wird, nicht *wie* dies geschieht.

Während ein Coretest nur einzelne Teile eines Schaltkreises testet, wird bei einem Schaltkreistest dieser als Ganzes getestet bzw. beim Leiterplattentest die Verbindung zwischen Schaltkreisen. Softwaretests spielen in Rahmen dieser Arbeit keine Rolle.

Der Leiterplattentest testet ausschließlich die Leiterplatte und geht davon aus, dass die verwendeten Schaltkreise fehlerfrei sind, da diese im Vorfeld getestet worden sind [5]. Das Testen ist mit erheblichem Aufwand und Kosten verbunden. Die Kosten steigen gravierend an, je später ein Fehler im Herstellungsprozess detektiert wird.[4]

[3] Es kann zwar das Vorhandensein von bestimmten Fehlern erkannt werden, genauso wie das Nicht-Vorhandensein, jedoch nicht die Fehlerfreiheit allgemein festgestellt werden. Eine Aussage zur Fehlerfreiheit ist immer nur in Bezug auf die durchgeführten Tests möglich.

[4] Generell rechnet man mit einer Verzehnfachung der entstehenden Kosten je Prozessschritt.

Laut [6] ist das Problem des Testens vergleichbar mit der Suche nach dem Heiligen Gral. Testen darf nach der Aussage von Wenzel und Ehrenberg [6] „*nichts kosten, weder Zeit und schon gar kein Geld, nicht beim Design und insbesondere nicht in der Produktion. Und wenn sich das Übel dann doch nicht vermeiden lässt, sind die Forderungen klar – vollautomatisierte Testentwicklung und -ausführung im SubSekundenbereich mit Equipment zum Nulltarif und Fehlerabdeckung von […] 100%*“.

Als testbar gilt ein Schaltkreis bzw. eine Leiterplatte dann, wenn hierfür Testmuster generiert, angewendet und ausgewertet werden können, und zwar so, dass vorher definierte Leistungsmerkmale bezüglich Fehlererkennung und –lokalisation innerhalb eines vorgegebenen Kosten- und Zeitrahmens zufriedenstellend durchgeführt werden können [7]. Die Realität sieht jedoch oft anders aus. Schuld sind hier vor allem die Komplexität der zu testenden Systeme, die Testzugriffsmöglichkeiten sowie die systemabhängige Testgeschwindigkeit.

1.3.1. LEITERPLATTENTEST

Die vorliegende Arbeit beschäftigt sich mit dem Thema Leiterplattentest. Eine bestückte Leiterplatte besteht dabei, wie in Abbildung 2 dargestellt, aus einem Träger, der eigentlichen Leiterplatte, auch PCB[5] genannt, und Schaltkreisen (IC[6]) sowie Eingabe-/Ausgabeschnittstellen (I/O[7]). Im weiteren Verlauf der Arbeit ist mit dem Begriff Leiterplatte immer die bestückte Leiterplatte gemeint.

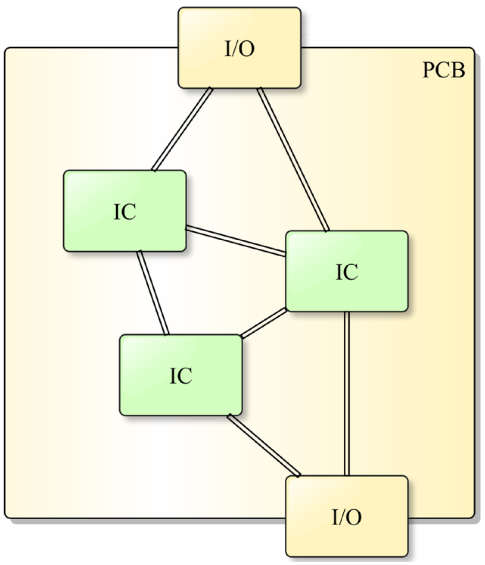


Abbildung 2: Schematischer Aufbau einer Leiterplatte

[5] PCB (engl.: Printed Circuit Board) – Leiterplatte

[6] IC (engl.: Integrated Circuit) – Integrierter Schaltkreis

[7] I/O (engl.: Input / Output) – Eingabe-- / Ausgabeblock

Ziel des Leiterplattentests ist es dabei, die Korrektheit bestimmter Eigenschaften in Zusammenhang mit der Leiterplatte nachzuweisen. Hierbei wird generell zwischen strukturellem und funktionalem Test unterschieden. Dies wird in Kapitel 2.1 „Arten des Testens“ näher betrachtet.

1.3.2. TESTABDECKUNG

Das Ziel eines Tests ist es, eine gewünschte Struktur oder Funktion nach bestimmten Vorgaben zu testen und ihre Korrektheit nachzuweisen. Hierbei wird eine Menge an möglichen Fehlern definiert, die für die vorliegende Struktur bzw. Funktion bekannt sind. Die Testabdeckung beschreibt wie viele dieser Fehler mit den gewählten Tests geprüft werden können. Grundsätzlich ist es das Ziel, eine Testabdeckung von 100% zu erreichen und somit alle bekannten Testfälle auch zu testen. Dies bedeutet nicht, dass eine Struktur oder Funktion nicht doch bestimmte Fehler aufweisen kann.

Auf die erreichbare Testabdeckung hat insbesondere die Möglichkeit des Zugriffs einen entscheidenden Einfluss. Hierbei ist nicht nur die prinzipielle Zugriffsmöglichkeit auf einzelne Strukturen oder Funktionen wichtig, sondern auch ob mit der ‚richtigen‘ Geschwindigkeit zugegriffen werden kann. Diese Eigenschaft wird als at-speed bezeichnet und bedeutet einen Zugriff mit der Geschwindigkeit durchzuführen, die dem späteren Einsatzfall entspricht.

1.3.3. TESTSYSTEM

Ein Testsystem für das strukturelle Testen von Leiterplatten ist in Abbildung 3 schematisch dargestellt. Hierbei gehört zu einem vollständigen Testsystem:

- ein Testingenieur, der den Test entwirft bzw. den Test-PC steuert
- ein Test-PC für die Ausführung der Testalgorithmen
- ein Testbed, dass die Verbindung zwischen Test-PC und Prüfling(en) herstellt[8]
- der zu testende Prüfling (oder mehrere)

[8] Test-PC und Testbed können zusammen auch als „Automated Test Equipment (ATE)“ bezeichnet werden. Diese Begriffe sind als gleichwertig anzusehen.

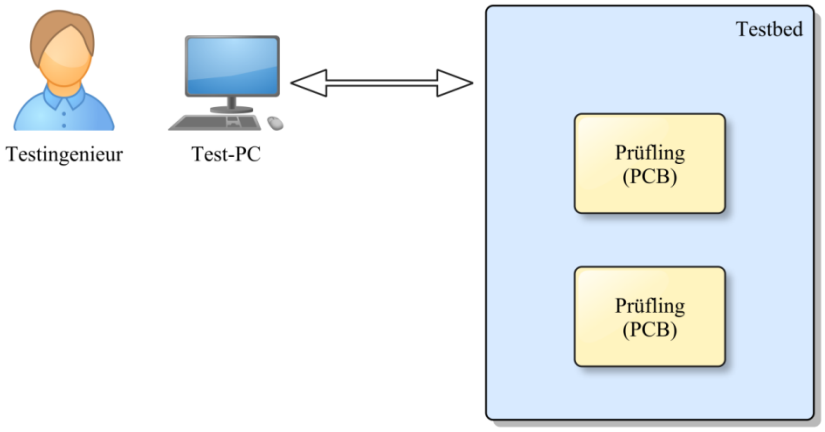


Abbildung 3: Allgemeiner Aufbau eines Testsystems

1.4. PROBLEME DES TESTENS

Die fortschreitende Entwicklung elektrischer Schaltkreise ermöglicht unter anderem immer kleinere Gehäuseformen, einen steigenden Funktionsumfang auf gleicher Chipfläche sowie eine zunehmende Verarbeitungsgeschwindigkeit. Dies erlaubt auf gleicher Leiterplattenfläche den Aufbau immer komplexerer und schnellerer Systeme bzw. führt bei gleichem Funktionsumfang zu deutlich kleineren Leiterplatten. Beides führt zu einer Reihe von Problemen beim strukturellen Testen dieser Systeme.

Diese Probleme sind:

- ein begrenzter Testzugriff auf die physikalischen Schnittstellen der Schaltkreise aufgrund der kleineren Strukturen der Leiterplatte und Schaltkreisgehäuse
- ein steigender Funktion sumfang bei gleicher Leiterplattengröße führt zu umfangreicheren Tests und erfordert eine aufwendige Testgenerierung
- eine unzureichende Testgeschwindigkeit durch zu langsames Abarbeiten der Testalgorithmen im Vergleich zur Geschwindigkeit mit der die Schaltkreise im realen Einsatz arbeiten

Hinzu kommen weitere Probleme, die sich bei aktuellen Leiterplatten ergeben. Hierzu gehören:

- kurze Design-Zyklen erfordern ständiges Adaptieren der Tests
- ständiger Kostendruck erfordert einfaches und schnelles Testen

Diese Probleme führen dazu, dass im Rahmen einer vorgegebenen Zeit bzw. eines definierten Kostenlimits immer weniger Funktionen getestet werden können. Diese Lücke zwischen vorhandenen und getesteten Funktionen bezeichnet man als sogenannte Testlücke (engl.: Testgap), die sich ständig weiter vergrößert, da die Entwicklung von Schaltkreisen seit Jahren schneller verläuft als die Entwicklung passender Test-methoden.

Um dem zunehmend größer werdenden Abstand zwischen verfügbarer und testbarer Funktionalität entgegenzuwirken wurden erste Strategien entwickelt. So findet aktuell ein Paradigmenwechsel statt und es beginnt das "*Zeitalter des Embedded System Access*" [8], bei dem Testfunktionen innerhalb des Prüflings ausgeführt werden. Zu dieser Klasse gehören unter anderem FPGA-basierte Tests.

Für den Begriff des FPGA-basierten Testens werden in der Literatur unterschiedliche Abkürzungen und Synonyme verwendet. Hierzu gehört sowohl der Begriff FBT (FPGA based test) als auch FAT (FPGA assisted test) oder FCT (FPGA controlled test), die jeweils von unterschiedlichen Herstellern der Testtools benutzt werden. Alle Begriffe beschreiben prinzipiell jedoch die Testunterstützung durch einen FPGA und werden im Rahmen dieser Arbeit als gleichwertig betrachtet, auch wenn es in der detaillierten Ausführung kleinere Unterschiede gibt, die für diese Arbeit jedoch nicht relevant sind.

1.5. ALLGEMEINE ZIELE UND AUFBAU DER DISSERTATION

Aus den in der Einleitung definierten Problemen beim strukturellen Testen von Leiterplatten lassen sich sechs allgemeine Ziele ableiten, die zu einer Verbesserung des Testens von FPGA-basierten Systemen beitragen können.

Diese Ziele sind:

- eine Beschleunigung von strukturellen Tests beim Einsatz von FPGAs direkt auf dem Prüfling, zur Realisierung von **at-speed Tests**, so dass diese unter gleichen Bedingungen wie im späteren Einsatz erfolgen
- eine **flexible Testgenerierung**, so dass die Verwendung von beliebigen Testvektoren möglich ist
- eine Anwendbarkeit in einem großen Bereich, um dadurch ein möglichst **großes Einsatzgebiet** und somit die Verwendbarkeit auf möglichst vielen FPGA-basierten Systemen abzudecken, ohne dafür auf spezielle Änderungen am FPGA angewiesen zu sein oder sich auf einen bestimmten Hersteller beschränken zu müssen
- eine **automatische Adaption** des Testsystems an die physikalischen Gegebenheiten des jeweiligen Prüfling, wie z.B. verwendeter FPGA, Verschaltung auf der Leiterplatte oder verwendete Takte, um auch bei kurzen Design-Zyklen ein optimales Testsystem zu realisieren; es ist dabei wichtig, ein Testsystem möglichst ohne manuellen Aufwand an einen beliebigen Prüfling anpassen zu können
- ein **umfangreicher Testzugriff**, um möglichst auf alle Verbindungen, die zwischen einem FPGA und anderen Schaltkreisen bestehen, zugreifen zu können

- eine **schnelle Testausführung** zur Reduzierung der benötigten Testzeiten und somit auch der Testkosten[9]

In Kapitel 2 werden die Grundlagen und in Kapitel 3 der aktuelle Stand der Technik in Bezug auf diese Ziele erläutert. Hieraus wird in Kapitel 3.5 eine detaillierte Zielstellung für diese Arbeit abgeleitet und es werden die zu lösenden Aufgaben konkretisiert. In Kapitel 4 wird der gewählte Lösungsansatz im Detail vorgestellt, gefolgt von den durchgeführten Untersuchungen in Kapitel 5. Den Abschluss der Arbeit bilden Zusammenfassung, Fazit und Ausblick in Kapitel 6.

1.5.1. UMFELD UND HINTERGRÜNDE DER DISSERTATION

Diese Dissertation wurde ihm Rahmen mehrerer Forschungsprojekte am Fachgebiet Integrierte Kommunikationssysteme der Technischen Universität Ilmenau durchgeführt. Im Folgenden wird das Umfeld der Forschungsprojekte, die Aufgabenverteilung sowie Abgrenzung innerhalb der Forschungsgruppe sowie der eigene Anteil näher erläutert.

Umfeld

Diese Dissertation ist im Rahmen der Forschungsprojekte ERADOS und ROBSY entstanden und wurde in Zusammenarbeit mit der Firma GÖPEL electronic GmbH in Jena durchgeführt. Die Firma GÖPEL electronic lieferte dabei Informationen zu den Problemen der Praxis, die sich letztendlich in der Zielstellung dieser Arbeit widerspiegeln. Es wurden durch die Firma GÖPEL electronic jedoch keine Vorgaben zur Art der Lösung gemacht, die in dieser Arbeit beschrieben ist.

Beide Projekte wurden vom Freistaat Thüringen unterstützt und durch Mittel der Europäischen Union im Rahmen des Europäischen Fonds für regionale Entwicklung (EFRE) kofinanziert.

Abgrenzung im Team

Die Arbeiten in den Forschungsprojekten ERADOS und ROBSY wurden im Team durch folgende Personen mit den genannten Aufgabenschwerpunkten realisiert:

- Heinz-Dietrich Wuttke: Projektleitung
- Jörg Sachße: Realisierung des Embedded Software Compilers zur Ausführung von Testfunktionen auf dem eingebetteten Testprozessor
- Jorge Hernán Meza Escobar: Realisierung des eingebetteten Testprozessors für das Ausführen von Testfunktionen in Embedded Software

[9] Diese Forderung ist nicht zwangsläufig durch die Forderung nach einem at-speed Test abgedeckt, da es beim at-speed Test nur darum geht, bestimmte Zeiten einzuhalten und nicht unbedingt darum, den gesamten Test so schnell wie möglich durchzuführen.

- Steffen Ostendorff: Realisierung der Co-Prozessoren für das Ausführen von Testfunktionen in Hardware

Folgende Punkte, die im weiteren Verlauf der Arbeit detailliert behandelt werden, sind als **Eigenanteil** dieser Dissertation anzusehen:

- Idee einer Testsystemarchitektur, die es erlaubt Funktionalität wahlweise in Hardware oder Software zu implementieren und bezogen auf die Leiterplattenherstellung kostenneutral ist und nur auf vorhandene Ressourcen zurückgreift.
- Die grundlegende Idee beim FPGA-basierten strukturellen Leiterplattentest die Funktionalität der Schaltkreise (DUTs), zu denen die Verbindungen getestet werden soll, übersichtlich in einem Ebenenmodell je DUT strukturiert abzubilden. Dies ermöglicht es jedes DUT getrennt zu modellieren und zwar so, dass die Ausführung der Testalgorithmen in Hardware oder Software später festgelegt werden kann. Das Ebenenmodell und die getrennte Modellierung für jedes DUT ermöglicht eine übersichtliche Strukturierung der Testfunktionen auch bei großen Leiterplatten mit vielen DUTs.
- Entwicklung eines ersten Ebenenmodells mit 7 Schichten [9]. Die Weiterentwicklung zu dem letztendlich im Rahmen dieser Arbeit dargelegten Modell mit 5 Schichten und 3 Interfaces ist gemeinsam im Projekt ERADOS erfolgt.
- Auswahl einer geeigneten Methode zur textuellen Beschreibung zeitlicher Abhängigkeiten und deren zugehöriger Aktionen für die Generierung von Zugriffsroutinen auf DUTs, so dass bei einer Realisierung in Hardware automatisch eine optimierte Lösung für eine parallele Ausführung gefunden werden kann, während eine Ausführung in Software weiterhin sequentiell erfolgt. Die Modellierung erfolgt dabei ohne jegliches Wissen über den späteren Einsatzfall des DUTs auf einer Leiterplatte.
- Erarbeiten einer Beschreibungsform für die Spezifikation elektrischer Parameter, zeitlicher Abhängigkeiten und L1 Zugriffsroutinen in RTDL. Die Modellierung höherer Ebenen (L2 bis L5) ist gemeinsam im Projekt ROBSY entstanden.
- Realisierung eines Hardware Designflows und des dazugehörigen Compilers zur automatischen Generierung von Co-Prozessoren für eingebettete Testinstrumente.
- Nachweis der Machbarkeit eines automatisch generierten Testsystems bestehend aus Prozessor und Co-Prozessor mit zeitlich korrekter Ansteuerung für at-speed Tests.
- Funktioneller Nachweis in Form von Simulationen.
- Der funktionelle Nachweis des Praxistests für ein Testsystem bestehend aus Prozessor und Co-Prozessor ist zusammen mit den Kollegen des Projekts ROBSY erfolgt.

2. GRUNDLAGEN

In diesem Kapitel sollen die Grundlagen, die für das Verständnis dieser Arbeit notwendig sind, dargestellt werden. Hierbei wird im Folgenden insbesondere darauf eingegangen, welche Arten des Testens es überhaupt gibt, wie Testsysteme im Allgemeinen aufgebaut sind und worauf insbesondere beim strukturellen Testen von Leiterplatten zu achten ist bzw. was hierbei die Besonderheiten sind. Weiterhin werden die nötigen Grundlagen für das Testen am Beispiel von Speichern und für die Synthese von Testsystemen dargestellt.

2.1. ARTEN DES TESTENS

Grundsätzlich kann man beim Testen zwei Zielstellungen unterscheiden. Entweder wird die Funktion der zu prüfenden Leiterplatte getestet oder es wird die Struktur getestet. Im ersten Fall spricht man von einem Funktionstest, während der zweite Test ein Strukturtest ist. Beide Arten des Testens werden im Folgenden näher betrachtet.

2.1.1. FUNKTIONSTEST

Der Funktionstest dient dazu, eine vorher definierte Funktionalität eines Prüflings, beim Leiterplattentest der Leiterplatte, nachzuweisen. Laut [5] ist hierbei nicht nur der Positivtest, sondern auch der Negativtest wichtig. Beim Positivtest wird geprüft, ob das System die gewünschten Funktionen bei bestimmten Eingangsbelegungen erfüllt. Beim Negativtest wird entsprechend das Verhalten auf falsche bzw. fehlerhafte Eingangsbelegungen untersucht. Was eine korrekte Funktion ist und was nicht, muss jedoch in beiden Fällen vorher definiert werden. Von daher unterscheiden sich laut [5] beide Testarten nur durch ihre Definition, nicht aber durch ihre Art der Ausführung.

Auch wenn es der Funktionstest theoretisch ermöglicht 100% Fehlerabdeckung zu erreichen, werden in der Praxis z.B. aufgrund des Zeit- und Kostendrucks oft nicht alle Testmöglichkeiten genutzt und somit eine geringere Testabdeckung erreicht [5].

Ein weiterer Nachteil dieser Testmethode ist die für die Durchführung von Funktionstests notwendige Grundfunktionalität des zu testenden Systems. Laut [5] tritt bei komplexen Systemen genau diese Problematik deutlich zutage, da sich starke Abhängigkeiten der einzelnen Systemkomponenten voneinander ergeben. So führt zum Beispiel eine defekte Speicheranbindung (Adress-, Daten- oder Steuerleitung) dazu, dass gar kein Test korrekt ausgeführt werden kann. Es ist somit nur schwer möglich, Rückschlüsse auf die eigentliche Fehlerursache zu ziehen. Weiterhin gibt der Funktionstest keine messbare Testabdeckung und ist deshalb insbesondere für Debugging und Diagnose nicht geeignet.

Grundsätzlich erschweren oder behindern Abhängigkeiten eine Fehlerdiagnose. Dies führt zu einer aufwendigen und somit teuren Fehlersuche und war laut [5] letztendlich auch der Grund, warum strukturelle Testverfahren entwickelt worden sind. Dies führt zunehmend zu einer Verdrängung des Funktionstests und dazu, dass mehr und mehr Tests auf struktureller Ebene durchgeführt werden. Für eine vertiefende Einführung in die Funktionsweise von Funktionstests sei an dieser Stelle auf [5] verwiesen.

2.1.2. STRUKTURTEST

Beim Strukturtest wird entgegen dem Funktionstest nicht die Funktion, sondern die Struktur eines Schaltkreises beziehungsweise einer Leiterplatte getestet. Im Falle von Leiterplattentests kann laut [5] nach drei wesentlichen Fehlerarten unterschieden werden.

Diese Fehlerarten sind:

- Bauteilfehler
- Bestückungsfehler
- Lötfehler

Bauteilfehler beschreiben dabei Fehler, die durch das Bauteil an sich verursacht werden. Dies können sowohl mechanische (z.B. verbogene Beine) als auch elektrische Fehler (z.B. Kurzschluss im Schaltkreis oder fehlerhafter Bauteilwert) sein. Laut [5] gehören in diese Kategorie sämtliche Fehler, die vor dem Bestückungsprozess in den Gesamtprozess der Leiterplattenfertigung eingebracht werden.

Bestückungsfehler beschreiben Fehler, die durch die Bestückung der Leiterplatte geschehen. Dies können versetzt platzierte Bauteile (z.B. nicht planar aufliegende BGA-Bausteine, hoch stehende Steckerleisten, verdrehte Bauelemente), aber auch falsche Bauteile (z.B. rote statt grüne LED, falscher Temperaturbereich eines Bausteins, falsche Gehäuseform) oder sogar fehlende oder zu viel bestückte Bauteile (z.B. bei Bestückungsvarianten) sein.

Lötfehler entstehen durch den nach der Platzierung erfolgten Lötprozess. Hierzu gehören alle Fehler, die durch eine nicht ordnungsgemäße Lötstelle entstehen. Es gibt je

nach verwendeter Fertigungstechnologie unterschiedliche Möglichkeiten eines Defekts. So sind generell THT-Lötfehler und SMD-Lötfehler an sichtbaren und unsichtbaren Stellen zu unterscheiden. Während im ersten Fall oft eine unzureichende Füllhöhe bzw. unzureichende Lötverbindung die Fehlerursache ist, so treten bei einer SMD-Bestückung oft Kurzschlüsse und unzureichende Lötverbindungen auf sowie an sichtbaren Stellen zusätzlich noch Benetzungsfehler und Grabsteineffekte und an unsichtbaren Stellen Lunker[10] und Black Pads[11].

Eine ausführlichere Darstellung der Fehlerarten mit ihren Unterkategorien, insbesondere zu Lötfehlern, ist [5] zu entnehmen.

2.2. STRUKTURELLE TESTVERFAHREN

Aufgrund der besseren Diagnosemöglichkeiten erfolgt im Rahmen dieser Arbeit einer Beschränkung auf die strukturellen Testverfahren. Bei diesen Testverfahren gibt es generell drei unterschiedliche Ansätze:

- die nicht-elektrischen Verfahren
- die invasiven Verfahren
- nicht-invasiven elektrischen Verfahren

Auf alle drei wird im Folgenden näher eingegangen.

2.2.1. NICHT-ELEKTRISCHE VERFAHREN

Zu den nicht-elektrischen Verfahren gehören in erster Linie optische Messmethoden, die durch das ‚Betrachten' der Leiterplatte Fehler detektieren.

Hierzu zählen:

- manuelle visuelle Inspektion,
- automatische visuelle Inspektion,
- automatische Röntgen-Inspektion.

Bei der **manuellen visuellen Inspektion** (OI[12]) wird die Leiterplatte durch den Testingenieur betrachtet und dieser entscheidet, ob ein Fehler vorliegt oder nicht. Diese Methode setzt einen gut geschulten Testingenieur voraus und ist relativ langsam. Dies macht das Testverfahren zeitaufwendig und führt somit zu hohen Personalkosten, insbesondere wenn nicht nur stichprobenartig getestet werden soll. Weiterhin ist es dem

[10] Lunker: Lufteinschlüsse in Lötverbindungen

[11] Black Pads: Durch Oberflächenveredlung verursachte schlechte Lötverbindung

[12] Optical Inspection

Testingenieur nur möglich, sichtbare Lötstellen zu prüfen. So können z.B. verdeckte Lötstellen, wie sie bei BGAs existieren, nicht untersucht werden [5] [10]. Ein Beispiel für einen Fehler, der gut mittels manueller visueller Inspektion detektiert werden kann, ist in Abbildung 4 dargestellt. Es handelt sich hierbei um einen Kurzschluss zwischen zwei benachbarten Pins eines Schaltkreises.

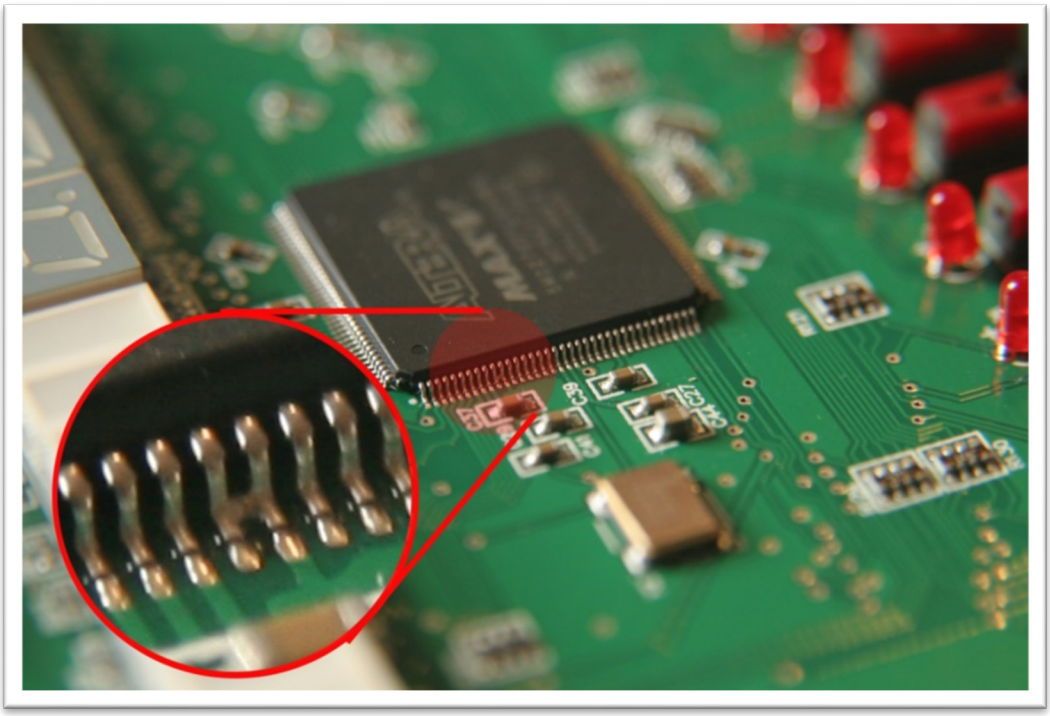


Abbildung 4: Lötfehler (manuelle visuelle Inspektion)

Die **automatische visuelle Inspektion** (AOI[13]) erlaubt durch den Einsatz von Kamerasystemen und computergestützter Auswertung die automatische Inspektion von Leiterplatten. Die Leiterplatten werden hierbei durch das Testsystem mit einer Referenzleiterplatte verglichen. So können die Ergebnisse der Bestückung und des Lötprozesses bewertet werden [5] [10].

Im Gegensatz zu den beiden bisher vorgestellten optischen Verfahren erlaubt die **Automatische Röntgen Inspektion** (AXI[14]) auch das Untersuchen von verdeckten Lötstellen. Dies kann entweder durch eine einfache Durchleuchtung (2D) erfolgen oder aber auch dreidimensional, was mit Hilfe einer entsprechenden computergestützten Auswertung eine noch bessere Fehlerdetektion ermöglicht. Aufgrund der notwendigen Testhardware sind diese Verfahren jedoch sehr kostenintensiv und werden daher oft nur bei Leiterplatten angewendet, die eine sehr hohe Zuverlässigkeit benötigen, da diese

[13] Automated Optical Inspection

[14] Automated X-Ray Inspection

entweder sicherheitskritisch sind oder aber deren Ausfall sehr hohe Folgekosten verursacht [5] [10]. Dies ist zum Beispiel in der Luftfahrt oder bei Weltraumtechnik der Fall.

Alle drei hier vorgestellten visuellen Verfahren erlauben es, neben Kurzschlüssen, offenen Lötstellen oder Bauteilfehlern zumindest auch teilweise die Qualität von Lötstellen zu beurteilen. Der Umfang dieser Beurteilung hängt von dem eingesetzten Verfahren ab, ist aber eine Eigenschaft, die mit elektrischen Testverfahren bisher so nicht möglich ist. Somit ist es möglich, Schwachstellen im Prozess frühzeitig zu erkennen und die Wahrscheinlichkeit eines späteren Ausfalls der Leiterplatte zu minimieren, jedoch zu den beschriebenen Nachteilen der relativ hohen Testkosten.

2.2.2. INVASIVE ELEKTRISCHE VERFAHREN

Die invasiven elektrischen Testverfahren zeichnen sich dadurch aus, dass von außen in die Leiterplatte eingegriffen wird.

Hierbei soll im Wesentlichen auf zwei Verfahren eingegangen werden:

- In-Circuit Test (ICT)
- Flying Probe (FPT)

Der **In-Circuit Test** war einer der ersten strukturellen Testverfahren auf dem Markt [5] und wurde 1979 von der Firma INGUN [11] entwickelt. Hierbei wird über Federkontaktstifte eine physikalische Verbindung zu Testpunkten auf der Leiterplatte hergestellt. Testpunkte sind kleine Kontaktflächen, die an den gewünschten Punkten der Leiterplatte platziert und mit der Schaltung verbunden sind. An diesen Punkten ist es möglich, Signale einzuspeisen und zu messen. Es ist somit möglich, jedes Bauteil auf der Leiterplatte für sich zu testen, statt, wie bis dahin üblich, nur eine ganze Baugruppe. Dies erleichtert die Fehlersuche sowie die genaue Lokalisation von Fehlern und erlaubt durch die Betrachtung einzelner Bauelemente eine deutlich größere Testtiefe auch bei komplexen Leiterplatten.

Ein weiterer Vorteil dieser Testmethode ist, dass alles elektrisch Messbare auch testbar ist [5]. Somit lassen sich Widerstände, Kondensatoren und Induktivitäten bestimmen, aber auch Dioden und Transistoren testen. Jedoch sind die Bauteile nicht vollständig aus der Schaltung herauslösbar, d.h. die Umgebung der Bauelemente muss weiterhin berücksichtigt werden.

Diese Methode ermöglicht es zwar, die Tests mit Hilfe von externem Test-Equipment (ATE) sehr schnell auszuführen (at-speed), jedoch wird das Testverfahren durch die zunehmende Miniaturisierung beschränkt. Dies führt zu Problemen bei der Testpunktsetzung [6], für die entweder kein Platz mehr auf der Leiterplatte vorhanden ist oder die aufgrund kapazitiver Effekte hohe Frequenzen auf der Leiterplatte beeinträchtigen.

Neben der Miniaturisierung gibt es noch weitere Einschränkungen für dieses Testverfahren. So ist es damit nur möglich, Testpunkte auf den Außenlagen der Leiterplatte zu testen. Modernen Leiterplatten werden jedoch heutzutage als Mehrlagenleiterplatten gefertigt, deren Lagenanzahl durchaus über 10 Lagen betragen kann. Insbesondere Leitungen für Signale mit hohen Taktfrequenzen werden jedoch aus kapazitiven und induktiven Gründen oft in den inneren Lagen platziert. Auch neue Technologien bei der Leiterplattenherstellung, wie z.B. flexible Leiterplatten, erschweren den Einsatz von ICT, da diese nicht mehr notwendigerweise starr sind bzw. sie sogar dreidimensional angeordnet sein können. Dadurch ist das zuverlässige Kontaktieren der Leiterplatte nicht mehr gewährleistet.

Neben den bisher genannten Nachteilen ist auch der Preis beim Einsatz von ICT zu berücksichtigen. So entstehen durch die Fertigung eines passenden Messadapters, des sogenannten Nadelbettadapters, hohe Einmalkosten. Schon kleine Änderungen an der Leiterplatte (z.B. bei einer Überarbeitung) erfordern eine kostenintensive Anpassung bzw. vollständige Neukonstruktion des Messadapters. Daher eignet sich dieses Verfahren nur für große Stückzahlen und ausgewählte Leiterplattenkonstruktionen.

Der Aufwand für die Fertigung bzw. die Anpassung des Nadelbettadapters war einer der Gründe für die Entwicklung des **Flying Probe** Verfahrens. Dieses wurde im Jahre 1987 von der Firma TAKAYA [12] entwickelt [5]. Hierbei werden die Testpunkte, wie in Abbildung 5 dargestellt, statt mittels eines starren Adapters über bewegliche Nadeln angefahren. Diese Prüfnadeln sind dynamisch verfahrbar und können die Testpunkte auf der Leiterplatte anfahren. Das Verfahren hat den Vorteil, keine individuell gefertigten Adapter zu benötigen, sondern programmierbar zu sein. Es ist daher sehr flexibel und verursacht geringere Einmalkosten pro zu testender Leiterplatte. Die Anschaffung eines solchen Gerätes ist jedoch sehr teuer und für kleinere Betriebe nicht wirtschaftlich.

Weiterhin ist der größte Vorteil des Flying Probe Test (FPT) auch ein großer Nachteil. Das notwendige Verfahren der Prüfnadeln macht den FPT deutlich langsamer im Vergleich zu ICT. Im Produktionsprozess lässt sich ein FPT nur bei einem entsprechend langsamen Fertigungstakt einsetzen.

Wie bereits beschrieben haben beide Verfahren den Nachteil, einen physikalischen Zugriff zu benötigen, was sie besonders bei verdeckten Lötstellen wie bei BGAs unbrauchbar macht. Dies war einer der Gründe, der zur Entwicklung der nicht-invasiven elektrischen Testverfahren geführt hat, die in Kapitel 2.2.3 beschrieben werden.

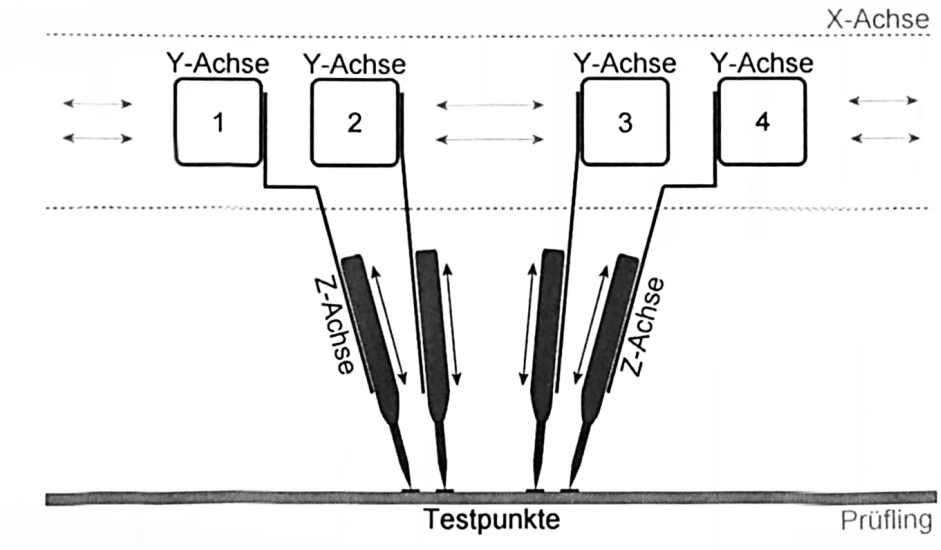


Abbildung 5: Testpunktkontaktierung mittels Flying Probe [5]

2.2.3. NICHT-INVASIVE ELEKTRISCHE VERFAHREN

Die nicht-invasiven elektrischen Testverfahren zeichnen sich dadurch aus, dass sie keine physikalische Kontaktierung zu den Testpunkten auf dem Prüfling benötigen. Ein weit verbreitetes Verfahren, das von der Joint-Test-Action-Group (JTAG) in den 1980er Jahren entwickelt worden ist, ist das Boundary-Scan-Verfahren (BScan). Dies wurde 1990 als Standard IEEE1149.1 [13] verabschiedet und ist seitdem, zuletzt 2013 [14], vielfach übererarbeitet und erweitert worden.

Boundary-Scan nutzt für den Zugriff auf die Leiterplatte interne Schaltungskomponenten, statt hierfür auf Kontaktnadeln angewiesen zu sein. Dies wird durch zusätzlich in den Schaltkreis integrierte Elemente ermöglicht. Diese Elemente sind:

- JTAG Interface
- TAP (Test Access Port) Controller
- Boundary-Scan-Zellen
- weitere teilweise optionale Boundary-Scan-Register, wie zum Beispiel das Bypassregister, ID Register oder Instruktionsregister

Abbildung 6 zeigt die schematische Struktur eines solchen Schaltkreises.

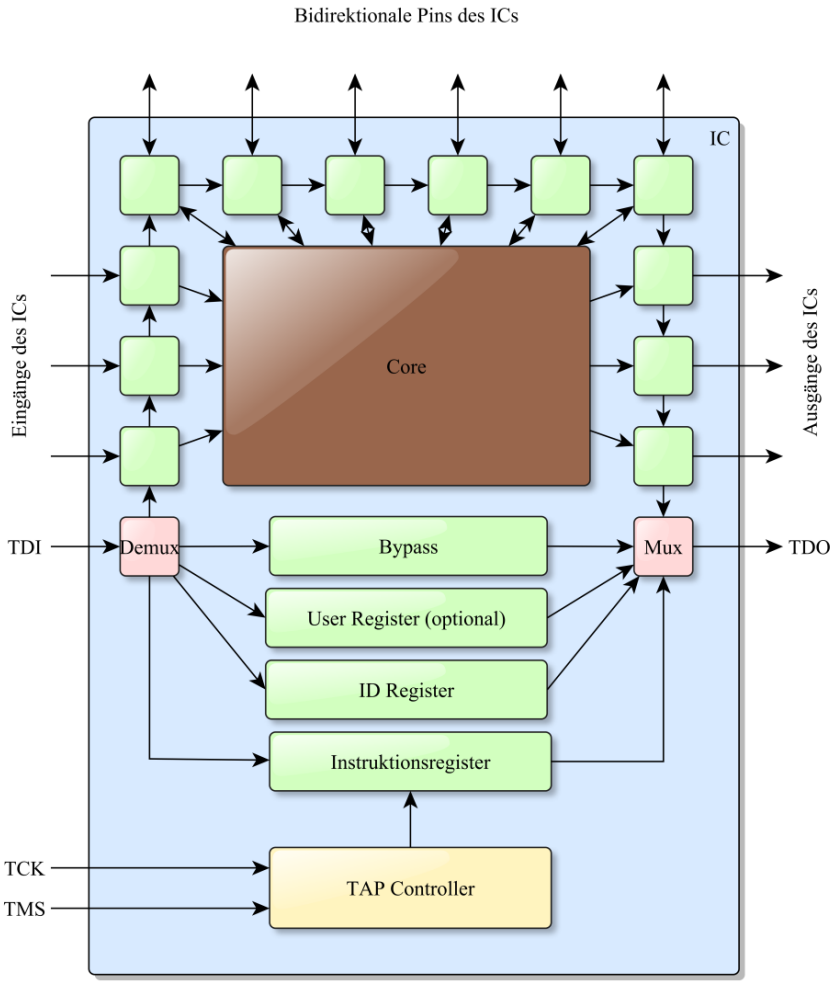


Abbildung 6: Struktur eines Boundary-Scan kompatiblen Schaltkreises

Über die vier[15] Leitungen des JTAG Interfaces (TDI, TDO, TMS und TCK) werden Daten mit dem Schaltkreis ausgetauscht. Über eine Zustandsmaschine im TAP Controller[16] wird das interne Verhalten aller Boundary-Scan Elemente gesteuert. Über zusätzliche Register zwischen den Pins des Schaltkreises und der eigentlichen Kernlogik (Core) ist es möglich, direkt Einfluss sowohl auf die Pins als auch auf die Kernlogik zu nehmen. Diese Verbindungen zwischen Pins und Kernlogik können über JTAG nicht nur voneinander getrennt, sondern auch gelesen oder geschrieben werden. Da die gesamte Logik, die für das Testen mit Boundary-Scan nötig ist, bereits im Schaltkreis integriert ist, sind auf der Leiterplatte lediglich vier zusätzliche Leitungen notwendig, um diese Funktionalität zu nutzen.

[15] Optional gibt es eine fünfte Leitung, die Reset Leitung, die jedoch nicht verpflichtend vorgeschrieben ist und in dieser Arbeit aus Gründen der Übersichtlichkeit daher nicht weiter betrachtet wird.

[16] Zur Übersichtlichkeit wurde die Verbindung des TAP Controllers und des Taktes zu allen Boundary-Scan Zellen und Register verzichtet.

Der Einsatz von nur vier Leitungen für die Steuerung und Datenübertragung bei Boundary-Scan führt dazu, dass alle Daten seriell übertragen werden müssen. So bedeutet ein Zugriff auf einen einzelnen Pin des Schaltkreises, dass dafür trotzdem die gesamte Boundary-Scan-Kette ‚durchgeschoben' werden muss. Je nach Anzahl der Zellen innerhalb dieser Kette führt dies auch bei schnellen Testtakten (TCK) zu einem langsamen Testverhalten an den Pins des Schaltkreises. Daher ist dieses Verfahren nur als statisches Testverfahren einzustufen, dessen ursprünglich primäre Aufgabe der Verbindungstest war. In Abbildung 7 ist ein solcher Verbindungstest zwischen zwei Boundary-Scan-fähigen Schaltkreisen dargestellt. Hierbei sendet der links dargestellte Schaltkreis IC1 bestimmte Testvektoren, die vom rechten IC2 empfangen werden. Je nach empfangenen Daten kann auf bestimmte Fehler geschlossen werden. Im gezeigten Beispiel sind dies ein open (stuck-at-0) Fehler und ein short (wired AND) Fehler. Mehr zu Fehlermodellen und Fehlerarten ist in Kapitel 2.3 dargestellt.

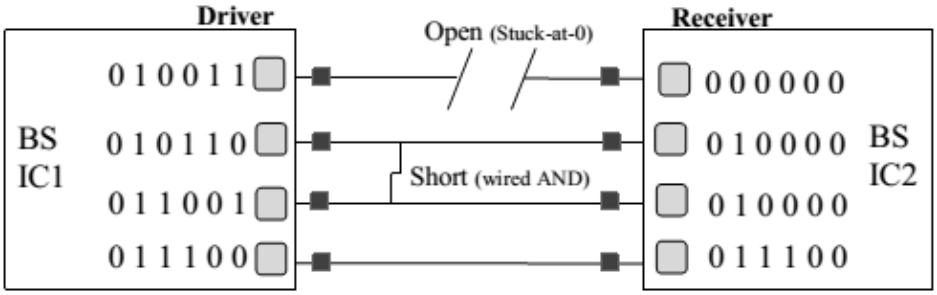


Abbildung 7: Verbindungstest mittels Boundary-Scan [15]

Neben der bisher beschriebenen Testaufgabe bieten die Strukturen, die mit Boundary-Scan eingeführt wurden, insbesondere das JTAG Interface und der TAP Controller jedoch zusätzliche Möglichkeiten.

Diese sind zum Beispiel [15]:

- Prozessordebugging und –emulation: Hierbei wird JTAG verwendet, um interne Register im Prozessor zu steuern und diesen so zu kontrollieren bzw. Register und Speicherinhalte auszulesen.
- BIST: Beim Built-In-Self-Test können bestimmte Strukturen im Schaltkreis über JTAG angestoßen werden, die dann autonom Testalgorithmen ausführen.
- Programmierung von Speicher- oder Logikschaltkreisen: Hierbei wird die JTAG Schnittstelle verwendet, um zum Beispiel Konfigurationsdaten für Schaltkreise mit Programmierbarer Logik oder aber Speicherabbilder für nicht flüchtige Speicherschaltkreise zu übertragen.

Laut einer Umfrage [16] aus dem Jahre 2009, die unter 240 Personen von 131 Firmen aus 27 Ländern erhoben wurde, sehen 79% der Befragten Boundary-Scan als eine wichtige Technik beim Erreichen ihrer Produktionsziele. Lediglich 2% geben an, diese Technik gar nicht zu verwenden. Trotz dieser großen Akzeptanz und der Verbreitung des Standards gibt es jedoch auch viele Probleme. Auf die Frage, wer

bereits Probleme beim Testen von *nicht-boundary-scan fähigen Schaltkreisen*[17] mittels Boundary-Scan hatte, äußerten sich 86% der Befragten bei dynamischen Speichern wie SDRAM positiv, während es bei Flashspeichern immerhin noch 82% waren. Diese Probleme werden immer größer, je höher die Taktfrequenzen werden. So stellen aktuell insbesondere DDR3 und GDDR5 die Testingenieure vor Herausforderungen [17]. Dies sind einige der Gründe, die zur Erweiterung des IEEE1149.1 geführt haben und in Kapitel 3.1 näher beschrieben werden.

2.3. FEHLERMODELLE

Nachdem die Arten des Testens sowie unterschiedliche strukturelle Testverfahren beschrieben worden sind, soll nun auf unterschiedliche Arten von Fehlern eingegangen werden. Diese generieren wiederum unterschiedliche Anforderungen an das verwendete Testsystem und die genutzte Testmethode, um eine akzeptable Testabdeckung bei möglichst niedrigen Testkosten zu liefern.

Laut [18] wird „*eine Änderung des Verhaltens einer Schaltung infolge eines physikalischen Defekts*“ als Fehler bezeichnet. Im Rahmen dieser Arbeit sind folgende Fehler zu unterscheiden:

- **Permanente Fehler**, hierzu gehören *Logische Fehler*, sowie *Parameterfehler*
- **Intermittierende Fehler**, hierzu gehören *Verzögerungsfehler*

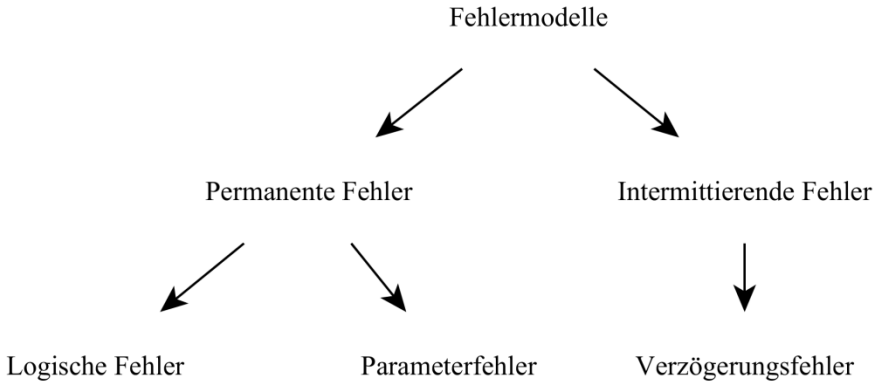


Abbildung 8: Fehlermodelle

„***Permanente Fehler*** *sind zeitinvariant*“ [18]. Zu dieser Art Fehler zählen zum Beispiel logische Fehler und Parameterfehler.

[17] Hierbei werden die Pins eines boundary-scan fähigen Schaltkreises so angesteuert, dass der daran angeschlossene und selber nicht boundary-scan-fähige zu prüfende Schaltkreis getestet werden kann.

„*Ein **logischer Fehler** ist ein Fehler, welcher sich im logischen Verhalten eines Schaltkreises niederschlägt*“ [18]. Zu dieser Kategorie gehören auch Haftungsfehler, im Englischen als ‚stuck-at faults‘ bezeichnet, bei denen eine Leitung einen dauerhaften Wert annimmt, sowie Brückenfehler (bridged faults), bei denen zwei Leitungen miteinander verbunden sind und sich die logischen Signale gegenseitig beeinflussen. Die Art der Beeinflussung kann dabei sehr unterschiedlich sein und sich im Verhalten wie eine ‚Oder‘, ‚Und‘ oder ‚dominante‘ logische Verbindung äußern.

„*Unter **Parameterfehlern** werden Fehler verstanden, welche Schaltkreisparameter wie Geschwindigkeit, Stromaufnahme, Temperaturverhalten etc. verändern*“ [18].

Fehler, die nicht permanent auftreten, werden als **intermittierende Fehler** bezeichnet. Diese sind besonders schwer zu testen, da sie von ganz bestimmten Randbedingungen abhängen, die oft nicht genau bekannt sind oder nur sehr schwer nachgestellt werden können.

„***Verzögerungsfehler** sind Fehlfunktionen des Schaltkreises hinsichtlich der Schaltgeschwindigkeit. Verzögerungsfehler können durch Parameterabweichungen verursacht werden*“ [18]. Diese Fehler sind nur bei at-speed Tests detektierbar, sofern die Verzögerung des Fehlers sich signifikant auf das Signal auswirkt. In der Praxis ist es jedoch möglich, dass eine solche Verzögerung erst im Laufe des Betriebs der Schaltung, also durch Alterung der Schaltung, zu signifikanten Auswirkungen führt. Diese Fehler sind während der Produktion nicht ohne weiteres detektierbar.

Alle aufgeführten Fehler können mit Hilfe von sogenannten Fehlermodellen modelliert werden. Zu den klassischen Fehlermodellen gehören unter anderem stuck-at-0 und stuck-at-1 Fehler. In Anlehnung an [4] wird ein Schaltkreis im Folgenden als C und ein Fehler als f modelliert. Eine Liste von Fehlern ist somit $F = \{f_1, \dots, f_k\}$. Ein Testmuster t für einen Fehler f im Schaltkreis C entspricht einer Belegung der primären Eingänge des zu testenden Schaltkreises. Weiterhin wird eine Menge von Testmustern als $T = \{t_1, \dots, t_n\}$ bezeichnet. Laut [4] wird f_i als testbar bezeichnet, wenn mindestens ein Testmuster t_j für C_f existiert. Ansonsten ist der entsprechende Fehler nicht testbar. Für eine ausführlichere Behandlung des Themas sind [4] und [18] zu empfehlen.

Ziel des Testens ist es, für jeden bekannten Fehler f (mindestens) ein Testmuster t zu finden und dieses anzuwenden. Testmuster werden auch als Testvektoren bezeichnet. Während ihre Generierung im folgenden Kapitel 2.4 kurz beschrieben wird, ist die Anwendbarkeit der Vektoren stark vom verwendeten Testsystem abhängig. Der Aufbau eines solchen Systems wird im Kapitel 2.5 beschrieben.

2.4. TESTVEKTORGENERIERUNG

Das Testen einer Leiterplatte auf einen bestimmten Fehler wird mittels Testvektoren durchgeführt. Diese Vektoren können nach unterschiedlichen Verfahren generiert werden. Hierzu gehören:

- Exhaustive Tests
- Pseudo-Exhaustive Test
- Pseudo-Random Test
- Weighted Pattern Test

Während beim **exhaustive Test** alle möglichen Kombinationen getestet werden, was bei einem kombinatorischen Schaltkreis mit n Eingängen 2^n Möglichkeiten entspricht, kann dies beim **pseudo-exhaustive Test** auf 2^k Elemente ($k < n$) reduziert werden. Dies ist laut [19] und [20] für viele Schaltungen nach einigen Modifikationen möglich und beschleunigt das Testen deutlich.

Beim **Pseudo-Random Test** wird die Vektorgenerierung über eine quasi zufällige, aber reproduzierbare Musterabfolge realisiert und wenn nötig durch **Weighted Pattern Tests** ergänzt, um „random pattern resistant faults“ zu detektieren. Für die genaue Generierung der Testvektoren sei an dieser Stelle auf [21] verwiesen.

Während die Testvektoren und deren Generierung Einfluss darauf haben, wie schnell bestimmte Fehler detektiert werden können, so ist für ihre Anwendung jedoch unerheblich, wie genau die Generierung erfolgt ist. Da diese Arbeit die Anwendung erzeugter Testvektoren verbessert werden soll, ist deren genaue Generierung nicht relevant, solange es mit denen im Rahmen dieser Arbeit entworfenen Methoden möglich ist die Anwendung beliebiger Testvektoren zu verbessern. Aus diesem Grund wird auf die genaue Art der Testvektorgenerierung im weiteren Verlauf der Arbeit nicht weiter eingegangen. In Kapitel 6.2 wird jedoch auf die Möglichkeit beliebige Testvektoren generieren zu können noch einmal eingegangen.

2.5. AUFBAU VON BOUNDARY-SCAN-TESTSYSTEMEN

Nachdem in Kapitel 1.3.3 der grobe Aufbau eines allgemeinen Testsystems vorgestellt worden ist, soll im Folgenden auf den detaillierten Aufbau eines Boundary-Scan-basierten Testsystems eingegangen werden. Dabei werden sowohl die einzelnen Teile des Testers als auch die des Prüflings dargestellt. Der Aufbau bezieht sich auf Testsysteme für den strukturellen Leiterplattentest. Abbildung 9 zeigt ein solches Boundary-Scan basiertes Testsystem. Dieses besteht aus einem Testingenieur, einem Test-PC sowie einem externen Tester und dem eigentlichen Prüfling.

Der **Testingenieur** ist die Person, die einen Test entwirft und ausführt und somit an oberster Stelle in der Testausführung steht.

Als **Test-PC** wird der Teil des Testsystems bezeichnet, der über seine Benutzerschnittstelle im Kontakt zum Testingenieur steht. Er übernimmt die Koordination des Testprozesses einschließlich der Kommunikation mit den weiteren Komponenten des Testsystems.

Als **externer Tester** (oder auch Testbed) werden alle Teile des Testsystems bezeichnet, die sich zwischen Test-PC und Prüfling befinden. Dies ist zumeist ein Stück Elektronik, das die Ansteuerung der Interfaces des Prüflings sowie die Umsetzung der Steuerbefehle des Test-PCs auf z.B. JTAG durchführt. Der externe Tester dient auch als Gegenstück, um Schnittstellen des Prüflings mit bestimmten Tests bedienen zu können. Dies können Interface Tests, Protokoll Tests, I/O Tests oder auch analoge Tests sein. Der externe Tester muss für alle diese Testarten entsprechende elektrische Verbindungen bereitstellen.

Der **Prüfling** ist der zu testende Teil des Testsystems. Im Kontext dieser Arbeit ist dies eine bestückte Leiterplatte. In einigen Ausarbeitungen ist hierfür auch der Begriff System-Under-Test (SUT) zu finden [15]. Diese beiden Begriffe sind als gleichwertig zu betrachten.

Auf dem Prüfling befinden sich einige zumeist über JTAG[18] verbundene Komponenten, z.B. FPGA und Prozessoren (Controller im weitesten Sinne) sowie daran angeschlossene Schnittstellen, Schaltkreise wie zum Beispiel Speicher, Treiber oder andere, sowie weitere diskrete Komponenten. Zu testende Schaltkreise werden als Devices-Under-Test, kurz DUT bezeichnet. Was genau als DUT definiert wird, ist davon abhängig, auf welchen Teil des Prüflings die Testvektoren angewendet werden und was getestet werden soll. Verfügt ein Prüfling zum Beispiel über zwei FPGAs, so kann jeder dieser FPGAs in einem Testdurchlauf als DUT dienen und die Verbindungen zu diesem Schaltkreis durch den anderen FPGA getestet werden.

In Abbildung 9, einer schematischen Darstellung eines möglichen Testszenarios, sind drei DUTs dargestellt, von denen zwei (DUT 1 und DUT 2) über JTAG (fett gezeichnete Verbindungen) mit dem FPGA bzw. Prozessor verbunden und somit Teil einer Boundary-Scan-Kette sind. DUT 3 hingegen hat keine Verbindung zu JTAG. Während im ersten Fall ein Testen wie in Abbildung 7 dargestellt möglich ist, und die Testvektoren vom Sender direkt am Empfänger gelesen werden können, ist dies im zweiten Fall nicht möglich. Eine Aussage über die Verbindung zu DUT 3 kann deshalb nur über Umwege getroffen werden, z.B. indem Daten zum DUT geschrieben und anschließend wieder gelesen werden. Aus der Antwort können Rückschlüsse auf die Verbindungen zwischen den Schaltkreisen geschlossen werden. Dieser Vorgang ist schematisch in Abbildung 10 dargestellt.

[18] JTAG – Joint Test Action Group: Serielle Schnittstelle mit 4 (optional 5) Leitungen

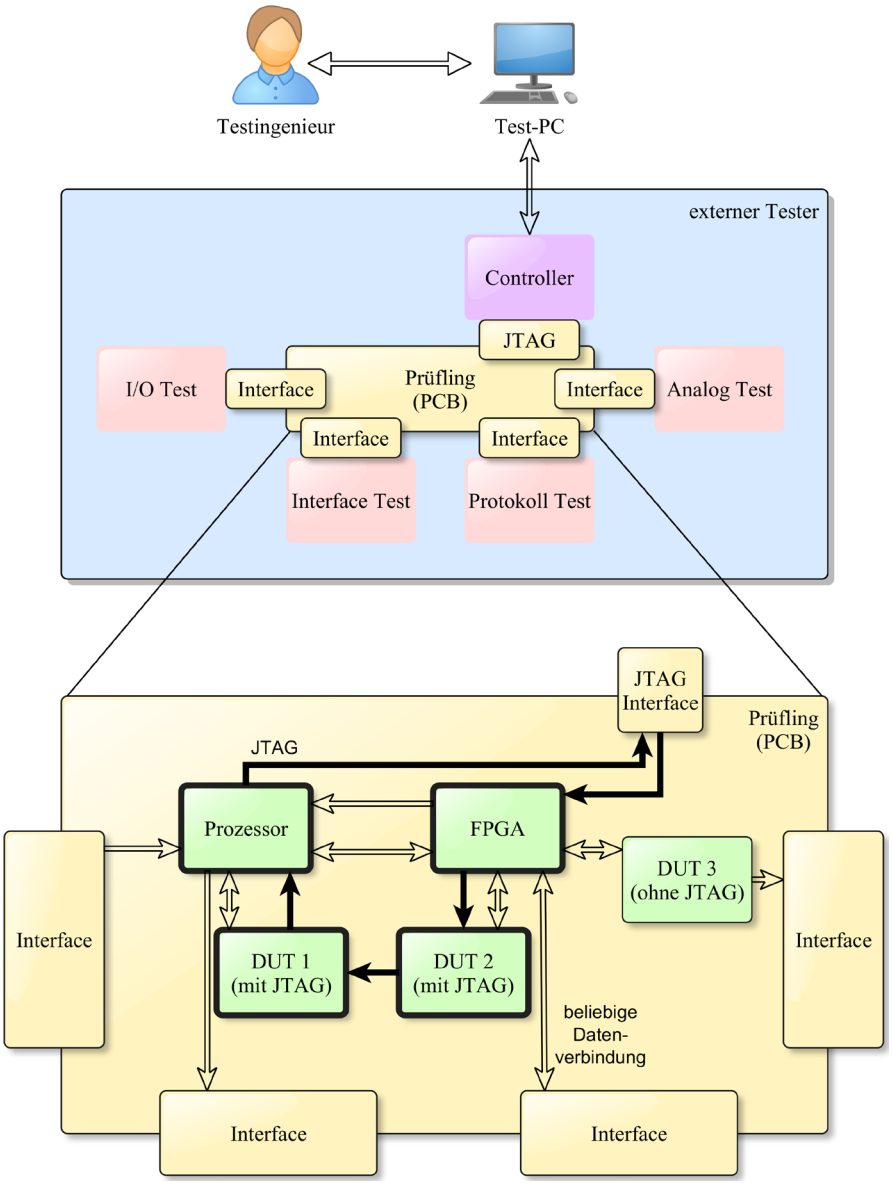


Abbildung 9: Detaillierter, schematischer Aufbau eines möglichen Testszenarios

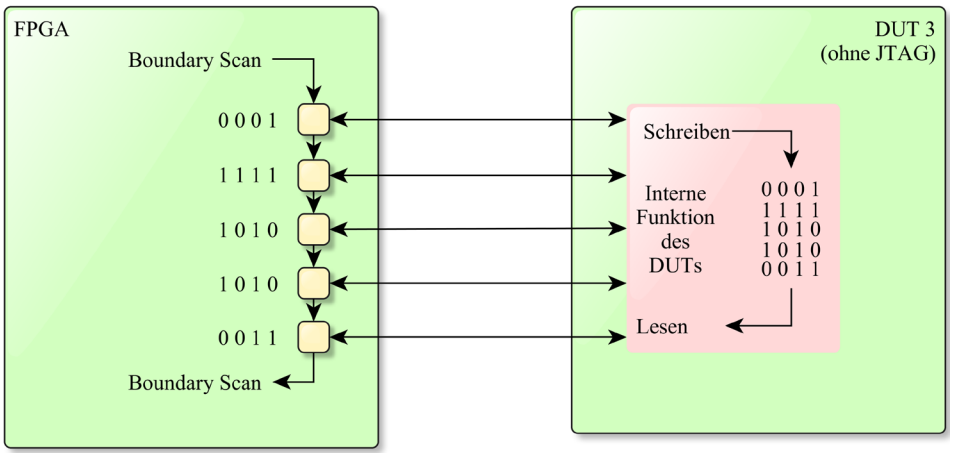


Abbildung 10: Indirektes Testen eines nicht Boundary-Scan-fähigen Schaltkreises

Neben der Struktur des technischen Aufbaus aus Abbildung 9 gibt es noch eine algorithmische Sicht auf ein Boundary-Scan-Testsystem. Deren Komponenten sind in Abbildung 11 dargestellt und werden im Folgenden näher erläutert.

Der **Testentwurf** wird vom Testingenieur durchgeführt und schließt den Entwurf aller weiteren aufgeführten Teile eine Tests ein.

Die **Testalgorithmen** beschreiben die durchzuführenden Tests in einer eindeutigen, jedoch ggf. noch abstrakten Form, die vor der Ausführung noch an die Art des genutzten externen Testers sowie der Ausführung auf dem Prüfling übersetzt werden müssen.

Als **Testprozess** wird die Ausführung von Testalgorithmen bezeichnet. Dieser wird vom Testingenieur gestartet und auch durch ihn beendet, bzw. das Ende wird durch ihn bewertet.

Als **Testprogramm** wird das Ergebnis der Umsetzung der Testalgorithmen in eine durch das Testsystem ausführbare Form bezeichnet. Hierbei muss das Testprogramm nicht zwingend komplett im FPGA ausgeführt werden, sondern kann auf dem Testsystem auch zeitlich sowie räumlich verteilt sein. So wäre es möglich, einen FPGA mit zwei verschiedenen Programmierungen nacheinander einzusetzen oder zeitgleich zwei FPGAs jeweils Teile eines Testprogramms ausführen zu lassen.

Eine **Testfunktion** beschreibt einen einzelnen Block innerhalb eines Testprogramms. Eine Testfunktion stellt dabei immer eine bestimmte Funktionalität über eine definierte Schnittstelle bereit. Die konkrete Implementierung einer Testfunktion kann jedoch von einem Testsystem zum anderen variieren.

Ein **Testvektor** ist die Datenausgabe einer Testfunktion, welche zum DUT gesendet wird. Der Testvektor beschreibt ein ganz bestimmtes Muster, um bestimmte Fehler zu testen.

Komplexe Testprogramme bestehen dabei meist aus einer Reihe von Testfunktionen, die die gesamten Testalgorithmen strukturiert abbilden. Hierbei greifen die einzelnen Testfunktionen jeweils auf bestimmte Funktionalitäten anderer Testfunktionen zu. Stellt man sich diese als eine hierarchisch angeordnete Struktur vor, so stellt die oberste Ebene eine Verbindung zum Testingenieur her, während die unterste die direkte Ansteuerung der DUTs übernimmt.

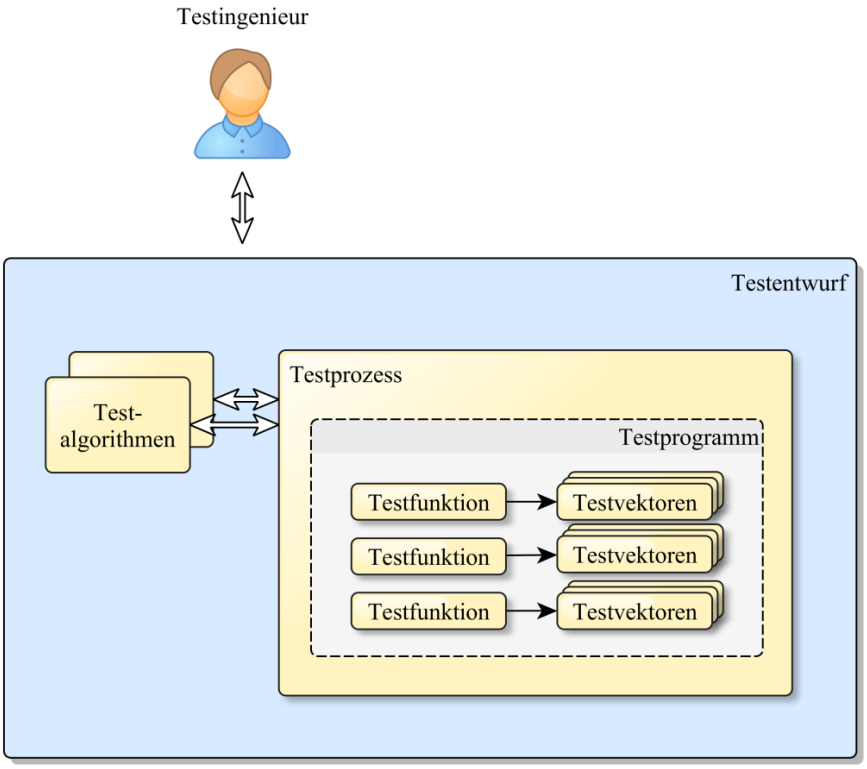


Abbildung 11: Algorithmische Sicht auf ein Boundary-Scan-Testsystem

2.6. SPEICHERTEST

Speichertests sind besonders wichtig, da Speicher sowohl innerhalb von Prozessoren, FPGAs und SoCs (System-On-Chip) vorhanden sind als auch als eigenständige Schaltkreise [4] in vielen eingebetteten Systemen. Auf Leiterplatten sind Speicher, bis auf serielle Speicher wie zum Beispiel SPI/I^2C EEPROMs/Flashs, über Adress-, Daten- und Steuerleitungen mit dem Prozessor, FPGA oder SoC verbunden. Deshalb können an diesen Verbindungen viele Fehler auftreten, so dass für eine akzeptable Testabdeckung entsprechend viele Testvektoren notwendig sind, die wiederum zu langen Testzeiten und hohen Testkosten führen. Zusätzlich arbeiten moderne Speicher mit hohen Taktfrequenzen und Tests müssen dies entsprechend berücksichtigen, um keine dynamischen Fehler zu übersehen, die erst bei der Verwendung der sogenannten Arbeitsfrequenz, also der Frequenz, mit der der Schaltkreis in der Praxis eingesetzt werden soll, entstehen.

Dies ist der Grund, warum sich diese Arbeit vorrangig auf den strukturellen Verbindungstest von Speichern fokussiert. Die Anwendbarkeit der entwickelten Methode im Allgemeinen wird jedoch auch für andere Schaltkreise (DUTs) geprüft und abschließend bewertet. Im Rahmen dieser Arbeit werden die Speichertypen SRAM, serieller Flash und DRAM berücksichtigt.

Die Ansteuerung und deren Besonderheiten werden nachfolgend näher erläutert.

2.6.1. SRAM

Bei SRAM handelt es sich um einen statischen Speicher, der entweder synchron mit einem Takt oder asynchron, also ohne Takt arbeitet. Die Datenübertragung wird in der Regel durch Adress-, Daten- und Steuerleitungen gesteuert. Die Daten in einem statischen RAM bleiben dauerhaft erhalten, solange die Spannungsversorgung erhalten bleibt.

Neben der ‚normalen' parallelen Ansteuerung über Adress-, Daten- und Steuerleitungen gibt es auch SRAMs mit einem seriellen Interface. Diese werden in der Regel über ein Protokoll wie zum Beispiel I^2C oder SPI angesprochen und sind bezüglich des Testens mit seriellem Flash (siehe Kapitel 2.6.2) zu vergleichen.

Für SRAMs, die über Adress-, Daten- und Steuerleitungen angesteuert werden, ist es üblich, einfache Testalgorithmen wie zum Beispiel die sogenannte walking-1 bzw. walking-0 oder True-Complement-Codes als Testvektoren zu verwenden. Die Vektoren werden dabei durch Schiebe- bzw. Zähloperationen generiert und erlauben das Detektieren von Kurzschlüssen zwischen Leitungen, von offenen Leitungen und unter bestimmten Bedingungen auch das Detektieren von Verzögerungsfehlern. Sie liefern prinzipiell eine gute Fehlerdiagnostik.

Als Speicher wird im Rahmen der Arbeit ein SRAM des Typs IS61LV25616AL [22] betrachtet. Dies ist ein asynchrones 4 Mbit High-Speed SRAM, organisiert als 256k x 16 Bit, welches über einen 18 Bit Adressbus, 16 Bit Datenbus sowie 5 Steuerleitungen verfügt. Auf den Inhalt des Speichers kann über unterschiedliche Lese- und Schreibbefehle zugegriffen werden. Das RAM gibt es für unterschiedliche Zugriffszeiten zwischen 8 und 15 ns.

2.6.2. SERIELLER FLASH

Bei seriellem Flash handelt es sich um einen nicht flüchtigen Speicher, der über ein serielles Interface angesprochen werden kann. Hierbei kommen in der Regel Protokolle wie zum Beispiel I^2C oder SPI zum Einsatz. Die Ansteuerung der meisten Funktionen wird dabei über dieses Interface ausgeführt. Im Gegensatz zu parallelem Flash, dessen Interface ähnlich zu dem von SRAM ist (siehe Kapitel 2.6.1), werden bei seriellem Flash nur sehr wenige Leitungen verwendet. Daher unterscheidet sich dieser bezüglich der Auswirkungen möglicher Fehler und den durchzuführenden Tests. So sind für eine Datenübertragung über das I^2C Protokoll lediglich zwei Leitungen notwendig. Diese sind jedoch für alle Operationen notwendig, weshalb eine Diagnose sich auf go-nogo Ergebnis beschränken muss, d.h. es gibt eine Kommunikation mit dem Speicher oder nicht. Für das Testen der Verbindung ist somit die Umsetzung des Protokolls wichtig, nicht die Art der zu übertragenden Daten wie beim SRAM in Kapitel 2.6.1.

2.6.3. DRAM

Bei DRAM handelt es sich um einen dynamischen Speicher, der stellvertretend für die Gruppe der dynamischen Speicher wie z.B. DDR2 bis DDR4 oder GDDR5 gewählt worden ist. Eine Besonderheit dieser Speicher ist die Notwendigkeit, den Speicherinhalt regelmäßig aufzufrischen, da dieser nicht statisch ist. Weiterhin wird die Ansteuerung durch das Einhalten strikterer zeitlicher Vorgaben als dies z.B. bei SRAM der Fall ist erschwert. So gibt es für viele zeitliche Randbedingungen neben minimalen Zeiten, die nicht unterschritten werden dürfen auch maximale Vorgaben, die nicht überschritten werden dürfen. Hinzu kommen insbesondere bei hohen Taktfrequenzen neuester Technologien, wie DDR4 oder GDDR5, immer mehr Randbedingungen im Subtaktbereich, die eingehalten werden müssen, um eine zuverlässige Funktion zu gewährleisten.

Bezüglich des Interfaces gibt es bei DDR DRAM weitere Besonderheiten gegenüber SRAM. So steht das Kürzel DDR für die Dual-Data-Rate-Ansteuerung, bei der beide Taktflanken für die Datenübertragung genutzt werden. Außerdem gibt es mehrere Konfigurationsregister, mit denen die Funktion des Speichers beeinflusst werden kann. Diese Register sind jedoch nicht über unabhängige Leitungen ansteuerbar, sondern über die Adressleitungen des Speichers. Weiterhin erfolgt der Datenzugriff aufgrund der internen Organisation des Speichers neben einem Adress- und Datenbus noch durch Signale für die Bank- sowie Zeilenselektion. Alle diese Eigenschaften erschweren die Ansteuerung des Speichers bzw. führen zu einer erschwerten Diagnose im Falle eines Fehlers. Als Testalgorithmen werden jedoch, ähnlich wie beim SRAM, walking-1 bzw. walking-0 oder True-Complement Codes verwendet.

2.7. ZUSAMMENFASSUNG

Im Kapitel 2 sind die Grundlagen für das Verständnis des strukturellen Testens dargestellt. Hierbei geht es insbesondere um den strukturellen Leiterplattentest, auch Verbindungstest genannt. In diesem Zusammenhang wurde besonders auf das strukturelle, nicht-invasive elektrische Testverfahren Boundary-Scan eingegangen, sowie unterschiedliche Fehlermodelle, die Testvektorgenerierung und deren Einfluss auf Testsysteme gezeigt. Es wurden drei unterschiedliche Arten von Speichern, die im weiteren Verlauf der Arbeit betrachtet werden sollen, kurz vorgestellt und ihre Anforderungen an die Testalgorithmen erläutert. Im folgenden Kapitel 3 wird auf den Stand der Technik bezüglich nicht-invasiver elektrischer Testverfahren näher eingegangen.

3. STAND DER TECHNIK

Nachdem in Kapitel 2.2 die Grundlagen zu unterschiedlichen strukturellen Testverfahren dargelegt worden sind, wird im folgenden Kapitel ein Überblick über den aktuellen Stand der Technik im Bereich des strukturellen Leiterplattentests gegeben. Dabei wird auf vorhandene Verfahren und deren Eigenschaften eingegangen, außerdem werden die Ergebnisse aktueller Forschungsarbeiten kritisch betrachtet.

3.1. NICHT-INVASIVE ELEKTRISCHE TESTVERFAHREN

In Kapitel 1.5 wurden die folgenden sechs Ziele, die für eine Verbesserung des strukturellen Testens von Verbindungen auf Leiterplatten erreicht werden sollen, ausführlich definiert:

- at-speed Tests
- flexible Testgenerierung
- großes Einsatzgebiet
- automatische Adaption
- umfangreicher Testzugriff
- schnelle Testausführung

Ziel dieses Kapitels ist es zu beurteilen, welche der aktuellen Testverfahren sich eignen, um die beschriebenen Ziele zu erreichen.

3.1.1. IEEE 1149.X

Neben dem in Kapitel 2.2.3 vorgestellten IEEE 1149.1 wurden im Laufe der letzten Jahre diverse Erweiterungen in Form weiterer Standards hierzu verabschiedet. Diese haben alle das Ziel, bestimmte Eigenschaften des ursprünglichen Standards zu verbessern.

Zum Standard IEEE 1149 gehören:

- 1149.1: Test Access Port and Boundary-Scan Architecture [14]
- P1149.2: Extended Digital Serial Subset[19] [23].
- P1149.3: System Test Bus[20] [23].
- 1149.4: Mixed-Signal Test Bus [24]
- 1149.5: Module Test and Maintenance Bus (MTM-Bus) Protocol [25]
- 1149.6: Boundary-Scan Testing of Advanced Digital Networks [26]
- 1149.7: Reduced-Pin and Enhances-Functionality Test Access Port [27]
- 1149.8.1: Boundary-Scan-Based Stimulus of Interconnections to Passive and/or Active Components [28]
- P1149.10: High Speed Test Access Port and On-chip Distribution Architecture[21] [29]

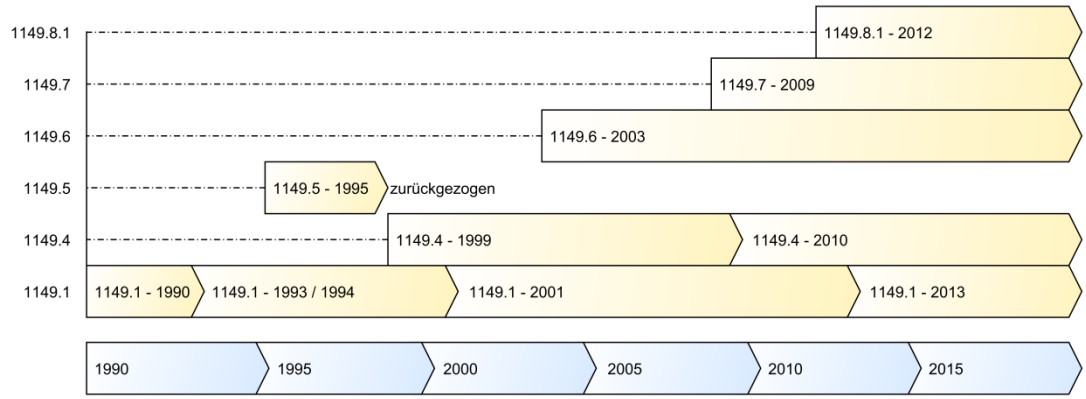


Abbildung 12: Übersicht zu IEEE 1149.X Standards

Während der Standard 1149.5 zurückgezogen worden ist und nicht mehr verwendet wird, befindet sich der Standard 1149.10 noch in der Entwicklung. Alle anderen Standards werden aktiv genutzt. Hierbei hat weiterhin 1149.1 die größte Verbreitung, gefolgt von 1149.7. Die anderen Standards konnten sich bisher nur in kleinerem Maße durchsetzen, weshalb bisher FPGAs und µC weitestgehend nur mit 1149.1 ausgestattet werden.

Der Standard 1149.6 wurde entwickelt, um dynamische Tests an differentiellen Leitungspaaren durchführen zu können. Es ist damit möglich, Impulsfolgen zwischen Sendern und Empfängern auszutauschen. Dies setzt jedoch entsprechend kompatible

[19] Entwurf eines Standards mit mehr I/O Pins zur Datenübertragung. Wurde nie als Standard verabschiedet und wird daher nicht weiter betrachtet.

[20] Entwurf eines Standards für einen System Test Bus. Wurde nie als Standard verabschiedet und wird daher nicht weiter betrachtet.

[21] Standard noch nicht verabschiedet.

Sender und Empfänger voraus, um den Test auch auswerten zu können. Das Übertragen von definierten Signalfolgen, wie dies z.B. für die Implementierung von Protokollen nötig wäre, ist hierbei nicht möglich.

Insgesamt betrachtet stellen die gesammelten IEEE 1149 Standards keine geeignete Option für die im Rahmen dieser Arbeit gesuchte Lösung dar, um die in Kapitel 1.5 beschriebenen Ziele zu erreichen. Insbesondere die fehlende Adaption an einen bestimmten Prüfling, die eingeschränkte Anwendbarkeit auf ein möglichst großes Einsatzgebiet durch die Unterstützung vieler FPGAs, sowie die weitestgehend fehlende Unterstützung von beliebigen at-speed Tests, ergeben für keinen dieser Standards ein angemessenes Erreichen der definierten Ziele. Eine Übersicht ist in Tabelle 1 dargestellt.

Tabelle 1: Übersicht zur Eignung von IEEE 1149.X[22]

Ziele / Standard	at-speed Test	flexible Test-generierung	großes Einsatz-gebiet	auto-matische Adaption	umfangreicher Testzugriff	Schnelle Test-ausführung
1149.1	×	✓	✓	×	×	×
1149.4	×	✓	×	×	× / ✓	×
1149.5	×	×	×	×	×	×
1149.6	× / ✓	×	×	×	× / ✓	×
1149.7	×	✓	×	×	×	×
1149.8.1	×	✓	×	×	× / ✓	×
1149.10	× / ✓	✓	×	×	×	× / ✓

×: Wird nicht unterstützt ✓: Wird unterstützt × / ✓: Mit Einschränkungen

3.1.2. BIST

Der Built-In-Self-Test (BIST) ist eine weitere Möglichkeit im Bereich der elektrischen Testverfahren, bei der zusätzliche Testhardware ins System integriert wird. Dies können zum Beispiel integrierte Testgeneratoren sein. Diese Generatoren können entweder für funktionale Tests oder für Verbindungs-Built-in-Self-Test genutzt werden [30] [31]. In beiden Fällen kann BIST prinzipiell den Nachteil der langsamen Testausführung von Boundary-Scan überwinden. Jedoch benötigt BIST die Integration zusätzlicher Strukturen im Schaltkreis, was aus Kostengründen oft nicht akzeptabel ist.

[22] Es werden nur die jeweils neuesten Versionen der Standards berücksichtigt.

Weiterhin muss die zu integrierende Logik bereits zur Zeit des Schaltkreisentwurfs definiert werden und kann danach nicht mehr geändert werden[23].

Bei [32] wird ein FPGA genutzt, um selbst Boundary-Scan-basierte Tests auszuführen. Es wird parallel zur bestehenden Boundary-Scan Hardware eine eigene Teststruktur, basierend auf Boundary-Scan, in die programmierbaren Ressourcen des FPGA implementiert. Zwar ist ein solches Verfahren auch als BIST zu betrachten, es eliminiert jedoch nicht die wesentlichen Nachteile von Boundary-Scan, wie die fehlende at-speed Unterstützung und eine aufgrund der seriellen Datenübertragung langsame Testausführung. Weiterhin benötigt das vorgestellte Verfahren ein paar dedizierte Pins am FPGA und muss daher bereits beim Entwurf der Leiterplatte berücksichtigt werden. Dies schränkt die Nutzung für beliebige Leiterplatten ein. Weiterhin kann selbst die Reservierung von einigen wenigen Pins beim Leiterplattenentwurf ein K.O.-Kriterium für die Umsetzung darstellen, da die Pinanzahl oft stark begrenzt ist und die Entwickler aus Kostengründen meist die kleinstmögliche Bauform eines FPGAs, die die gestellte Aufgabe noch erfüllt, wählen (müssen).

Während bei der Testausführung der Ansatz, bestimmte Testfunktionalität fest in den Schaltkreis zu integrieren, viele positive Eigenschaften hat, so ist dies zur Entwurfszeit des Schaltkreises bei FPGAs keine Option, da das spätere Einsatzumfeld dieser Schaltkreise nicht bekannt ist und sehr vielfältig sein kann. Auf die Möglichkeit die programmierbaren Ressourcen des FPGAs für die Implementierung von Testfunktionalität zu nutzen wird in Kapitel 3.2 näher eingegangen.

3.1.3. KOMBINIERTE TESTVERFAHREN

Durch die Kombination von Testverfahren wird beabsichtigt, deren jeweilige Vorteile zu vereinen und wenn möglich die Nachteile zu kompensieren. Im Folgenden wird nur auf die Kombination von elektrischen Verfahren miteinander näher eingegangen. Andere Varianten, wie die Kombination mit optischen Testverfahren, werden aufgrund der Zielstellung elektrische Verfahren zu nutzen nicht weiter betrachtet.

Bei der Kombination von **Funktionstest und In-Circuit Test / Flying-Probe** ist es möglich die Testlücken, die durch fehlende Testpunkte beim In-Circuit bzw. Flying-Probe Test entstehen, durch Funktionstest zu füllen.

Um im Falle eines Fehlers die ungenaue Aussage des Funktionstests über eine mögliche Fehlerursache zu verfeinern, kann ein **Funktionstest mit Boundary-Scan** kombiniert werden und so eine detaillierte Diagnose ermöglichen. Weiterhin ist besonders im digitalen Bereich die Testtiefe von Boundary-Scan größer, während der Funktionstest

[23] Dies gilt für die Hardware des BIST. Führt BIST eine bestimmte Software aus, so kann diese ggf. auch nach der Schaltkreiserstellung geändert werden.

dort, wo Boundary-Scan nicht funktioniert, also bei analogen Schnittstellen und schnellen Datenverbindungen, noch eine Aussage über deren Funktion ermöglicht.

Bei der Kombination von **In-Circuit und Boundary-Scan** ist es von Vorteil, den schlechten Testzugriff des In-Circuit Tests, der durch fehlende Testpunkte entsteht, durch Boundary-Scan ausgleichen zu können. Da Boundary-Scan im Vergleich zum In-Circuit Test kostengünstiger ist, wird in der Praxis oft gegenteilig argumentiert. Es wird möglichst viel mit Boundary-Scan getestet und erst im Grenzbereich, wo Boundary-Scan keine Testergebnisse mehr liefert, also bei analogen Schnittstellen oder schnellen Datenverbindungen, wird der In-Circuit Test eingesetzt.

Bei der Kombination von **Flying-Probe und Boundary-Scan** ergeben sich die gleichen Vorteile wie beim In-Circuit Test und Boundary-Scan, jedoch erfolgen hierbei die Einsparungen nicht durch geringere Kosten für den Nadeladapter, sondern durch eine verkürzte Testzeit, da der Flying-Probe Tester weniger Stellen mit seinen Nadeln kontaktieren muss, da das Stimulieren in vielen Fällen durch Boundary-Scan erfolgen kann.

Alle diese Verfahren ergeben eine Verbesserung beim Testen, die sich entweder in einer besseren Testabdeckung, schnelleren Testausführung oder geringeren Kosten beim Testen auszeichnet. Jedoch bleiben auch einige Probleme offen. So gelten für Tests, die statt mit In-Circuit oder Flying-Probe nun mit Boundary-Scan ausgeführt werden, auch die Einschränkungen dieses Verfahrens. Somit sind Tests quasi nur noch statisch ausführbar und ein at-speed Testen ist nicht mehr möglich. Zum Beispiel sind Speicher oder schnelle Schnittstellen aufgrund der fehlenden Testpunkte und ihrer schnellen Signalwechsel weder mit In-Circuit oder Flying-Probe noch Boundary-Scan sinnvoll testbar. Ein Funktionstest wäre zwar möglich, jedoch setzt dieser ein umfangreiches Wissen über den FPGA voraus, auf dem dieser Test ausgeführt werden soll. Weiterhin ist, wie bereits beschrieben, die Diagnose nur sehr grob möglich. Folglich ist keines der vorgestellten Verfahren ausreichend, um alle gestellten Ziele aus Kapitel 1.5 zu erreichen.

3.2. ANSÄTZE ZU REUSE-OF-BOARD-COMPONENTS

Im Gegensatz zu Ansätzen, die bestimmte Änderungen am Schaltkreis voraussetzen, gibt es auch die Möglichkeit, ausschließlich auf bestehende Strukturen zurückzugreifen. Diese Methoden haben den großen Vorteil, sofort auf die betrachteten Systeme anwendbar zu sein, da keine Voraussetzungen an die Schaltkreis- oder Leiterplatten-herstellung bestehen und nur Elemente verwendet werden, die bereits vorhanden sind. Diese Ansätze werden als „Reuse-of-Board-Components“ bezeichnet und im Rahmen dieser Arbeit, wie in Abbildung 13 dargestellt, strukturiert. Hierbei ist der besondere Fokus auf FPGA-basierte Verfahren gelegt.

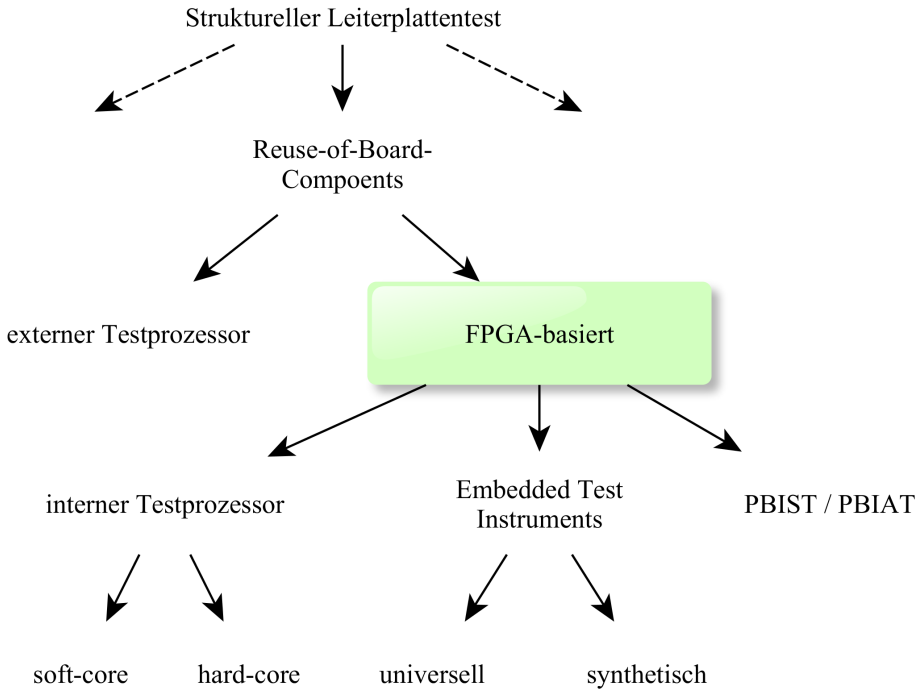


Abbildung 13: Strukturierung der „Reuse-of-Board-Components“ Ansätze

Aus Abbildung 13 gehen drei zu untersuchende Ansätze der FPGA-basierten Verfahren hervor. Diese sind:

- die internen Testprozessoren
- der Programmable Built-In Self Test (PBIST) und Programmable Built-In Assisted Test (PBIAT)
- sowie die Embedded Test Instruments

3.2.1. TESTPROZESSOR

In vielen Testverfahren spielen Prozessoren eine wichtige Rolle während des Testablaufs. In klassischen Verfahren, wie ICT oder Boundary-Scan, werden die Prozessoren auf dem externen Test-Equipment zur Steuerung und Synchronisierung des Testablaufs sowie der Verarbeitung der Testdaten verwendet. In anderen Testverfahren sind sie im Prüfling selbst eingebettet und werden als aktive Testkomponenten verwendet. Hierbei kann man zwischen Prozessoren, die ausschließlich für den Testzweck verwendet werden und solchen, die eigentlich ‚normale‘ Funktionen ausführen und nur im Falle eines Tests umkonfiguriert werden, unterscheiden.

Aus Sicht des FPGA-basierten Testens sind nur die internen (eingebetteten) Prozessoren relevant, die im Gegensatz zu externen Prozessoren nicht neben dem FPGA auf der Leiterplatte platziert werden, sondern im FPGA integriert sind. Bei diesen

Prozessoren kann wiederrum zwischen hard-core und soft-core Prozessoren unterschieden werden. Während erste fest im FPGA vorhanden sind, werden soft-core Prozessoren in der programmierbaren Logik des FPGAs implementiert.

Das soft-core prozessor-basierte Testen unterscheidet sich vom hard-core prozessorbasierten Testen nur durch die physische Umsetzung des Prozessors. Somit können die Vor- und Nachteile, die für eigenständige Prozessoren gelten, im Wesentlichen auch auf die soft-core Prozessoren übertragen werden. Bei den soft-core Prozessoren ergibt sich jedoch ein entscheidender Vorteil gegenüber den festverdrahteten Prozessoren. So ist es möglich, die Prozessoren an die Aufgabenstellung des Testens anzupassen. Dies kann zwar mit erheblichem Aufwand verbunden sein, liefert aber die Möglichkeit, den Prozessor für die Testausführung zu optimieren. Da der FPGA programmierbar ist, kann sich die Optimierung für die Testausführung auf den Zeitpunkt des Testens beschränken und muss nicht dauerhaft im FPGA integriert bleiben und somit Ressourcen belegen. Eine ausführliche Gegenüberstellung verschiedenster soft-core Prozessoren mit direktem Bezug zu dieser Dissertation ist [33] zu entnehmen.

Weiterhin muss man zwei Arten von eingebetteten Prozessoren (egal ob soft-core oder hard-core) unterschieden.

- Universelle Prozessoren
- Spezielle Testprozessoren

3.2.1.1. UNIVERSELLE PROZESSOREN

Universelle Prozessoren sind Prozessoren ohne spezielle Testfunktionen, die entweder bereits im integrierten Schaltkreis für die geplante Endanwendung eingebettet sind oder nur zum Testen genutzt werden. Sie werden zur Ausführung der sogenannten Software-basierten-selbst-Test (SBST) Verfahren [34] [35] [36] verwendet, die als effiziente Alternative entstanden sind, um die klassischen strukturellen Testverfahren zu ergänzen. In SBST wird keine Implementierung von zusätzlichen Testressourcen benötigt, sondern die Testfunktionen werden in Form von Programmcode vom externen Test-Equipment auf die eingebetteten Prozessoren verschoben. Dies ermöglicht nichtinvasive Tests der Prozessor-Peripherie sowie die Ausführung der Tests at-speed, sofern der Prozessor auf die zu testenden Strukturen in angemessener Weise zugreifen kann.

Die Ergebnisse aus [35] und [36] zeigen die Wirksamkeit des SBST. Auf diese Weise wird eine hohe Fehlerabdeckung, ohne die Notwendigkeit von teurem externen Test-Equipment oder zusätzlichen eingebetteten Testkomponenten erreicht.

Der aktuelle Trend bei modernen kommerziellen Testsystemen für den strukturellen Test von Verbindungen auf Leiterplatten geht ebenfalls dahin, die Testfunktionen nicht mehr auf externem Test-Equipment auszuführen, sondern diese in den Prüfling selbst einzubetten. Dieses wird bei Prozessoren als *Boundary-Scan-Emulation* bzw. *JTAG-Emulation* [37] [38] [39] bezeichnet. Auch in [40] wird ein entsprechender „processor-

controlled test“ Ansatz beschrieben. Hierbei werden Testvektoren nicht direkt über Boundary-Scan übertragen, sondern einem Prozessor als algorithmische Beschreibung übergeben und dieser kann die Testvektoren dann selbst generieren und gezielt an das DUT anlegen. Dies beschleunigt die Testabarbeitung deutlich und ermöglicht at-speed Tests. Ein Nachteil dieses Verfahrens ist, dass viel Wissen über den Prozessor und dessen Interna vorhanden sein muss [41] [42].

3.2.1.2. SPEZIELLE TESTPROZESSOREN

Im Gegensatz zu den universellen Prozessoren gibt es verschiedene Ansätze mit speziellen Testprozessoren zu arbeiten, die auf einen spezifischen Testfall zugeschnitten sind. Die Testprozessoren werden nur zum Zweck des Testens in den integrierten Schaltkreis (intern im FPGA) bzw. auf der Leiterplatte (extern vom FPGA) platziert, um die Testperformance bezüglich Testzeit, Testabdeckung und/oder Testzugriff zu verbessern, sowie die Testkosten von externem Test-Equipment zu senken. Eine Eigenschaft dieser Testprozessoren ist die Implementierung von Adaptierungsmechanismen, die zur Anpassung an die Testumgebung benötigt werden.

Es gibt zwei Klassen von Testprozessoren:

- fest-verdrahtete Testprozessoren ohne Befehlssatz
- Testprozessoren mit einem Befehlssatz

Die fest-verdrahteten Test-Prozessoren [43] [44] [45] [46] bestehen aus Patterngenerierungs- und Signaturanalyse-Schaltungen, die zum Beispiel mittels LFSRs (Linear Feedback Schieberegister) implementiert sind. Die Testpattern werden parallel an die Prüflingseingänge angelegt und seriell mittels Scan zu den Prüflingsregistern geschoben. In diesem Fall wird die Anpassung an die Testumgebung und Testanforderungen mittels programmierbaren Registern und einer skalierbaren Architektur erreicht.

Die Testprozessoren mit einem Befehlssatz sind standardisierte RISC Prozessoren mit speziellen konfigurierbaren Testfunktionen. So wurde in [47] ein Testprozessor für das Testen von System on Chips (SoC) entwickelt. Er unterstützt LFSRs, die als Teil des Registerfiles zur Patterngenerierung und -analyse implementiert sind, sowie eine spezialisierte Ein-/Ausgabe Testschnittstelle mit parallelen und seriellen Ports, um die Testvektoren anzulegen. In [48] und [49] werden verschiedene Versionen des Testprozessors zur Anpassung an die Testumgebung und Testanforderungen dargestellt. Die Versionen besitzen verschiedene Implementierungsmerkmale bezüglich Registeranzahl, Pipeline-Struktur, On-line- und Selbst-Testfunktionen und unterstützter arithmetischer Operationen. In [50] wurde der Testprozessor für einen speziellen Testfall angepasst, um das Testen von asynchronen Schnittstellen zu ermöglichen. In diesem Fall wurde eine spezielle Ein-/Ausgabe Schnittstelle als Teil des Testprozessors entwickelt, die sich für verschiedene asynchrone Handshake-Protokolle konfigurieren lässt.

Keiner der hier beschriebenen Ansätze ermöglicht jedoch eine *automatische* Adaption des Testprozessors basierend auf Modellen der zu testenden Leiterplatte bzw. Modellen der sich auf der Leiterplatte befindlichen DUTs.

Einen FPGA parallel zur eigentlichen Schaltung auf der Leiterplatte zu platzieren, um darin einen dedizierten Testprozessor zu implementieren, führt neben zusätzlichen Kosten bei der Leiterplattenherstellung zu weiteren Problemen. Während eine parallele Anbindung eines Testprozessors innerhalb von SoC mit etwas Aufwand in den meisten Fällen noch möglich ist [48] [49], so erfordert dies bei Leiterplatten unter Umständen einen erheblich erhöhten Layoutaufwand, der insbesondere bei Verbindungen mit kritischen zeitlichen Randbedingungen wie z.B. bei DRAM oder SerDes zu einer extremen Kostensteigerung führen kann. Teilweise wird der Einsatz des Verfahrens sogar ganz verhindert, da physikalische Leitungseffekte, die durch das parallele Verbinden des Testprozessors entstehen, unweigerlich die Signalqualität negativ beeinflussen und somit die eigentliche Funktion der Leiterplatte stören. Als Konsequenz hat ein dedizierter Testprozessor in praktischen Anwendungen nicht auf alle Verbindungen der Leiterplatte Zugriff und weist somit eine geringere Testabdeckung auf, als solche Verfahren, die auf alle Verbindungen zugreifen können.

3.2.2. PBIST / PBIAT

Die Methode der Programmable Built-In Self Test beziehungsweise Programmable Built-In Assisted Test wird in [51] vorgestellt. Es wird beschrieben, welche Vorteile es bietet, einen FPGA für Testzwecke zu verwenden, um damit Testfunktionalität in das zu testende System selbst zu übertragen.

Es wird zwischen dem BIST und BIAT unterschieden. Man spricht von BIST, wenn nicht nur das Generieren von Testvektoren, sondern auch die Analyse vollständig im Schaltkreis realisiert sind. Als BIAT wird es bezeichnet, wenn der FPGA nur unterstützend zu anderen Testmethoden sein soll und so zum Beispiel nur bestimmte Testvektoren generiert, diese aber nicht auswertet. BIAT kann somit den Testprozess vereinfachen [51].

Ein wichtiges Fazit in [51] ist jedoch die Aussage, dass es sehr aufwendig ist, den Code für den FPGA zu schreiben, so dass dieser in gewünschter Weise das Testen ausführen bzw. unterstützen kann. Insbesondere wenn die gleiche Testfunktionalität an unterschiedliche Leiterplatten anzupassen ist, fehlen geeignete Methoden, die diesen Aufwand reduzieren oder gar ganz vermeiden. Somit ist zwar der Ansatz gut, Testfunktionalität in den FPGA zu übertragen, jedoch ist die Adaption an den Prüfling sehr aufwendig. Eine Weiterentwicklung des Ansatzes aus [51] wird in Kapitel 3.2.3 vorgestellt [52].

3.2.3. EINGEBETTETE TESTINSTRUMENTE

Bei eingebetteten Testinstrumenten handelt es sich um integrierte, also eingebettete Funktionalität im weitesten Sinne, hier Instrumente genannt. Bei diesen Instrumenten handelt es sich im Rahmen dieser Arbeit immer um Instrumente für das Testen. Daher ist der Begriff „*Eingebettetes Instrument*“ immer mit „*Eingebettetes Testinstrument*“ gleichzusetzen.

Die Entwicklung, immer mehr Funktionen einzubetten und dann extern z.B. über JTAG zu steuern, zeichnet sich auch bei der Weiterentwicklung des Boundary-Scan-Standards ab. Dieser wird aktuell für den optimierten Zugriff auf eingebettete Test-instrumente unter dem Begriff IJTAG [53] erweitert und wurde bereits vor der endgültigen Verabschiedung des Standards aktiv genutzt [54]. Da es sich hierbei aber nur um Methoden für den Zugriff auf Test-instrumente und nicht um das Testinstrument selber handelt, wurden diese Verfahren in Kapitel 3.1.1 betrachtet und für die weiteren Betrachtungen in diesem Kapitel ausgeschlossen.

Der Bereich der eingebetteten Testinstrumente beschreibt eine ganze Reihe von teil-weise unterschiedlichen Ansätzen. Eine Gemeinsamkeit aller Ansätze ist es jedoch, eine gewisse Funktionalität des Testens vom Test-PC in Richtung des Prüflings zu verlegen. So beschreibt der Ansatz in [8] ein Testinstrument, bestehend aus einem Prozessor und ggf. unterstützt durch Co-Prozessoren welches im FPGA platziert wird. Diese Tendenz, bestimmte Funktionen in Hardware auszulagern, wird ebenfalls von den Schaltkreisher-stellern als Zukunftstrend, auch im Bereich des Testens, bestätigt [55] [56]. Neben den beschriebenen Prozessoren sind auch andere Verfahren aktuell als state-of-the-art zu betrachten. Diese werden entsprechend der Klassifizierung in den folgenden Unter-kapiteln näher betrachtet und bewertet.

In der Literatur muss zwischen internen und externen Testinstrumente unterschieden werden. Letztere setzen zwar zusätzliche Hardware auf oder neben dem Prüfling voraus, jedoch können die Ansätze dieser Testinstrumente (z.B. [57]) oft auch auf interne über-tragen werden, um so die Vorteile dieser Verfahren zu nutzen, jedoch keine Kosten durch zusätzliche Hardware zu verursachen. Daher werden solche Ansätze in diesem Kapitel ebenfalls betrachtet, um zu prüfen ob mit ihnen die in Kapitel 1.5 definierten Ziele erreicht werden können.

In [58] werden die Vorteile, insbesondere die at-speed Fähigkeit eines FPGA-basierten Testinstruments hervorgehoben. Das Testinstrument befindet sich zwar extern vom Prüfling, sollte aber prinzipiell auch intern funktionieren. [58] geht jedoch nicht auf die Art der Generierung eines solchen Testinstruments ein, weshalb von einer manuellen Generierung ausgegangen werden muss. Dies erschwert die Adaption des Testinstruments auf beliebige Prüflinge.

Eine andere Möglichkeit ein Testinstrument zu generieren wird in [59] beschrieben. Hierbei werden die Stimuli aus einer HDL Simulation extrahiert und dann mit Hilfe eines FPGA-basierten Testinstruments angewendet. Dieser Ansatz setzt jedoch eine

vollständige Simulation des Prüflings voraus. Weiterhin bezieht sich dieser Ansatz auf ein externes Testinstrument und kann nicht einfach übertragen werden, da er entweder große FPGAs voraussetzt oder auf externen Speicher angewiesen ist. Dieser müsste jedoch vorab getestet werden, um seine Funktion zu garantieren. Weiterhin werden bei diesem Verfahren die Testvektoren zwar at-speed angewendet, jedoch müssen alle über eine serielle Schnittstelle zum FPGA übertragen werden. Dies reduziert die Testgeschwindigkeit, da die Übertragung bei einem internen Testinstrument für jeden Prüfling erneut zu erfolgen hat. Somit ist dieses Verfahren für weitere Betrachtungen nicht geeignet.

In [60] wird ebenfalls ein FPGA für die Unterstützung beim Testen von SRAM verwendet. Der Fokus der Veröffentlichung liegt jedoch auf den Testalgorithmen, die ausschließlich auf einem soft-core Prozessor ausgeführt werden. Es werden die berechtigten Vorteile einer solchen Implementierung wie Flexibilität und einfache Adaptierbarkeit an den Prüfling hervorgehoben, jedoch die Nachteile (z.B. kein at-speed) verschwiegen.

Die Autoren von [61] und [62] sowie von [63] beschreiben einen Ansatz zur Generierung von Testinstrumenten für den VLSI Schaltkreistest anhand einer Verhaltensbeschreibung (BBATG - behavior-based automatic test generation). Diese Beschreibung beinhaltet neben einer Funktionsbeschreibung auch die Darstellung von zeitlichen Abhängigkeiten. Die Details im Paper legen jedoch nahe, dass die Funktionsbeschreibung nicht in algorithmischer Form vorliegt, sondern Testvektoren zum FPGA übertragen und dort direkt in Hardware ausgeführt werden. Dies verursacht einen entsprechend Datentransfer über die beschriebene serielle Schnittstelle, der den gesamten Testprozess verlangsamt. Weiterhin wird die aktuelle maximale Taktfrequenz der realisierten Implementierung mit lediglich 6 MHz angegeben. Dies ist jedoch für moderne Schaltkreise und Schnittstellen deutlich zu gering, so dass in diesem Fall trotz der Umsetzung in Hardware nicht von at-speed Test gesprochen werden kann.

Nachfolgend wird eine kurze Definition von unterschiedlichen Begriffen zur Klassifikation von eingebetteten Testinstrumenten gegeben, auf die im weiteren Verlauf der Arbeit Bezug genommen wird. Die Klassifizierung ist von der jeweiligen Sichtweise auf die Testinstrumente abhängig. Diese kann sich auf den Testprozess oder die Testhardware beziehen. Im Rahmen dieser Arbeit wird immer von einer Sicht auf die Hardware ausgegangen.

Es werden drei Arten von eingebetteten Instrumenten betrachtet:

- Virtuelle Testinstruments
- Universelle Testinstruments
- Synthetische Testinstruments

Als „Virtuelle Testinstrumente“ werden Testinstrumente definiert, deren gesamtes Testverhalten ausschließlich auf dem PC realisiert ist. So könnte dies zum Beispiel ein

I^2C Tester sein, der am Test-PC für das Testen von zwei Leitungen verwendet werden soll. Wie die Umsetzung im Prüfling erfolgt, ist hierbei nicht relevant. In der realen Umsetzung könnte dieses Testinstrument über IEEE 1149.1 ausgeführt werden. Ein virtuelles Testinstrument stellt somit eine komfortable Möglichkeit der Testgenerierung dar (z.B. durch einfaches Drag&Drop von Bibliothekselementen), sagt jedoch nichts über die physikalische Realisierung und somit die Leistungsfähigkeit aus.

Als „Virtuelle Testinstrumente“ werden laut [15] und [64] Testinstrumente bezeichnet, die einmal entworfen werden und unabhängig vom DUT oder der Testspezifikation sind. Die Eigenschaft ‚virtuell’ beschreibt jedoch lediglich den Testprozess, nicht aber die Hardware, die für das Testen genutzt wird. Die Hardware eines solchen Testinstruments ist nicht virtuell, sondern real im FPGA vorhanden, lediglich ihre genaue Funktion ist noch nicht definiert und wird erst zu einem späteren Zeitpunkt durch Konfiguration festgelegt.

Da diese Art der Testinstrumente einmal entworfen und dann lediglich konfiguriert werden, sind sie sehr robust und liefern ein definiertes Verhalten, da alle Implementierungen einer identischen Beschreibung folgen, im Gegensatz zu „Synthetischen Testinstrumenten“, die vom Synthesetool unter Umständen bei jeder Synthese im Detail anders realisiert werden. Somit ist insbesondere das zeitliche Verhalten der Testinstrumente in [15] besser vorherzusagen und entsprechende Fehler sind im Vorfeld einfacher zu beheben. Ihr entscheidender Nachteil ist die nicht optimale Anpassbarkeit an den Prüfling.

Diese Testinstrumente sind aus Sicht der Hardware jedoch nicht ‚virtuell‘ sondern eher als ‚universell‘ zu klassifizieren. Dies deckt sich auch weitestgehend mit der Klassifizierung in [65], an der der Autor von [15] ebenfalls beteiligt war und somit seiner eigenen früheren Definition widerspricht. Entsprechend der vorliegenden Klassifikation wird auf „Virtuelle Testinstrumente“ im Verlauf dieser Arbeit nicht weiter gesondert eingegangen, da die Hardwareumsetzung von Testinstrumenten im Fokus dieser Arbeit liegt und nicht der Testprozess. Stattdessen werden die Testinstrumente aus [15] den „Universellen Testinstrumenten“ zugeordnet und zusammen mit den „Synthetischen Testinstrumenten“.

Universelle Testinstrumente

In [52] wird ein „Universelles Testinstrument“ als eines definiert, das neben Mechanismen für die Testausführung, also z.B. das Übertragen von Testvektoren zum DUT auch die eigentliche Testvektorgenerierung integriert hat und somit eigenständig nutzbar ist. In [15] wird dagegen sowohl für „Synthetische Testinstrumente“ als auch „Universelle Testinstrumente“ davon ausgegangen, dass nur die Hardware im FPGA implementiert wird, die eigentliche Teststeuerung aber auf dem Test-PC verbleibt.

Im Rahmen dieser Arbeit erfolgt die Klassifikation unabhängig von der Platzierung der Logik für die Testvektorgenerierung. Diese kann sowohl im FPGA wie auch auf dem Test-PC erfolgen oder auch verteilt auf beidem.

Zu den „Universellen Testinstrumenten“ gehören unteranderem Ansätze, bei denen weiterhin die Boundary-Scan-Zellen sowie ausschließlich Standard Boundary-Scan-Operationen genutzt werden, um die Daten zum FPGA zu übertragen [66]. Statt diese jedoch direkt an den zu prüfenden Schaltkreis anzulegen, werden die Testvektoren einem „Universellen Testinstrument“ zur Verfügung gestellt. Dieses übernimmt dann das eigentliche Anlegen der Daten an das DUT. Hierdurch lässt sich die Übertragung der Testvektoren zum FPGA beschleunigen. Es müssen aber weiterhin alle Daten übertragen werden, da die Testalgorithmen immer noch auf dem Test-PC ausgeführt werden. Weiterhin ist das verwendete Testinstrument nicht speziell an den zu testenden Prüfling adaptiert und ist somit nicht optimal angepasst.

Dieser Grundgedanke wurde in [15] aufgegriffen und führt zu einer Kommunikationsarchitektur, bei der ein Testcontroller über eine so genannte ‚glue logic‘ mit dem TAP Controller des JTAG Interface verbunden wird. Dieser Testcontroller ist abstrakt gehalten und kann prinzipiell beliebige Testaufgaben übernehmen. In [15] wird dieser Testcontroller jedoch nicht individuell für jeden Prüfling adaptiert, sondern ist universell und somit nicht optimal angepasst.

Bei dem Ansatz aus [52] werden die Zugriffsmechanismen auf die DUT Funktionen, sowie grundlegende Analysefunktionen ebenfalls unabhängig von den Testszenarien implementiert. Diese werden manuell erzeugt und müssen bei einer Veränderung des Prüflings angepasst werden, was einen hohen Aufwand und Kosten verursacht.

Die Realisierung von DUTs Zugriffsfunktionen erfordert eine Einhaltung von bestimmten zeitlichen Randbedingungen und Parametern. Dies würde bei „Universellen Testinstrumenten“ für eine optimale Anpassung eine sehr große Anzahl an verschiedenen Testinstrumenten erfordern, um alle Variationen abzudecken oder eine Adaption des zeitlichen Verhaltens zur Laufzeit notwendig machen. Die Adaption eines FPGA-basierten Testinstruments aufgrund zeitlicher Anforderungen wurde bisher in der Literatur nicht näher betrachtet.

In [67] wird die Leistungsfähigkeit von Testinstrumenten am Beispiel der Programmierung von Flashspeichern beschrieben. Obwohl in der Veröffentlichung keine Details zur Umsetzung enthalten sind, wird der Geschwindigkeitsgewinn im Wesentlichen durch drei Punkte begründet:

- Reduzierung der Boundary-Scan Kettenlänge
- partielle Generierung der Testvektoren im FPGA
- Verwendung von Testdatenkomprimierung

Die Ergebnisse zeigen eine Beschleunigung des Programmierverfahrens anhand von vier Beispielen für vermutlich zwei unterschiedliche Speicher, drei unterschiedliche zu programmierende Daten und vier unterschiedliche FPGAs. Als Vergleich wird jeweils die Programmierdauer des Flash über die Boundary-Scan-Kette des FPGAs angegeben. Somit ist zwar eine Beschleunigung des Programmierverfahrens festzustellen, die ge-

nauen Zahlen der Beschleunigung zwischen dem 8-fachen und 72-fachen im Vergleich zu Boundary-Scan sind jedoch von sehr vielen Faktoren abhängig und somit nicht mit anderen Verfahren vergleichbar, sofern diese nicht mit genau den gleichen Ausgangsbedingungen durchgeführt wurden. Insbesondere für den Vergleich zur Programmierzeit über Boundary-Scan erlaubt die Verwendung eines anderen FPGAs jede ermittelte Programmierdauer um den Faktor 10 oder mehr zu verändern (sowohl zum Positiven, wie zum Negativen) und erschwert somit die Vergleichbarkeit der Ergebnisse.

Synthetische Testinstrumente

Laut [15] werden „Synthetische Testinstrumente“ speziell für einen Anwendungsfall bzw. Prüfling entwickelt und bestehen aus einer synthetisierbaren Beschreibung z.B. in VHDL. Der Vorteil ist eine mögliche optimale Adaption an den Prüfling, die jedoch zu Lasten einer aufwendigeren Entwicklung geht.

In [65] werden weitere Vorteile der „Synthetischen Testinstrumente“, sowie der „Universellen Testinstrumente“ genannt. So wird neben den bereits in [67] genannten positiven Ergebnissen auf die Vorteile beim RAM Test und der Möglichkeit der Frequenzbestimmung bei Takten hingewiesen und vielversprechende Ergebnisse präsentiert. Die genauen Werte der erreichten Beschleunigung sind jedoch auch hier wieder stark von der gewählten Hardware abhängig, sprich dem DUT und dem FPGA.

In Rahmen des ELESIS Projekts [68] wurde ebenfalls der Einsatz von eingebetteten synthetischen Testinstrumenten untersucht. Als Lösung wurde eine umfangreiche Bibliothek spezieller IPs für ganz gezielte Testaufgaben entwickelt. Diese liegen teilweise jedoch nur als bereits synthetisierte Netzlisten vor und sind daher nicht adaptierbar und auf speziell vorher definierte Anwendungsfälle zugeschnitten. Lediglich die Verbindungen zum DUT sind variabel, nicht jedoch der interne Aufbau dieser Instrumente.

Allen beschriebenen „Synthetischen Testinstrumenten“ gemein ist, dass diese auf die zu testende Hardware angepasst werden müssen. Somit muss der gesamte Generierungsprozess bei der kleinsten Änderung am Prüfling oder an den Testalgorithmen wiederholt werden. Sind hierfür manuelle Eingriffe notwendig, können schnell hohe Kosten entstehen. Dies würde durch eine automatische Adaption eliminiert werden können. Zwar ist der Syntheseprozess weiterhin notwendig, jedoch ist diese vollständig automatisierbar, so dass lediglich die Zeit für die Generierung eines neuen FPGA-bitfiles notwendig ist. Dieses kann jedoch selbst bei aufwendigen Testsystemen in deutlich weniger als einer Stunde erfolgen. Es ist ein einmaliger Prozess für das Erzeugen des Testsystems und hat keinen Einfluss auf die Anwendung und somit die Testdauer.

3.3. MODELLIERUNGSSPRACHEN

Eine zentrale Forderung der in Kapitel 1.5 definierten Ziele ist die Adaption an den Prüfling. Damit dies realisiert werden kann, ist es notwendig, die Bestandteile des

Prüflings so zu beschreiben, dass daraus später automatisch ein Testinstrument generiert werden kann. Dieser Vorgang wird als Modellierung bezeichnet. Im Idealfall kann für ein vorgegebenes Modellierungsproblem eine bestehende Modellierungssprache genutzt werden, die im vollen Umfang die benötigten Eigenschaften zur Modellierung bietet und trotzdem nicht zu komplex ist. Ist das nicht möglich, kann entweder eine bestehende Sprache erweitert werden oder bei komplexen Sprachen ist die Definition von Einschränkungen notwendig, um sie geeignet einsetzen zu können. Alternativ kann eine neue Sprache entwickelt werden, die nur genau die Elemente enthält, die für eine Modellierung notwendig sind.

Im Rahmen dieser Arbeit werden Modellierungssprachen aus drei unterschiedlichen Domainen betrachtet:

- standard Sprachen aus dem Bereich des strukturellen Leiterplattentests
- „Design und Specification“ Sprachen für Hardware- und Automatenmodellierung
- spezielle Sprachen von Testtool-Herstellern

In der ersten Rubrik der Standard Testsprachen sind alle Sprachen zusammengefasst, die sich entweder im Laufe der letzten Jahrzehnte beim strukturellen Testen bewährt haben oder die neu entwickelt worden sind, um aktuellen Trends gerecht zu werden.

Zu diesen Sprachen gehören:

- BSDL – Boundary-Scan Description Language
- SVF– Serial Vector Format
- STAPL – Standard Test and Programming Language
- ICL – Instrument Connectivity Language
- PDL – Procedure Description Language
- STIL – Standard Test Interface Language
- CTL – Core Test Language

In der zweiten Rubrik sind die Sprachen zusammengefasst, die sich für eine Modellierung von Hardware oder Automaten bewährt haben.

Hierzu gehören:

- VHDL (IEEE 1076) / Verilog (IEEE 1364)
- SystemC (IEEE 1666)
- PSL – Property Specification Language (IEEE 1850)
- TSC – Timed Statecharts
- Erweiterte Petrizetze
- SDL-RT – Specification and Description Language – Real Time
- UML – Unified Modeling Language

In der letzten Rubrik wird die Sprache des Testtoolherstellers GÖPEL electronic untersucht. GÖPEL electronic gehört zu den weltweit führenden Anbietern von

Testtools im Bereich der strukturellen Leiterplattentests. Diese Sprache ist speziell für die bisherigen Anforderungen an den strukturellen Test von Verbindungen auf Leiterplatten entwickelt worden.

Diese Sprache ist:

- CASLAN – CASCON Language

In der zu verwendenden Modellierung muss die Sicht auf das System nach unterschiedlichen Gesichtspunkten möglich sein. Zum einen ist eine abstrakte Sicht bezüglich der späteren Umsetzung gefordert. Der Testingenieur soll sich nicht damit befassen müssen, wie die Testalgorithmen später implementiert werden sollen.

In einer funktionalen Sicht muss es möglich sein, die unterschiedlichsten Testalgorithmen in einer angemessenen Art und Weise zu beschreiben. Die Testalgorithmen, die für einen SRAM Test verwendet werden sollen und dem Vergleich der Modellierungssprachen zugrunde liegen, wurden im Kapitel 2.6.1 vorgestellt. Weiterhin ist es nötig, die zeitlichen Zusammenhänge der Signale zwischen FPGA und DUT hinreichend genau zu beschreiben.

Es muss außerdem möglich sein, das System aus struktureller Sicht zu betrachten. Dies ist für das Verbinden der DUTs mit dem FPGA nötig und um spezielle Anforderungen der Verbindungen zu modellieren. Hierzu gehört neben der Vorgabe der nötigen Treiberstärken auch die Wahl bestimmter I/O-Standards.

Im Folgenden werden die Anforderungen, die an die Modellierungssprachen gestellt werden, erläutert. Diese werden bei den Betrachtungen in diesem Kapitel berücksichtigt und in der Bewertung der Modellierungssprachen in Kapitel 4.2.2 erneut aufgegriffen:

- um ein einfaches Modellieren, sowie das Erstellen und die Weitergabe von Modellen zu vereinfachen, ist eine **Text-basierte** Modellierung gefordert
- die Modellierung soll **einfach zu lesen** und **einfach zu schreiben** sein. Dies gilt sowohl für die zeitlichen Randbedingungen, die Ansteuerung der DUTs sowie für die Beschreibung der Testalgorithmen. Es bezieht sich sowohl auf die Darstellungsform als auch auf das nötige Fachwissen, um korrekten Code in der Modellierungssprache zu schreiben
- die Modellierung muss so beschaffen sein, dass sie bzw. ein aus ihr generiertes Ergebnis **direkt für die Synthesetools der FPGA Hersteller nutzbar** ist
- die Pinbelegung des Interfaces eines DUTs muss beschreibbar sein. Hierzu gehören alle nötigen Informationen eines DUTs, wie Namen der Pins oder Anschlüsse im Gehäuse
- die **elektrischen Interface Parameter** eines DUTs müssen beschrieben werden, damit der FPGA später korrekt darauf zugreifen kann
- es muss möglich sein, **Zuweisungen und Vergleiche** für Testvektoren durchzuführen

- zum Generieren einfacher Testvektoren sind **Grundbefehle** wie Zählen, Schieben, Addieren und Subtrahieren notwendig
- für eine einfache Ablaufsteuerung sind **Verzweigungen und Schleifen** vorgesehen
- für eine gute Strukturierung im Code ist ein **Hierarchisches / Prozedurales Modellieren** gefordert
- es soll sowohl **taktgesteuertes als auch kontinuierliches Zeitverhalten** unterstützt werden
- Ziel ist es, bei der **Modellierung den gesamten Lösungsraum** der zeitlichen Randbedingungen abzudecken

3.3.1.1. BSDL – BOUNDARY-SCAN DESCRIPTION LANGUAGE

Die Boundary-Scan Description Language wurde zum ersten Mal als Standard in den IEEE 1149.1-1994 „Supplement to IEEE Std. 1149.1-1990, IEEE standard test access port and boundary-scan architecture“ eingeführt. [13]. Sie dient der Beschreibung der Boundary-Scan Funktionalität von JTAG kompatiblen elektronischen Bauteilen und basiert auf einem Subset von VHDL. Die Beschreibungen werden zumeist von den Schaltkreisherstellern angeboten und in Form einer lesbaren Textdatei bereitgestellt.

Ein gekürztes und kommentiertes Beispiel einer solchen Datei ist im Anhang H zu finden. Dieses basiert auf einer Beschreibung des in dieser Arbeit zugrunde gelegten Xilinx FPGAs vom Typ XC3S50-4 FTG256C. Die ausführliche BSDL Beschreibung ist in [69] zu finden.

Die BSDL Dateien werden von den Testtools dazu benutzt, Testprogramme für IC- und Leiterplattentests zu generieren. Sie enthalten Informationen über den Aufbau der Boundary-Scan-Kette, über die einzelnen Zellen dieser Kette und die unterstützten Befehle. Für jeden JTAG-kompatiblen IC muss eine BSDL Beschreibung vorhanden sein, welche die Boundary-Scan-Eigenschaften des ICs beschreibt und die Funktionalität extern nutzbar macht.

BSDL wurde im Laufe der Jahre durch viele nutzerseitige Erweiterungen ergänzt. Diese unterliegen jedoch keinem Standard und führen somit zu teilweise unterschiedlichen BSDL Dateien. Dies ist einer der Gründe, der zur Entwicklung des neuen Standards IJTAG und den mit diesem Standard verbundenen zwei neuen Sprachen ICL (siehe Kapitel 3.3.1.3) und PDL (siehe Kapitel 3.3.1.4) führte.

3.3.1.2. SVF / STAPL

Das Serial Vector Format (SVF) [70] dient der Beschreibung von JTAG Operationen. Es beschreibt dabei nicht explizit jeden einzelnen Zustand des IEEE 1149.1 Zustandsdiagramms, jedoch Bewegungsabläufe durch dieses Diagramm. Dies wird genutzt, um Datenübertragungen über JTAG zu beschreiben.

Die „Standard Test and Programming Language“ (STAPL) [71] ist ähnlich wie SVF und dient ebenfalls zur Beschreibung der Datenübertragung über JTAG, ist im Gegensatz zu SVF jedoch flexibler. Während in SVF lediglich Befehle der Reihe nach abgearbeitet werden können, entspricht STAPL eher einer prozeduralen Programmiersprache und kann sowohl interpretiert als auch kompiliert ausgeführt werden.

Es ist üblich, dass beide Sprachen auf die in BSDL beschriebene Hardware zugreifen und darauf aufbauen. Somit entfällt die Notwendigkeit, diese in SVF / STAPL erneut modellieren zu müssen.

3.3.1.3. ICL – INSTRUMENT CONNECTIVITY LANGUAGE

Die Instrument Connectivity Language (ICL) wurde zusammen mit der Procedure Description Language (PDL – siehe Kapitel 3.3.1.4) als Teil des IEEE 1687 (IJTAG) entwickelt. Zusammen sollen sie die Wiederverwendung von IP Testbeschreibungen[24] erlauben, unabhängig vom Einsatzort des IPs. Sie dienen der standardisierten und ausreichenden Beschreibung von IJTAG Netzwerken, Testinstrumenten und deren Zugriffsfunktionen. Während ICL die Hardware beschreibt, die dem Testinstrument zugrunde liegt, dient PDL zur Ansteuerung dieser Hardware.

Zusammen können ICL und PDL als ‚Nachfolgepaar‘ zu BSDL und SVF / STAPL angesehen werden, ohne dass mit der Einführung von ICL alle BSDL Beschreibungen automatisch obsolet geworden sind. Im Gegensatz zu BSDL eignet sich ICL zur Beschreibung von dynamischen Registern und zur Strukturierung großer Designs, in dem Teile der Boundary-Scan-Kette umgangen bzw. übersprungen werden können. Es ermöglicht die Beschreibung von hierarchischen Strukturen bei der Verwendung von IJTAG und erlaubt eine einfachere Betrachtung für den Testentwickler, der die tieferen Strukturen und die Funktionsweise von Testinstrumenten nicht kennen muss. Zwar sind ICL und PDL Beschreibungen größer, was die Codegröße betrifft, jedoch wesentlich leichter wartbar und leichter anzuwenden.

3.3.1.4. PDL – PROCEDURE DESCRIPTION LANGUAGE

Die Procedure Description Language (PDL) ist neben ICL die zweite Sprache des IEEE 1687 Standards (IJTAG). Sie ist eine Erweiterung der Tool Command Language (TCL) und weist den zuvor in ICL definierten Strukturen eine Funktion zu. PDL beschreibt, wie in ICL definierte Elemente untereinander interagieren. Dies ist auf unterschiedlichen Hierarchieebenen wie Block-, Core- oder Top-Level möglich. Mit Hilfe von Electronic Design Automation (EDA) Tools können die PDL Beschreibungen

[24] IP bezeichnet den Begriff ‚Intellectual Property‘ und beschreibt einen Funktionsblock, der von Dritten entworfen worden ist und ohne genaue Kenntnisse der internen Umsetzung genutzt werden kann.

von den untersten Ebenen automatisch auf höhere Ebenen übersetzt werden. Dort ist dann eine Beschreibung mittels gebräuchlicher Testsprachen wie zum Beispiel STIL möglich [72].

Im Gegensatz zu SVF, welches für die Durchführung von Tests, z.B. für den Vergleich von Daten, eine höhere Programmiersprache benötigt, kann dies mit PDL direkt beschrieben werden. So ist es möglich, mit PDL und ICL selbst komplexe Testinstrumente zu beschreiben und anzusteuern.

Ein Beispiel, welches ein einfaches Testinstrument realisiert und die Unterschiede zwischen den Sprachpaaren BSDL und SVF sowie ICL und PDL anschaulich darstellt, ist in [73] zu finden. Besonders die Wiederverwendung von Ansteuerungen für Testinstrumente wird darin behandelt. Die Erweiterbarkeit eines Designs und die entstehenden Schwierigkeiten werden vorgestellt. Hieraus werden die beschriebenen Vorteile von ICL und PDL sichtbar. Dieses Beispiel ist in Anhang I in Auszügen kommentiert dargestellt.

In PDL werden unterschiedliche Ausprägungen, sogenannten Flavors unterstützt. Während Flavor-0 lediglich Setup- und Aktionskommandos unterstützt, können mit Flavor-1 bereits Schleifen und Verzweigungen realisiert werden. Folglich ist die Umsetzung von Test- und Debugszenarien hiermit bereits möglich.

3.3.1.5. STIL – STANDARD TEST INTERFACE LANGUAGE

Die Standard Test Interface Language (STIL) wird in IEEE 1450.0 beschrieben und ist ursprünglich eine Testsprache, mit der Testmuster und Signale beschrieben werden. Sie wurde 1999 erstmals eingeführt und ist seitdem durch unterschiedliche Erweiterungen von IEEE 1450.1 bis 1450.8 ergänzt worden [74].

Die wesentlichen Elemente der Sprache sind Signale, Muster, Makros und Funktionstabellen. Ein einfaches mittels STIL beschriebenes Testmuster (nach[74]) ist im Folgenden dargestellt: $W\ t1; V\ \{CLK = K; DATA = AA55\}$. Hierbei entspricht $t1$ dem Namen der Waveform (W) und im zu bearbeitenden Testvector (V) wird K als Taktsignal referenziert und $AA55$ entspricht den anzulegenden Daten.

STIL ermöglicht auch das Beschreiben von kontinuierlichen Zeiten, wie dies in [75] anschaulich dargestellt ist. Neben einfachen Zuweisungen ist es somit mit STIL auch möglich, Ereignissen beliebige zeitliche Abhängigkeiten zuzuweisen. Entgegen der bisher betrachteten Bechreibungsmöglichkeiten können sich diese in STIL auch relativ auf andere Ereignisse beziehen. Weiterhin unterstützt STIL modulares Arbeiten, bei dem einmal entworfene Komponenten wiederverwendet werden können. STIL stellt damit eine sehr leistungsfähige Modellierungssprache dar.

Während der ursprüngliche Standard 1450.0 im wesentlichen Signale, Testvektoren sowie zeitliche Parameter beschrieben hat, so ist es das Ziel von 1450.4, auch Testabläufe entwickeln zu können. Bisher ist dieser Standard jedoch noch nicht

verabschiedet worden [76] und befindet sich noch in der Entwicklung. Darauf aufbauend ist ein weiterer Standard 1450.5 geplant, der zur Spezifikation von Testmethoden geeignet sein soll. Die Entwicklung an diesem Standard wird jedoch erst nach dem Verabschieden von 1450.4 begonnen. Somit stehen diese beiden vielversprechenden Ansätze nicht für den Einsatz im Rahmen dieser Arbeit zur Verfügung. Zum aktuellen Zeitpunkt ist somit eine algorithmische Beschreibung von Testprogrammen in STIL nicht möglich.

3.3.1.6. CTL – CORE TEST LANGUAGE

Die Core Test Language (CTL) ist mit IEEE 1450.6 eingeführt worden. Sie ist die offizielle Beschreibungssprache für den Standard Embedded Core Test gemäß IEEE 1500.

CTL dient dazu Schaltkreise mit eingebetteten Prozessoren und deren Peripherie zu testen. Dabei wird ein Prozessor isoliert und mit einem CTL Interface umschlossen. Auf dieses kann dann standardisiert zugegriffen werden. Grundlage für CTL bildet die Standard Interface Language (STIL), deren Syntax teilweise erweitert wurde.

Die Kernelemente der CTL wie „Design Configuration Information“, „Structural Information“ und „Test Pattern Information“ werden, wie auch bei BSDL, vom Schaltkreishersteller mitgeliefert und erlauben es dem Systemintegrator, mit diesen Informationen einen Prozessorkern-bezogenen Test an eingebetteten Systemen mit Prozessoren durchzuführen.

Da es CTL auch erlaubt, IEEE 1149.1 (JTAG) Hardware zu beschreiben [77], ist es möglich, auf einem Schaltkreis unterschiedliche Testmechanismen zu vereinen und den in IEEE 1149.1 enthaltenen Test Access Port auch über die Eingabeschnittstelle von IEEE 1500 anzusteuern.

Laut [73] kann CTL als Sprache angesehen werden, die sowohl das Interface zu Testinstrumenten beschreibt als auch die Testvektoren bzw. die Zugriffsroutinen für den Core. Ein Beispiel für einen Speichertest ist in [78] dargestellt.

3.3.1.7. VHDL / VERILOG

VHDL und Verilog erlauben neben einer Beschreibung von Hardwarestrukturen auch die Beschrebeibung von zeitlichen Abläufen. Diese werden zum Beispiel in Simulationen genutzt, um die korrekte Funktion einer Schaltung zu prüfen. So bieten zum Beispiel viele Speicherhersteller entsprechende Modelle ihrer Schaltkreise in VHDL / Verilog an, die somit nicht nur eine funktionale sondern auch genaue zeitliche Kontrolle der eigenen Ansteuerung ermöglichen. In Kapitel 4.2.5.1 sind Teile eines entsprechenden Verilog Modells zu finden.

Während zur Kontrolle von zeitlichen Abläufen Aussagen wie

$$if\ \left((tm_{tdqss} < tck_{avg}/2.0)\ \&\&\ \left(tm_{tdqss} > TDQSS * tck_{avg}\right)\right) \qquad \text{(nach [79])}$$

möglich sind, um das Verhalten eines Signals innerhalb einer bestimmten Zeitspanne zu prüfen, so sind direkt synthesefähige Anweisungen nur für Zeitpunkte zulässig, für die es eine steigende oder fallende Taktflanke gibt. Kontinuierliche Zeiten können in VHDL / Verilog zwar auch definiert werden, diese sind jedoch nicht direkt synthesefähig und nur für Simulationen zulässig.

Werden durch ein DUT bestimmte Werte für minimale und/oder maximale Zeitparameter vorgegeben, so muss der Testingenieur bereits bei der Modellerstellung den später zu verwendenden Takt kennen, um eine definierte Ansteuerung zu entwerfen. Eine andere Wahl des Taktes würde eine Adaption des DUT Modell nötig machen. Der Testingenieur kann also mit jedem Modell, das er generiert, nur eine Lösung und nicht den gesamten Lösungsraum für eine DUT Ansteuerung modellieren.

Komplexe Modelle wie z.B. für DDR2 werden in VHDL schnell unübersichtlich, wenn sie hardwarenah beschrieben werden. So ist in [80] eine Vielzahl an Modellen vorhanden, die einen Einblick in die Komplexität solcher Beschreibungen geben. Weiterhin beinhalten viele der in VHDL oder Verilog vorliegenden Modelle auch die interne Funktionsweise von Schaltkreisen. Dies ist für das strukturelle Testen jedoch oft nicht relevant, da nur die Funktion der Schnittstelle getestet werden soll. Daher sind die Modelle in VHDL / Verilog oft viel zu komplex, um sie für die automatische Generierung von Testinstrumenten nutzen zu können. Insbesondere quelloffene Modelle können für den professionellen Einsatz in Testsystemen nicht direkt verwendet werden, da ihre Qualität nicht bekannt ist und unklar ist, ob diese fehlerfrei sind. Die Modelle müssten vor einem Einsatz einzeln überprüft werden. Dies ist bei komplexen Modellen sehr zeitaufwendig. Somit kommt der Vorteil einer bereits bestehenden großen Datenbasis bei der Wahl von VHDL / Verilog als Modellierungssprache für DUTs nicht zum Tragen.

3.3.1.8. SYSTEMC

SystemC ist im Gegensatz zu VHDL / Verilog für den Entwurf von Software- und Hardwarekomponenten entwickelt worden. Es stellt keine eigene Sprache dar, sondern ist eine Erweiterung von C++. Sie erweitert die Sprache mittels Makros und Funktionen um die notwendigen Mittel, die es erlaubt, hardwarespezifische Besonderheiten wie Parallelität und Synchronisation zu beschreiben.

3.3.1.9. PSL – PROPERTY SPECIFICATION LANGUAGE

Die Property Specification Language (PSL) ist als IEEE 1850 Standard definiert und wird zur formalen Beschreibung von Spezifikationen elektronischer Systeme eingesetzt.

PSL ist kompatibel zu unterschiedlichen Hardwarebeschreibungssprachen wie unter anderem VHDL, Verilog oder SystemC und ermöglicht eine einheitliche Spezifikation, Simulation und Verifizierung [81]. Auch wenn PSL ursprünglich zur Verifikation von Eigenschaften entwickelt worden ist, so können die mittels PSL definierten Zusammenhänge auch für eine Schaltungssynthese genutzt werden. In [82] wird ein entsprechender Ansatz vorgestellt, beim dem das Verhalten eines Bus-Arbiters mit PSL spezifiziert wird. Hierbei wird deutlich, dass zeitliche Zusammenhänge einem festen Zeitraster folgen. Ein Beispiel für eine solche Spezifikation ist in Abbildung 14 dargestellt. Hierbei wird definiert, dass, *wenn kein Zugriff (START) im nächsten Zeitschritt gestartet wird, wird der Bus (HMASTER) nicht zugewiesen und HMASTLOCK wird nicht verändert*.

$$\begin{aligned}&\forall i : \texttt{always}\,(\texttt{next!}\ \neg\mathrm{START} \rightarrow \\ &\quad((\mathrm{HMASTER} = i \leftrightarrow \texttt{next!}\ \mathrm{HMASTER} = i) \wedge \\ &\quad(\mathrm{HMASTLOCK} \leftrightarrow \texttt{next!}\ \mathrm{HMASTLOCK})))\end{aligned}$$

Abbildung 14: Beispiel einer in PSL spezifizierten Eigenschaft

3.3.1.10. TIMED STATE CHARTS

Bei State Charts handelt es sich um eine Erweiterung der klassischen Zustandsdiagramme. Mit ihnen ist es möglich, sowohl Parallelität als auch Hierarchie einfacher abzubilden, als dies sonst möglich wäre.

Typischerweise werden State Charts zur *grafischen* Darstellung von Zustandsdiagrammen verwendet. Es gibt jedoch auch die Möglichkeit, diese textuell zu beschreiben. In [83] wird eine entsprechende Möglichkeit für zeitliche (Timed State Charts) und hybride (Hybrid State Charts) Zustandsmaschinen vorgestellt. Ein Auszug ist in Abbildung 15 und Abbildung 16 dargestellt. Für das vollständige Beispiel sei auf [83] verwiesen.

Im vorliegenden Beispiel haben alle drei Transistionen kein Triggerevent, sondern nur zeitliche Abhängigkeiten, die durch eine untere (l=lower) und obere (u=upper) Schranke beschrieben werden. Die Transition *good-part* führt als einzige eine Aktion (put) aus, die der Transition zugeordnet ist.

In der textuelle Abbildung des Timed State Charts sind die Zustände *Idle* und *Busy*, sowie die drei Transitionen *set-up*, *bad-part* und *good-part* und deren zeitliche Abhängigkeiten zu erkennen. Trotzdem ist die Darstellung nicht intuitiv und erschwert das direkte Formulieren von zeitlichen Abhängigkeiten in textueller Form. Somit wird deutlich, dass selbst in einfachen Fällen zuerst eine graphische Darstellung des State Charts erstellt werden sollte. Dieses erfordert einen aufwendigen Generierungsprozess, insbesondere wenn die Generierung lediglich auf vorgegebenen Signalwechsel aufbaut, wie diese zum Beispiel in Abbildung 22 dargestellt worden sind.

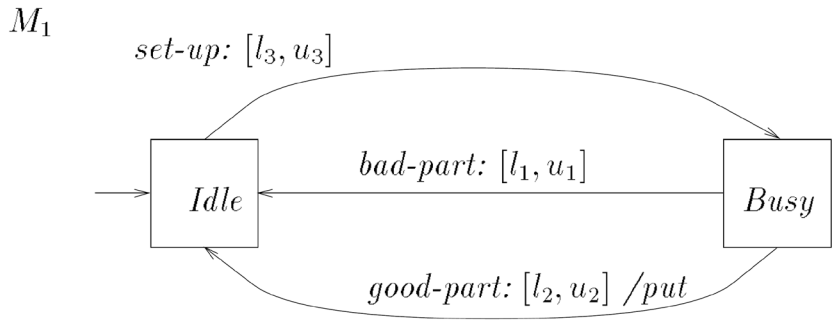


Abbildung 15: Ausschnitt eines Timed State Charts [83]

$$
M_1 :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{ll}
Idle: & \textbf{skip} \\
\text{set-up}: & \textbf{delay } [l_3, u_3] \\
Busy: & \left[\begin{array}{llll}
\textbf{skip}; & bad\text{-}part: & \textbf{delay } [l_1, u_1] & \\
 & & \sqcup & \\
\textbf{skip}; & good\text{-}part: & \textbf{delay } [l_2, u_2]; & put!
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Abbildung 16: Textuelle Beschreibung eines Timed State Charts Ausschnitts [83]

3.3.1.11. ERWEITERTE PETRI-NETZE

Petri-Netze können dazu genutzt werden, dynamische Systeme mit nebenläufigen Vorgängen zu beschreiben, zu analysieren und zu simulieren. Im Laufe der Jahre wurden Petri-Netze immer wieder um neue Semantiken erweitert. Man spricht dann allgemein von erweiterten Petri-Netzen. Eine dieser Neuerungen der erweiterten Petri-Netze sind Transitionen mit Zeitverzögerungsfunktionen. Nachdem eine solche Transition schaltfähig geworden ist, schaltet sie nicht sofort, sondern erst nach der festgelegten Verzögerungszeit. Für diese Zeitverzögerung ist es auch möglich, ein Intervall anzugeben. So können zeitliche Abhängigkeiten mit minimalen und maximalen Werten modelliert werden.

Petri-Netze werden üblicherweise graphisch dargestellt. Sie sind intuitiv und relativ einfach zu verstehen. Ihnen liegt ein mathematisches Modell zugrunde, weswegen sie auch ganz formal und rein textuell dargestellt werden können.

3.3.1.12. SDL-RT

SDL-RT [84] basiert auf dem Specification and Description Language (SDL) Standard der ITU-T[25] und erweitert diesen um Konzepte zur Beschreibung von Echtzeitsystemen. Mit dieser Erweiterung ist es in der aktuellsten Version 2.3 möglich,

[25] ITU-T: International Telecommunication Union – Telecommunication Standardization Sector

graphische Repräsentationen von zeitlichen Abhängigkeiten zwischen Ereignissen zu beschreiben. Als zeitliche Bedingungen kommen dabei sowohl absolute als auch relative Zeiten bzw. Zeitintervalle in Frage.

Während SDL (ohne RT) nur wenig Möglichkeiten bietet, Zeiten zu beschreiben, wurde die Erweiterung RT für die Modellierung von Systemen entwickelt, deren korrektes Verhalten nicht nur von der korrekten Antwort abhängen, sondern auch davon, wann diese Antworten produziert werden [85]. In [86] werden drei Ansätze beschrieben, wie SDL durch RT erweitert werden kann.

3.3.1.13. UML 2

UML 2 ist die zweite Version der Unified Modeling Language und soll eine möglichst universelle Modellierung erlauben. Die Sprache ist sehr vielseitig und deckt die unterschiedlichsten Modellierungsdomainen und -anforderungen ab. Mit Hilfe von Timing-Diagrammen und Sequenzdiagrammen kann das Zeitverhalten von Objekten exakt beschrieben werden. Diese Form der Darstellung ist hauptsächlich auf die graphische Beschreibung von zeitlichen Abläufen ausgerichtet.

3.3.1.14. CASLAN

Die Sprache CASLAN ist eine von der Firma GÖPEL electronic [87] genutzte Testsprache des Testsystems CASCON GALAXY [88]. Sie wurde entwickelt, um Boundary-Scan-basierte Tests zu beschreiben und zu steuern. Die Sprache erlaubt es, JTAG Befehle auszuführen und eigene Prozeduren zu definieren. Einfache Abfragen werden unterstützt, jedoch keine Schleifen oder mathematische Operationen. Weiterhin ist eine Beschreibung zeitlicher Abhängigkeiten nicht vorgesehen.

3.4. ZUSAMMENFASSUNG

Aktuell ist ein Paradigmenwechsel zu beobachten [89], bei dem immer mehr Test-funktionen in den Prüfling integriert werden. Hierbei steuert z.B. der FPGA selbst Testfunktionen und testet direkt mit ihm verbundene Schaltkreise. Mit diesen Verfahren sind teilweise beeindruckende Beschleunigungsfaktoren im Vergleich zu Boundary-Scan zu erzielen [65] [67].

Bei allen hier vorgestellten Ansätzen des prozessor-basierten Testens, PBIST und PBIAT, sowie den Eingebetteten Testinstruments wird eine gewisse Testfunktionalität in den Prüfling übertragen. Jedoch ist die größte Schwachstelle die optimale Adaption an den Prüfling, die entweder nicht erfolgt oder sehr aufwendig ist. Eine Zusammenfassung der untersuchten Testverfahren und ihre Eignung bezüglich der definierten Ziele ist in Tabelle 2 dargestellt.

Tabelle 2: Zusammenfassung zum Stand der Technik aktueller Testverfahren

Ziele / Standard	at-speed Test	flexible Test-generierung	großes Einsatz-gebiet	auto-matische Adaption	umfangreicher Testzugriff	schnelle Test-ausführung
1149.X[26]	✗	✓	✓	✗	✗ / ✓[27]	✗ / ✓[28]
BIST	✓	✗	✓	✗	✓	✓
Kombinierte Testverfahren	✗ / ✓	✗ / ✓	✗ / ✓	✓	✓	✓
Testprozessoren[29]	✗ / ✓	✓	✓	✗ / ✓	✗ / ✓	✓
Universelle Testinstrumente	✗ / ✓	✓	✓	✗	✓	✗ / ✓
Synthetisch Testinstrumente	✓	✓	✓	✓	✓	✓

✗: Wird nicht unterstützt ✓: Wird unterstützt ✗ / ✓: Mit Einschränkungen

Es zeigt sich, dass Prozessoren nach den untersuchten Kriterien viele der Kriterien für die Testverbesserung erfüllen, sofern sie effizient auf die zu testenden Strukturen zugreifen können. Lediglich at-speed Tests bereiten aufgrund der generell in Software erfolgenden Algorithmenverarbeitung weiterhin Probleme, falls der Zugriff auf die DUTs nicht über spezielle Module hardwaremäßig unterstützt wird. Diese Idee ist besonders bei FPGAs interessant, da es bei diesen prinzipiell möglich ist einen *beliebigen*[30] Prozessor in dessen Ressourcen zu realisieren und diesen Prozessor mit einer entsprechenden Unterstützung für den gewünschten Testzugriff auszustatten.

Bei den eingebetteten Testinstrumenten kann zwischen universellen, im Vorfeld synthetisierten Testinstrumenten und „Synthetischen Testinstrumenten“, die an jeden Prüfling angepasst werden, unterschieden werden. Beide Verfahren zeichnen sich durch bestimmte Vorteile und Nachteile aus. Die vorgefertigten „Universellen Test-instrumente“ benötigen keine Synthese, um an den Prüfling angepasst zu werden. Dies erlaubt es, das Testinstrument im Vorfeld ausgiebig zu testen und die Wahrschein-

[26] Bezieht sich auf die Überlagerung der Teilergebnisse der einzelnen Unterstandards unter Berücksichtigung deren aktueller Verbreitung. Siehe Kapitel 3.1.1.

[27] Einige Unterstandards erweitern zwar den Testzugriff, jedoch sind diese insbesondere bei FPGAs noch nicht weit verbreitet.

[28] Einige Unterstandards beschleunigen zwar das Testen, sie sind jedoch bisher nicht ausreichend weit verbreitet und werden bei den meisten FPGAs nicht unterstützt.

[29] Bezieht sich auf die Überlagerung der Teilergebnisse der unterschiedlichen Testprozessoren.

[30] entsprechend der Möglichkeiten und Ressourcen des genutzten FPGAs

lichkeit für Logik- und Laufzeitfehler innerhalb des Testinstruments zu reduzieren. Für ihren Einsatz müssen „Universelle Testinstrumente“ lediglich programmiert und konfiguriert werden. Mit ihnen sind bereits gute Ergebnisse zu erzielen und die Test- sowie Programmierdauer für einzelne Beispiele zeigen ein durchweg positives Ergebnis.

Aus den Vorteilen der „Universellen Testinstrumente“ ergeben sich jedoch zugleich auch die gravierendsten Nachteile. So werden die Testinstrumente im Vorfeld ohne Wissen über den späteren Prüfling generiert. Somit ist ein solches Testinstrument nie optimal an den Prüfling angepasst. Insbesondere bei Tests mit hochfrequenten Signalen wie zum Beispiel bei dynamischen Speichern, ist dies jedoch besonders wichtig, um möglichst schnell und innerhalb ihrer Spezifikation testen zu können.

„Synthetische Testinstrumente“, die für jeden Prüfling individuell adaptiert werden können, ermöglichen solche Anpassungen. Dies erfordert neben der eigentlichen Synthese des FPGA-Bitfiles, die weitestgehend automatisiert werden kann, jedoch auch gewisse Änderungen an der Beschreibung des Testinstruments. Hierfür ist unter Umständen viel Wissen des Testingenieurs über die Testinstrumente und deren Aufbau notwendig. Weiter benötigt die Umsetzung solcher Änderungen Zeit. Beides verursacht Kosten im Testprozess, die vermieden werden sollten, bzw. für einige Anwendungen nicht akzeptabel sind und den Einsatz von „Synthetischen Testinstrumenten“ verhindern.

Prinzipiell benötigt FPGA-basiertes Testen keine zusätzliche Hardware und ist somit sehr kostengünstig einzusetzen, insbesondere da auf Testpunkte und teures Testequipment (Automated Test Equipment – ATE) verzichtet werden kann. Eine Herausforderung ist jedoch die Generierung eines solchen FPGA-basierten Testinstruments. Dies führt zur Forderung, den Generierungsprozess weitestgehend zu automatisieren und somit die Testkosten zu reduzieren.

Bezogen auf die Zielstellungen aus Kapitel 1.5 zeigen „Synthetische Testinstrumente“ folgende Möglichkeiten:

- Beschleunigung von strukturellen Tests und Realisierung von at-speed Tests, durch die optimale Adaption des Testsystems an den Prüfling
- flexible Testgenerierung durch die Platzierung beliebiger Testalgorithmen innerhalb des FPGAs
- Anwendbarkeit in einem großen Bereich, da das Konzept auf alle bestehenden Leiterplatten mit einem FPGA angewendet werden kann und keine speziellen Voraussetzungen an die Hardware gegeben sind
- automatische Adaption des Testsystems an den Prüfling durch die Automatisierung des gesamten Entwurfsprozesses der Testinstrumente, um auch bei kurzen Design-Zyklen ein optimales Testsystem realisieren zu können
- umfangreicher Testzugriff, da auf alle Pins des FPGAs zugegriffen werden kann und die Testinstrumente beliebige Funktionen im FPGA realisieren können

- einfache Testgenerierung und schnelle Testausführung, falls eine parallele Ausführung verschiedener Testalgorithmen im FPGA und eine Automatisierung der Testsystemgenerierung möglich sind

Der aktuelle Stand der Technik zeigt jedoch, dass genau die automatische Adaption und Generierung bei synthetischen Testinstrumenten bisher nicht akzeptabel gelöst sind. Dies führt im Folgenden zur detaillierten Zielstellung dieser Arbeit, die in Kapitel 3.5 dargestellt ist.

3.5. ZIELSTELLUNG

Im Folgenden wird, aufbauend auf dem Stand der Technik und den dort beschriebenen Eigenschaften der Verfahren, eine Zielstellung abgeleitet. Diese Zielstellung ist die Grundlage für den in Kapitel 4 erarbeiteten Lösungsansatz.

Es soll eine Lösung für den ausschließlichen Einsatz auf FPGAs gefunden werden, da diese Schaltkreise durch ihre Konfigurierbarkeit ein besonders hohes Maß an Flexibilität bieten. Es sollen weiterhin keine speziellen Anforderungen an die zu nutzende Hardware gestellt werden, so dass das Lösungskonzept auf möglichst vielen Prüflingen eingesetzt werden kann, um so ein breites Einsatzspektrum abzudecken.

Das zu entwerfende Testsystem soll ausschließlich auf dem Prüfling bereits vorhandene Ressourcen verwenden. D.h. bei der Entwicklung der Leiterplatte sollen keine zusätzlichen Ressourcen für das Testsystem eingeplant werden müssen. Dies würde die Kosten der Leiterplatte erhöhen und würde in der Praxis zu einer Ablehnung des Testsystems führen.

Das Testsystem soll lediglich temporär in den FPGA geladen werden und nicht parallel zur eigentlichen Funktionalität der Leiterplatte vorhanden sein.

Als zu betrachtende Fehlermodelle werden zuerst ausschließlich stuck-at- und wired-Fehler aus der Menge an Fehlerarten von Kapitel 2.3 betrachtet.

Bei der vorliegenden Arbeit steht nicht die Art der Testmustergenerierung für einzelne Fehlerarten im Vordergrund, sondern die Anwendung dieser Muster at-speed, also mit der späteren Arbeitsgeschwindigkeit des Prüflings. Dies bedeutet, dass die Testmuster so angewendet werden sollen, dass sie auch dynamische Fehler aufdecken, die durch den Einsatz bei Arbeitsgeschwindigkeit entstehen könnten. Hierfür werden meist einzelne Testmuster mit 'voller' Geschwindigkeit angewendet. Laut [90] ist es für at-speed nicht zwingend notwendig, dass *alle* Testmuster mit voller Geschwindigkeit aufeinanderfolgend ausgegeben werden müssen. Während dies zwar Auswirkungen auf die Testdauer und somit die Testkosten hat, hat dies nichts mit at-speed zu tun, sondern ist ein sekundärer Wunsch nach einer schnellen Testausführung. Ziel dieser Arbeit ist es somit, at-speed zu garantieren und, wenn möglich, die *gesamte* Testausführung zu beschleunigen.

Abgeleitet von den aktuellen Problemen struktureller Leiterplattentests und den in Kapitel 3.4 zusammengefassten Möglichkeiten von synthetischen Testinstrumenten und Testprozessoren, die sich aus der Analyse vom aktuellen Stand der Technik ergeben, soll das Lösungskonzept folgende Eigenschaften erfüllen:

- systematische Beschreibung der Schaltkreise (DUTs), zu denen die Verbindung getestet werden soll, unabhängig von deren Platzierung auf der Leiterplatte und somit unabhängig vom späteren Prüfling
- Anpassen des Testsystems an die Umgebung des DUTs und Adaption an die Leiterplatte, die getestet werden soll; hierbei soll die Anpassung automatisch erfolgen
- es sollen die Vorteile der Testausführung in Software z.B. auf einem softcore-basierten Testprozessor innerhalb des FPGAs mit der Testausführung in Hardware, also die in diskreter Logik im FPGA, kombiniert werden
- automatische Generierung der nötigen Beschreibungen, um das FPGA-Bitfile synthetisieren zu können
- at-speed Testen, sowie schnelle Testausführung des Gesamttests
- geringe Kosten für die Testgenerierung und Testausführung durch weitest-gehende Automatisierung

Diese Arbeit konzentriert sich auf die Modellierung der DUTs und die hardwarenahe Generierung von Testinstrumenten, um die beschriebenen Ziele zu erreichen. Eine detaillierte Zielstellung zu diesen beiden Punkten ist in Kapitel 3.5.1 zu finden.

3.5.1. AUTOMATISCHEN GENERIERUNG VON TESTINSTRUMENTEN

Entscheidend für eine spätere automatische Generierung des Testsystems ist eine ausreichend genaue Modellierung. Man kann hierbei auch von *modell-basierten synthetischen Testinstrumenten* sprechen. Die Modellierung soll folgendes berück-sichtigen:

- vom späteren Prüfling unabhängige Beschreibung aller Testsystemkomponenten; das bedeutet sowohl alle DUTs, wie auch der verwendete FPGA und die Leiter-platte sollen vollständig unabhängig voneinander beschrieben werden
- speziell die Schwächen bisheriger DUT Modellierungen sollen eliminiert werden. Die angestrebten Verbesserungen sind:
    - systematisches Erfassen zeitlicher Abhängigkeiten von DUT Zugriffs-funktionen
    - übersichtliche Strukturierung der Testfunktionalität
    - Beschreibung von Hierarchie in den Funktionen der Testinstrumente
- Ausführung der Algorithmen in Software und Hardware muss basierend auf einer gemeinsamen Beschreibung möglich sein, um wahlweise die Vorteile einer software-basierten oder hardware-basierten Ausführung zu nutzen

- bei der Spezifikation unbekannte Randbedingungen wie z.B. der verwendete Takt des FPGAs müssen berücksichtigt werden

Das zu generierende Testsystem soll auf einem FPGA implementiert werden. Daher ist es nötig, eine entsprechende Beschreibung in Form eines Bitfiles zu erzeugen. Um die recht komplexe Generierung solcher Systeme zu vereinfachen, soll im Rahmen dieser Arbeit eine Methode erarbeitet werden, die es erlaubt, alle für die Synthese nötigen Informationen automatisch zu generieren. Es soll das modellierte Testsystem mit Hilfe von Regeln in eine Hardwarebeschreibungssprache überführt werden. Der nötige Designflow für eine solche Transformation soll im Rahmen dieser Arbeit dargestellt werden.

Alle nötigen Funktionen zum Testen der Verbindungen eines einzelnen DUTs sollen in einem Testinstrument zusammengefasst werden. Dieses Testinstrument soll sowohl in Hardware als auch in Software umgesetzt werden können. Weiterhin müssen die Testinstrumente für mehrere DUTs korrekt zu einem Gesamtsystem zusammengebaut werden, das dann als ein Bitfile in den FPGA programmiert werden kann.

Auch wenn der gesamte Generierungsprozess automatisch erfolgt, so dass der Anwender nur fertige Bibliothekselemente benötigt und sich das Testsystem selbständig anhand des vorgegebenen zu testenden DUTs generiert, sollen dennoch manuelle Änderungen möglich sein. Es ist somit wichtig, dass neben einem gut lesbaren Modell für die DUTs auch das Ergebnis des Generierungsprozesses lesbar und gut strukturiert ist. Dies kann unter anderem hilfreich sein, wenn bestimmte Besonderheiten eines Prüflings (noch) nicht im Modell berücksichtigt werden können und daher doch manuelle Eingriffe am generierten Testsystem gewünscht sind.

Auch wenn der Entwurfsprozess an ausgewählten Beispielen demonstriert wird, soll er allgemeine Gültigkeit erhalten und möglichst vielseitig anwendbar und erweiterbar sein.

4. ANSATZ FÜR EIN TESTSYSTEM

Im folgenden Kapitel wird, aufbauend auf den definierten Zielen aus Kapitel 3.5, der gewählte Lösungsansatz für ein Testsystem erläutert. Dabei wird, ausgehend von der Vorstellung des Konzeptes und der gewählten Architektur in Kapitel 4.1, in den danach folgenden Abschnitten der Entwurfsprozess eines Testsystems von der Modellierung bis zur endgültigen Systemgenerierung vorgestellt. In Kapitel 5 werden die durchgeführten Untersuchungen und die Ergebnisse dargestellt.

Die in dieser Arbeit beschriebenen Ergebnisse sind als Teil der Forschungsprojekte „*ERADOS – Experimentelle Forschung zur adaptiven Fehlerdiagnose basierend auf strukturellen ‚multi-core' Emulationstest*“ und „*ROBSY – Rekonfigurierbares On Board selbst-test-System*“ entstanden. Beide Projekte wurden vom Freistaat Thüringen unterstützt und durch Mittel der Europäischen Union im Rahmen des Europäischen Fonds für regionale Entwicklung (EFRE) kofinanziert.

4.1. TESTSYSTEMARCHITEKTUR

Eine Leiterplatte besteht, wie in Abbildung 2 bereits dargestellt, aus diversen miteinander verschalteten Komponenten. Wie in Kapitel 3.5 begründet, schränkt sich die vorliegende Arbeit auf Leiterkarten mit FPGAs. Zum besseren Lesbarkeit wird im Folgenden immer von nur von einem FPGA gesprochen. Das Konzept erlaubt aber auch das Erstellen von Testinstrumenten für mehrere FPGAs. Auf die Besonderheiten bei gegenseitigen Abhängigkeiten wird in Kapitel 6.3 eingegangen.

Der Stand der Technik in Kapitel 3 hat gezeigt, dass in erster Linie die Verlagerung der Testfunktionalität vom Test-PC auf den Prüfling, also in diesem Fall in den FPGA, viele Vorteile bringt. Weiterhin gibt es unterschiedliche Ansätze, wie diese Algorithmen im Prüfling ausgeführt werden. Hierbei haben sowohl Hardwarelösungen als auch Softwarelösungen für die Ausführung von Testalgorithmen ihre Vorteile. Im vorliegenden Konzept sollen diese Vorteile miteinander kombiniert werden.

Das Testsystem, welches in den FPGA implementiert wird, besteht dabei als logische Konsequenz aus einer Kombination von Prozessor(en)[31] für die Ausführung von Software und Co-Prozessor(en)[32] für die Ausführung von Algorithmen in Hardware. Eine schematische Darstellung eines solchen Testsystems (ohne dargestellte Verbindung zum Test-PC) ist in Abbildung 17 zu sehen. Da der Prozessor ausschließlich für Testaufgaben konzipiert ist und nur während des Testens in den FPGA programmiert wird, wird dieser im weiteren Verlauf als Testprozessor bezeichnet. Details zum Testprozessor und den Co-Prozessoren werden in Kapitel 4.1.2 näher erläutert.

Eine Kombination aus Testprozessor und Co-Prozessoren erlaubt es nicht nur, die Testalgorithmen entweder vollständig in Software oder in Hardware auszuführen, sondern ebenfalls deren Ausführung aufzuteilen. Eine solche Aufteilung macht das System sehr flexibel und ermöglicht es, die Vorteile sowohl von Testprozessoren (siehe Kapitel 3.2.1) und eingebetteter Testfunktionalität wie z.B. BIST (siehe Kapitel 3.1.2) als auch von Embedded Test Instrumenten (siehe Kapitel 3.2.3) zu vereinen.

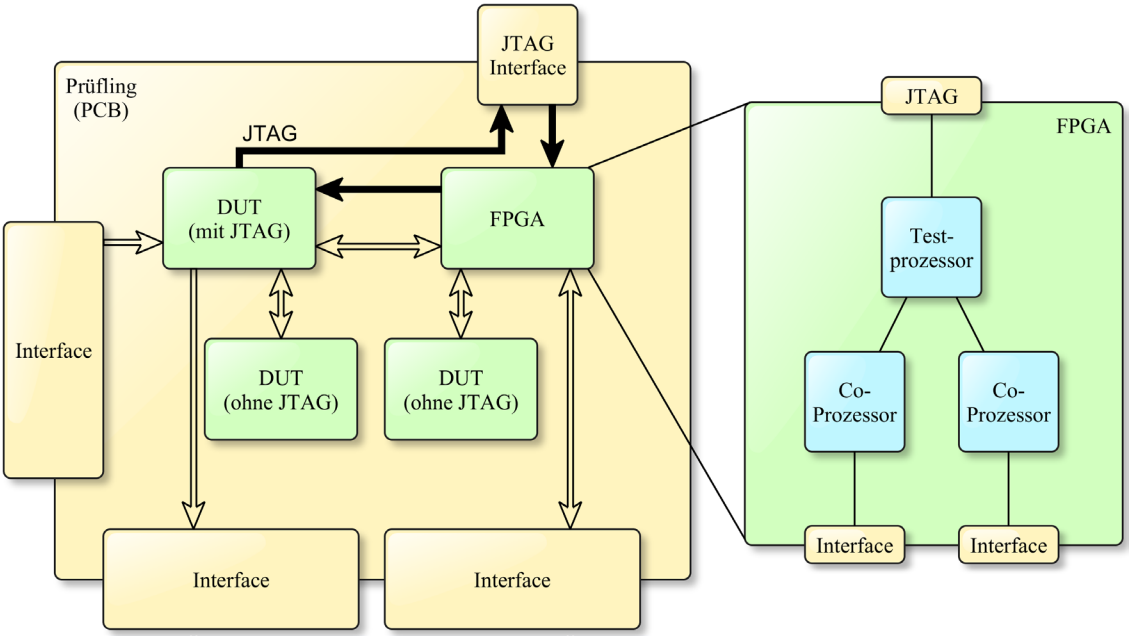


Abbildung 17: Testsystemarchitektur (ohne Test-PC)

[31] Aufgrund besserer Lesbarkeit wird im weiteren Verlauf der Arbeit nur von einem Prozessor gesprochen. Dies stellt keine generelle Einschränkung dar. Grundsätzlich sind die vorgestellten Ansätze und Konzepte auch für Mehrprozessorsysteme gültig.

[32] Aufgrund besserer Lesbarkeit wird im weiteren Verlauf der Arbeit zumeist nur von einem Co-Prozessor gesprochen. Dies ist keine generelle Einschränkung. Grundsätzlich sind die vorgestellten Ansätze und Konzepte auch für mehrere Co-Prozessoren gültig.

4.1.1. EBENENKONZEPT

Das strukturelle Testen einer Leiterplatte benötigt, wie in Kapitel 2.5 beschrieben, eine ganze Reihe von Algorithmen, sowohl für die DUT Ansteuerung als auch die Generierung von Testvektoren und die Testkoordination. Damit diese Algorithmen, wie oben beschrieben, wahlweise in Software oder in Hardware ausgeführt werden können, ist ein Modell entwickelt worden, das es erlaubt, die Testkomplexität zu strukturieren und die einzelnen Teile getrennt voneinander zu betrachten. Dies reduziert die Testkomplexität im Ganzen, macht die Darstellung übersichtlicher und erlaubt eine einfachere Wiederverwendung von Funktionen. Die Modellierung erfolgt hierbei für jedes DUT getrennt voneinander.

Das Modell eines DUTs dient der Beschreibung von Testfunktionen und stellt Abstraktionsmechanismen zur Verfügung, die es erlauben, die Eigenschaften des DUTs und der Testalgorithmen ohne Kenntnis der späteren Einsatzumgebung zu beschreiben. Das Modell unterteilt die Algorithmen der Tests und DUT Ansteuerung dabei in Ebenen. Diese Ebenen (auch Layer genannt) beschreiben jeweils eine feste Testfunktionalität. Der gesamte Test wird insgesamt auf fünf Ebenen verteilt. Aufgrund dieser Unterteilung wird das Modell als Ebenenmodell bezeichnet. Dies ist in Abbildung 18 dargestellt.

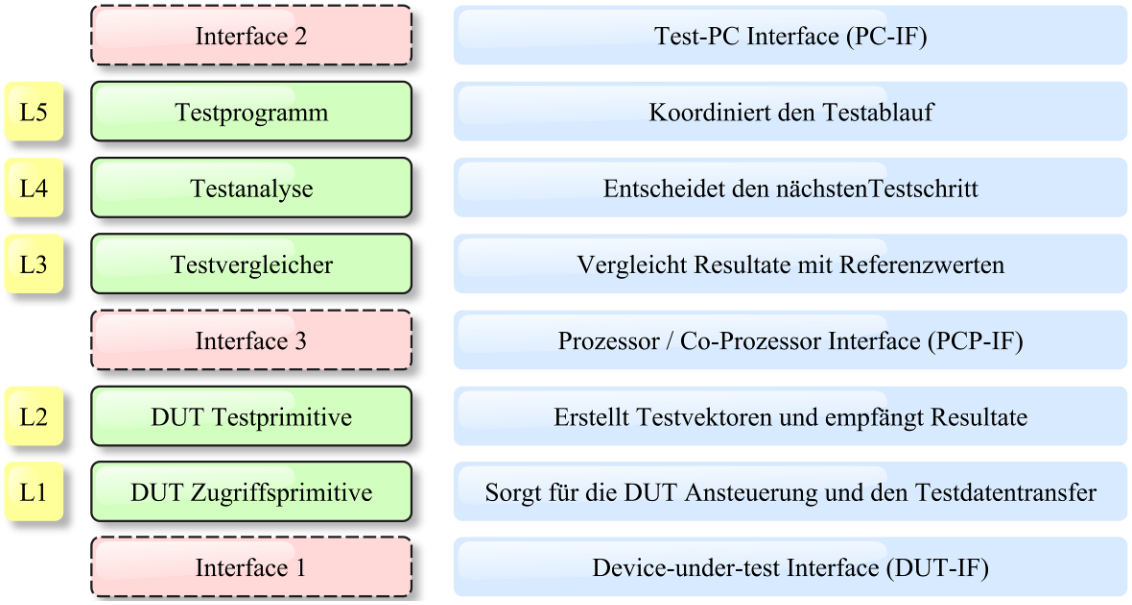


Abbildung 18: Ebenenkonzept

Dieses Ebenenmodell stellt eine hierarchische Struktur dar, in der jeweils die untergeordneten Ebenen eine gewisse Funktionalität realisiert und diese als Dienste über eine definierte Schnittstelle der übergeordneten Ebene zur Verfügung stellen. Die Funktionen der Ebenen reichen von einfachen Zugriffsroutinen der DUTs, wie Lesen und Schreiben auf der untersten Ebene L1, bis hin zur Testsanalyse, die für die Abarbeitung der Tests eines einzelnen DUTs zuständig sind auf Ebene L4 und der Koordination von Tests mehrerer DUTs auf Ebene L5.

Das Ebenenmodell besteht nicht nur aus Ebenen, die eine bestimmte Funktionalität bündeln, sondern verfügt auch über Schnittstellen. Es gibt dabei drei unterschiedliche Schnittstellenarten, die nach ihrer Art der Kommunikation unterschieden werden:

- Kommunikation mit dem DUT
- Kommunikation mit dem externen Test-PC
- Kommunikation zwischen den in Hardware und Software realisierten Ebenen innerhalb des FPGAs.

Die Kommunikation **mit dem DUT** ist in Abbildung 18 als Interface 1 gekennzeichnet und hängt ausschließlich von den vorhandenen Pins des DUT ab. Über diese Schnittstelle werden die eigentlichen strukturellen Tests ausgeführt und die Testvektoren an das DUT angelegt.

Die Kommunikation **mit dem externen Test-PC** erfolgt über Interface 2 und dient als Schnittstelle zum Test-PC und zum Testingenieur. Über diese Schnittstelle können alle Funktionen, die auf dem Prüfling ausgeführt werden gesteuert und die Ergebnisse zum Test-PC übertragen werden.

Die Schnittstelle **zwischen Prozessor und Co-Prozessor** wird als Interface 3 bezeichnet. Über diese Verbindung kann der Prozessor auf die vom Co-Prozessor bereitgestellte Funktionalität zugreifen.

Auf alle drei Kommunikationsarten wird genauer in Kapitel 4.1.4 eingegangen.

4.1.2. KOMPONENTEN

Die Ausführung der Ebenen innerhalb des FPGAs kann gemäß dem Ebenenkonzept entweder in Software oder in Hardware erfolgen. Abbildung 19 veranschaulicht drei von vielen Möglichkeiten der Aufteilung der Ebenen auf die unterschiedlichen Ausführungsarten. Zur besseren Unterscheidung wird die auf dem Test-PC ausgeführte Software weiterhin als Software (SW) bezeichnet, während die auf dem Testprozessor ausgeführte Software als Embedded Software (ESW) bezeichnet wird. Funktionen, die als Co-Prozessor implementiert werden, werden als ‚Ausführung in Hardware' (HW) bezeichnet.

Während Interface 1 immer zwischen DUT und L1 platziert ist und somit nicht verschoben werden kann, ist die Anordnung von Interface 2 und Interface 3 variabel und hängt von der gewählten Verteilung der Ebenen auf die möglichen Ausführungsarten ab. I2 stellt dabei immer die Schnittstelle zwischen SW und ESW dar und muss zwingend oberhalb von I3 platziert werden. I3 hingegen kann sich zwischen I1 und I2 befinden und stellt die Verbindung von SW zu HW dar.

Abbildung 19 zeigt drei verschiedene Platzierungen. Diese sind:

- a) wenn I3 direkt an I2 angrenzt, da keine Funktionalität im Testprozessor ausgeführt werden soll

- b) wenn sowohl im Co-Prozessor als auch im Prozessor die Funktionalität von zwei Ebenen realisiert werden soll
- c) wenn I3 direkt an I1 angrenzt, da die gesamte Funktionalität im Testprozessor ausgeführt werden soll

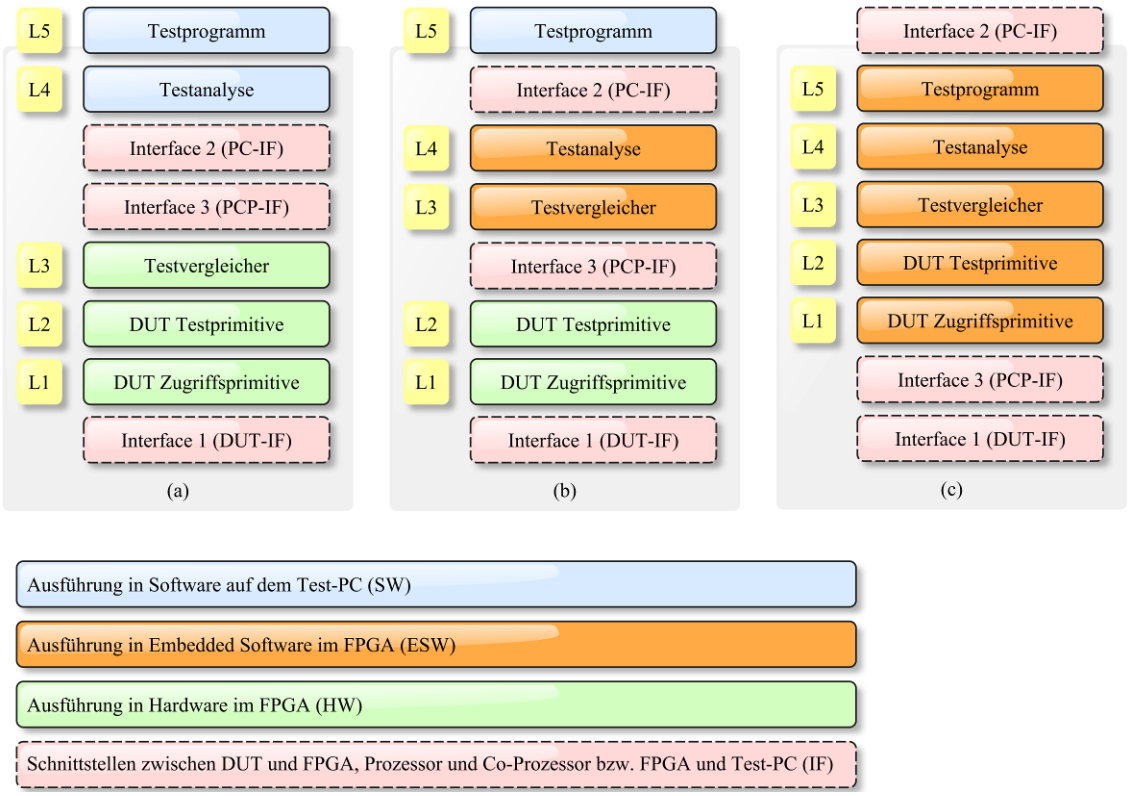


Abbildung 19: Unterschiedliche Implementierungen des Ebenenkonzepts

Für diese Ausführung der Ebenen und die notwendige Kommunikation mit dem DUT sowie dem Test-PC sind jeweils bestimmte Komponenten innerhalb des FPGAs notwendig.

Diese sind:

- ein Testprozessor für die Ausführung als Embedded Software
- ein Co-Prozessor für die Ausführung als Hardware
- weitere Komponenten, wie Funktionsböcke innerhalb des FPGAs, die zur Nutzung der JTAG Schnittstelle als Interface 2 erforderlich sind

Als **Testprozessor** wird ein RISC Prozessor mit Harvard Architektur verwendet. Dieser wurde, ausgehend von den genannten Forderungen an das Testsystem bezüglich Flexibilität und Adaptierbarkeit an den Prüfling speziell entwickelt und ist sehr variabel. So ist es nicht nur möglich, genau zu spezifizieren, welche Befehle die ALU unterstützen soll, sondern auch die Registeranzahl, Speichergrößen und sogar Bitbreiten des Adress- und Datenbusses sind in einem großen Rahmen frei zu wählen. Die genaue Beschreibung des Prozessors sowie die Begründungen zu den getätigten Design-

entscheidungen können in der Dissertation von [33] nachgelesen werden. In dieser Ausarbeitung wird auch die Frage nach den Vor- und Nachteilen eines Prozessor-basierten Testens ausführlich behandelt. Für den Kontext dieser vorliegenden Arbeit ist nur die Existenz eines entsprechend konfigurierbaren und flexiblen Testprozessors relevant und wird als gegeben vorausgesetzt.

Als **Co-Prozessor** wird jeweils der Teil der DUT Modellierung beschrieben, der nicht in Form von Embedded Software auf dem Testprozessor ausgeführt wird, sondern direkt in FPGA Logik implementiert ist. Durch die Umsetzung in Hardware ist es dem Co-Prozessor möglich, speziell an die Anforderungen des DUTs angepasst zu werden. Dies erlaubt potentiell die schnellste Ausführung von Algorithmen. Durch die Nutzung dedizierter Hardware steigt jedoch der Ressourcenverbrauch.

Notwendiger Bestandteil des Testsystems sind neben dem beschriebenen Prozessor und den Co-Prozessoren **weitere Komponenten**. Hierzu gehören unter anderem ein JTAG Interface, das es ermöglicht über JTAG zwischen Test-PC und Testprozessor zu kommunizieren. Im Ebenenmodell ist dies Teil des Interface 2. Diese Schnittstelle ist jedoch austauschbar, um so zukunftssicher zu bleiben. So kann ein neuerer JTAG Standard wie zum Beispiel IJTAG [53] prinzipiell vom Testsystem sofort verwendet werden, sobald der FPGA über die entsprechende Schnittstelle verfügt.

Ebenfalls ist es möglich, parallel oder als Alternative zur JTAG Schnittstelle auch andere Schnittstellen für eine Übertragung zwischen Test-PC und FPGA zu nutzen. So könnte eine Übertragung großer Datenmengen zum Beispiel über USB oder Ethernet wesentlich schneller erfolgen. Dies wirkt sich positiv auf die Dauer der gesamten Testausführung aus, erhöht jedoch den Ressourcenbedarf innerhalb des FPGAs. Für das Konzept des Testsystems ist die genaue Ausführung der Schnittstelle jedoch nicht relevant. Im Rahmen dieser Arbeit wurde JTAG als Kommunikationsschnittstelle zum Test-PC festgelegt, da nahezu jeder FPGA über diese Schnittstelle verfügt und sie die Anwendung des Konzepts auch bei kleinsten FPGAs ermöglicht.

4.1.3. KONFIGURIERBARKEIT

Durch die Aufteilung der Testsystemfunktionalität auf den Test-PC, Testprozessor und Co-Prozessor ist es möglich, das gewünschte Testsystem genau auf die eigenen Anforderungen und gegebenen Randbedingungen abzustimmen.

Diese Anforderungen können sein:

- ein geringer Ressourcenverbrauch, da der verwendete FPGA nur eine begrenzte Anzahl an Ressourcen hat und diese nicht überschritten werden können
- at-speed Testen, bzw. bestimmte zeitliche Randbedingungen der DUTs, die zwingend eingehalten werden müssen, um ein korrektes Testergebnis zu erhalten
- eine schnelle Testausführung, bezogen auf die Gesamtdauer des Tests

- oder eine besonders gute Fehlerdiagnose, für die jedoch komplexe Algorithmen notwendig sind

Vergleicht man die drei Realisierungsmöglichkeiten eines Test-PCs, eines Testprozessors und eines Co-Prozessors, so werden die Gegensätze bezüglich dieser Anforderungen sichtbar. Diese sind in Tabelle 3 dargestellt.

Tabelle 3: Eigenschaften der Testsystemkomponenten

Eigenschaft \ Testsystemkomponente	Test-PC	Prozessor	Co-Prozessor
Ressourcen schonend[33]	✓	o	×
At-speed	×	o	✓
Schnelle Testausführung	×	o	✓
Besonders gute Fehlerdiagnose	✓	o	×

×: Wird nicht gut erfüllt ✓: Wird gut erfüllt o: Mit Einschränkungen

Bei der Entscheidung über die geeignete Zuordnung der Testalgorithmen zu einer Ausführung auf dem Test-PC, Testprozessor oder Co-Prozessor müssen alle Randbedingungen und Möglichkeiten der Realisierungsvarianten berücksichtigt werden.

Damit sich das generierte Testsystem möglichst gut an die Forderungen und Bedingungen anpassen kann, muss es hoch konfigurierbar sein. Hierbei müssen folgende Fragen beantwortet werden:

- Ist das gesamte Testsystem in einem FPGA-Bitfile enthalten oder werden mehrere FPGA-Bitfiles generiert?
- Sollen unabhängige DUTs parallel jeweils durch eigene Prozessoren getestet werden?
- Soll die gesamte Embedded Software ins FPGA-Bitfile integriert werden oder wird diese über JTAG nachgeladen?
- Werden die gesamten Testalgorithmen für jedes DUT (L1 bis L5) im FPGA implementiert oder befinden sich einige Teile auf dem Test-PC?
- Welche Testalgorithmen sollen in einem Co-Prozessor ausgeführt werden?
- Welche Operationen muss der jeweilige Testprozessor unterstützen?
- Welche Parameter (Busbreite, Registeranzahl, etc.) sind für den Testprozessor zu wählen?

[33] Es werden nur die Ressourcen auf dem Prüfling betrachtet. Die Ressourcen auf dem Test-PC werden nicht betrachtet, da diese relativ zum Prüfling in sehr großer Menge zur Verfügung stehen.

Alle Entscheidungen haben direkten oder indirekten Einfluss auf das entstehende Testsystem, dessen Ressourcenverbrauch und Leistungsfähigkeit. Auf die Komplexität dieser Entscheidungen wird in Kapitel 4.3 *Partitionierung* näher eingegangen.

4.1.4. KOMMUNIKATION

Durch die mögliche Aufteilung der Testalgorithmen auf drei unterschiedliche Teile des Testsystems (extern – Test-PC und intern – Prozessor und Co-Prozessor) ergeben sich auch drei unterschiedliche Ausführungsarten der Algorithmen. Entweder als Software auf dem Test-PC, als Embedded Software auf dem Testprozessor oder als Hardware in den Co-Prozessoren.

Die drei Ausführungsarten führen zu einer Reihe an sinnvollen Kommunikationsvarianten der Testsystemkomponenten untereinander. Hierbei wird immer davon ausgegangen, dass eine Kommunikation von einer höheren Ebene aus gesteuert wird (Quelle). Die zulässigen Kommunikationsvarianten sind in Tabelle 4 dargestellt. So kann eine in Software realisierte Testfunktion (Quelle = SW) mit untergeordneten Testfunktionen in allen drei Ausführungsarten kommunizieren, egal, ob diese als SW, ESW oder HW implementiert sind (1. Zeile von Tabelle 4). In Hardware realisierte Testfunktionen (Quelle = HW) hingegen können nur auf Testfunktionen zugreifen, die ebenfalls in Hardware realisiert sind (Ziel = HW) (3. Zeile in Tabelle 4).

Tabelle 4: Kommunikationsvarianten im Schichtenmodell

Quelle \ Ziel	SW	ESW	HW
SW	✓	✓	✓
ESW	×	✓	✓
HW	×	×	✓

4.1.4.1. EXTERNE KOMMUNIKATION

Für die Kommunikation zwischen den auf dem Test-PC realisierten Ebenen (SW) und Ebenen, die auf dem FPGA implementiert sind (ESW und HW), wurde, wie bereits in Kapitel 4.1.2 beschrieben, JTAG gewählt, da diese Schnittstelle bei nahezu allen FPGAs vorhanden ist. Diese Schnittstelle wird als Interface 2 bezeichnet (siehe Abbildung 19).

Die Verwendung von JTAG als externe Kommunikationsschnittstelle Interface 2 erlaubt unterschiedliche Realisierungsvarianten für den Datenaustausch.

Diese können in zwei Gruppen unterteilt werden:

- eine Intest-basierte Realisierung
- eine TAP-interface-basierte Realisierung

Beim **Intest-basierten** Verfahren wird die Möglichkeit einiger Boundary-Scan-Zellen genutzt, Daten im FPGA nicht nur nach extern, sondern auch nach intern bereitzustellen. Hierfür muss jeweils die gesamte Boundary-Scan-Kette für den Datentransport genutzt werden. Genau hieraus ergibt sich einer der entscheidenden Nachteile dieser Methode. So muss auch für eine Datenübertragung von wenigen Bits immer die gesamte Kette verwendet werden. Es ist zwar möglich, mehrere Datenblöcke auf einmal über die Kette in den FPGA zu schieben und diese dort dann nacheinander zu nutzen, jedoch wird Bandbreite verschenkt, wenn die Daten überhaupt nicht benötigt werden.

Beispiel: Es sollen 100 Testvektoren zu je $4\,Bit$ am FPGA ausgeben werden. Die Boundary-Scan Kette beträgt 1000 Zellen. Müsste man beim klassischen Boundary-Scan noch $100 * 1000\,Bit$ schieben, so reduziert sich diese Anzahl auf $1 * 1000\,Bit$ bei Intest-basierten Ansatz, da die benötigten $100 * 4\,Bit$ vollständig in einer Kettenlänge integrierbar sind. Da diese jedoch vollständig geschoben werden muss, werden immer noch $600\,Bit$ unnötig übertragen.

Weiterhin ist das Intest-basierte Verfahren nicht bei jedem FPGA anwendbar. Insbesondere die meisten FPGAs von Lattice [91] verfügen nicht über entsprechende Boundary-Scan-Zellen, die den Intest unterstützen. Dies liegt daran, dass der Intest zwar im Boundary-Scan-Standard definiert ist, jedoch nicht verpflichtend implementiert werden muss.

Der zweite Ansatz ist das **TAP-basierte** Verfahren. Hierbei wird im FPGA eine direkte Verbindung zur JTAG Schnittstelle geschaffen. Über meist zwei, bei einigen FPGAs aber auch über mehrere so genannter Benutzerinstruktionen kann, der Datenstrom in die frei programmierbare Logik des FPGAs geleitet werden. Dieses Verfahren ist sehr flexibel und es können unterschiedliche Möglichkeiten der Datennutzung umgesetzt werden.

In Abbildung 20 wird die JTAG Struktur innerhalb eines FPGA über zwei benutzerdefinierte Registerverbindungen, den sogenannten User Registern, an die programmierbare Logik angebunden. Dabei dient USER 1 als Adressregister, um über USER 2 mehrere Register innerhalb der FPGA Logik zu adressieren. Auf dieser Weise kann eine flexible Struktur von mehreren, auch unterschiedlich langen Registern im FPGA realisiert werden.

Bei der TAP-basierten Kommunikationsarchitektur, die in Abbildung 20 dargestellt ist, ist es prinzipiell nicht relevant, was genau sich in einem der User-Register verbirgt.

Dies kann:

- ein einfaches Schieberegister sein, dass Daten für einen Test entgegen nimmt und lediglich an bestimmte Pins des FPGAs weiterleitet
- oder aber auch komplexere Funktionen beinhalten, wie z.B. ein Register eines Prozessors, über das neue Testvektoren und Testalgorithmen direkt in den Speicher des Prozessors geschrieben werden können.

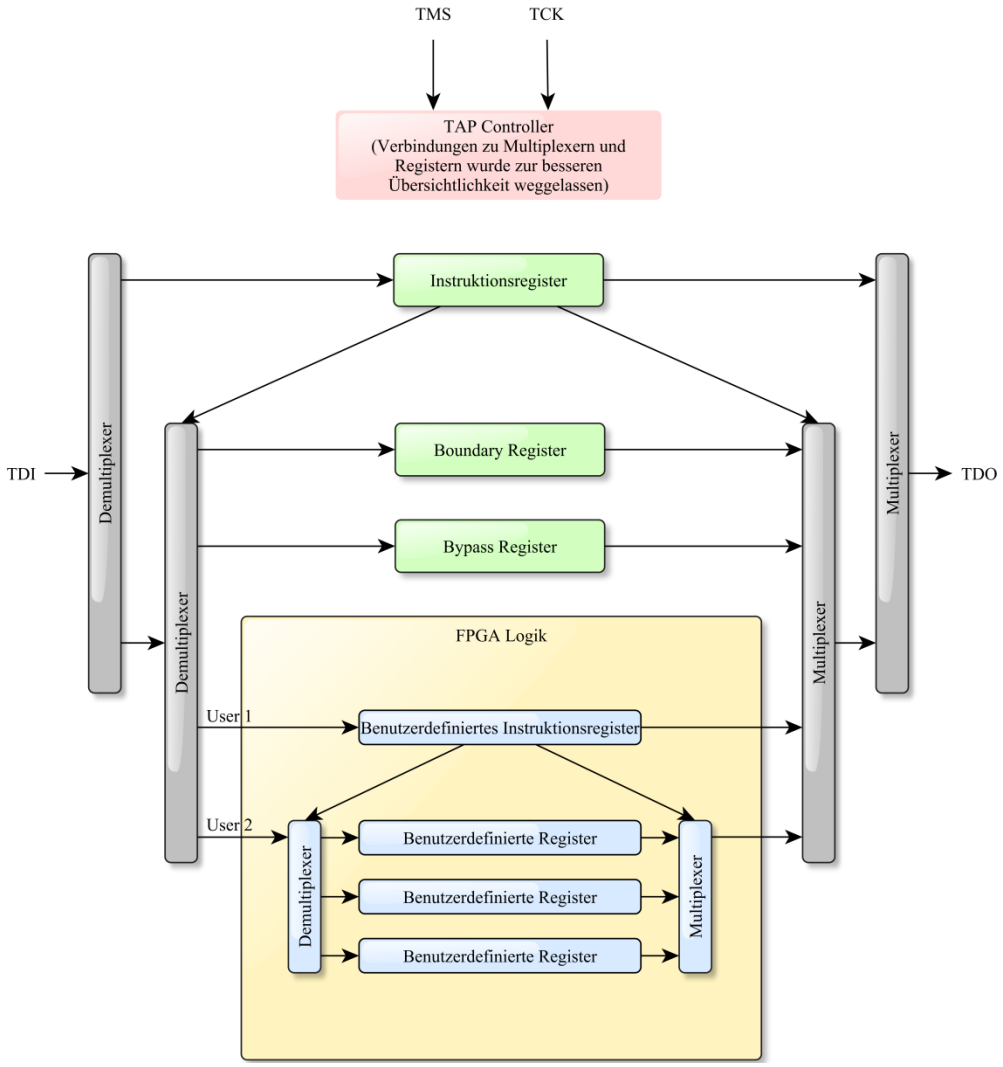


Abbildung 20: JTAG Architektur mit benutzerdefinierten Registern in FPGA Logik

Im Folgenden sollen einige Ansätze aus der Literatur für ihre Eignung im Rahmen der definierten Testsystemarchitektur hin untersucht werden.

In [15] werden für die Implementierung einer eigenen Scankette unterschiedliche Typen an Scanzellen in der programmierbaren Logik des FPGAs untersucht und ausführlich gegenübergestellt. Dies betrifft sowohl Ansätze, bei dem die Daten direkt zu den FPGA Pins geleitet als auch zur internen Nutzung einem Testcontroller übergeben werden. Die Ansteuerung erfolgt über zwei User-Register.

Diese Ergebnisse zeigen eine Beschleunigung zwischen dem ca. 13-fachen und 60-fachen je nach gewählter Registerstruktur und abhängig von der durchzuführenden Art der Datenübertragung im Vergleich zu Boundary-Scan. An dieser Stelle zeigt sich ein generelles Problem bei der Umsetzung und Bewertung von unterschiedlichen Realisierungen von Teststrukturen im FPGA. Die Ergebnisse sind generell stark abhängig vom gewählten FPGA und den Testvektoren, die genutzt werden, so dass ein

genauer Vergleich von Testzeiten nur unter exakt gleichen Bedingungen erfolgen kann und dann auch nur für diesen Testfall Gültigkeit hat.

Da sich [15] hauptsächlich um die Programmierung von Flash und nicht mit dem Testen von Speichern beschäftigt, sind die Ergebnisse, also die erreichbaren Beschleunigungfaktoren, nicht direkt auf das Testen von Speichern übertragbar. Beim Programmieren von Speichern geht es darum, möglichst schnell eine ggf. große Datenmenge zum Speicher zu übertragen, während dies beim Testen meist nicht der Fall ist, da es dabei um das schnelle Übertragen weniger ausgewählter Testvektoren geht. Die Arbeit in [15] zeigt jedoch mögliche Strukturen für eine Datenübertragung auf, die auch für das Testen nutzbar sind.

In [92] wurde die Möglichkeit vorgestellt in Abhängigkeit der verwendeten Testvektoren unterschiedlich lange Scanketten in der FPGA Logik zu implementieren und diese sehr ähnlich zum Boundary-Scan zu verwenden. Durch die reduzierte Länge im Vergleich zu den oft sehr langen Boundary-Scan-Ketten ergibt sich eine Beschleunigung bei der Testausführung.

Ein weiterer Vorteil des TAP-basierten Verfahrens gegenüber dem Intest-basierten Verfahren ist es, dass auch, während Teile des FPGAs einen Test über ihre Pins ausführen, über JTAG und die interne Scankette weitere Daten nachgeladen werden können, ohne dass der Test unterbrochen werden muss.

Die TAP-basierte Kommunikation zeigt somit deutliche Vorteile gegenüber der Intest-basierten Variante. Dies betrifft besonders den Vorteil der flexiblen Kettenlänge, die innerhalb der programmierbaren Logik der FPGAs generiert werden kann und unabhängig von der Länge der Boundary-Scan-Kette ist. Weiterhin ist es beim TAP-basierten Verfahren von Vorteil, unabhängig von den verwendeten Boundary-Scan-Zellen und somit bei nahezu allen FPGAs anwendbar zu sein. Der Nachteil der zusätzlichen Ressourcen, die benötigt werden, um die Registerketten zu implementieren, relativiert sich, da dieser Verbrauch sehr gering ist und außerdem auch das Intest-basierte Verfahren weitere Ressourcen benötigt, um die bereitgestellten Daten sinnvoll anzuwenden. Ein genauer Vergleich des Ressourcenbedarfs und der erreichten Testzeiten hängt vom jeweiligen Testfall, also den verwendeten Testalgorithmen, dem zu testenden DUT und dem genutzten FPGA ab und kann nicht pauschal bestimmt werden.

Auch ohne genaue Zahlen vergleichen zu können, lässt das TAP-basierte Verfahren generell eine schnellere und flexiblere Kommunikation gegenüber dem Intest-basierten Verfahren erwarten. Zwar ist es laut [15] auch möglich, dass bei bestimmten Konfigurationen Intest-basierte Kommunikation schneller ist als TAP-basierte, dies ist jedoch nicht der Regelfall. Die Abwägung der Vor- und Nachteile für die Nutzung einer TAP-basierten Kommunikation für den Datenaustausch zwischen Test-PC und FPGA deckt sich mit den Ergebnissen aus [15] sowie den Erkenntnissen aus [92]. Hiernach ist die TAP-basierte Kommunikation für einen Datenaustausch vorzuziehen, und wird im Folgenden für eine Umsetzung gewählt.

4.1.4.2. INTERNE KOMMUNIKATION

Neben der TAP-basierten Kommunikation zwischen SW und ESW bzw. HW (siehe Tabelle 4) wird für die interne Kommunikation innerhalb des FPGAs zwischen ESW und HW ein anderes Verfahren verwendet. Da im Rahmen des Projekts *ROBSY* sowohl der Testprozessor als auch der Co-Prozessor entwickelt wurden, mussten für die Auswahl einer geeigneten Kommunikation zwischen diesen Komponenten keine externen Vorgaben berücksichtigt werden. Für die Kommunikation zwischen Testprozessor und den Co-Prozessoren wurde auf den flexiblen und frei zugänglichen Wishbone-Bus-Standard [93] gesetzt. Diese Schnittstelle ist in Abbildung 19 als Interface 3 dargestellt.

Folgende Gründe sprechen für den Wishbone-Bus-Standard:

- eine frei zugängliche Dokumentation
- eine einfache Struktur des Busses
- kein Ballast durch ungenutzte Funktionalität
- eine weite Verbreitung. So kann später eine einfache Einbindung von Co-Prozessoren von Drittanbietern ermöglicht werden

Die prinzipielle Kommunikation zwischen einem Testprozessor (ESW) und einem Co-Prozessor (HW) über Wishbone ist in Anhang C dargestellt.

Für die Kommunikation zwischen den einzelnen, ausschließlich in Hardware realisierten Ebenen, die innerhalb eines Co-Prozessors implementiert werden, wurde ein einfaches Handshakeverfahren gewählt. Diese Kommunikation entspricht der untersten Zeile in Tabelle 4. Die Daten werden über Signale bereitgestellt und müssen von der jeweils übergeordneten Schicht für die Dauer der Kommunikation konstant gehalten werden. Die Synchronisation zwischen mehreren parallel implementierten Co-Prozessoren erfolgt über ein einfaches Enable Signal. Die Synchronisation zwischen zwei Schichten eines Co-Prozessors erfolgt über die Request- und Acknowledge-Signale. Dies hat folgende Vorteile und ist grafisch in Anhang D dargestellt:

- geringer Ressourcenverbrauch, da Signale im FPGA keine Logik-Ressourcen benötigen
- einfache und schnelle Kommunikation durch Auswerten von nur zwei Signalen

Nachdem die ausgewählte Architektur des Testsystems mit den verwendeten Komponenten und deren Schnittstellen untereinander beschrieben und begründet wurde, ergibt sich an dieser Stelle folgende Frage:

- Wie modelliert man diese Funktionen, um daraus automatisch eine Implementierung des Testsystems generieren zu können?

Mit dieser Frage beschäftigt sich das folgende Kapitel 4.2.

4.2. MODELLIERUNG

Um neben den Vorteilen nicht auch die Nachteile, wie z.B. die aufwendige manuelle Generierung der beschriebenen Testsystemarchitekuren zu übernehmen, ist es das Ziel, das hier beschriebene Testsystem automatisch generieren zu lassen. Hierfür ist eine entsprechende Modellierung notwendig, die es einem Programm erlaubt, das Testsystem möglichst ohne manuelle Eingriffe durch den Testingenieur zu erstellen. Der folgende Abschnitt beschäftigt sich mit der Fragestellung der Modellierung der einzelnen Komponenten eines solchen Testsystems.

Allgemein wird eine Modellierung dazu genutzt, einen Sachverhalt oder ein Objekt so zu beschreiben, dass alle nötigen Informationen dargestellt werden, während unnötige verborgen bzw. vernachlässigt werden. Die entscheidende und schwierige Frage hierbei ist: Was ist wichtig, was ist unwichtig? Ebenso die Frage, welcher Modellierungsansatz der geeignete ist. [94] hat hierzu gesagt „*There is not a single correct model, only adequate and inadequate ones*“[34]. Dies zeigt, dass es durchaus mehrere Ansätze geben kann, die alle ihre Berechtigung haben, da sie für eine Modellierung geeignet sind, aber keiner der Ansätze kann als *der einzig richtige* bezeichnet werden.

Für die vorliegende Aufgabe der automatischen Testsystemgenerierung ist es notwendig, folgende Dinge zu modellieren:

- die relevanten Merkmale der Leiterplatte (PCB), bestehend aus der Netzliste und den Verbindungen aller Komponenten untereinander
- die Parameter des FPGAs, der für die Testausführung verwendet werden soll
- die Schnittstellen und Testfunktionen der Devices-Under-Test (DUTs), zu denen die Verbindung getestet werden soll

Entscheidend bei der Wahl einer geeigneten Modellierung ist auch die Frage, wie am Ende die Programmierung des FPGAs realisiert wird. Diese erfolgt zumeist durch ein sogenanntes Bitfile, das nach einem Syntheseprozess[35] von den Entwicklungstools der FPGA Hersteller generiert wird. Will man nicht den gesamten Syntheseprozess selber durchführen – und hiervon ist aufgrund der Komplexität und der zumeist nicht offengelegten Informationen der FPGA Hersteller über den Aufbau der Bitfiles auch abzuraten – so muss man eine Modellierung wählen, die es erlaubt daraus Eingangsdaten für die FPGA-Tools zu generieren.

[34] „Es gibt kein einziges korrektes Modell, nur passende und unpassende“.

[35] Der Prozess, der für eine Bitfile-Generierung notwendig ist, besteht aus den Schritten Synthese, Translate, Mapping, Place-and-Route sowie das eigentliche Erzeugen des Bitfiles. Zur Vereinfachung wird dieser Prozess im Verlauf dieser Arbeit nur als Synthese bezeichnet.

Eine Übersicht über die zu modellierenden Elemente eines Testsystems ist in Abbildung 21 dargestellt. Auf diese Elemente wird im Folgenden näher eingegangen.

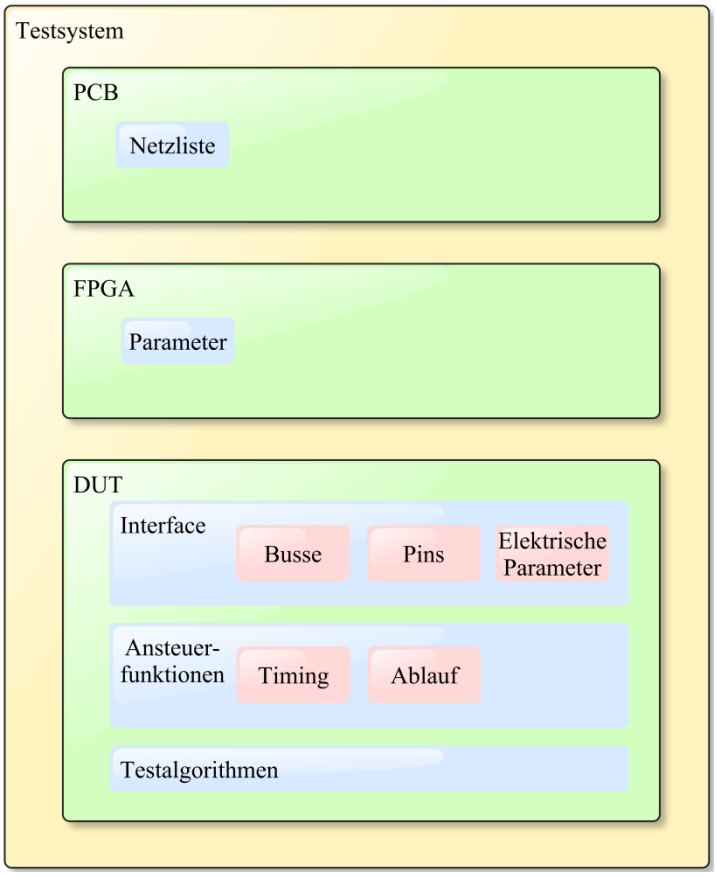


Abbildung 21: Modellierung eines Testsystems

Für die **Modellierung der Leiterplatte (PCB)** kann auf bestehende Ansätze zurückgegriffen werden. So ist es üblich, alle benötigten Daten, die eine Leiterplatte beschreiben, direkt aus den CAD-Tools, die für den Entwurf von Leiterplatten genutzt werden, zu erhalten. Diese Daten beinhalten alle Bauelemente sowie deren Verbindungen untereinander (Netzliste) und können in unterschiedlichen Formaten bereitgestellt werden. Im Rahmen dieser Arbeit wird ein Format der Firma GÖPEL electronic verwendet, um eine spätere Integration des Testsystems in deren Tools zu erleichtern. Da diese Daten vom Entwickler der Leiterplatten bereitgestellt bzw. vom Testtoolhersteller aufbereitet werden, werden diese als gegeben betrachtet und im weiteren Verlauf der Modellierung nicht ausführlicher analysiert. Ein gekürztes und kommentiertes Beispiel eines solchen Modells ist in Anhang F zu finden.

Für die **Modellierung des FPGAs** sind nur wenige Informationen notwendig. Diese sind zum Beispiel Typenbezeichnung, Gehäusebezeichnung und verwendeter Speedgrade. Diese Daten sind für die spätere Synthese notwendig. Weiterhin sind Informationen über die Anzahl der vorhandenen Ressourcen im FPGA wichtig, wie zum Beispiel die verfügbaren Taktgeneratoren, Speicherblöcke oder Logikzellen, da diese nötig sind, um bestimmte Eigenschaften der Optimierung und Partitionierung zu steuern

bzw. das generierte Bitfile zu bewerten. Da es sich bei dieser Modellierung nur um das Speichern weniger Begriffe und Zahlen handelt, ist eine Analyse bestehender Modellierungsansätze an dieser Stelle nicht sinnvoll. Es wird daher eine einfache Text-basierte Beschreibung gewählt. Ein Beispiel hierfür ist in Anhang G zu finden.

Die **Modellierung der DUTs** stellt den zentralen Teil der Modellierung des Testsystems dar. In ihr müssen bestimmte Funktionsweise der Schnittstellen der DUTs, wie zum Beispiel Zugriffsroutinen, aber auch die physikalischen Schnittstellen an sich (Pinbelegungen, elektrische Parameter, etc.), sowie die Testalgorithmen zum Testen bestimmter Fehler modelliert werden.

Neben den Schnittstellen und Testalgorithmen muss sich auch das Ebenenkonzept angemessen modellieren lassen, da sich nur so später eine Aufteilung der Algorithmen auf die unterschiedlichen Ausführungsarten realisieren lässt. Dies erfordert eine geeignete Modellierungssprache. Was genau ‚geeignet' bedeutet hängt von den detaillierten Anforderungen ab und wird ausführlicher in Kapitel 4.2.1 beschrieben.

4.2.1. MODELLIERUNGSANFORDERUNGEN FÜR DUTS

Damit ein Testsystem automatisch generiert werden kann, müssen eine Reihe von Merkmalen eines DUTs modelliert werden.

Diese sind:

- Schnittstellen
- Elementare Zugriffsfunktionen und deren zeitliche Zusammenhänge
- Testalgorithmen für die Testvektorgenerierung

Aus den Vorgaben der Zielstellungen in Kapitel 3.5 und der gewählten Testsystemarchitektur aus Kapitel 4.1 ergeben sich für eine DUT Modellierung folgende Anforderungen an die Modellierungssprache:

- die Spezifikation muss modular für jedes DUT separat erfolgen können
- alle Teile eines DUT-Modells, unabhängig von einer späteren Realisierung in Software oder Hardware, sollen innerhalb einer einzelnen Datei vollständig beschreibbar sein
- die Modellierung sollte textbasiert in einer einfachen Sprache erfolgen, so dass jeder Testingenieur ein Modell editieren bzw. neue erstellen kann
- sie muss unabhängig von der Netzliste der Leiterplatte und von gewählten Einsatzparametern, wie zum Beispiel der genutzten Taktfrequenz sein
- alle nötigen elektrischen Eigenschaften der DUTs, wie die Pinbelegungen des Interfaces, zulässige I/O-Spannungen, benötigte I/O-Standards und geforderte Treiberstärken sowie Flankensteilheit der Signale müssen beschreibbar sein

- es muss eine strukturierte Darstellung der Testalgorithmen und Zugriffsroutinen möglich sein. Hierfür sollen mindestens Schleifen, Vergleiche und Sprünge, sowie Addition, Subtraktion und Schiebeoperationen unterstützt werden, um neben einer einfachen Ablaufsteuerung auch die Testvektorgenerierung auf Basis der in Kapitel 2.4 und 2.6 definierten Testalgorithmen durchführen zu können. In Kapitel 6.3 wird im Ausblick noch einmal auf die Möglichkeit komplexerer Befehle eingegangen
- weiterhin muss die Modellierung hierarchisch erfolgen, gemäß des vorgestellten Ebenenkonzepts aus Kapitel 4.1.1. Diese strukturierte, hierarchische Darstellung erfordert eine prozedurale Beschreibung
- um at-speed Tests zu unterstützen und gleichzeitig nicht schon bei der Modellierung eine bestimmte Realisierung vorzugeben, sollen die zeitlichen Randbedingungen der DUTs exakt beschrieben werden. Im Gegensatz zu anderen Verfahren wie z.B. Boundary-Scan, bei denen die Testalgorithmen relativ langsam ausgeführt werden, ist es mit einem FPGA möglich, sehr schnell auf die DUTs zuzugreifen. Während bei einer langsamen Ansteuerung minimale Zeitvorgaben zumeist automatisch eingehalten wurden, waren maximale Zeitvorgaben oft gar nicht einhaltbar. Der Testingenieur konnte somit keinen wirklichen Einfluss auf die Einhaltung dieser Zeiten nehmen und musste hoffen, dass der entwickelte Test trotzdem funktionsfähig war und die gewünschte Testabdeckung lieferte. Bei einem Einsatz eines FPGA ergibt sich jedoch der Vorteil, dass die Verbindungen zu DUTs prinzipiell mit ihrer auch später im Betrieb genutzten Geschwindigkeit getestet werden können, sich die Tests also at-speed durchführen lassen. Somit sind sowohl minimale als auch maximale Zeitvorgaben exakt einzuhalten. Damit dies möglich ist, müssen diese entsprechend spezifiziert werden
- um auch komplexe zeitliche Abhängigkeiten und wiederkehrende Signalfolgen, wie sie bei Protokollen und seriellen Datenübertragungen auftreten können, beschreiben zu können, ist ein hierarchisches Modellieren zeitlicher Abhängigkeiten erforderlich
- da zum Zeitpunkt der Spezifikation eines DUTs die spätere Einsatzumgebung wie zum Beispiel die genaue Pinbelegung am FPGA, und auch die genutzte Taktfrequenz nicht bekannt ist, müssen die tatsächlich erlaubten Zeitparameter und somit der gesamte Lösungsraum modelliert werden und es reicht nicht aus, nur eine gültige Lösung zu beschreiben

Bei der Modellierung der hardwarenahen Schichten gibt es ein paar Besonderheiten gegenüber der klassischen Modellierung von Testalgorithmen. So ist für die Ansteuerung von DUTs eine bestimmte Abfolge mehrerer Signale notwendig. Dieses ist für ein abstraktes Beispiel mit drei Signalen einer asynchronen Ansteuerung in Abbildung 22 dargestellt. Hierbei können einige Signalwechsel gleichzeitig oder aber auch nacheinander erfolgen. Die zeitlichen Abhängigkeiten für dieses fiktive Beispiel sind in

Tabelle 5 dargestellt. Hieraus lassen sich eine ganze Reihe unterschiedlicher, jedoch gleichermaßen gültiger Ansteuerungen generieren, von denen drei beispielhaft in Abbildung 23 dargestellt sind.

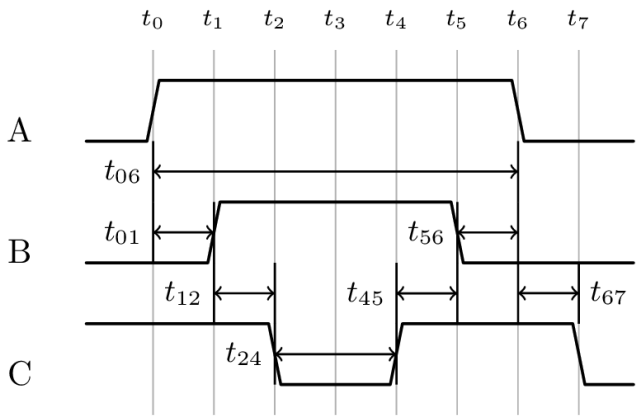


Abbildung 22: Waveform einer fiktiven DUT Zugriffsfunktion

Tabelle 5: Zeitliche Bedingungen einer fiktiven DUT Zugriffsfunktion

\ zulässige Zeiten Zeitspanne	min. [ns]	max. [ns]
t_{06}	20	50
t_{01}	0	10
t_{12}	5	-
t_{24}	10	-
t_{45}	5	10
t_{56}	0	5
t_{67}	0	5

Alle Realisierungen aus Abbildung 23 basieren auf derselben zeitlichen Beschreibung und erfüllen alle zeitlichen Randbedingungen aus Tabelle 5. Auch bei asynchronen Ansteuerungen erfordert ein FPGA jedoch intern eine getaktete Umsetzung, weshalb für die Realisierungen eine Taktperiode von 10 ns frei gewählt worden ist.

Eine Bewertung der unterschiedlichen Abläufe aus Abbildung 23 hängt vom späteren Einsatzfall ab und kann im Vorfeld nicht bestimmt werden. Somit kann die optimale Ansteuerung erst im Einsatzumfeld ermittelt werden. Sollen nicht alle Möglichkeiten einer Ansteuerung explizit spezifiziert werden, müssen sämtliche Randbedingungen und somit der vollständige Lösungsraum beschrieben werden. Hieraus lässt sich dann zum Zeitpunkt der Testsystemgenerierung eine optimale Lösung erzeugen.

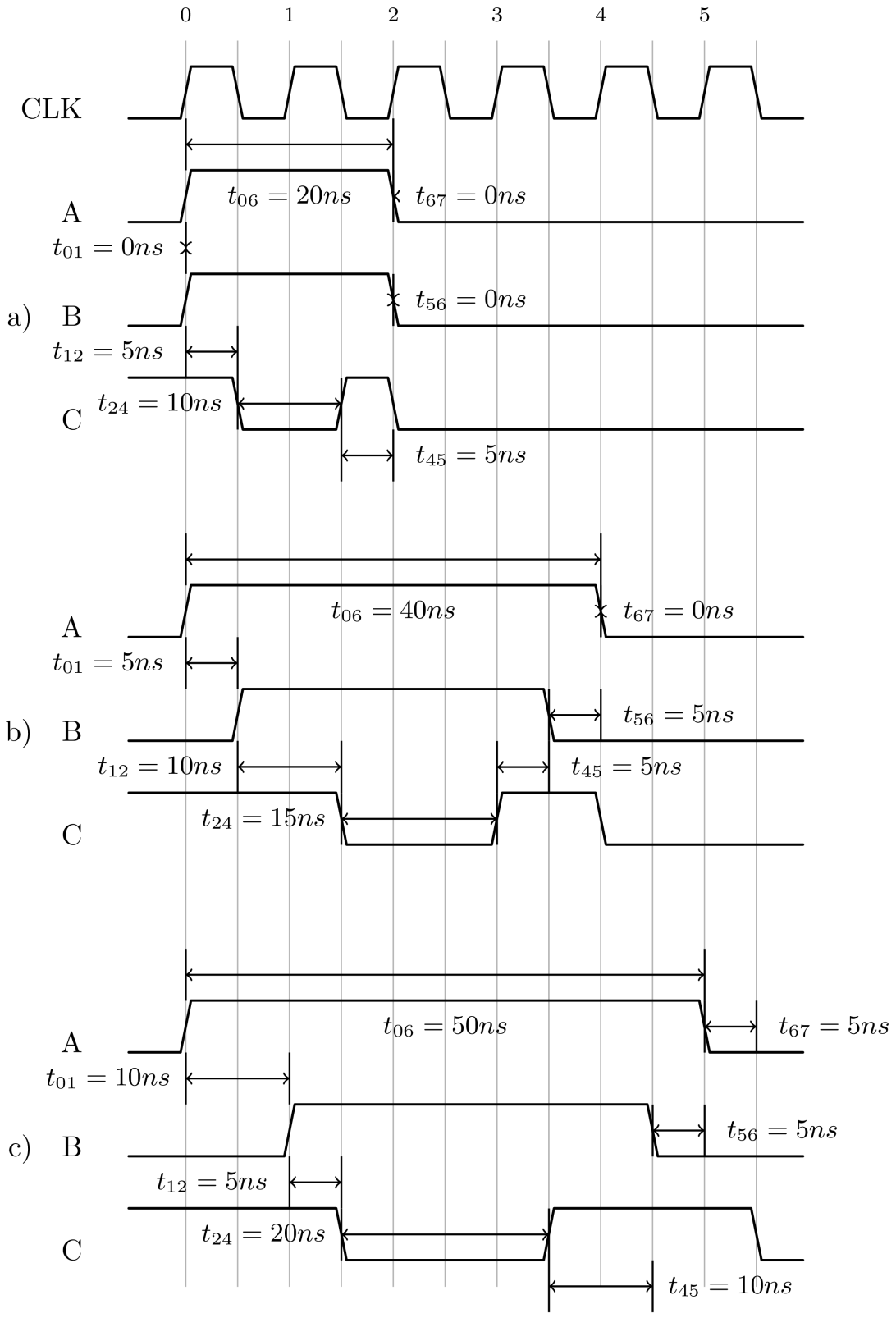


Abbildung 23: 3 mögliche Realisierungen einer fiktiven DUT Zugriffsfunktion

4.2.2. BEWERTUNG DER MODELLIERUNGSSPRACHEN

Basierend auf den Anforderungen an eine Modellierung aus Kapitel 4.2.1, werden die in Kapitel 3.3 betrachteten Modellierungssprachen im Folgenden bewertet. Auf Grundlage dieser Bewertung erfolgt in Kapitel 4.2.3 die Auswahl einer geeigneten Beschreibung, sowie die Darstellung der Vorteile und Nachteile der gewählten Sprache.

Eine Übersicht der betrachteten Sprachen und der bewerteten Eigenschaften ist in Tabelle 6 dargestellt.

Tabelle 6: Vergleichsübersicht Modellierungssprachen

Sprachen / Eigenschaften	Testsprachen							Hardware- / Automatensprachen							Test
	BSDL	SVF	STAPL	ICL	PDL	STIL	CTL	VHDL Verilog	System C	PSL	Timed State Charts	Erweiterte Petrinetze	SDL-RT	UML	CASLAN
1. Text-basiert															
2. Modelle einfach zu lesen															
3. Modelle einfach zu schreiben															
4. Direkt nutzbar für Synthesetools der FPGA Hersteller															
5. Pinbelegung des Interfaces															
6. Elektrische Interface-Parameter															
7. Zuweisungen / Vergleiche															
8. Grundbefehle (Zählen, Schieben, Addieren, Subtrahieren)															
9. Verzweigungen, Schleifen															
10. Hierarchisches- / Prozedurales-Modellieren															
11. Taktgesteuertes Zeitverhalten															
12. Kontinuierliches Zeitverhalten															
13. Hierarchisches Zeitverhalten															
14. Abdecken des gesamten Lösungsraums															

Eigenschaft erfüllt
Eigenschaft nur teilweise erfüllt
Eigenschaft nicht ausreichend erfüllt

BSDL – Boundary-Scan Description Language:

Bezüglich der Anforderungen, die an eine Modellierung gestellt werden, kann BSDL einige Forderungen erfüllen, besonders die strukturelle Sichtweise (Eigenschaften 1.-3., 5. in Tabelle 6). Dies ist jedoch nicht ausreichend, um die gestellten Anforderungen vollständig zu erfüllen. Insbesondere um Testalgorithmen mit Verzweigungen und Schleifen darzustellen (8.-9.), ist BSDL nicht geeignet. Die Möglichkeit, komplexe kontinuierliche zeitliche Zusammenhänge darzustellen (11.-14.), wie dies für die korrekte at-speed Ansteuerung von DUTs nötig ist, ist ebenfalls nicht möglich. Somit ist BSDL als Modellierungssprache nicht geeignet.

SVF / STAPL:

Sowohl SVF als auch STAPL lassen sich ebenfalls nicht für die Modellierung verwenden, da Abläufe in beiden Sprachen taktgesteuert beschrieben werden. Es sind keine absoluten Zeiten darstellbar (12.-14.). Weiterhin ist insbesondere in SVF geschriebener Code nur schwer wiederverwendbar und lesbar (2.), da weder die Verwendung von Aliassen noch Enumerationen unterstützt werden. Weiterhin erfüllen beide Sprachen die beschriebenen Anforderungen 4.-6. sowie bei SVF 8.-9. nicht ausreichend und sind folglich ungeeignet für die geforderte Modellierung.

ICL – Instrument Connectivity Language und PDL – Procedure Description Language:

Bezogen auf die Anforderungen im Rahmen dieser Arbeit bietet PDL zusammen mit ICL mehr Möglichkeiten der Modellierung als SVF und BSDL, insbesondere für Testalgorithmen und Abläufe (7-9.). Weiterhin eignen sich PDL und ICL für die Beschreibung der Verbindungen des Testprozessors und der Co-Prozessoren zu JTAG. Auf diese Weise können im FPGA genutzte Testinstrumente über eine Standardmodellierung einem Test-PC bekannt gegeben werden, was deren Nutzung vereinfacht.

Aus dem Beispiel in Anhang I, der Ansteuerung eines Testinstruments in SVF (Abbildung 74) wie auch in PDL (Abbildung 76), geht jedoch auch hervor, dass es sich hierbei um eine taktgesteuerte Programmierung handelt (11.). PDL dient zur Steuerung des Testinstruments über JTAG. Weder PDL noch ICL beschreiben das Verhalten des Testinstruments in Richtung des DUT. Ziel von PDL und ICL ist es, genau dieses Verhalten zu abstrahieren, so dass sich der Testingenieur damit nicht befassen muss. Die Modellierung von kontinuierlichen Zeiten wird von den Sprachen nicht unterstützt (12.-14.). Dies kommt daher, dass beide Sprachpaare für die Ansteuerung, Nutzung und den einfachen Austausch und die Wiederverwendung von Embedded Testinstrumenten und nicht für deren Entwurf entwickelt worden sind. Weiterhin unterstützten ICL und PDL die Anforderungen 4.-6. unzureichend, so dass somit beiden Sprachen für die Nutzung im Rahmen dieser Arbeit ungeeignet sind.

STIL – Standard Test Interface Language:

Ein kommentiertes Beispiel in Anhang J zeigt, dass STIL nicht besonders einfach zu lesen und zu schreiben ist (2.-3.). Es setzt viel Erfahrung voraus, um sicher und schnell neue Modellierungen mit STIL zu erstellen oder bestehende anzupassen.

Weiterhin ist STIL nicht darauf ausgelegt, die beschriebenen Testalgorithmen sowohl in Software als auch in Embedded Software oder Hardware umzusetzen (8.-9.). Dieses sind wesentlichen Merkmale, die einen Einsatz von STIL für die Modellierung nicht sinnvoll machen.

CTL – Core Test Language:

Obwohl CTL Mechanismen zur Verfügung stellt, Testvektoren bei der Integration eines Cores auf der nächst höheren Hierarchieebene wiederzuverwenden, kann es laut [73] das Verhalten eines Testinstruments mit komplexen Zugriffsroutinen nicht vollständig beschreiben. So fehlt CTL die Flexibilität einer Programmiersprache, um in Abhängigkeit von Antworten vorhergehender Testvektoren Entscheidungen über den weiteren Testverlauf zu treffen (8.-9.). Weiterhin ist es nicht möglich, komplexes Zeitverhalten von DUTs zu beschreiben (13.-14.). Somit ist auch CTL für eine Modellierung im Rahmen dieser Arbeit nicht geeignet.

VHDL / Verilog:

VHDL / Verilog sind Hardwaremodellierungssprachen und erfüllen daher eine ganze Reihe von Anforderungen (1.,4.,7.-12.) für die Modellierung im Rahmen dieser Arbeit, insbesondere die Möglichkeit, zeitliche Abläufe zu beschreiben Jedoch setzt dies viel Wissen des Testingenieurs voraus und deshalb sind Modelle nicht einfach zu schreiben (3.). Weiterhin fehlt die Flexibilität, zeitliche Abläufe hierarchisch zu spezifizieren (13.) und den ganzen Lösungsraum zeitlicher Abhängigkeiten abzudecken (14.).

Zusammenfassend ist VHDL bzw. Verilog gut für einen Systementwurf geeignet. Für eine Modellierung im Rahmen dieser Arbeit sind beide aber ungeeignet, da die Anforderungen 3., 13. und 14. nicht erfüllt werden.

SystemC:

SystemC bringt viele Eigenschaften mit, die für die Modellierung im Rahmen dieser Arbeit nötig sind. Insbesondere die Abstraktion einer bestimmten Realisierung in Software, Embedded Software oder Hardware ist für die Modellierung vorteilhaft. Jedoch wird für das Schreiben von korrekten Modellen immer noch viel Wissen des Testingenieurs vorausgesetzt. Somit wird die Anforderung 3 nicht erfüllt, die Sprache ist also nicht für eine Modellierung von DUTs geeignet.

PSL – Property Specification Language:

Sowohl die fehlende Beschreibung von kontinuierlichem Zeitverhalten (12.) als auch die nicht einfach zu lesende Beschreibung (2.-3.) verhindert die Nutzung von PSL für eine Modellierung im Rahmen dieser Arbeit.

Timed State Charts:

In [95] werden „Zeitliche Automaten (Timed Automata)“ verwendet, um zeitliches Verhalten darzustellen. Zur Beschreibung werden in diesem Fall UML State Charts verwendet. Die Darstellung ist somit sowohl graphisch als auch textuell möglich. Jedoch ist die textuelle Beschreibung eher für den digitalen Datenaustausch als für die Erstellung und Bearbeitung solcher Modelle gedacht. Entsprechend schlecht ist eine solche Darstellung zu lesen bzw. zu schreiben. Aufgrund der mangelhaften Unterstützung von Eigenschaft 3. ist diese Methode für die gesuchte Modellierung nicht geeignet.

Erweiterte Petri-Netze:

Auch wenn erweiterte Petri-Netze formal textuell beschrieben werden können, so werden sie doch zumeist graphisch entworfen. Insbesondere komplexe zeitliche Abhängigkeiten sind nur mit viel Erfahrung sicher in ein entsprechendes Petri-Netz zu überführen (3.). Hinzu kommen fehlende Möglichkeiten, das Interface eines DUTs zu beschreiben (5.-6.). Somit sind auch erweiterte Petri-Netze trotz umfangreicher Modellierungsmöglichkeiten (7.-14.) nicht ausreichend geeignet für die geforderte Modellierung von Testinstrumenten.

SDL-RT:

Eine Ausarbeitung, die sich ausführlich mit dem Thema der Modellierung von Systemen mit harten Echtzeitanforderungen in SDL befasst, um daraus Hardwarebeschreibungen für die FPGA Synthese zu generieren, ist [96]. Diese Arbeit zeigt sehr ausführlich die Möglichkeiten der Modellierung, macht jedoch zugleich die Komplexität bei der in Kapitel 4.2.1 geforderten Beschreibung sichtbar.

In [86] werden drei Ansätze beschrieben, wie SDL durch RT erweitert werden kann. Für einen Einsatz mit den hier geforderten Randbedingungen, insbesondere der einfachen Beschreibung, sind diese Methoden jedoch ungeeignet, da die Beschreibung bei der Darstellungen von DUT Signalverläufen unübersichtlich wird.

In [79] hat sich der Autor mit der Eignung dieser Beschreibung zur Modellierung von Testinstrumenten beschäftigt. Er schreibt hierzu: „*Der SDL-RT Standard beschreibt neben der graphischen Repräsentation der Diagramme auch eine textuelle Form. Diese folgt dem XML*[36] *Standard und wird in einer Document Type Definition (DTD) definiert. Damit bietet SDL-RT sowohl eine Möglichkeit, die zeitlichen Abhängigkeiten […] zu beschreiben als auch diese im DUT-Modell in textueller Form darzustellen. Allerdings ist diese Darstellung in XML-Form nicht besonders anschaulich. Außerdem wurde SDL-RT konzipiert, um komplette Systeme zu beschreiben, was in der*

[36] XML: Extensible Markup Language (engl. „erweiterbare Auszeichnungssprache“)

Anwendung im DUT-M in der Form von Beschreibungen der Abhängigkeiten auf einer niedrigen Abstraktionsebene zu viel Overhead führen würde“.

Folglich bietet SDL-RT zwar viele Möglichkeiten zur Modellierung zeitlicher Abhängigkeiten und algorithmischer Abläufe (7.-14.), ist jedoch für die geforderte Modellierung im Rahmen dieser Arbeit trotzdem nicht geeignet ist, da Modelle in Textform schwer zu schreiben sind (3.).

UML:

UML bringt durch entsprechende Erweiterungen und viele Notationselemente die Möglichkeit mit, Bedingungen, Attribute, Nachrichten, Zeitbedingungen und Zeitverlaufslinien ausreichend zu beschreiben (7.-14.). Für die Modellierung von Testinstrumenten hat [79] UML untersucht und ist dabei zu folgender Erkenntnis gekommen: „*Die zweite Version der Unified Modeling Language (UML 2) bietet Methoden zur Modellierung von technischen Systemen mit Echtzeitanforderungen. Dazu gehört unter anderem das neu eingeführte Timing-Diagramm, eine spezielle Version des Sequenzdiagramms, mit welchem das Zeitverhalten von Objekten und Systemen genau beschrieben werden können soll*“.

Es fehlt jedoch an einer geeigneten textuellen Umsetzung (2.-3.), die es erlaubt, intuitiv und einfach DUT Modelle zu lesen und zu erstellen.

CASLAN:

Die Sprache CASLAN erlaubt das Erstellen einfacher Abfragen, unterstützt Schleifen oder mathematische Operationen aber nur unzureichend. Weiterhin ist eine zeitliche Modellierung nicht vorgesehen und es können nur taktbasierte Verzögerungen verwendet werden. Die Anforderungen 12.-14 werden folglich nicht erfüllt, deshalb ist auch diese Sprache für die geforderten Modellierung ungeeignet.

Weitere Hardwaremodellierungssprachen:

Die zunehmende Komplexität der heutigen Hardware macht die Entwicklung auf einem höherem Abstraktionsgrad notwendig, d.h. es reicht oft nicht aus, die Hardware auf der relativ niedrigen Register Transfer Ebene (RTL) zu beschreiben, so dass Hochsprachenbeschreibungen der Hardware notwendig werden, die dann entweder in eine RTL-Beschreibung oder direkt in eine Netzliste transformierbar sind. Mit der Beschreibung der Hardware auf hoher Abstraktionsebene ist es dem Hardware-Entwickler möglich, sich vollständig auf den zu realisierenden Algorithmus zu konzentrieren. Dies eliminiert das "in Hardware denken" des Designers. Somit kann Hardware direkt wie in der Software-Entwicklung programmiert werden, was einerseits die Hardwareentwicklung beschleunigt und andererseits keine Hardwareerfahrung des Entwicklers voraussetzt. Die heutigen Beschreibungen von Hardware lassen sich prinzipiell in textuelle und visuelle (graphische) Beschreibungen unterteilen.

Neben den oben beschriebenen Sprachen sind noch weitere Sprachen für die Hardwareentwicklung betrachtet worden. Für die textuelle Beschreibung kommen in breitem

Maße C-ähnliche Sprachen zur Anwendung. Diese Sprachen werden seit 1980 vorgeschlagen und sind in vielen Veröffentlichungen (unter anderem [97] [98] [99] [100]) diskutiert.

C ist eine unter Hardware- und Software-Entwicklern bekannte und beliebte Sprache, mit welcher auch sehr hardwarenah programmiert werden kann. Eine weitere Motivation für C ist, dass heutige Systeme gewöhnlich aus Hardware- und Software-Teilen bestehen und es anfangs oft nicht vollständig definiert ist, welche Teile des Systems wie implementiert werden sollen. Eine gemeinsame Modellierungssprache vereinfacht in diesem Falle die Entwicklung. Einige der verwendeten C-ähnlichen Sprachen neben SystemC (siehe Kapitel 3.3.1.8) wie Transmogrifier C oder DIME-C synthetisieren ein Subset von AnsiC, andere wie Impulse-C, Handel-C, SpecC oder BachC erweiten C um Konstrukte für die Hardware-Entwicklung und c2Verilog unterstützt den AnsiC Standard fast vollständig.

Allen diesen Sprachen ist gemeinsam, dass sie die Parallelität von Prozessen, wie sie in Hardware ablaufen, ungenügend oder gar nicht beschreiben und keine exakte Modellierung der Zeit stattfindet. Außerdem muss es einen Mechanismus geben, welcher Zeit exakt in Clock-Zyklen überführt. Die Aufspaltung in parallele Prozesse wird in den meisten Fällen implizit von Compiler übernommen. Dieses gilt auch für die Clock-Zyklen und Zeitvorgaben wie im Fall von Transmogrifier C oder es werden explizite Zeiten spezifiziert, welche entweder Teil der Sprache sind, wie in Handel-C und HardwareC oder außerhalb der Sprache angegeben werden, wie in C2Verilog.

Für eine Modellierung im Rahmen dieser Arbeit sind somit alle diese Arten von Sprachen ungeeignet, da die Anforderungen 12.-14. nicht erfüllt werden.

Fazit:

Die untersuchten Modellierungssprachen decken alle bestimmte Eigenschaften, der geforderten Modellierung ab, jedoch erfüllt keine der Sprachen alle Anforderungen vollständig (siehe hierzu Tabelle 6). Während Sprachen wie BSDL, SVF, STAPL, ICL oder PDL einfach zu schreiben sind, fehlen ihnen bestimmte Eigenschaften für die Darstellung von Testalgorithmen oder elektrischen Eigenschaften der DUTs.

Die Sprachen STIL, CTL, VHDL/Verilog sowie SystemC und PSL eignen sich zwar besser zur Modellierung von Testalgorithmen und den elektrischen Eigenschaften der DUTs, sie sind jedoch für den geforderten Anwendungsfall nicht mehr einfach zu verwenden und erfordern vom Testingenieur ein ausgeprägtes Verständnis in der Hardwareentwicklung. Außerdem fehlt ihnen eine ausreichende Möglichkeit der Modellierung von komplexem zeitlichem Verhalten.

Während es mit den Timed State Charts, erweiterten Petrinetzen, SDL-RT sowie UML und seinen Erweiterungen möglich ist, auch komplexes zeitliches Verhalten zu beschreiben, so sind diese Methoden und Sprachen jedoch durchweg kompliziert in der Anwendung und erfordern viel Wissen vom Testingenieur.

Die Sprache CASLAN, die speziell für Tests entwickelt worden ist, kann auch einige der geforderten Eigenschaften erfüllen. Da die Sprache bisher hauptsächlich mit Boundary-Scan eingesetzt worden ist und keine FPGA-Bitfiles generiert worden sind, war es jedoch unter anderem nicht nötig, die elektrischen Eigenschaften der DUTs zu beschreiben. Während diese Eigenschaft sich noch relativ leicht ‚nachrüsten' lassen würde, ist die Modellierung von komplexen zeitlichen Zusammenhängen, insbesondere die Modellierung des gesamten Lösungsraum für eine gültige DUT Ansteuerung, gar nicht vorgesehen. Es existiert keinerlei Modellierungskonzept hierfür in CASLAN, weshalb auch diese Sprache nicht eingesetzt werden kann.

Für eine vollständige DUT Modellierung müssen Algorithmen für Tests ebenso wie auch elektrische Parameter und zeitliche Abhängigkeiten definiert werden. Dies erfordert es, unterschiedlichste Modellierungen zu vereinen. Dies wäre durch die Kombination mehrerer Sprachen zwar möglich, jedoch muss dann der Testingenieur in diesem Fall viele ggf. unterschiedliche Modellierungssprachen verinnerlichen. Dies führt als Konsequenz dazu, einen eigenen Modellierungsansatz mit einer eigenen Modellierungssprache zu verwenden, welcher im folgenden Kapitel 4.2.3 vorgestellt wird.

4.2.3. GEWÄHLTER MODELLIERUNGSANSATZ

Im folgenden Abschnitt wird der gewählte Modellierungsansatz genauer erläutert. Die Entscheidung basiert auf den definierten Anforderungen und der Bewertung der untersuchten bestehenden Modellierungssprachen.

Neben der gewählten Struktur der Modellierung wird auch die genutzte Sprache vorgestellt, sowie auf die Besonderheiten der Anforderungen für die Modellierung von hardwarenahen Schichten entsprechend des in Kapitel 4.1.1 vorgestellten Ebenenkonzepts näher eingegangen.

Die Tabelle 6 zeigt, dass als einzige Sprache VHDL bzw. Verilog direkt von allen FPGA Herstellern als Sprache für die FPGA Synthese unterstützt wird. D.h. aus Beschreibungen in VHDL bzw. Verilog kann direkt ein FPGA-Bitfile generiert werden. Da die FPGA Synthese weiterhin durch die Tools der FPGA Hersteller erfolgen soll, wird VHDL als Zwischensprache für die Testsystemerstellung gewählt. Somit braucht die Anforderung der „direkten Nutzbarkeit durch ein Synthesetool" von den Modellierungssprachen nicht mehr erfüllt zu werden. Stattdessen reicht es aus, wenn aus der gewählten DUT Modellierungssprache VHDL bzw. Verilog generiert werden kann.

Für die Entwicklung einer eignen Sprache stellt CASLAN eine gute Grundlage dar, da diese Sprache viele der grundlegenden Forderungen an eine Modellierung erfüllt. So bietet CASLAN neben der Tatsache, einfach zu lesen und zu schreiben zu sein, sowie ein paar weitere Aspekte der geforderten Modellierung abzudecken, insbesondere den Vorteil, bereits von einigen Testtools unterstützt zu werden. Dies ist der ideale Ausgangspunkt für eine eigene Entwicklung, die sich an dieser Sprache orientiert und die fehlenden Modellierungen in einer geeigneten Form realisiert.

Die Entwicklung einer eigenen Sprache erleichtert die Darstellung und Speicherung aller Funktionen, die für das Testen der Verbindungen zu einem DUT notwendig sind, innerhalb einer Datei. Dies ist vorteilhafter als eine getrennte Modellierung für Funktionen, die in Hardware bzw. Software umgesetzt werden sollen. Die daraufhin entwickelte Sprache sowie deren Struktur, Syntax und Merkmale, um die geforderten Modellierungen zu ermöglichen, sind im folgenden Kapitel näher erläutert.

4.2.3.1. STRUKTUR UND SPRACHE

Im Vordergrund der Entwicklung stand eine Programmiersprache, die es erlaubt, Testalgorithmen sowie auch hardwarenahe Steuerfunktionen der DUTs auf einfache Weise zu beschreiben. Hierbei sollte die Beschreibung frei von Kenntnissen über die später gewählte Implementierung des DUTs erfolgen.

Die gewählte Lösung wurde *RTDL – ROBSY Test Description Language* genannt und ist von der Syntax her an die Sprache CASLAN angelehnt, wurde aber an mehreren Punkten verändert und erweitert, so dass die nötigen Modellierungen möglich sind. Es ist die Absicht von RTDL, sowohl Implementierungsdetails der FPGA Umsetzung zu verstecken als auch die Wiederverwendbarkeit von Funktionen für unterschiedliche Testfälle zu ermöglichen. In RTDL geschriebene Modelle sollen als Eingangsdaten für eine automatisierte Generierung von Testinstrumenten dienen. Die Erweiterte Backus-Naur-Form (EBNF) von RTDL ist in Anhang E zu finden. Ein kommentiertes RTDL Beispiel ist Anhang L (Interfaces und L1) und Anhang M (L2 bis L5) zu entnehmen.

Die Struktur von RTDL besteht aus 8 Bereichen, die jeweils durch Schlüsselworte eingeleitet werden (siehe Abbildung 24 links). Die zugehörigen Ebenen des Ebenenmodells (mitte) sowie das Verhältnis von RTDL zu CACLAN (rechts) sind ebenfalls dargestellt. Im Folgenden werden die Inhalte der einzelnen Bereiche näher erläutert.

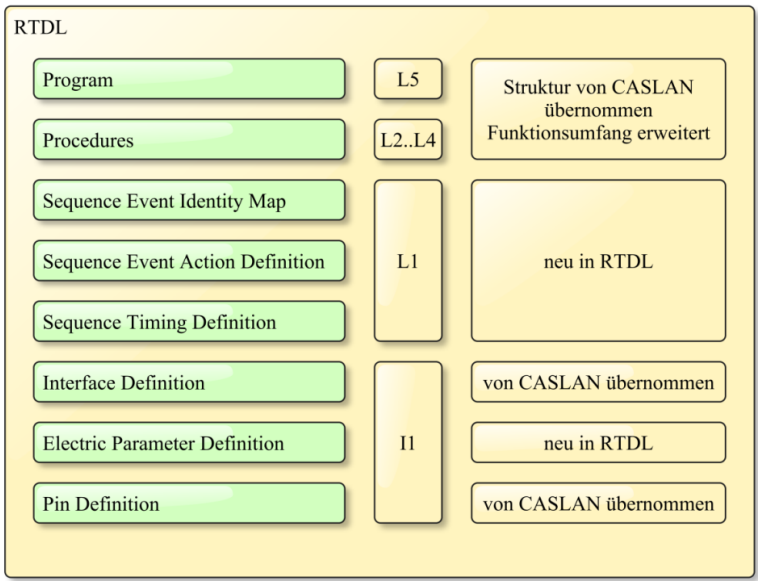


Abbildung 24: Modellierungsstruktur

Im Abschnitt **Pin Definition** wird die Zuordnung jedes Signals zu einem physikalischen Pin am Gehäuse des Schaltkreises gemacht. Dieser Abschnitt unterscheidet sich in Abhängigkeit vom verwendeten Gehäuse des DUTs.

Im Abschnitt **Electric Parameter Definition** werden die für die Synthese des FPGA-Bitfiles nötigen Angaben zu den elektrischen Anforderungen der im Abschnitt Interface definierten Signale festgelegt. Hierzu gehören Angaben, wie der zu verwendende I/O-Standard, die nötige Treiberstärke oder die nötige Flankensteilheit.

Im Abschnitt **Interface Definition** wird die Schnittstelle des DUT beschrieben. Hierzu gehört die Definition der vorhandenen Zugriffssignale und ggf. deren Zuordnung zu einzelnen Bussen. Auch wird an dieser Stelle das Verhalten der Signale bei Inaktivität festgelegt, und es werden ggf. weitere Signalabhängigkeiten definiert. Dies ist zum Beispiel bei bidirektionalen Leitungen wie Datenbussen notwendig. Die definierten Signale und Busse stehen später sowohl dem Testprozessor als auch den Co-Prozessoren als Schnittstelle für die Ansteuerung des DUTs zur Verfügung.

Im Abschnitt **Sequence Timing Definition** wird für jede Zugriffsroutine des DUTs das genaue Timing hinterlegt. Mit Hilfe dieser Angaben ist es möglich, das zu generierende Testinstrument so zu spezifizieren, dass es beim späteren Einsatz des DUTs automatisch an die genaue Einsatzumgebung angepasst werden kann. In diesem Abschnitt wird somit der gesamte Lösungsraum für jede Zugriffsroutine beschrieben. Es stellt das zentrale Element der Modellierung von zeitlichen Abhängigkeiten dar und wird daher ausführlicher in Kapitel 4.2.4 erläutert.

Im Abschnitt **Sequence Event Action Definition** werden zu jeder Sequence Timing Definition die dazugehörigen auszuführenden Aktionen definiert. Dies erlaubt es, Aktionen separat von den dazugehörigen zeitlichen Abhängigkeiten übersichtlich zu beschreiben. Die Sequence Event Actions werden zusammen mit dem Sequence Timing in Kapitel 4.2.4 ausführlich beschrieben.

Im Abschnitt **Sequence Event Identity Map** wird bei einer hierarchischen Timing-modellierung die Verknüpfung über die einzelnen Hierarchieebenen dargestellt. Näheres hierzu ist in Kapitel 4.2.7 erläutert. Zusammen mit der Sequence Timing Definition und der Sequence Event Action Definition stellen diese drei Abschnitte die vollständige Modellierung der Ebene L1 des Ebenenmodells dar.

Diese Ebene L1 beschreibt die hardwarenahe Funktionalität. Die Funktionen dieser Ebene werden ausschließlich durch das DUT bestimmt und sind unabhängig von den anzuwendenden Testalgorithmen. Um einen Test entsprechend der Spezifikation des DUTs durchzuführen, ist es wichtig diese entsprechend zu modellieren. Dies betrifft insbesondere die genauen Zeiten, mit denen die Zugriffsfunktionen ausgeführt werden müssen.

Während eine Ausführung in Software in Abhängigkeit des verwendeten Test-prozessors zumeist sequentiell erfolgt, ist eine Ausführung in Hardware beliebig parallel

möglich. Das heißt, bei der Modellierung muss festgelegt werden, welche Elemente parallelisiert werden können, um bei einer Umsetzung in Hardware nicht die Nachteile einer sequentiellen Bearbeitung zu übernehmen. Von einer getrennten Modellierung für Hardware und Software ist abzuraten, da in diesem Fall zwei ggf. voneinander abweichende Beschreibungen der gleichen Funktion im Modell vorhanden sein würden.

Im Abschnitt **Procedures** werden die Testalgorithmen der Ebenen L2 bis L4 des Ebenenkonzepts definiert.

Im Abschnitt **Programm** wird die oberste Ebene jedes DUT Modells definiert. Diese Ebene L5 wird beim Einsatz von mehreren DUT Modellen innerhalb eines Testsystems zusammengefasst, so dass es für jede Implementierung eines Testsystems immer genau eine Ebene L5 gibt, die den gesamten Testablauf koordiniert.

Mit RTDL ist es möglich, die einzelnen Teile eines Testsystems unabhängig voneinander zu beschreiben, in Bibliotheken bereitzustellen und dann daraus das gesamte Testsystem unter Berücksichtigung der jeweils gültigen Randbedingungen des verwendeten Prüflings automatisch abzuleiten. Dies ist nötig, da jedes strukturell zu testende System anders ist und somit die Testsystemgenerierung nicht vom Entwickler der Testtools gemacht werden kann, sondern vom Testingenieur, der den Test durchführt, zu erfolgen hat.

4.2.4. TIMINGMODELLIERUNG

In diesem Abschnitt wird die gewählte Timingmodellierung im Detail beschrieben. Diese erlaubt eine flexible Darstellung von zeitlichen Schranken und stellt den gesamten Lösungsraum für DUT Zugriffsroutinen dar, statt nur eine einzige Lösung zu spezifizieren.

Für eine kompakte und übersichtliche Darstellung innerhalb der RTDL Modelle wurde eine sogenannte Difference Bound Matrix (DBM) für die Modellierung von zeitlichen Abhängigkeiten gewählt. Difference Bound Matrizen modellieren zeitliche Abhängigkeiten, indem für jedes Ereignis sowohl die minimale als auch maximale Zeit zu anderen Ereignissen notiert wird. Sie sind einfach zu beschreiben und können textuell gut dargestellt werden. Diese Matrizen sind seit vielen Jahren bekannt und dienen unter anderem der finiten Darstellung von zeitlichen Automaten, finden aber auch im Bereich des Model-Checkings ihren Einsatz [101] [102] [103].

In Anlehnung an eine Präsentation in [104] soll nachfolgend ein kurzes Beispiel für eine einfache Modellierung dargestellt werden, um das Prinzip der Difference Bound Matrix zu veranschaulichen. Hierbei wird vom Anwendungsfall der DUT Ansteuerung ausgegangen und ein einzelnes Signal mit zwei Ereignissen und zwei zeitlichen Randbedingungen vorgegeben. Diese Randbedingungen sind:

- der generierte Impuls soll mindestens $10\ ns$ lang sein
- der generierte Impuls darf maximal $20\ ns$ lang sein

Der in Abbildung 25 gezeigte Signalverlauf entspricht dem in Abbildung 26 dargestellten Automaten. Im Zustand A erfolgt die Ausgabe des Signals $y = 0$, sowie im Zustand B entsprechend $y = 1$. Weiterhin gelten $t_o = 0\,ns$ sowie $t_1 \geq 10\,ns$ und $t_1 \leq 20\,ns$ bzw. $10\,ns \leq t_1 \leq 20\,ns$ als zeitliche Randbedingungen. Als Startbedingung in Abbildung 26 muss ein Signal ‚request' auf 1 gesetzt sein.

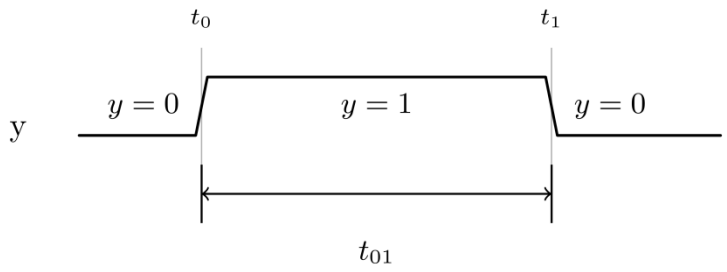


Abbildung 25: Signalverlauf für DBM Beispiel

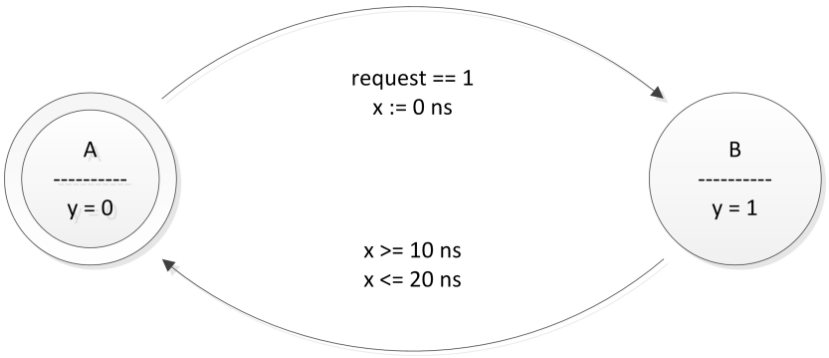


Abbildung 26: FSM für DBM Beispiel

In einer Difference Bound Matrix lassen sich diese zeitlichen Abhängigkeiten übersichtlich notieren. Hierfür wird eine Difference Bound Matrix $A \in (\mathbb{R} \cup \#)^{n \times n}$ [37] definiert. Das Kodieren der Randbedingungen erfolgt immer getrennt für jeweils zwei Ereignisse E_i und E_j mit den zugehörigen Zeitpunkten t_i bzw. t_j, sowie deren zeitlichen Abhängigkeiten.

Die Elemente der Matrix heißen $A_{i,j}$. Die Indizes der Matrix werden als *i* und *j* definiert, wobei *i* die Zeilennummer beschreibt und *j* entsprechend die Spalte. Gemäß der Definition der Difference Bound Matrix mit $t_j - t_i \leq C$ entspricht der Eintrag $A_{i,j} = C$ einer oberen Schranke bzw. einer maximal Zeitvorgabe von $t_j - t_i \leq A_{i,j}$. Es darf somit zwischen den Zeitpunkten t_j und t_i maximal eine Zeit $A_{i,j}$ vergehen. Es gilt $A_{i,j} \geq 0$.

[37] Das Zeichen # dient als Platzhalter und stellt die Abwesenheit entsprechender Randbedingungen zwischen zwei Ereignissen dar, während n hierbei die Anzahl der vorher definierten Ereignisse ist.

Entsprechendes gilt für minimale Zeitvorgaben in der Form $t_j - t_i \geq D$, die umgeformt $t_i - t_j \leq -D$ ergibt. Somit ergibt sich $A_{j,i} = -D$ als untere Schranke und es gilt $A_{j,i} \leq 0$.

Wird entsprechend dieser Definitionen $t_i = t_0 = 0$ und $t_j = t_1$ gesetzt, so ergibt sich für die maximale Zeitvorgaben folgende Darstellung:

$$\begin{aligned} & t_j - t_i \leq A_{i,j} \\ \rightsquigarrow\ & t_1 \leq A_{0,1} \end{aligned} \tag{4.1}$$

Entsprechend gilt dies für die Definition von minimalen Zeitvorgaben.

$$\begin{aligned} & t_i - t_j \leq A_{j,i} \\ \rightsquigarrow\ & -t_1 \leq A_{1,0} \\ \rightsquigarrow\ & t_1 \geq |A_{1,0}| \end{aligned} \tag{4.2}$$

Die Abbildung 27 enthält somit eine zeitlich vollständige Repräsentation des beschriebenen Beispiels mit $A_{0,1} = 20\text{ns} \geq 0$ und $A_{1,0} = -10\text{ns} \leq 0$. Abbildung 28 stellt einen Ausschnitt des zulässigen Lösungsraums für das Beispiel in zweidimensionaler Form dar.

X	t_0	t_1
t_0	#	20
t_1	-10	#

Abbildung 27: DBM Beispiel

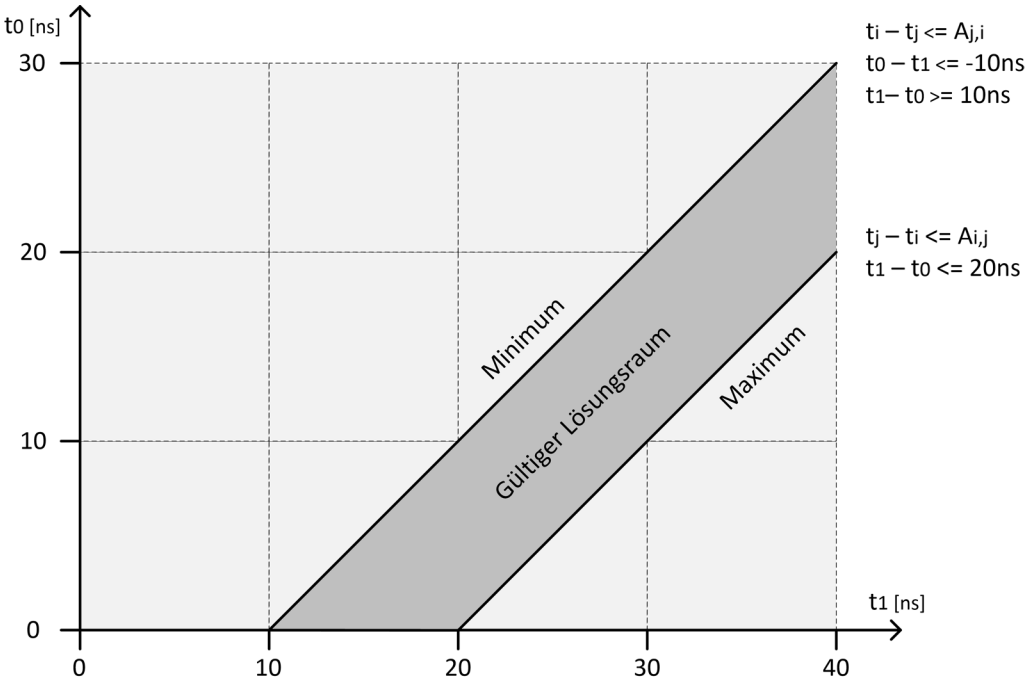


Abbildung 28: 2D Repräsentation des DBM Beispiels

Neben der einfachen Modellierung von zeitlichen Zusammenhängen in DBMs, wie sie der vorliegenden Arbeit zugrunde liegt, gibt es in der Literatur noch weitere Verwendungszwecke für diese Art der Darstellung.

So wird in [105] eine Methode vorgestellt, wie mittels erweiterter DBMs automatisch die Korrektheit von Echtzeitsystemen mittels Model-Checking von Zeitlichen Automaten geprüft wird.

In [106] werden DBMs dazu genutzt, zeitliche Abläufe zu koordinieren und zu optimieren. Hierbei handelt es sich im beschriebenen Fall um die Entwicklung von Online Schedulern in biologischen Analysesystemen.

In [107] wird das Problem des Model-Checkings bei Echtzeitsystemen beleuchtet, bei denen sich mehrere Prozesse eine Ressource teilen müssen. Hierbei geht es insbesondere um Subtraktionsoperationen bei Difference Bound Matrizen und die dabei entstehenden Probleme.

Auch wenn alle diese Artikel DBMs zum Prüfen und Optimieren von zeitlichen Abhängigkeiten nutzen, so ist die Generierung von Co-Prozessoren für Testinstrumente auf FPGAs mittels DMBs ein neuer Einsatzfall und in der Literatur bisher nicht dokumentiert.

4.2.5. INFORMATIONSQUELLEN

Die beschriebene Timingmodellierung der DUTs mittels Difference Bounded Matrizen soll ausschließlich auf den Informationen für die Ansteuerung der DUTs basieren und keine weiteren Informationen über den Prüfling oder die zu verwendenden Testparameter enthalten. Dies würde den Einsatz einschränken, da bereits zum Zeitpunkt der Entwicklung des DUT Modells Informationen über den späteren Einsatz des DUTs nötig wären.

Für die Schnittstellendefinition und die Ansteuerung der DUTs sind Daten vom Hersteller des DUTs notwendig. Diese können entweder in digitaler Form zum Beispiel als Modell in einer Hardwarebeschreibungssprache vorliegen oder aber als Datenblatt. Unabhängig von der Form der Informationen müssen die für einen Test relevanten Ansteuerungen eines DUTs aus diesen Daten extrahiert werden. Im Folgenden werden zwei unterschiedliche Informationsquellen untersucht und auf ihre Eignung hin bewertet.

4.2.5.1. SIMULATIONSMODELL

Einige Hersteller, insbesondere die von Speicherschaltkreisen, stellen für ihre Schaltkreise Simulationsmodelle zur Verfügung, die es ermöglichen, in einer Simulation das korrekte Verhalten der selber entworfenen Ansteuerung sowohl logisch als auch zeitlich zu prüfen. In [108] wird ein solches Modell für einen DDR2 Schaltkreis untersucht. Dieses Modell wurde von Hersteller Micron als Verilog Modell zur Verfügung gestellt.

Hierbei ist zu erwähnen, dass die Funktionen und Parameter im Modell in zwei getrennten Dateien definiert werden. So wird in der Parameterdatei z.B. festgelegt, welcher Speedgrade oder welche Latency verwendet wird oder wie viele Bänke der Speicher nutzt. Viele dieser Parameter können durch eine einfache Suche im Funktionsmodell zugeordnet werden. Erschwert wird dieser Prozess jedoch dadurch, dass einige Timings als Picosekunden andere als Faktor, bezogen auf den gewählten Taktzyklus angegeben werden. Die korrekte Interpretation jedes Parameters ist nur durch die Analyse der Kommentare hinter den Platzhaltern im Funktionsmodell möglich. Ebenso ist bei den Parametern nicht offensichtlich, ob es sich um eine Minimale- oder Maximale-Schranke handelt. Hierfür ist der Kontext zu prüfen, in dem der Parameter genutzt wird.

Untersuchungen in [108] haben gezeigt, dass es teilweise sehr schwierig ist, die zeitlichen Randbedingungen aus dem Verilog-Modell korrekt als Minimum- bzw. Maximumrestriktionen zu erkennen. Insbesondere die Zuordnung der Randbedingungen zu den Signalflanken von Takt-, Daten- und Steuerleitungen ist nicht trivial. Zwei einfache Beispiele verdeutlicht diese Problematik in Abbildung 29 (aus [108]).

- t_{DSH}
  - Textsuche nach „TDSH“ in ddr2_parameters.vh ergibt :

```
[...]  TDSH = 0.20; // tDSH tCK [...]
```

  - $t_{DSH} = 0.2 * 3.0ns = 600ps$
  - Textsuche nach „TDSH“ in ddr2.v ergibt :

```
if  ($time − tm_ck_pos < $rtoi(TDSH∗tck_avg))
    [...]  ERROR: tDSH violation [...]
```

  - t_{DSH} ist Minimumrestriktion

- t_{DQSS}
  - Textsuche nach „TDQSS“ in ddr2_parameters.vh ergibt :

```
[...]  TDQSS = 0.25; // tDQSS tCK [...]
```

  - $t_{DQSS} = 0.25 * 3.0ns = 750ps$
  - Textsuche nach „TDQSS“ in ddr2.v ergibt :

```
if  ((tm_tdqss < tck_avg/2.0) && (tm_tdqss > TDQSS∗tck_avg))
    [...]  ERROR: tDQSS violation[...]
```

  - t_{DQSS} muss größer als der halbe Taktzyklus (1500ps) und kleiner als 750ps sein, also mindestens 1500ps und höchstens 750ps

Abbildung 29: Parameterextraktion für DDR2 Modell [108]

4.2.5.2. DATENBLATT

Eine weitere Möglichkeit, die Daten für die Ansteuerung eines DUTs zu gewinnen ist es, die Datenblätter der Schaltkreishersteller zu verwenden. Diese beinhalten eigentlich[38] alle nötigen Informationen zur Ansteuerung des DUT. In ihnen sind alle Schnittstellen sowie die erlaubten Zugriffsroutinen mit den zeitlichen Abhängigkeiten dargestellt.

In [108] wurde neben der Extraktion zeitlicher Abhängigkeiten aus Simulationsmodellen auch die Verwendung von Datenblättern am Beispiel von DDR2 SDRAM untersucht. Das Ergebnis der Untersuchungen zeigt, dass zwar auch bei diesem Verfahren auf evtl. vorhandene Randbemerkungen wie Fußnoten geachtet werden muss, um die Zeiten korrekt zu interpretieren, jedoch ist generell die Parameterextraktion und somit ein

[38] Man muss in diesem Fall *eigentlich* sagen, da viele Datenblätter weder vollständig noch korrekt sind.

Füllen der Difference Bound Matrix einfacher möglich als die Extraktion aus den Simulationsmodellen.

Die ausschließliche Verwendung von Simulationsmodellen für die Modellierung ist nicht möglich, da diese nicht von jedem Hersteller angeboten werden. Somit muss die Extraktion aus Datenblättern von der hier vorgestellten Modellierungsmethode zwangsläufig unterstützt werden, da diese Art der Dokumentation für alle Schaltkreise vorhanden ist[39]. Die Verwendung eines Simulationsmodells in der bisherigen Form hat gegenüber der Verwendung eines Datenblatts keine signifikanten Vorteile. Daher wird im Folgenden nur die Modellierung mittels Datenblatt unterstützt.

Die Datengewinnung für das in dieser Arbeit beispielhaft verwendete SRAM wird in Kapitel 4.2.6 gezeigt. Es werden alle nötigen Informationen des DUT Modells erfasst und in RTDL umgesetzt. Da diese Arbeit sich vorrangig mit dem Konzept der Modellierung und der Generierung der Co-Prozessoren befasst, und sich insbesondere mit der Einhaltung zeitlicher Abhängigkeiten beschäftigt, beschränkt sich der beschriebene Modellierungsablauf mit der Umsetzung auf die Ebene L1 und das Interface I1 des Ebenenkonzepts.

4.2.6. MODELLIERUNGSABLAUF

In diesem Abschnitt wird die Modellierung eines DUTs, insbesondere die Modellierung von zeitlichen Abhängigkeiten, beispielhaft dargestellt. Hierfür wird der Schreibzugriff auf ein SRAM des Typs IS61LV25616 beschrieben. Das Datenblatt ist in [22] zu finden. Der hier vorgestellten Modellierung und den Regeln, nach denen die Difference Bound Matrix generiert wird, liegt eine Ausarbeitung [109] zugrunde, die im Rahmen des Projekts ROBSY erfolgt ist.

Für eine Modellierung ist eine Tabelle mit den zeitlichen Parametern (Abbildung 30) sowie ein Waveformdiagramm (Abbildung 31) notwendig. Diese zeigen Abhängigkeiten zwischen bestimmten Ereignissen der betroffenen Steuer-, Adress- und Datensignale.

[39] Bei einigen Schaltkreisherstellern gibt es diese Unterlagen nicht frei zugänglich, sondern nur unter bestimmten Auflagen. Auf diese Problematik soll im Rahmen dieser Arbeit nicht eingegangen werden.

Symbol	Parameter	-10 Min.	-10 Max.	-12 Min.	-12 Max.	Unit
t_{WC}	Write Cycle Time	10	—	12	—	ns
t_{SCE}	$\overline{CE}$ to Write End	8	—	8	—	ns
t_{AW}	Address Setup Time to Write End	8	—	8	—	ns
t_{HA}	Address Hold from Write End	0	—	0	—	ns
t_{SA}	Address Setup Time	0	—	0	—	ns
t_{PWB}	$\overline{LB}$, $\overline{UB}$ Valid to End of Write	8	—	8	—	ns
t_{PWE1}	$\overline{WE}$ Pulse Width	8	—	8	—	ns
t_{PWE2}	$\overline{WE}$ Pulse Width ($\overline{OE}$ = LOW)	10	—	12	—	ns
t_{SD}	Data Setup to Write End	6	—	6	—	ns
t_{HD}	Data Hold from Write End	0	—	0	—	ns
t_{HZWE}	$\overline{WE}$ LOW to High-Z Output	—	5	—	6	ns
t_{LZWE}	$\overline{WE}$ HIGH to Low-Z Output	2	—	2	—	ns

Abbildung 30: Zeitliche Parameter für Back-to-Back Write verschiedener Speedgrades [22]

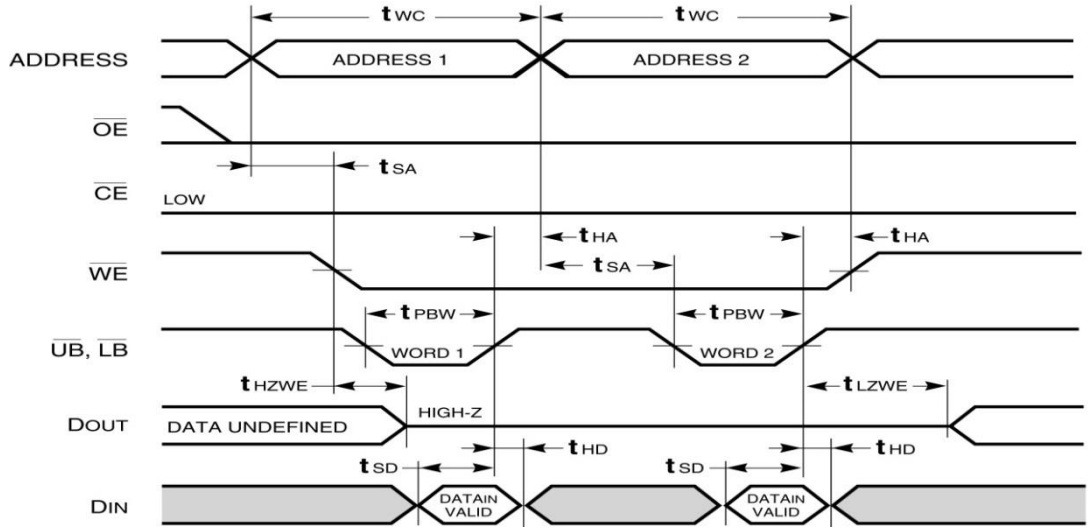


Abbildung 31: Waveform für Back-to-Back Write [22]

Jeder Signalwechsel im Diagramm, zu erkennen an den vertikalen Linien, wird als ein Ereignis bezeichnet. Im ersten Schritt werden zur eindeutigen Identifikation die Ereignisse E_i für das Waveformdiagramm aus Abbildung 31 definiert. Dies erfolgt, indem der Reihe nach für jedes Ereignis im Diagramm ein eindeutiger Name vergeben wird.

Für das vorliegende Beispiel ist das Ergebnis mit 15 definierten Ereignissen in Abbildung 32 dargestellt. Für jedes Ereignis E_i beschreibt t_i den Zeitpunkt zu dem E_i stattfindet.

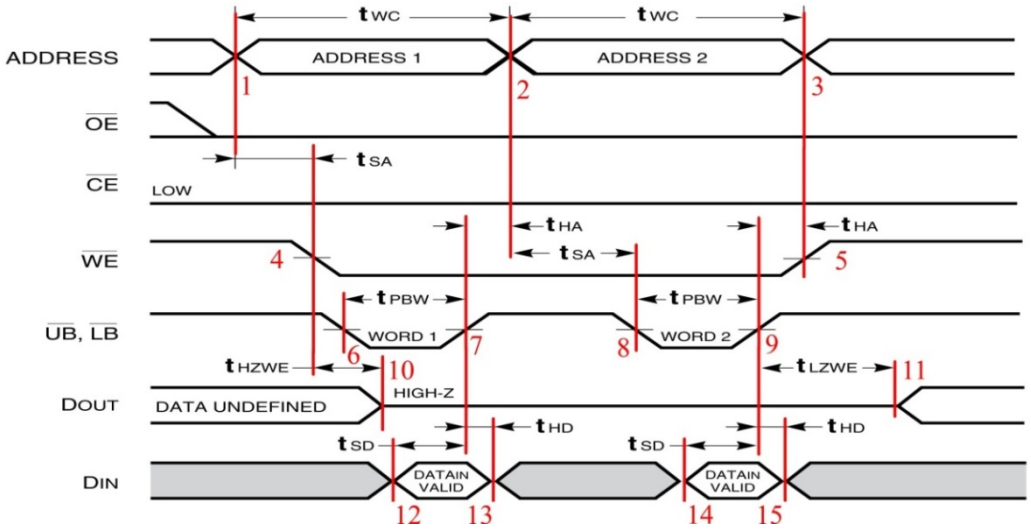


Abbildung 32: Wavediagram mit Ereignissen

An dieser Stelle sei gleich auf drei Mehrdeutigkeiten bzw. Fehler im Datenblatt hingewiesen. Es ist die Aufgabe des Testingenieurs, diese zu erkennen und zu korrigieren. Dieser Prozess kann nicht automatisiert werden.

Die Mehrdeutigkeiten sind:

- Im Waveformdiagramm wird den Ereignissen E_3 und E_5 nur ein gemeinsamer Zeitpunkt zugewiesen. Es werden keine zeitlichen Randbedingungen zwischen diese Ereignissen definiert. Müssen diese Ereignisse tatsächlich gleichzeitig stattfinden, so ist dies über die Definition sowohl einer minimalen als auch einer maximalen Randbedingung zwischen den beiden Ereignissen möglich. Hierfür werden zwischen beiden Ereignissen jeweils ein minimaler zeitlicher Abstand von 0 ns definiert. Beide Ereignisse werden also zeitgleich ausgeführt.
- Laut Abbildung 31 verstreicht die angegebene Zeitdifferenz t_{SA} zwischen den Ereignissen E_1 und E_4 sowie E_2 und E_8, nicht jedoch zwischen E_1 und E_6. Hier fehlt eine entsprechende Markierung, da t_{SA} in beiden Fällen berücksichtigt werden muss, bzw. es fehlt ein zeitlicher Zusammenhang zwischen E_4 und E_6.

Der Fehler ist:

- Im Waveformdiagramm wird die Zeitspanne zwischen Ereignis 6 und 7, sowie zwischen 8 und 9 mit t_{PBW} bezeichnet. In der Tabelle in Abbildung 30 ist diese jedoch als t_{PWB} bezeichnet (Konsistenzfehler).

Nach Korrektur dieser Fehler bzw. Mehrdeutigkeiten kann mit der Matrixgenerierung begonnen werden.

Alle Betrachtungen für die Generierung von Testinstrumenten beziehen sich immer auf die Sichtweise und das Verhalten des FPGAs. Wenn die betroffene Zeitvorgabe aus Sicht des DUTs formuliert ist, das Ereignis E_i zum Zeitpunkt t_i also ein Signal beschreibt, dass vom DUT getrieben wird, so sind die dargestellten zeitlichen

Randbedingungen invertiert zu betrachten. Aus einer maximalen Zeitvorgabe wird entsprechend eine minimale und umgekehrt. Dies bedeutet für das vorliegende Beispiel $t_{HZWE} = 5\,ns$, dass das DUT maximal $5\,ns$ braucht, damit das Signal D_{out} reagiert. Dies entspricht einer Wartezeit des FPGAs von mindestens $5\,ns$ auf E_{10}. Dies führt zum Eintrag $A_{10,4} = -5$ in der Difference Bound Matrix.

Das Generieren der Matrix A beginnt mit einer ‚leeren' Matrix, in der jeder Eintrag als # festgelegt wird. Dies stellt einen Platzhalter für eine beliebige reelle Zahl dar. Danach wird $A_{i,j} = C$ gesetzt, wenn die zeitliche Bedingung $t_j - t_i \leq C$ erfüllt ist. C entspricht hierbei der oberen Schranke, also einer maximalen Zeitvorgabe. Entsprechend wird $A_{j,i} = -D$ gesetzt, wenn die zeitliche Bedingung $t_j - t_i \geq D$ erfüllt ist. D entspricht hierbei der unteren Schranke, also einer minimalen Zeitvorgabe.

Als Resultat sind in Matrix A (Abbildung 33) alle Daten aus dem Waveformdiagramm und der Tabelle der zeitlichen Parameter enthalten. Diese sind gelb dargestellt. Weiterhin sind die Daten, die aufgrund der Korrektur der Mehrdeutigkeiten ermittelt worden sind, in rot dargestellt.

X	A_0 (E_1)	A_1 (E_2)	A_2 (E_3)	nWE_0 (E_4)	nWE_1 (E_5)	nUB_0 (E_6)	nUB_1 (E_7)	nUB_2 (E_8)	nUB_3 (E_9)	$DOUT_0$ (E_{10})	$DOUT_1$ (E_{11})	DIN_0 (E_{12})	DIN_1 (E_{13})	DIN_2 (E_{14})	DIN_3 (E_{15})
A_0 (E_1)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
A_1 (E_2)	-10	#	#	#	#	#	0	#	#	#	#	#	#	#	#
A_2 (E_3)	#	-10	#	#	0	#	#	#	0	#	#	#	#	#	#
nWE_0 (E_4)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nWE_1 (E_5)	#	#	0	#	#	#	#	#	#	#	#	#	#	#	#
nUB_0 (E_6)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nUB_1 (E_7)	#	#	#	#	#	-8	#	#	#	#	#	-6	#	#	#
nUB_2 (E_8)	#	0	#	#	#	#	#	#	#	#	#	#	#	#	#
nUB_3 (E_9)	#	#	#	#	#	#	#	-8	#	#	#	#	#	-6	#
$DOUT_0$ (E_{10})	#	#	#	-5	#	#	#	#	#	#	#	#	#	#	#
$DOUT_1$ (E_{11})	#	#	#	#	#	#	#	#	-2	#	#	#	#	#	#
DIN_0 (E_{12})	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
DIN_1 (E_{13})	#	#	#	#	#	#	0	#	#	#	#	#	#	#	#
DIN_2 (E_{14})	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
DIN_3 (E_{15})	#	#	#	#	#	#	#	#	0	#	#	#	#	#	#

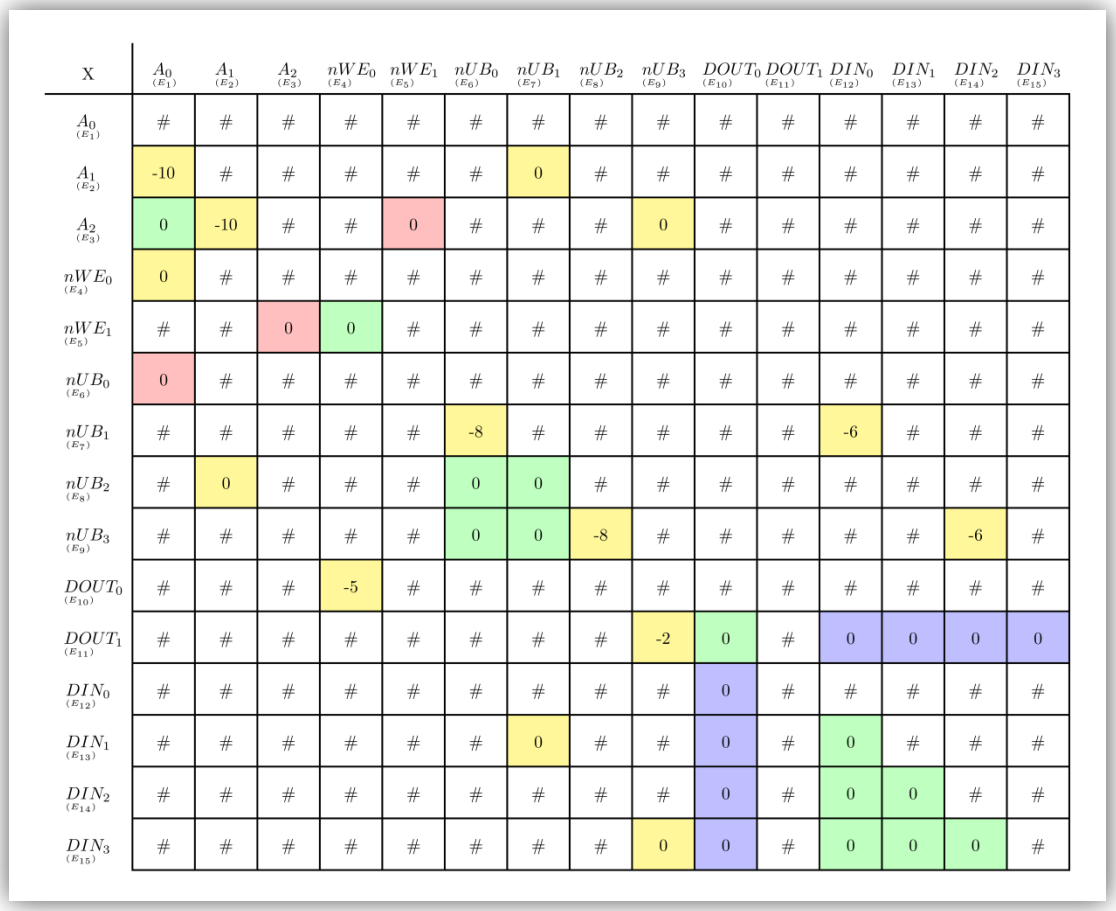
Abbildung 33: Ergebnis DBM nach Extraktion des Datenblatts

Die Matrix in Abbildung 33 ist jedoch noch unvollständig, da bestimmte, für die Matrixgenerierung wichtige Eigenschaften im Datenblatt anders dokumentiert worden sind. Daher muss die Matrix nach der folgenden Regel vervollständigt werden.

Regel zur Matrixvervollständigung:

(i) Wenn aus dem Waveformdiagramm ersichtlich ist, dass sich zwei Ereignisse auf dasselbe Signal beziehen, so darf das Ereignis $E_{i+1} = E_j$ zeitgleich oder nach E_i eintreten, jedoch nicht vorher. Dies bedeutet per Definition, dass E_j nicht vor E_i eintritt, so gilt $t_j - t_i \geq 0$ und entsprechend muss $A_{j,i} = 0$ gesetzt werden, falls noch keine weitere Bedingung notiert ist. Dies gilt zum Beispiel für $A_{3,1} = 0$, da aus dem Waveformdiagramm erkennbar ist, dass E_3 nach E_1 stattfinden muss. Diese Werte sind in Abbildung 34 grün markiert und können automatisch bestimmt werden.
Diese Regel gilt auch für unterschiedliche Signale, die dieselbe Leitung nutzen. Dies ist zum Beispiel der Fall beim Daten-Input- und Daten-Output-Signal einer bidirektionalen Leitung. So trifft diese Regel für das vorliegende Beispiel auf die Signale D_{IN} und D_{OUT} aus Abbildung 32 zu, weshalb $A_{12,10}$, $A_{13,10}$, $A_{14,10}$ und $A_{15,10} = 0$ zu setzen sind. Das gleiche gilt für $A_{11,12}$, $A_{11,13}$, $A_{11,14}$ und $A_{11,15}$. Diese sind ebenfalls als 0 zu definieren, so dass E_{11} zwingend nach E_{12}, E_{13}, E_{14} und E_{15} stattfindet. Diese Werte sind in Abbildung 34 blau markiert und müssen manuell ergänzt werden.
Eine obere Schranke $t_j - t_i \leq C$ besagt, dass t_j spätestens C nach t_i stattfinden muss. Es sagt jedoch nichts darüber aus, dass t_j nicht auch vor t_i eintreten darf. Diese Information ist formal nicht in der oberen Schranke enthalten und somit auch nicht in $A_{i,j} = C$. Falls dies jedoch gefordert ist, muss entsprechend zusätzlich $A_{j,i} = 0$ definiert werden. Dieser Fall ist im vorliegenden Beispiel nicht vorhanden, da es von Seiten des FPGA keine oberen Schranken zu berücksichtigen gibt.

Die Difference Bound Matrix aus Abbildung 34 ist das Ergebnis der Matrixgenerierung und enthält alle Informationen aus dem Waveformdiagramm in Abbildung 31 und den zeitlichen Parametern aus Abbildung 30.



X	A_0 (E_1)	A_1 (E_2)	A_2 (E_3)	nWE_0 (E_4)	nWE_1 (E_5)	nUB_0 (E_6)	nUB_1 (E_7)	nUB_2 (E_8)	nUB_3 (E_9)	$DOUT_0$ (E_{10})	$DOUT_1$ (E_{11})	DIN_0 (E_{12})	DIN_1 (E_{13})	DIN_2 (E_{14})	DIN_3 (E_{15})
A_0 (E_1)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
A_1 (E_2)	-10	#	#	#	#	#	0	#	#	#	#	#	#	#	#
A_2 (E_3)	0	-10	#	#	0	#	#	#	0	#	#	#	#	#	#
nWE_0 (E_4)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nWE_1 (E_5)	#	#	0	0	#	#	#	#	#	#	#	#	#	#	#
nUB_0 (E_6)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nUB_1 (E_7)	#	#	#	#	#	-8	#	#	#	#	#	-6	#	#	#
nUB_2 (E_8)	#	0	#	#	#	0	0	#	#	#	#	#	#	#	#
nUB_3 (E_9)	#	#	#	#	#	0	0	-8	#	#	#	#	#	-6	#
$DOUT_0$ (E_{10})	#	#	#	-5	#	#	#	#	#	#	#	#	#	#	#
$DOUT_1$ (E_{11})	#	#	#	#	#	#	#	#	-2	0	#	0	0	0	0
DIN_0 (E_{12})	#	#	#	#	#	#	#	#	#	0	#	#	#	#	#
DIN_1 (E_{13})	#	#	#	#	#	#	0	#	#	0	#	0	#	#	#
DIN_2 (E_{14})	#	#	#	#	#	#	#	#	#	0	#	0	0	#	#
DIN_3 (E_{15})	#	#	#	#	#	#	#	#	0	0	#	0	0	0	#

Abbildung 34: Ergebnis DBM nach der Vervollständigung

Betrachtet man die Matrix, die man durch A^T und das Löschen aller strikt positiven Einträge[40] erhält, als Adjazenzmatrix (siehe Abbildung 35) eines gerichteten Graphen mit der Knotenmenge $\{E_1, \dots, E_n\}$, so entspricht jeder gerichtete Pfad in diesem Graphen einer zeitlichen Reihenfolge von Ereignissen. Daher kann man die Terminalknoten, die keine ausgehenden Kanten besitzen (im Beispiel nur E_{11}), als kritische Ereignisse betrachten. Diese sind in der Adjazenzmatrix an leeren Zeilen zu erkennen. Initialknoten hingegen sind als leere Spalten zu erkennen (im Beispiel nur E_1). Hierauf wird in Kapitel 4.4.2 noch einmal eingegangen.

[40] Strikt positive Einträge entsprechen maximalen zeitlichen Randbedingungen. Diese beinhalten, wie weiter oben schon beschrieben, formal keine Aussage darüber, welches der Ereignisse zuerst stattfindet. Sie werden daher nicht zur Bildung der Adjazenzmatrix genutzt.

X	A_0 (E_1)	A_1 (E_2)	A_2 (E_3)	nWE_0 (E_4)	nWE_1 (E_5)	nUB_0 (E_6)	nUB_1 (E_7)	nUB_2 (E_8)	nUB_3 (E_9)	$DOUT_0$ (E_{10})	$DOUT_1$ (E_{11})	DIN_0 (E_{12})	DIN_1 (E_{13})	DIN_2 (E_{14})	DIN_3 (E_{15})
A_0 (E_1)	#	1	1	1	#	1	#	#	#	#	#	#	#	#	#
A_1 (E_2)	#	#	1	#	#	#	#	1	#	#	#	#	#	#	#
A_2 (E_3)	#	#	#	#	1	#	#	#	#	#	#	#	#	#	#
nWE_0 (E_4)	#	#	#	#	1	#	#	#	#	1	#	#	#	#	#
nWE_1 (E_5)	#	#	1	#	#	#	#	#	#	#	#	#	#	#	#
nUB_0 (E_6)	#	#	#	#	#	#	1	1	1	#	#	#	#	#	#
nUB_1 (E_7)	#	1	#	#	#	#	#	1	1	#	#	#	1	#	#
nUB_2 (E_8)	#	#	#	#	#	#	#	#	1	#	#	#	#	#	#
nUB_3 (E_9)	#	#	#	#	1	#	#	#	#	#	1	#	#	#	1
$DOUT_0$ (E_{10})	#	#	#	#	#	#	#	#	#	#	1	1	1	1	1
$DOUT_1$ (E_{11})	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
DIN_0 (E_{12})	#	#	#	#	#	#	1	#	#	#	1	#	1	1	1
DIN_1 (E_{13})	#	#	#	#	#	#	#	#	#	#	1	#	#	1	1
DIN_2 (E_{14})	#	#	#	#	#	#	#	#	1	#	1	#	#	#	1
DIN_3 (E_{15})	#	#	#	#	#	#	#	#	#	#	1	#	#	#	#

Abbildung 35: Adjazenzmatrix

Neben der Tatsache, dass die Matrix in Abbildung 35 eine kompakte und eindeutige Darstellung der Abhängigkeiten der Ereignisse ist, kann hiermit auch eine direkte graphische Visualisierung der zeitlichen Zusammenhänge generiert werden.

Der resultierende Graph kann automatisch von links nach rechts angeordnet werden und jedes Ereignis wird auf einer horizontalen Linie mit seinem zugehörigen Signal dargestellt. Abbildung 36 zeigt das Ergebnis einer solchen Graphendarstellung, in der der Terminalknoten E_{11} zu erkennen ist. Dies entspricht der Darstellung der kritischen Ereignisse aus Abbildung 32.

Da der Prozess der Graphengenierung ausschließlich die Daten aus der Difference Bound Matrix verwendet, kann dieser Prozess bereits zur Detektion erster Fehler genutzt werden. So lassen sich falsche oder fehlende Werte ggf. bereits frühzeitig erkennen. Näheres zu den Möglichkeiten der Fehlerdetektion ist Kapitel 4.5.2 zu entnehmen. Dieser gesamte hier beschriebene Prozess der Matrixgenerierung ist in [110] veröffentlicht worden.

Diese Modellierung der zeitlichen Randbedingungen für eine Ansteuerung des DUTs dient als Grundlage der späteren Generierung des Testinstruments. Diese wird in Kapitel 4.4 beschrieben.

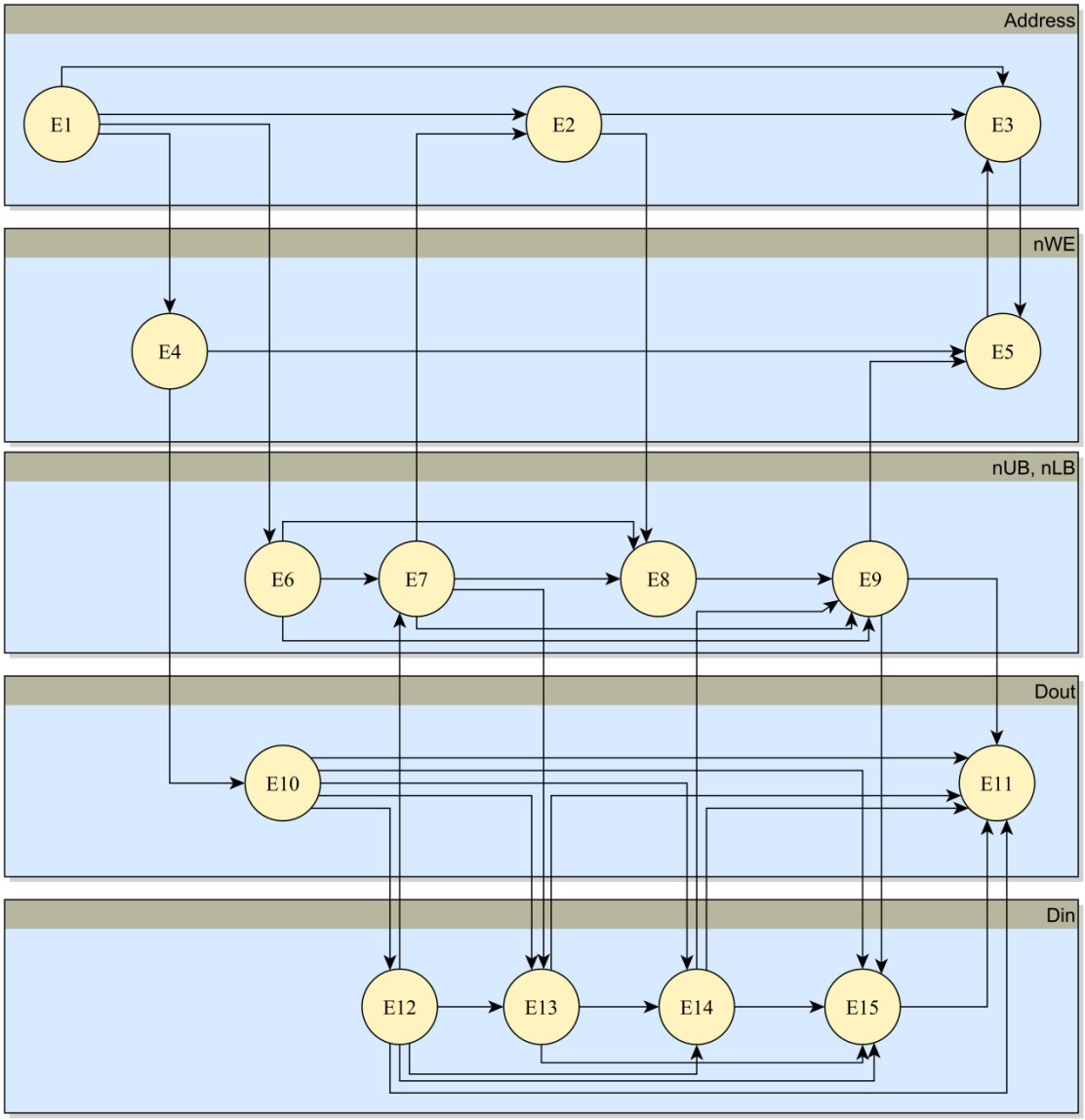


Abbildung 36: Gerichteter Graph der Adjazenzmatrix

4.2.7. ERWEITERUNG HIERARCHISCHE TIMINGMODELLIERUNG

In den letzten Kapiteln wurde erläutert, wie mit Hilfe von Difference Bound Matrizen einzelne zeitliche Abläufe so beschrieben werden können, dass daraus später eine Ansteuerung für DUTs generiert werden kann. Dies wurde am Beispiel einfacher Abläufe, die in einem Waveformdiagramm dargestellt werden konnten, demonstriert.

Für komplizierte Abläufe ist es üblich, diese auf mehrere Waveformdiagramme zu verteilen. Bei Protokollen, wie z.B. I^2C oder SPI, werden die zeitlichen Zusammenhänge einzelner Teile der Datenübertragung oft in mehreren Waveformdiagrammen dargestellt. So wird zum Beispiel das Senden eines einzelnen Bytes in einem Waveformdiagramm dargestellt, während andere Darstellungen die Reihenfolge bestimmter Bytesequenzen festlegen.

Der Zugriff auf ein EEPROM des Typ M24256 des Herstellers ST Micro Electronics [111] benötigt z.B. für das Schreiben eines Bytes an Nutzdaten die Übertragung folgender Informationen:

- Sende Startbedingung
- Sende 7-Bit IC Adresse und Write Bit
- Sende Bit 15..8 der Registeradresse
- Sende Bit 7..0 der Registeradresse
- Sende Byte der eigentlichen Nutzdaten
- Sende Stoppbedingung

Gemäß des Datenblatts [112] besteht eine Startbedingung aus 2 Ereignissen, ein zu sendendes Bit aus 4 Ereignissen, ein Byte (egal, ob Adresse oder Daten) aus 32 Ereignissen und eine Stoppbedingung aus 2 Ereignissen. Somit ergibt sich für das Aneinanderreihen, also das Übertragen eines einzelnen Bytes an Nutzdaten, eine Kette von 2+(4*8*4)+2=132 Ereignissen.

Würde man diese 132 Ereignisse in einer Matrix darstellen, hätte diese eine Größe von 132x132 und wäre damit sehr unübersichtlich. Außerdem gibt es in den Datenblättern keine komplette Darstellung dieses Ablaufs, sondern nur Darstellungen der einzelnen Teile, die vom Testingenieur zu einem Gesamtablauf zusammengestellt werden müssten.

Dies legt eine Modellierung nahe, die ebenfalls jeden Teil der Datenübertragung einzeln beschreibt und es dann erlaubt, diese, wenn möglich automatisch, aneinander zu reihen und in ein Testinstrument zu überführen.

Um den Lösungsansatz für dieses Problem näher zu erläutern, wird dieser im Folgenden an einem abstrakten Beispiel verdeutlicht. Dieses Beispiel ist in [79] ausführlich dargestellt[41].

Das Beispiel beschreibt den zeitlichen Ablauf in drei Hierarchieebenen. Diese sind als TOP, SUBx und SUBx.y bezeichnet, wobei x von der Nummer der übergeordneten Ebene abhängt und y eine fortlaufende Nummer in der jeweiligen Ebene ist. In Abbildung 37 wird deutlich, dass auf jeder Ebene nur die Abhängigkeiten der einzelnen Ereignisse zueinander und innerhalb eines Graphen dargestellt sind. Jeder dieser Graphen kann unabhängig von den anderen beschrieben werden, woraus sich in diesem Beispiel fünf DB-Matrizen ergeben (siehe Abbildung 38).

In jedem Subgraphen wird ein Ereignis als Anker bezeichnet und durch einen Strich oberhalb des Ereignisses markiert. Diese Anker geben die Verknüpfung zwischen den

[41] Bei [79] handelt es sich um eine Masterarbeit, die im Rahmen dieser Dissertation durchgeführt worden ist und von dessen Autor betreut wurde.

einzelnen Hierarchieebenen an. Alle Ereignisse, die als Anker vertikal über eine oder mehrere Hierarchieebenen markiert sind, finden zur selben Zeit wie das Ereignis in der obersten Ebene statt.

Die Bezeichnungen der Ereignisse jeder Ebene setzen sich aus den Bezeichnungen der übergeordneten Ereignisse, einem Punkt als Trennzeichnen für die neue Hierarchieebene und einer fortlaufenden Nummer je Subebene zusammen. Anhand dieser Bezeichnungen und der Ankermarkierungen in den Subgraphen können die einzelnen DB-Matrizen automatisch zu einer einzelnen Matrix zusammengesetzt werden. Diese Matrix beschreibt dann den gesamten zeitlichen Ablauf aller beteiligten Graphen. Für den Graph in Abbildung 37 ist die resultierende DB-Matrix in Abbildung 39 dargestellt. Die dargestellten Farben ergeben sich dabei aus den Farben der einzelnen Matrizen und geben somit Auskunft über die Herkunft der dargestellten Zeiten.

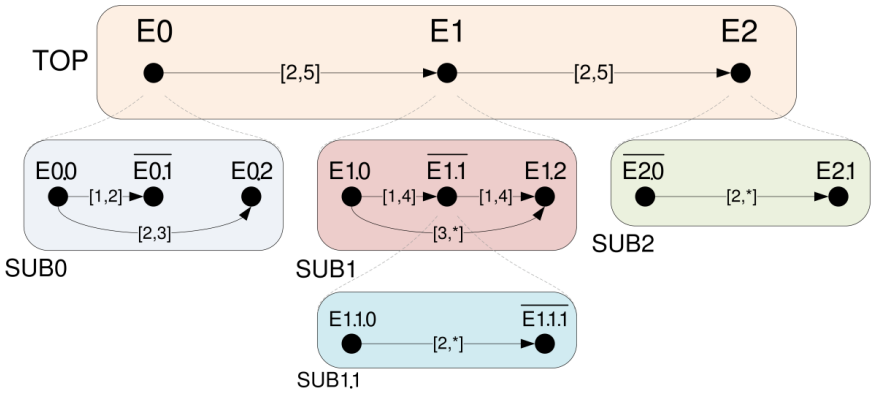


Abbildung 37: Beispiel zur hierarchischen Timingmodellierung [79]

TOP	E0	E1	E2
E0	#	5	#
E1	-2	#	5
E2	#	-2	#

SUB0	E0.0	/E0.1	E0.2
E0.0	#	2	3
E0.1	-1	#	#
E0.2	-2	#	#

SUB1	E1.0	/E1.1	E1.2
E1.0	#	4	#
E1.1	-1	#	4
E1.2	-3	-1	#

SUB2	/E2.0	E2.1
E2.0	#	#
E2.1	-2	#

SUB11	E1.1.0	/E1.1.1
E1.1.0	#	2
E1.1.1	-1	#

Abbildung 38: DB-Matrizen zur hierarchischen Timingmodellierung [79]

	E0.0	E0, E0.1	E0.2	E1.0	E1.1.0	E1, E1.1, E1.1.1	E1.2	E2, E2.0	E2.1
E0.0	#	2	3	#	#	#	#	#	#
E0, E0.1	-1	#	#	#	#	5	#	#	#
E0.2	-2	#	#	#	#	#	#	#	#
E1.0	#	#	#	#	#	4	#	#	#
E1.1.0	#	#	#	#	#	2	#	#	#
E1, E1.1, E1.1.1	#	-2	#	-1	-1	#	4	5	#
E1.2	#	#	#	-3	#	-1	#	#	#
E2, E2.0	#	#	#	#	#	-2	#	#	#
E2.1	#	#	#	#	#	#	#	-2	#

Abbildung 39: Fusionierte DB-Matrix zur hierarchischen Timingmodellierung [79]

An diesem Beispiel werden die Möglichkeiten der hierarchischen Timingmodellierung deutlich. Im Falle der Ansteuerung eines EEPROM über I^2C reduziert sich die Komplexität der Modellierung von einer Matrix mit der Größe von 132x132 auf lediglich sechs kleinere Matrizen mit einer Größe zwischen 2x2 und 8x8, wobei sich vier dieser Matrizen nur durch die zu sendenden Daten, aber nicht durch die zeitlichen Abhängigkeiten unterscheiden. Somit reduziert sich die zeitliche Modellierung im Falle von I^2C auf drei kleine Matrizen und eine übergeordnete Matrix, die die Zusammenhänge zwischen den Submatrizen beschreibt.

Hieraus kann, wie im Beispiel dargestellt, die automatische Generierung einer Gesamtmatrix erfolgen. Diese hat für das Beispiel der I^2C Ansteuerung wieder eine Größe von 132x132. Somit zeigt dieser Ansatz eine vereinfachte Modellierung, die jedoch keinen Einfluss auf die Realisierung hat, wenn der Ausgangspunkt dieses Prozesses in beiden Fällen eine große Matrix ist und nicht auf die hierarchische Beschreibung zurückgegriffen wird. In Kapitel 4.4.7 wird ein alternativer Ansatz vorgestellt, der den Vorteil der hierarchischen Modellierung auch für die Generierung nutzbar macht.

4.2.8. GRENZEN DER MODELLIERUNG

Mit der bisher beschriebenen Modellierung lassen sich viele Ansteuerungen für DUTs spezifizieren. Jedoch gibt es auch Grenzen, so dass bestimmte Modellierungen nicht unterstützt werden.

Die Modellierung zeitlicher Abhängigkeiten ist auf die Umsetzung der Ebene L1 des Testsystem-Ebenenmodells (siehe Abbildung 18) beschränkt. Bei Funktionen höherer Ebene ist dies nicht möglich. Gibt es dort ebenfalls zeitliche Abhängigkeiten, so müssen diese explizit im Code angegeben werden. Da die modellierten Funktionen der Ebenen L2 bis L5 im Wesentlichen abhängig sind von den Testalgorithmen und nicht von den DUTs, sind hier andere Modellierungsansätze notwendig.

Bei der Ansteuerung von DDR Speicher mit älteren FPGAs wie zum Beispiel dem Spartan3 von Xilinx, wird eine bestimmte Konfiguration von PLL- bzw. DLL-Blöcken sowie eine genaue Platzierung der später zu generierenden Logik vorausgesetzt. Dies kann in RTDL nicht spezifiziert werden. Es ist jedoch anzumerken, dass diese Art der Umsetzung seit Jahren bei Neuentwicklungen nicht mehr verwendet wird und somit nur Systeme von dieser Einschränkung betroffen sind, die nicht nach dem Stand der Technik von 2010 entwickelt worden sind.

Für diese und andere Spezialanwendungen kann es durchaus sinnvoll sein, nicht die gesamte Ansteuerung eines DUT in RTLD zu beschreiben, sondern zum Beispiel auf vorgefertigte Elemente für die Ansteuerung zurückzugreifen, die evtl. beim späteren Einsatz des zu testenden Systems verwendet oder vom FPGA Hersteller bereitgestellt werden und bereits viel Funktionalität beinhalten. Diese Ansteuerungen können als

Blackbox betrachtet werden. Die Testsystemarchitektur erlaubt es, diese als Modul mit definierter Schnittstelle in RTDL einzubinden.

Weiterhin ist die Nutzung von speziellen I/O Pins, zum Beispiel SerDes-Einheiten bisher, nicht vorgesehen und wird daher auch in der Modellierung nicht unterstützt. Diese Schnittstellen wären ebenfalls über den Blackbox-Ansatz in ROBSY ansprechbar.

4.2.9. ZUSAMMENFASSUNG DER MODELLIERUNG

Es wurde eine Testsystemarchitektur sowie ein Ebenenkonzept für die Generierung von Embedded Testinstruments vorgestellt, das in hohem Maße flexibel sowie konfigurierbar ist und es erlaubt, die benötigten Funktionen zum Testen eines DUTs wahlweise in Software, Embedded Software oder Hardware zu realisieren.

Für die Modellierung der DUTs wurden Anforderungen definiert und diverse bestehende Sprachen aus unterschiedlichen Anwendungsdomainen auf ihre Eignung hin untersucht. Als Ergebnis wurde eine eigene Modellierungssprache, die alle Anforderungen aus Kapitel 4.2.1 erfüllt, geschaffen und vorgestellt.

Der gewählte Ansatz ist universell und erlaubt es, DUTs auf einfache und kompakte Weise zu beschreiben. Der gewählte Ansatz ist modular und erweiterungsfähig. Er macht es möglich, sowohl Gegebenheiten der Hardware, Zugriffsroutinen als auch Testalgorithmen in einer Datei darzustellen. Hierbei ist die Beschreibung unabhängig von der später gewählten Umsetzung in Software, Embedded Software oder Hardware.

Insbesondere wurde eine Möglichkeit geschaffen, auf einfache Art die zeitlichen Abhängigkeiten der Ansteuerung selbst von komplexen DUTs zu beschreiben. Dies geschieht nicht nur für einen ausgewählten Anwendungsfall, sondern es wird der gesamte zulässige Lösungsraum für die Ansteuerung des DUTs durch das Testinstrument abdeckt. Hieraus kann dann zur Zeit der Testsystemgenerierung eine für den Einsatzfall optimale Lösung generiert werden, ohne dass dieser vorher bekannt sein müssen.

Es wurde weiterhin gezeigt, wie insbesondere die zeitkritischen Informationen zur DUT-Modellierung den Datenblättern der DUTs entnommen werden und wie daraus ein DUT-M generiert wird. Weiterhin wurde auf die Erweiterung bezüglich einer hierarchischen Timingmodellierung eingegangen und es wurden die Grenzen der aktuellen Realisierung gezeigt.

Das Konzept des Testsystems erlaubt es, komplexe Systeme, die an die jeweiligen Bedürfnisse der Nutzer angepasst sind und aus mehreren Testprozessoren und Co-Prozessoren bestehen können, zu generieren. Durch die starke Konfigurierbarkeit ist es auch möglich, nicht nur an den Prüfling optimal angepasste „Synthetische Testinstrumente“ zu generieren, sondern beliebige und somit auch „Universelle Testinstrumente“. Somit können z.B. die in [15] definierten Testinstrumente als Untermenge der mit ROBSY generierbaren Instrumente betrachtet werden.

4.3. PARTITIONIERUNG

Nachdem die benötigten DUTs modelliert sind, und bevor der passende Co-Prozessor generiert werden kann, ist eine Partitionierung des Testinstruments notwendig. Dies bezieht sich auf die Ebenen des in Kapitel 4.1.1 vorgestellten Ebenenkonzepts. Dabei muss für jede der fünf Ebenen eine Ausführungsart gewählt werden. Entsprechend Tabelle 4 wird dabei, aufsteigend beginnend bei Ebene L1, eine Ausführungsart festgelegt. Die Entscheidung, wie eine Ebene ausgeführt, wird hängt von folgenden Eigenschaften ab:

- benötigte Geschwindigkeit bei der Testausführung
- gefordertes Echtzeitverhalten bei der Testausführung
- verfügbare Ressourcen des FPGAs
- den zu erwartenden benötigten Ressourcen für den Co-Prozessor[42]
- der zu erwartenden Codegröße für die in Embedded Software zu realisierenden Komponenten des Testinstruments[43]
- Fähigkeiten und Parameter des Testprozessors

Alle diese Eigenschaften haben gegenseitige Abhängigkeiten. So beschleunigt zum Beispiel ein leistungsfähiger Testprozessor die Testausführung, benötigt im Gegenzug aber auch mehr Ressourcen, so dass es sinnvoller sein kann, die Ausführung bestimmter Algorithmen gar nicht in Embedded Software, sondern direkt in Hardware durchzuführen. Dies führt jedoch wieder zu unterschiedlichen Ausführungszeiten der Testalgorithmen sowie einem anderen Ressourcenverbrauch. Eine zufriedenstellende Konfiguration des Testsystems muss durch Experimentieren mit verschiedenen Varianten bestimmt werden und ist sehr zeitaufwendig.

Abbildung 40 zeigt die Abhängigkeiten der einzelnen Eigenschaften des Testsystems. Parameter des Prozessors sind dabei in rosa, des Co-Prozessors in grün und des FPGAs bzw. der Testausführung in orange dargestellt. Hierbei wird von einer vollständigen Realisierung des Testsystems auf dem FPGA ausgegangen. Sollen einige Funktionen auf den Test-PC ausgelagert werden, erweitern sich die Abhängigkeiten noch einmal entsprechend, da die Frage, ob eine Funktion überhaupt im FPGA implementiert wird, Auswirkungen auf alle in Abbildung 40 dargestellten Eigenschaften hat.

Dieses zeigt die teilweise komplexen Abhängigkeiten der Eigenschaften und macht die Schwierigkeit einer optimalen Partitionierung deutlich. Im Rahmen dieser Arbeit

[42] Der zu erwartenden Ressourcenverbrauch muss vor der Generierung des Co-Prozessors geschätzt werden. Wie genau dies geschieht ist nicht Bestandteil dieser Arbeit.

[43] Die zu erwartende Codegröße ist vom Softwarecompiler zu bestimmen. Dieser ist nicht Bestandteil dieser Arbeit.

soll nicht auf die Möglichkeiten einer Optimierung dieser Eigenschaften eingegangen werden, es soll lediglich dargelegt werden, dass die gewählte Modellierung und das Konzept des vorgestellten Testsystems solche Optimierungen erlauben und welches Potential für weitere Forschungen sich dahinter verbirgt.

Für den weiteren Verlauf dieser Arbeit wird eine manuelle Partitionierung mit einem Testprozessor in einer festen Konfiguration gewählt.

Abbildung 40: Abhängigkeiten für eine optimale Partitionierung

4.4. GENERIERUNG

Im folgenden Kapitel wird auf die Generierung eines Co-Prozessors eingegangen. Die Umsetzung von Funktionen in Software oder Embedded Software wird nur an den Stellen erwähnt, an denen es für das Verständnis hilfreich ist. Die Existenz eines entsprechenden Test-PCs und Testprozessors wird vorausgesetzt.

4.4.1. DESIGNFLOW

Wie in Kapitel 4.2.3 beschrieben, wurde die Sprache RTDL für die Modellierung von DUTs entwickelt. Diese DUT Modelle (DUT-M), die Netzliste der zu prüfenden Leiterplatte, Informationen zur Prozessorkonfiguration und einige durch den Testingenieur gewählte Eigenschaften für das Testsystem stellen zusammen die Eingangsinformationen für die Generierung eines solchen Systems dar.

In den folgenden Unterkapiteln wird speziell auf die Generierung der Ebene L1 eines einzelnen Co-Prozessors eingegangen und die Generierung einer Ansteuerung basierend auf einer DB-Matrix beschrieben. Dieser Schritt ist Teil des Bearbeitungsschritts „Informationen verarbeiten“ im Abschnitt „Hardware“ in der Darstellung des allgemeinen Designflows in Abbildung 41 und liefert als Ergebnis eine Beschreibung des „Gesamtsystem Hardware“, aus der dann der VHDL Code generiert wird und die „Synthese Konfiguration“, die für die Generierung des FPGA-Bitfiles notwendig ist.

Neben dem Co-Prozessor, der die in Hardware realisierten Funktionen eines Testinstruments beinhaltet, sind für ein vollständiges Testsystem weitere Elemente notwendig, die Bestandteil eines solchen Testsystems sein müssen:

- Integration eines Prozessors
- Anbinden des Prozessors an einen Wishbone-Bus
- Anbinden des Co-Prozessors an den zugeordneten Wishbone-Bus
- Realisieren des Interface I2 zur Verbindung mit dem Test-PC
- Verbindungslogik, die für den Zusammenbau aller Bestandteile zu einem Gesamtsystem notwendig ist

Hinzu kommen weitere optionale Bestandteile, deren Notwendigkeit von der Konfiguration des Testsystems abhängt. Diese sind:

- weitere Prozessoren, falls ein Multiprozessorsystem generiert werden soll
- weitere Co-Prozessoren
- weitere Wishbone-Busse[44] und das Zuordnen von Prozessoren und Co-Prozessoren
- Multiplexen der Verbindungen mehrerer Co-Prozessoren auf gemeinsam genutzte Pins

[44] Bei mehreren Prozessoren können mehrere Wishbone-Busse unabhängig voneinander existieren und jeder Co-Prozessor muss einem dieser Busse zugeordnet werden.

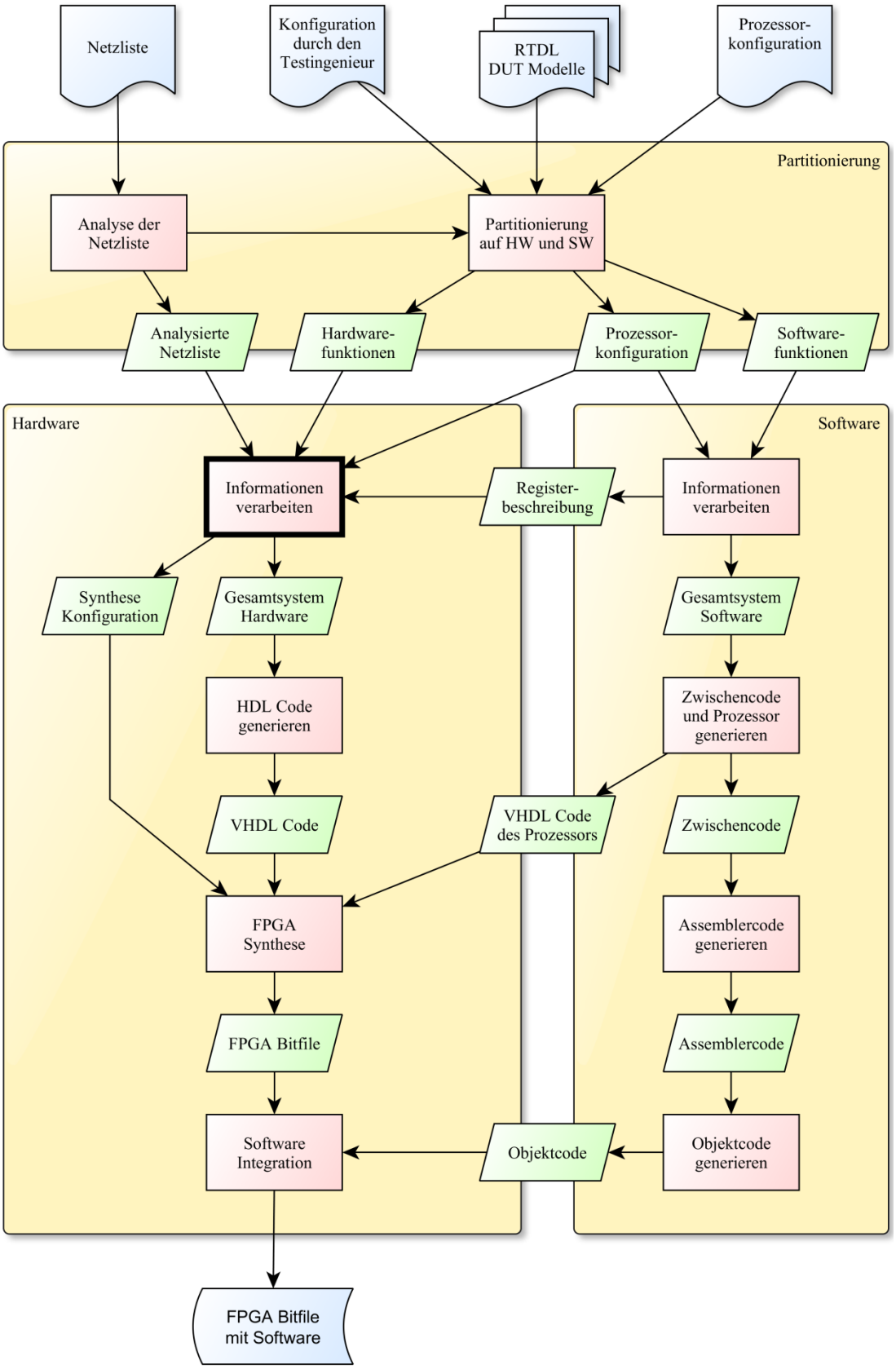


Abbildung 41: Designflow für die Generierung eines Testsystems

In diesem Kapitel wird der Ablauf für die Generierung eines vollständigen Testsystems mit einem Prozessor und einem Co-Prozessor beschrieben. Aufgrund diverser Abhängigkeiten muss die Generierung des Test-Prozessors in enger Kopplung mit der Generierung des restlichen Systems erfolgen. Da der Test-Prozessor nicht Bestandteil dieser Arbeit ist, wird nur auf die Schnittstellen zwischen dem Prozessor und dem restlichen Testsystem eingegangen, jedoch nicht dessen eigentliche Generierung beschrieben.

Die *automatische* Generierung des Testsystems ist eine der Hauptvoraussetzungen für die praktische Nutzung des vorgestellten Konzepts. Nur, wenn sich das Testsystem automatisch an die zu testende Leiterplatte anpasst und entsprechend generiert werden kann, wird der Testingenieur das System akzeptieren.

Zwischen Prozessor und den Co-Prozessoren bestehen folgende Abhängigkeiten, die bei einer Generierung berücksichtigt werden müssen:

- die Partitionierung (siehe Kapitel 4.3) hat Einfluss auf Parameter des Prozessors wie den verfügbaren Befehlssatz, die Registeranzahl und Busbreiten
- die Daten- und Adressbreite des Prozessors haben Einfluss auf die über Wishbone angeschlossenen Register innerhalb der Co-Prozessoren
- die verwendeten Register und deren Adressen in den Co-Prozessoren haben wiederum Einfluss auf die generierte Software des Test-Prozessors

Die Generierung des „Gesamtsystem Hardware“ ist detaillierter in Abbildung 42 dargestellt. Hierbei ist der modulare Aufbau der Systemgenerierung zu erkennen. So erfolgt trotz der Abhängigkeiten der einzelnen Schritte die Generierung für jeden Co-Prozessor getrennt. Die Generierung der Ebene L1 eines Co-Prozessors ist fett hervorgehoben und noch einmal in detaillierterer Form in Abbildung 43 dargestellt.

Die Generierung eines Co-Prozessors läuft dabei wie folgt ab:

- anhand der analysierten Netzliste wird die Verschaltung des DUTs mit dem FPGA bestimmt
- die Prozessorkonfiguration legt den Aufbau der über Wishbone verfügbaren Register und den zu verwendenden Adressbereich fest
- die Hardwarefunktionen beinhalten die eigentlich zu generierende Ansteuerung jedes Co-Prozessors sowie die Interfacebeschreibung und –parameter der DUTs, in Abhängigkeit der gewählten Partitionierung

Aufbauend auf diesen Informationen kann mit der Generierung der Co-Prozessoren begonnen werden. Dies erfolgt jeweils unabhängig für jeden Co-Prozessor und für jede in Hardware zu implementierende Ebene getrennt und wird abschließend für jeden Co-Prozessor in einem Wrapper gruppiert. Alle Wrapper werden wiederum im Top-Level

zusammengefasst und bilden mit dem Prozessor[45] eine Beschreibung des gesamten Testinstruments. Diese dient als Eingangsinformation für die VHDL Code Generierung wie in der Übersicht in Abbildung 41 dargestellt. Weiterhin wird eine Konfiguration für das Place-and-Route der später zu erfolgenden FPGA-Synthese generiert.

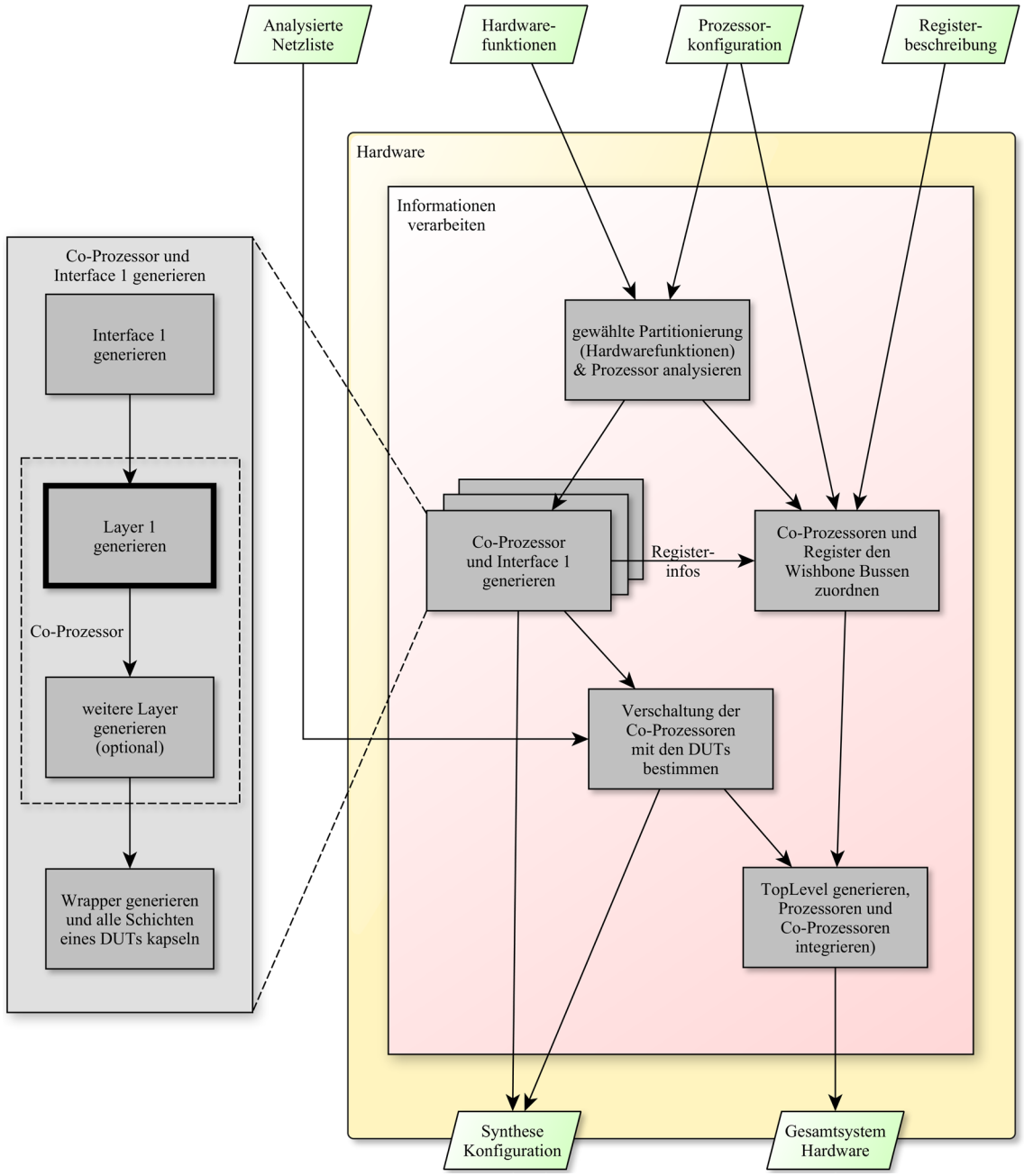


Abbildung 42: Partieller Designflow Hardware

[45] Im Hardware Designflow werden nur die Quellen des Prozessor in das Top-Level eingebunden. Die Generierung dieser Quellen ist Teil des Software Designflows und nicht Bestandteil dieser Arbeit.

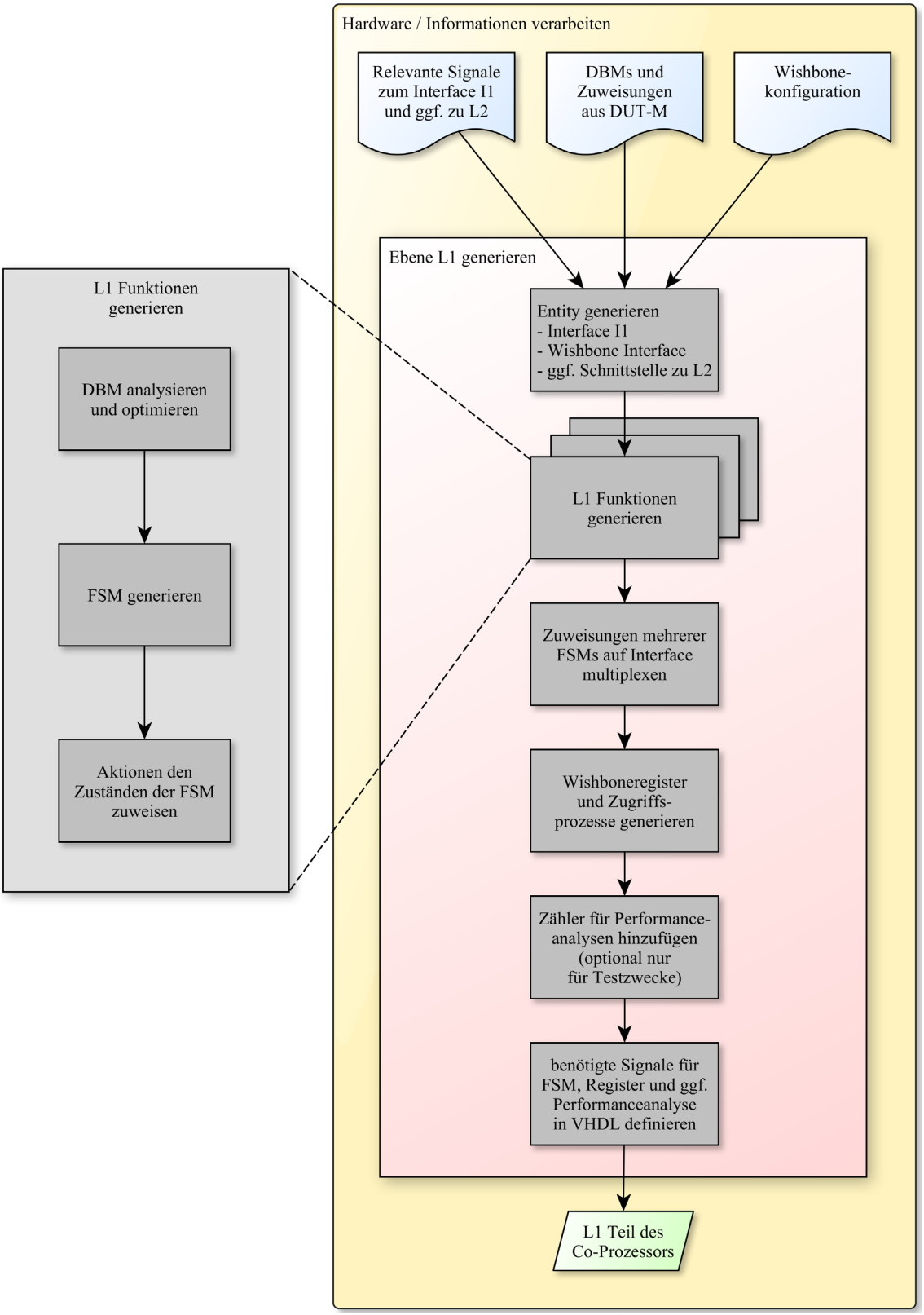


Abbildung 43: Designflow Ebene L1 der Co-Prozessor-Generierung

Die Generierung der höheren Ebenen L2 bis L5 erfolgt ähnlich zur Ebene L1, jedoch ohne das Analysieren und Optimieren der DB-Matrizen (siehe Abbildung 44). Ein Auszug aus einem DUT-Modell (Abbildung 45) sowie die dazugehörige FSM sind in Abbildung 46 und Abbildung 47 dargestellt. Auf die Darstellung der Aktionszuweisungen wurde aufgrund der Übersichtlichkeit an dieser Stelle verzichtet. Hierfür wird auf Anhang U verwiesen.

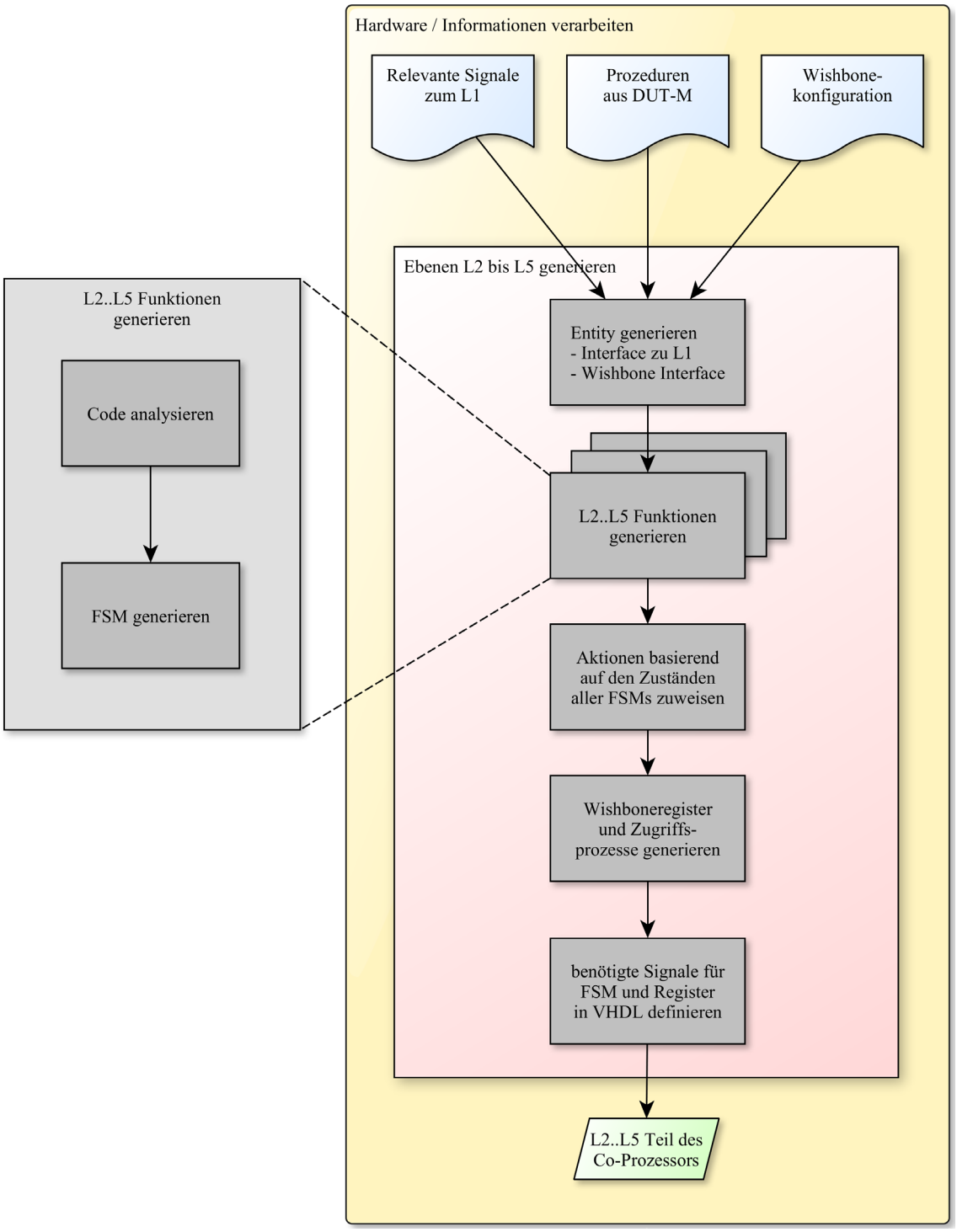


Abbildung 44: Designflow Ebene L2 bis L5 der Co-Prozessor-Generierung

```
L2: PROCEDURE l2_apply_pattern(IN wr_rd[2], IN p_data[16], IN p_addr[18], OUT result[16])
VAR dummy[1];
BEGIN
   IF ( ( wr_rd AND 1 ) == 1 )
      CALL l1_write_WE(addr(p_addr), data(p_data));
   ENDIF ;
   IF ( ( wr_rd AND 2 ) == 2 )
      CALL l1_read_oe(addr(p_addr),data(result));
   ENDIF ;
END
```

Abbildung 45: L2_apply_pattern Prozedur aus DUT-M

```
type tl2_apply_pattern is (
                idle_0,
                If_1,
                call_1_0,
                return_1_1,
                If_3,
                call_3_0,
                return_3_1,
                finish_5
            );
```

Abbildung 46: Definition der FSM für L2_apply_pattern Prozedur

```
process_sequence_l2_apply_pattern : process(CLK, RST)
begin
   if RST = '1' then
      stl2_apply_pattern <= idle_0;
   elsif rising_edge(CLK) then
      case stl2_apply_pattern is

         when idle_0 =>
            if l2_apply_pattern_req = "1" then
               stl2_apply_pattern <= If_1;
            else
               stl2_apply_pattern <= idle_0;
            end if;

         when If_1 =>
            if ( ( l2_apply_pattern_wr_rd and "01") ) = "01" )  then
               stl2_apply_pattern <= call_1_0;
            else
               stl2_apply_pattern <= If_3;
            end if;

         when call_1_0 =>
               stl2_apply_pattern <= return_1_1;

         when return_1_1 =>
            if PORT_l1_write_we_ack = "1" then
               stl2_apply_pattern <= If_3;
            else
               stl2_apply_pattern <= return_1_1;
            end if;

         when If_3 =>
            if ( ( l2_apply_pattern_wr_rd and "10" ) = "10" )  then
               stl2_apply_pattern <= call_3_0;
            else
               stl2_apply_pattern <= finish_5;
            end if;

         when call_3_0 =>
               stl2_apply_pattern <= return_3_1;

         when return_3_1 =>
            if PORT_l1_read_oe_ack = "1" then
               stl2_apply_pattern <= finish_5;
            else
               stl2_apply_pattern <= return_3_1;
            end if;

         when finish_5 =>
               stl2_apply_pattern <= idle_0;

         when others =>
            stl2_apply_pattern <= idle_0;

      end case;
   end if;
end process;
```

Abbildung 47: FSM für L2_apply_pattern Prozedur

4.4.2. AUTOMATISCHES SCHEDULING

Nachdem in Kapitel 4.2.6 der prinzipielle Ablauf einer Modellierung beschrieben worden ist, wird im aktuellen Kapitel eine Methode vorgestellt, um mit diesen Daten eine automatische Zeitplanung[46] durchzuführen, an deren Ende die benötigten Zustände für die Steuerungs-FSM der Ebene L1 des Testinstruments festgelegt sind.

Abbildung 48 zeigt den generierten Graphen aus Kapitel 4.2.6 sowie den längsten Pfad innerhalb dieses Graphen vom Initialknoten E_1 zum Terminalknoten E_{11}.

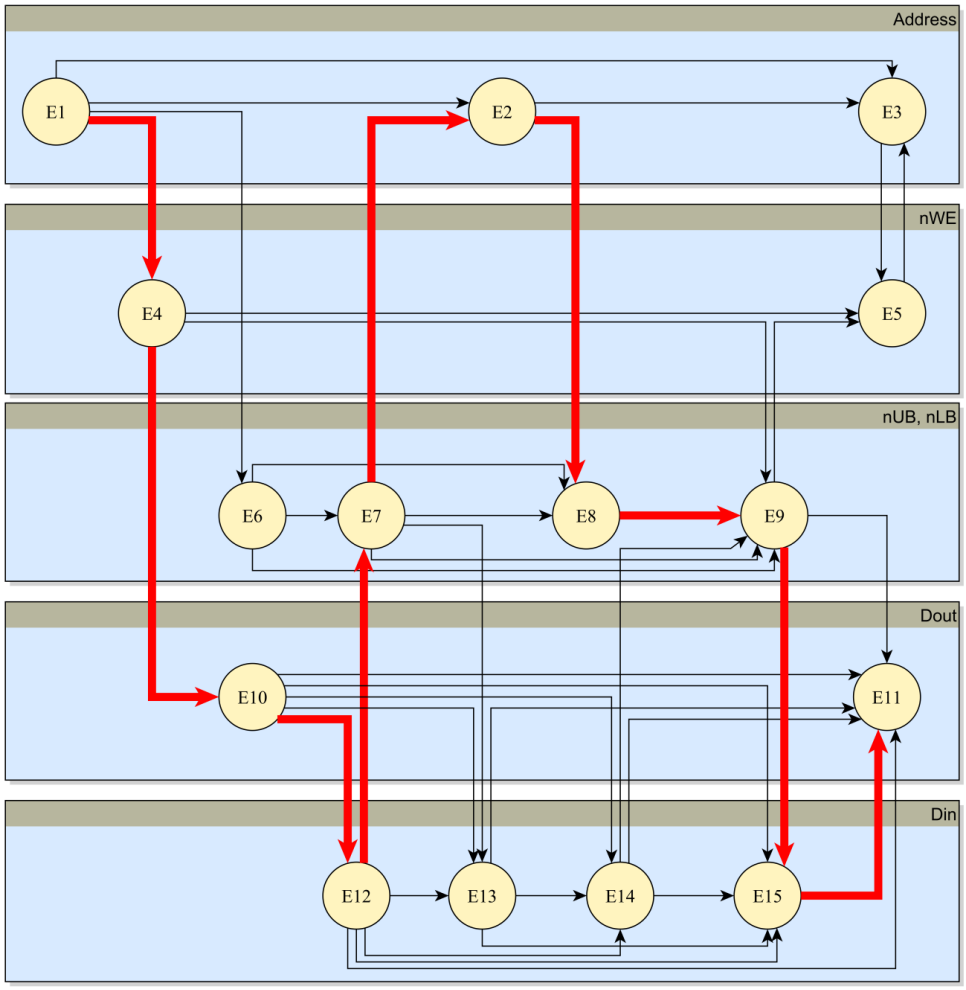


Abbildung 48: Längster Pfad vom Initialknoten zum Terminalknoten im gerichteten Graph der Adjazenzmatrix

Ziel der automatischen Zeitplanung ist es, den kürzesten Ablauf bzw. das schnellste Timing für $\{E_1, \dots, E_{15}\}$ zu finden, das alle zeitlichen Vorgaben aus der Difference Bound Matrix einhält.

[46] Diese wird auch im Deutschen oft mit dem englischen Begriff Scheduling bezeichnet.

Hierzu muss das folgende lineare Optimierungsproblem gelöst werden:

$$min\{f^T t | Bt \leq b, t \geq 0, t \epsilon \mathbb{R}^n\} \tag{4.3}$$

Hierbei ist die Difference Bound Matrix definiert als $A \in (\mathbb{R} \cup \#)^{n \times n}$. $n \in \mathbb{N}$ gibt dabei die Anzahl der Zeitpunkte der DB-Matrix an. Außerdem bezeichnet $m \in \mathbb{N}$ die Anzahl der Einträge in A und $t \in \mathbb{R}^n$ den Zeitvektor der Ereignisse $\{E_1, \dots, E_n\}$.

Für das Beispiel aus Abbildung 34 ist somit $n = 15$, $m = 38$ und $t = \{t_1, \dots, t_n\}$.

Es werden alle Einträge aus A in beliebiger Reihenfolge angeordnet. Die Elemente werden entsprechend $a^1 = A_{i_1,j_1}, \dots, a^m = A_{i_m,j_m}$ nummeriert. Für die ersten fünf Einträge des Beispiels könnte dies wie folgt aussehen:

$$\begin{aligned} a^1 &= A_{1_1,2_1} \\ a^2 &= A_{1_2,3_2} \\ a^3 &= A_{1_3,4_3} \\ a^4 &= A_{1_4,6_4} \\ a^5 &= A_{2_5,3_5} \end{aligned} \tag{4.4}$$

Diese Elemente ergeben den Vektor b, der somit alle zeitlichen Schranken der DB-Matrix in der Form $b = (a^1, \dots, a^m)^T$ enthält.

Für die Definition von B ergibt sich die l-te Zeile B_l aus $a^l = A_{i_l,j_l}$ durch die Umsetzung der Gleichung $B_l t = t_{j_l} - t_{i_l} \leq A_{i_l,j_l}$, die für alle $k = 1, \dots, n$ und $l = 1, \dots, m$ gemäß folgender Formel $B_{l,k}$ definiert ist:

$$B_{l,k} = \begin{cases} 1 & if\ k = j_l \\ -1 & if\ k = i_l \\ 0 & sonst \end{cases} \tag{4.5}$$

Die ersten fünf Einträge der Opti-Matrix B sind im Folgenden dargestellt:

$$B = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

Alle zeitlichen Randbedingungen sind somit durch $Bt \leq b$ abgedeckt[47]

[47] Es sei an dieser Stelle angemerkt, dass $t \geq 0$ eine allgemeingültige Voraussetzung für die Existenz einer minimalen Lösung ist, da alle zeitlichen Bedingungen relative Bedingungen sind.

Der Vektor f dient als Gewichtsvektor und ermöglicht unterschiedliche Optimierungsstrategien. Wird $f = (1, \dots 1)^T$ gesetzt, so entspricht dies einer einfachen ASAP[48] Strategie. Eine ALAP[49] Strategie wird hingegen erreicht, wenn das kritische Ereignis E_{11} einen sehr großer Wert erhält, während allen anderen Ereignissen kleine negative Werte zugewiesen werden. Die Ergebnisse der automatischen Zeitplanung für die Difference Bound Matrix aus Abbildung 34 entsprechend der Strategien ASAP und ALAP sind in Tabelle 7 dargestellt.

Tabelle 7: Ergebnis unterschiedlicher Zeitplanungen

Funktionen & Zeiten / Ereignisse	f_{ASAP}	t_{ASAP} *[ns]*	f_{ALAP}	t_{ALAP} *[ns]*
E_1	1	0	-1	0
E_2	1	11	-1	11
E_3	1	21	-1	21
E_4	1	0	-1	0
E_5	1	21	-1	21
E_6	1	0	-1	3
E_7	1	11	-1	11
E_8	1	11	-1	11
E_9	1	19	-1	19
E_{10}	1	5	-1	5
E_{11}	1	21	100	21
E_{12}	1	5	-1	5
E_{13}	1	11	-1	13
E_{14}	1	11	-1	13
E_{15}	1	19	-1	21

Alle Ergebnisse in Tabelle 7 zeigen für das Ereignis E_{11} die gleichen Ergebnisse und geben für die gesamte Abarbeitung eine Zeit von 21 ns vor. Aus den Ergebnissen wird ersichtlich, welche Zustände zeitgleich ausgeführt werden können. So sind zum Beispiel bei der ASAP Strategie die Ereignisse E_1, E_4 und E_6 alle dem Zeitpunkt 0 zugeordnet. Während die zeitgleiche Ausführung innerhalb des Co-Prozessors in Hardware möglich ist, muss die Ausführung in Software sequentiell erfolgen. Dies verlangsamt die Testausführung und führt unter Umständen auch zu einer Verletzung der zulässigen zeitlichen Bedingungen zwischen den Ereignissen.

[48] ASAP: As Soon As Possible – Zum frühestmöglichen Zeitpunkt

[49] ALAP: As Late As Possible – Zum spätestmöglichen Zeitpunkt

Die in Tabelle 7 dargestellten Ergebnisse sind für die Umsetzung im FPGA jedoch nicht direkt nutzbar. Dieser benötigt für die Umsetzung eine getaktete Lösung. Hierbei hängt der Takt von den Randbedingungen des Testsystems ab. Die getaktete Lösung erhält man durch die Umrechnung von $A_{i,j}$ gemäß folgender Formel

$$A_{i,j}^{c} = \begin{cases} -\left\lceil \frac{|A_{i,j}|}{T} \right\rceil & if\ A_{i,j} \leq 0 \\ \left\lfloor \frac{A_{i,j}}{T} \right\rfloor & if\ A_{i,j} > 0 \end{cases} \tag{4.6}$$

Für einen gegebenen Takt wird aus $A \in (\mathbb{R} \cup \#)^{n \times n}$ die Matrix $A^c \in (\mathbb{N} \cup \#)^{n \times n}$. Hiermit kann erneut durch eine identische Zeitplanung eine gültige Lösung für einen vorgegebenen Takt gefunden werden, sofern eine Lösung existiert. Wird dieser Takt mit der Taktzeit T multipliziert erhält man eine zeitliche Lösung, die sowohl getaktet ist als auch alle zeitlichen Randbedingungen einhält.

Tabelle 8 zeigt die Lösung der ASAP und ALAP Optimierung für eine gewählte Taktzeit von $T = 10ns$. In Abhängigkeit der verfügbaren Takte des verwendeten FPGAs kann das Testinstrument für die unterschiedlichsten Takte generiert werden.

Tabelle 8: Ergebnis getakteter Optimierung für $T = 10ns$

Funktionen & Zeiten / Ereignisse	f_{ASAP}	c_{ASAP}	t_{ASAP} *[ns]*	f_{ALAP}	c_{ALAP}	t_{ALAP} *[ns]*
E_1	1	0	0	-1	0	0
E_2	1	2	20	-1	2	20
E_3	1	3	30	-1	3	30
E_4	1	0	0	-1	0	0
E_5	1	3	30	-1	3	30
E_6	1	0	0	-1	1	10
E_7	1	2	20	-1	2	20
E_8	1	2	20	-1	2	20
E_9	1	3	30	-1	3	30
E_{10}	1	1	10	-1	1	10
E_{11}	1	4	40	100	4	40
E_{12}	1	1	10	-1	1	10
E_{13}	1	2	20	-1	2	20
E_{14}	1	2	20	-1	2	20
E_{15}	1	3	30	-1	4	40

4.4.3. ERZEUGUNG DES ZUSTANDSAUTOMATEN

Basierend auf den Ergebnissen der automatischen Zeitplanung der Ebene L1 in Kapitel 4.4.2 kann nun ein Zustandsautomat für die spätere Implementierung im Testinstrument generiert werden. Dieser Automat ist ein endlicher Zustandsautomat und kann direkt aus Tabelle 8 abgeleitet werden. Diese Zustandsmaschine ist ein zyklisch gerichteter Graph, welcher direkt für eine VHDL Implementierung zur Steuerung des DUTs genutzt werden kann. Der Graph erfüllt alle zeitlichen Randbedingungen, die in Abbildung 30 und Abbildung 31 definiert worden sind. Abbildung 49 zeigt die dazugehörige Lösung. Wie zu sehen ist, wurden den Zuständen q_1 bis q_5 genau die Aktionen zugeordnet, die zu den dazugehörigen Ereignissen E_1 bis E_{15} gehören und ebenfalls im DUT-Modell definiert werden. Die zeitliche Zuordnung mit $q_{n+1} = c_n$ aus Tabelle 8 ist in Tabelle 9 dargestellt.

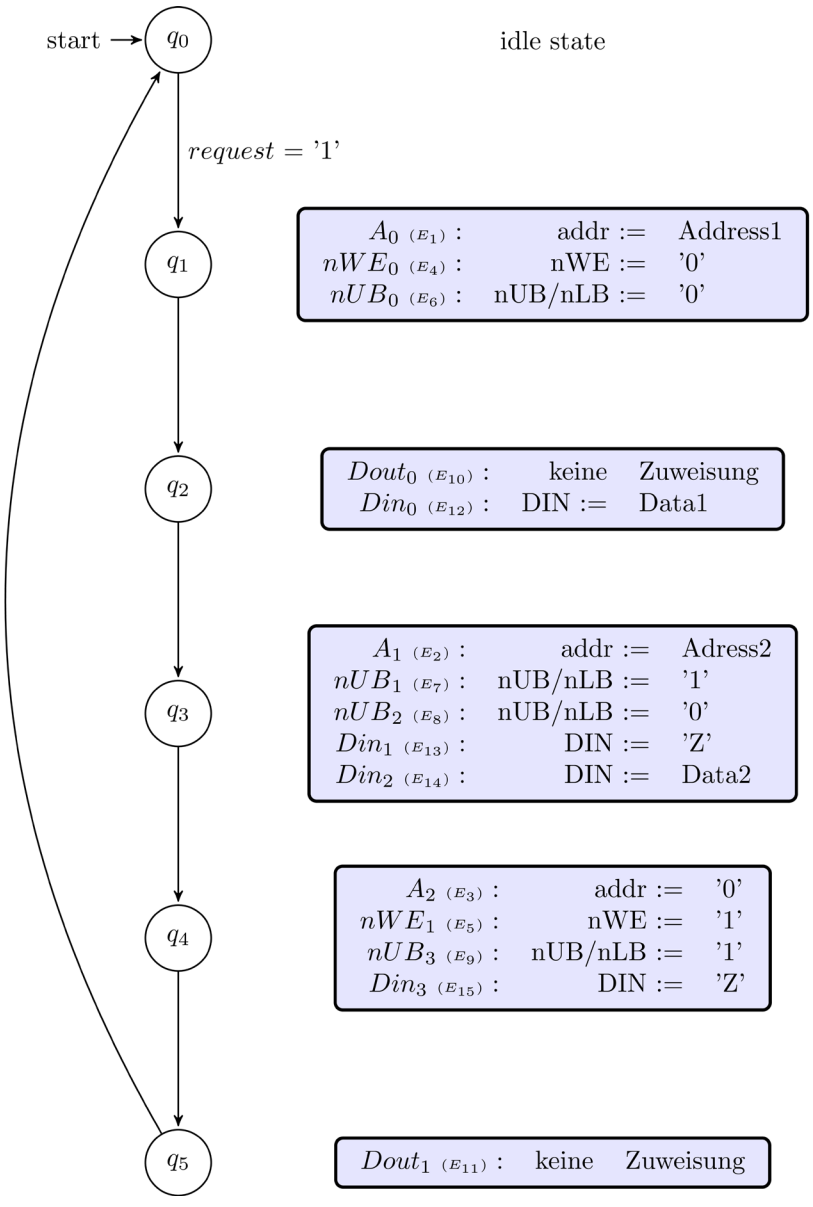


Abbildung 49: Zustandsmaschine für c_{ASAP}

Tabelle 9: Zuordnung FSM Zustände

Zustände & Zeiten / Ereignisse	c_{ASAP}	t_{ASAP} [ns]	FSM – Zustände
E_1	0	0	q_1
E_2	2	20	q_3
E_3	3	30	q_4
E_4	0	0	q_1
E_5	3	30	q_4
E_6	0	0	q_1
E_7	2	20	q_3
E_8	2	20	q_3
E_9	3	30	q_4
E_{10}	1	10	q_2
E_{11}	4	40	q_5
E_{12}	1	10	q_2
E_{13}	2	20	q_3
E_{14}	2	20	q_3
E_{15}	3	30	q_4

4.4.4. KONFLIKTDETEKTION / -LÖSUNG

In Abbildung 49 wird deutlich, dass einige Zustände widersprüchliche Zuweisungen enthalten. Dies betrifft die Zuweisungen E_7 und E_8 sowie E_{13} und E_{14} im Zustand q_3. Hierbei handelt es sich um Konflikte, die korrigiert werden müssen, um eine funktionierende Implementierung der FSM zu erhalten. In diesem Kapitel sollen entsprechende Methoden zur Konfliktdetektion und Konfliktlösung beschrieben werden.

Konflikte sind Zuweisungen von unterschiedlichen Werten zu einem Signal zum gleichen Zeitpunkt. Diese können verschiedene Ursachen haben.

- Fehlende Informationen in den Datenquellen der DUT Beschreibung, die nicht durch die Regeln der Matrixvervollständigung in Kapitel 4.2.6 behoben wurden
- Das Aneinanderreihen von sich wiederholenden Aktionsteilen

Nachfolgend werden diese zwei Fälle am generierten Zustandsautomaten aus Abbildung 49 verdeutlicht. Dort sind beide Konflikte in Zustand q_3 sichtbar.

Der **erste Konflikttyp** betrifft die Ereignisse E_7 und E_8. In diesem Fall wird dem Signal nUB/nLB im Zustand q_3 gleichzeitig eine logische 1 und eine logische 0 zugewiesen. Da dies gegensätzliche Zuweisungen sind, gibt es hier einen **nicht automatisch lösbaren Konflikt**.

Ein nicht automatisch lösbarer Konflikt kann vom Co-Prozessor Compiler nicht selbständig gelöst werden. Ein solcher Konflikt impliziert immer einen Fehler in den Datenquellen oder beim Generieren der Difference Bound Matrix.

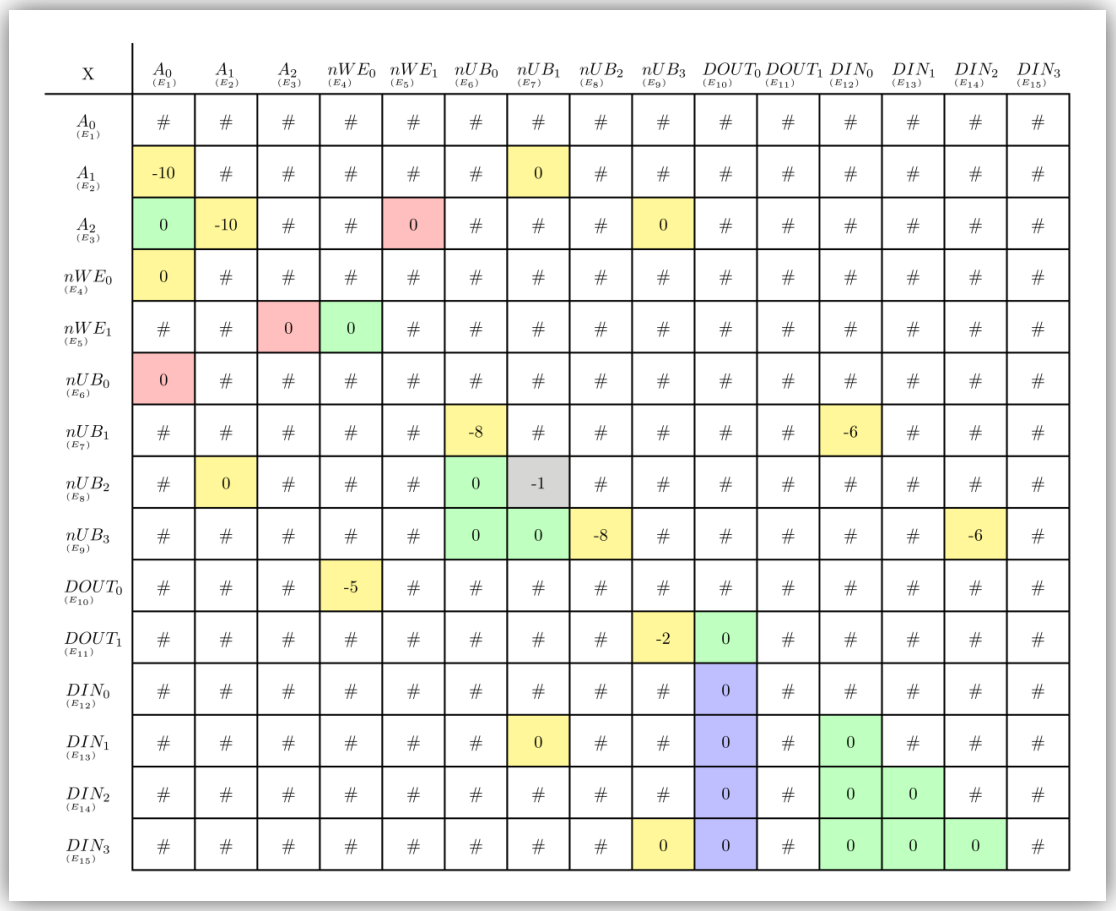
Diese Art von Konflikten kann bei der Matrixvervollständigung noch nicht geprüft werden, da es nicht zwingend zwischen aufeinanderfolgenden Ereignissen direkte zeitliche Randbedigungen geben muss. Zwar definieren die Regeln zur Matrixvervollständigung in Kapitel 4.2.6, dass E_8 nicht vor E_7 stattfinden darf, aber eine zeitgleiche Ausführung ist durch diese Regeln zulässig.

Statt direkter zeitlicher Abhängigkeiten können auch indirekte Abhängigkeiten zwischen anderen Ereignissen bestehen, die dafür sorgen, dass das gewünschte Verhalten zwischen den betrachteten Ereignissen eintritt. So ist im beschriebenen Beispiel zwar eine Hold-Zeit t_{HA} zwischen den Ereignissen E_7 und E_2, sowie eine Setup-Zeit t_{SA} zwischen E_2 und E_8, definiert, jedoch dürfen diese beiden Zeiten $0ns$ betragen. Die Optimierung kann somit die Zeit zwischen E_7 und E_8 zu 0 setzen und es werden die beiden angesprochenen Ereignisse dem gleichen Zeitpunkt zugeordnet.

Der Fehler kann also auf eine unvollständige Beschreibung im Datenblatt des DUT zurückgeführt werden. Der Konflikt wird durch das Einführen einer zusätzlichen zeitlichen Randbedingung gelöst. Wie groß diese Randbedingung sein muss, hängt vom DUT ab und kann entweder aufgrund von Erfahrungen des Testingenieurs festgelegt werden oder ist beim Hersteller des DUT zu erfragen. Im vorliegenden Fall wird aufgrund von Erfahrungen eine beliebige Zeit > 0 festgelegt, da diese durch die eingeführte Taktung automatisch auf einen ganzen Takt erweitert wird. Die resultierende DBM ist in Abbildung 50 dargestellt. Die hinzugefügte Randbedingung ist grau hinterlegt.

Der **zweite Konflikttyp** betrifft die Ereignisse E_{13} und E_{14}. Dort wird dem Signal D_{in} einmal ein hochohmiger Zustand (Z) zugewiesen, der Bus also abgeschaltet, das andere Mal werden die nächsten Daten angelegt. Diese können für jedes Bit entweder 1 oder 0 sein und werden als DATA(0/1) bezeichnet. Diese zwei Zuweisungen ergeben sich aus dem Aneinanderreihen der Teilaktionen „write“ zur hier dargestellten Aktion „back-to-back-write“ des Schreibbefehls. Hierbei handelt es sich um einen **automatisch lösbaren Konflikt**.

Lösbare Konflikte können vom Compiler selbständig behoben werden. Im vorliegenden Beispiel ist der Konflikt durch eine logische Verknüpfung des hochohmigen Zustands ‚Z‘ und DATA(0/1) lösbar, bei der ‚Z‘ von einer logischen 0 bzw. 1 dominiert wird. Dies entspricht dem elektrischen Verhalten von zwei Treibern, von denen einer Z und der andere eine 0 oder 1 ausgibt. Entsprechend dieser Vorschrift kann das Ereignis E_{13} aus der FSM gelöscht werden und nur die Zuweisung von E_{14} bleibt in q_3 bestehen.



X	A_0 (E_1)	A_1 (E_2)	A_2 (E_3)	nWE_0 (E_4)	nWE_1 (E_5)	nUB_0 (E_6)	nUB_1 (E_7)	nUB_2 (E_8)	nUB_3 (E_9)	$DOUT_0$ (E_{10})	$DOUT_1$ (E_{11})	DIN_0 (E_{12})	DIN_1 (E_{13})	DIN_2 (E_{14})	DIN_3 (E_{15})
A_0 (E_1)	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
A_1 (E_2)	-10	#	#	#	#	#	0	#	#	#	#	#	#	#	#
A_2 (E_3)	0	-10	#	#	0	#	#	#	0	#	#	#	#	#	#
nWE_0 (E_4)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nWE_1 (E_5)	#	#	0	0	#	#	#	#	#	#	#	#	#	#	#
nUB_0 (E_6)	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#
nUB_1 (E_7)	#	#	#	#	#	-8	#	#	#	#	#	-6	#	#	#
nUB_2 (E_8)	#	0	#	#	#	0	-1	#	#	#	#	#	#	#	#
nUB_3 (E_9)	#	#	#	#	#	0	0	-8	#	#	#	#	#	-6	#
$DOUT_0$ (E_{10})	#	#	#	-5	#	#	#	#	#	#	#	#	#	#	#
$DOUT_1$ (E_{11})	#	#	#	#	#	#	#	#	-2	0	#	#	#	#	#
DIN_0 (E_{12})	#	#	#	#	#	#	#	#	#	0	#	#	#	#	#
DIN_1 (E_{13})	#	#	#	#	#	#	0	#	#	0	#	0	#	#	#
DIN_2 (E_{14})	#	#	#	#	#	#	#	#	#	0	#	0	0	#	#
DIN_3 (E_{15})	#	#	#	#	#	#	#	#	0	0	#	0	0	0	#

Abbildung 50: Ergebnis DBM nach der Konfliktlösung

4.4.5. VERGLEICH DER ERGEBNISSE

Das Ergebnis der generierten Ansteuerung aus Abbildung 51 für einen Taktperiode von 10 ns und der zugrunde gelegten Waveform aus Abbildung 52 ist in Abbildung 53 zu sehen. Zur besseren Übersichtlichkeit sind Zeiten, die in der Realisierung 0 ns betragen nicht dargestellt worden. Hierzu gehören t_{HA} sowie t_{HD} und eine t_{SA} Randbedingung.

Es ist eine gewisse Asymmetrie der Realisierung in Abbildung 53 zu sehen. Dies betrifft insbesondere die Signale nUB/nLB und Din. Der Grund hierfür ist die im FPGA erforderliche Taktung, die Ereignisse und somit Signalwechsel nur zu Taktflanken ermöglicht. Dies verzögert das Anlegen von $Data1$ auf Din und erfordert einen Takt Wartezeit, bis $Dout$ einen hochohmigen Zustand angenommen hat. Hinzu kommt ein extra Takt für den Signalwechseln von nUB/nLB, da Daten nur bei einem Signalwechsel von 0 auf 1 von Din übernommen werden. Es ist somit nötig, nUB/nLB für die Übernahme von $Data1$ und $Data2$ jeweils einmal von 0 auf 1 wechseln zu lassen.

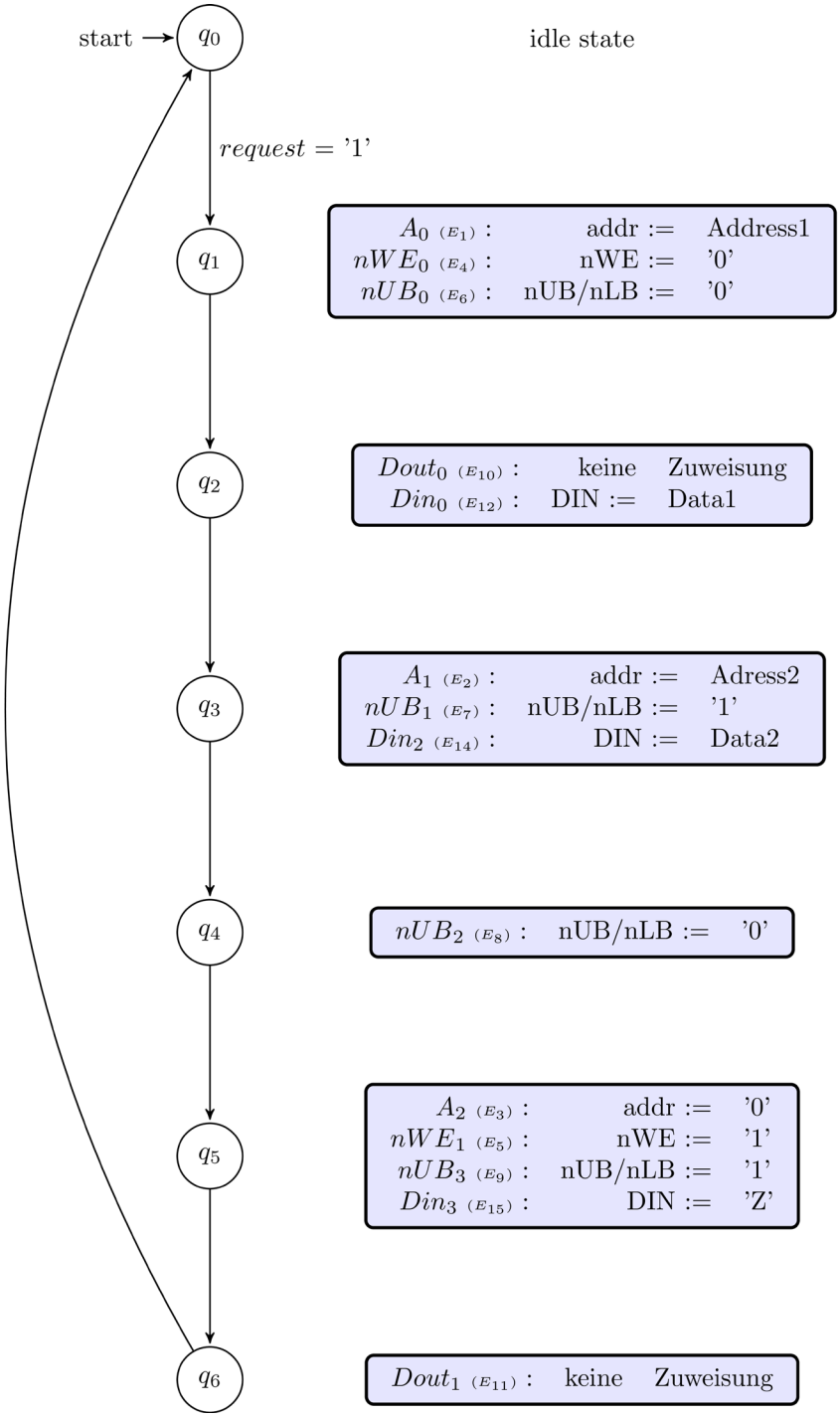


Abbildung 51: Zustandsmaschine für c_{ASAP} nach der Konfliktlösung

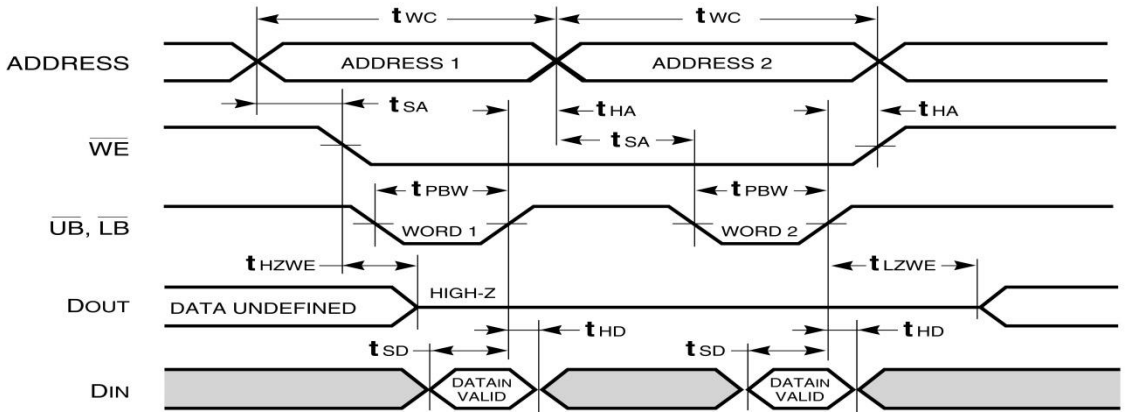


Abbildung 52: Reduzierte Waveform aus dem Datenblatt [22]

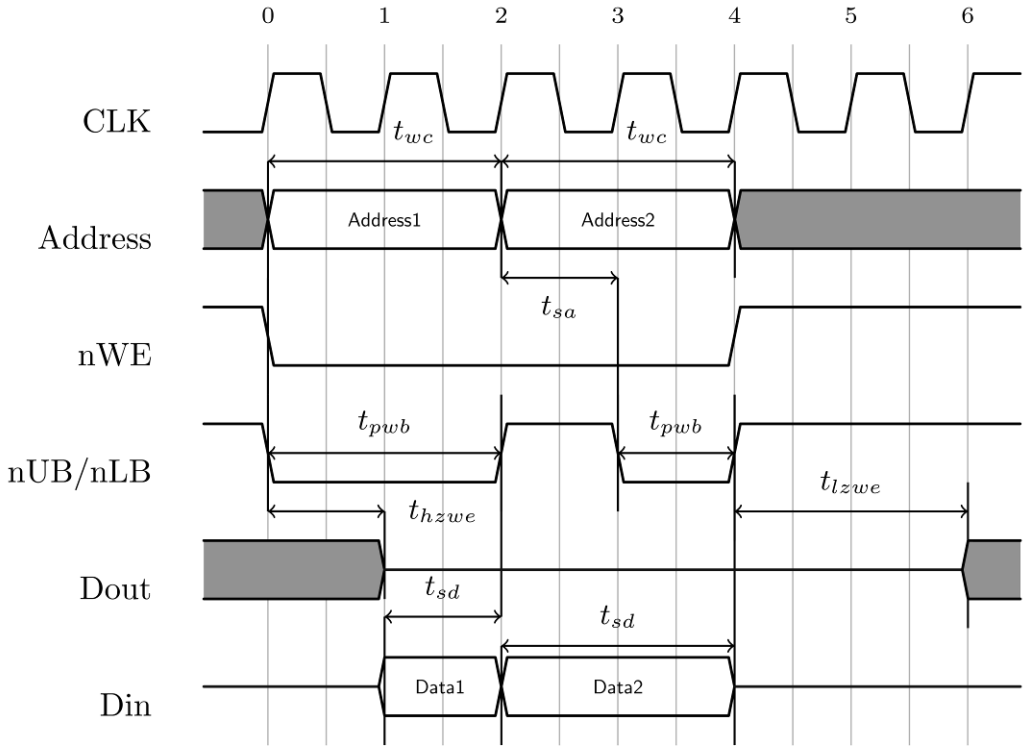


Abbildung 53: Waveform basierend auf DBM-basierter Ansteuerung

4.4.6. GENERIEREN DES CO-PROZESSORS

Die Generierung eines Co-Prozessors ist abhängig von der gewählten Partitionierung und wird für jeden Co-Prozessor nacheinander und unabhängig voneinander durchgeführt. Für jeden Co-Prozessor werden entsprechend Abbildung 54 die in Hardware zu generierenden Schichten sowie das Interface I1 in einem Wrapper zusammengefasst.

Im Folgenden wird näher auf den Inhalt der Ebene L1 eines Co-Prozessors eingegangen, da diese Ebene die wichtigste für ein korrektes zeitliches Verhalten des Testinstruments ist und daher möglichst immer in Hardware realisiert werden sollte.

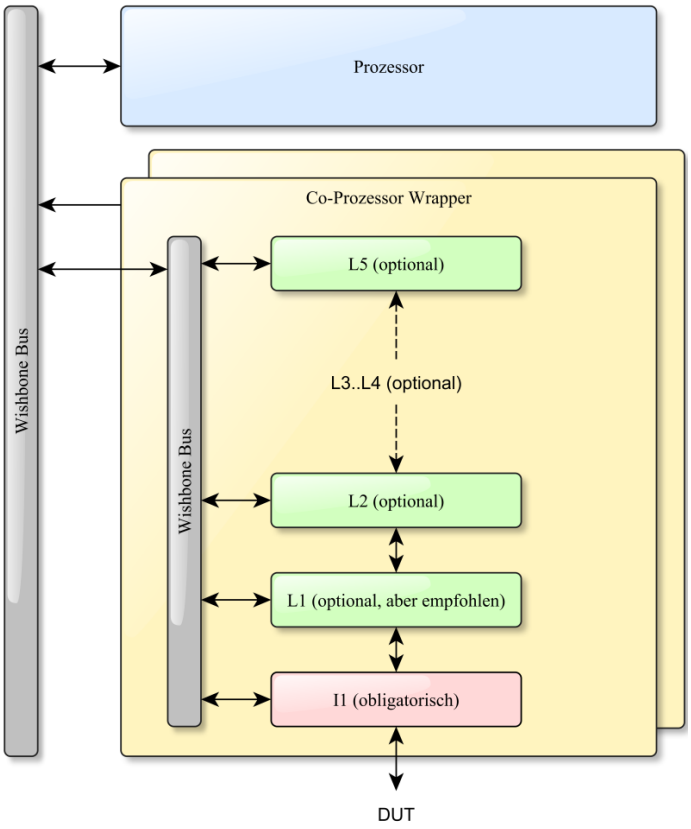


Abbildung 54: Struktur eines Co-Prozessors

Generell wird die Generierung für jeden Co-Prozessor getrennt durchgeführt. Es gibt aber einige Informationen, die zusätzlich zum entsprechenden DUT-Modell des Testinstruments notwendig sind. Diese sind:

- bereits belegte Registeradressen durch andere generierte Co-Prozessoren
- Eigenschaften des verwendeten Testprozessors

Die Generierung berücksichtigt die Prozessorkonfiguration wie z.B. Adress- und Datenbusbreiten und generiert das Wishbone-Interface entsprechend. Die Bestimmung und Zuweisung der benötigten Register erfolgt automatisch. So benötigt ein 16 Bit Datenbus bei der Verwendung eines 8 Bit Testprozessors zwei Register, während bei einem 16 Bit Prozessor ein Register ausreichend ist. Die Zuweisung der Wishbone-Register zum Interface I1, egal ob dies eins, zwei oder mehrere lesende oder schreibende Register sind, erfolgt ebenfalls automatisch.

Weiterhin wird ein Signalisierungsbit generiert, das in Abhängigkeit aller vorhandenen FSMs in L1 signalisiert, ob Ebene L1 aktuell aktiv ist. Dies wird auf den übergeordneten Ebenen im Wrapper und Top-Level genutzt, um die Aktivitäten zwischen mehreren Co-Prozessoren zu steuern und auszuwählen, welcher Zugriff auf die Pins des FPGAs hat. Dies ist insbesondere bei Pins notwendig, die von mehreren Co-Prozessoren genutzt werden, wie dies zum Beispiel bei einem gemeinsamen Datenbus der Fall ist.

Bestandteile der VHDL Beschreibung der Ebene L1 des Co-Prozessors sind:

- Entity mit Takt, Reset, Enable Signal und Wishbone-Bus-Interface, Schnittstelle zu Interface I1 sowie ggf. zu Modulen der Ebene L2
- Typen und Signaldefinitionen der FSM [50], der Wishbone-Register und Verbindungsleitungen der Schnittstellen zu Interface I1 und ggf. Modulen der Ebene L2
- Wishbone-Prozesse zum Lesen und Schreiben der Register
- FSM(s) entsprechend der definierten DBM(s)
- Signalzuweisungen für das Interface I1 entsprechend der Abhängigkeiten der FSM(s) und der Wishbone-Register

Ein kommentiertes RTDL Beispiel von I1 und L1 für die Generierung eines Co-Prozessors für die Ansteuerung des SRAMs IS61LV25616 [22] ist in Anhang L zu finden.

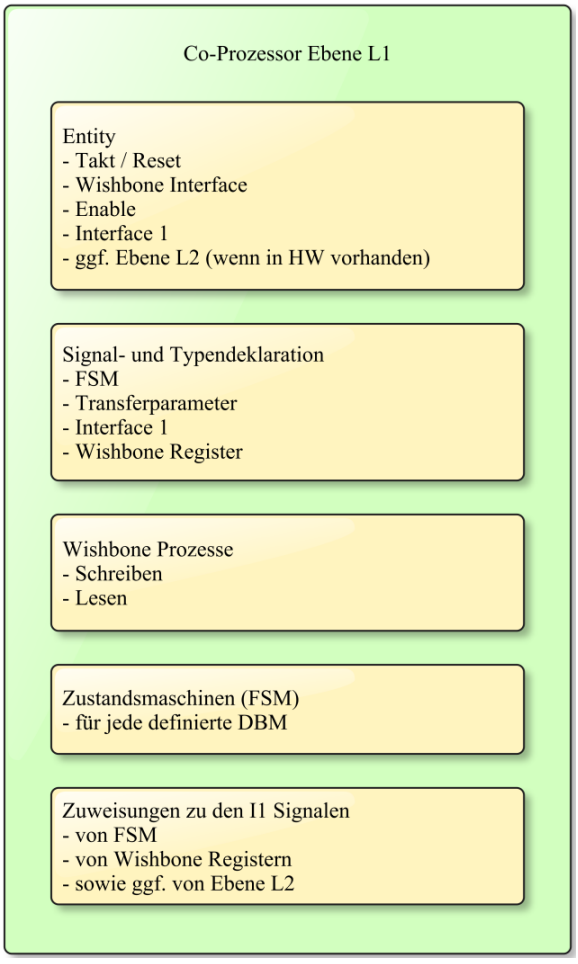


Abbildung 55: Struktur der Ebene L1 eines Co-Prozessors

[50] Es können auch mehr als eine FSM auf der Ebene L1 vorhanden sein.

4.4.7. HIERARCHISCHE GENERIERUNG

Neben der Generierung *einer* FSM basierend auf *einer* DB-Matrix, gibt es noch die Möglichkeit, einen Co-Prozessor basierend auf einer hierarchischen Modellierung durchzuführen. Diese Art der Modellierung ist in Kapitel 4.2.7 beschrieben worden. Wenn die hierarchischen Matrizen nicht vorher in eine DB-Matrix überführt werden sollen, ist für eine hierarchische Modellierung auch eine hierarchische Generierung notwendig.

Die Vorteile einer hierarchischen Generierung sind:

- Kompakte FSMs, da diese weniger Zustände haben
- ggf. geringerer Ressourcenverbrauch durch Wiederverwenden von Strukturen
- strukturierter VHDL Code

Das Problem bei einer getrennten Realisierung jeder DB-Matrix in einer FSM ist die Synchronisation über Hierarchieebenen hinweg. Während für die Ebenen L2 bis L5 ein zusätzlicher Takt für das Handshake zwischen den Hierarchieebenen unkritisch ist, da dort keine zwingend einzuhaltenden zeitlichen Vorgaben existieren, ist dies auf L1 nicht so einfach. So müssen sich z.B. die einzelnen Bestandteile des in Kapitel 4.2.7 verwendeten I^2C Protokolls nahtlos aneinanderreihen, um die zeitlichen Vorgaben der Modellierung einzuhalten.

Als Realisierungsvarianten der FSM über mehrere Hierarchieebenen sind sowohl Moore- als auch Mealy-Automaten betrachtet worden.

Der Moore-Automat benötigt für die Synchronisation zwischen jeweils zwei Hierarchieebenen einen extra Takt, generiert dann jedoch nur in Abhängigkeit des Zustands eine Ausgabe und lässt sich somit höher takten als ein vergleichbarer Mealy-Automat. Dieser realisiert hingegen eine schnellere Reaktionszeit, da er bereits auf Wechsel der Eingangssignale reagieren kann und somit für die erste Aktion nicht auf einen Zustandswechsel in der übergeordneten Hierarchieebene angewiesen ist. Dies setzt jedoch zusätzliche Logik in der Ausgabefunktion des Mealy-Automaten voraus, die wiederum ihren Teil zur gesamten Signallaufzeit beiträgt und damit die maximale Taktfrequenz reduziert.

Prinzipiell ist der Einsatz von Mealy-Automaten einfacher, da hierbei zwar die erreichbare Taktfrequenz sinkt, jedoch können die in den DB-Matrizen definierten Zeiten direkt verwendet werden, da die hierarchischen Zustandsmaschinen ohne zusätzliche Takte zur Synchronisation auskommen. Ob die erreichte Taktfrequenz den Vorgaben des verwendeten Taktes entspricht, wird vom Synthesetool geprüft und muss somit nicht vom Compiler der Co-Prozessoren berücksichtigt werden.

Beim Einsatz eines Moore-Automaten hingegen verändert der zusätzliche Takt für die Synchronisation der Hierarchieebenen das zeitliche Verhalten. Dies muss vom Compiler berücksichtigt werden, um nicht doch zeitliche Randbedingungen zu verletzen. Die Verwendung beider Möglichkeiten bietet Spielraum für Optimierung, um die kürzeste

Testzeit zu erreichen. Aufgrund des komplexen Optimierungsprozesses wird diese Optimierung jedoch im Rahmen dieser Arbeit nicht weiter betrachtet. Auf die Komplexität des Optimierungsprozesses wurde bereits in Kapitel 4.3 näher eingegangen.

In [113] werden die beiden beschriebenen Ansätze ausführlich anhand der Ansteuerung eines seriellen Flashs untersucht. Insbesondere wird dort auf die Kaskadierung mehrerer Matrizen bei einer hierarchischen Generierung eingegangen. Für den weiteren Verlauf dieser Arbeit wird die hierarchische Generierung jedoch nicht verwendet, es wird immer eine flache Struktur für die Realisierung gewählt.

4.5. VERIFIKATION UND VALIDIERUNG

In diesem Abschnitt sollen die Möglichkeiten der Verifikation und Validierung des generierten Systems beschrieben werden.

In [18] wird besagt, „*Die Verifikation ist ein Beweisverfahren. In Zusammenhang mit dem Schaltungsentwurf versteht man darunter die Prüfung der Konsistenz zweier unterschiedlicher Beschreibungen für das gleiche Schaltungsmodul. Sie wird üblicherweise eingesetzt nach dem Übergang von einer Beschreibungsebene auf die nächst tiefer liegende Ebene*“.

Bei der Validierung hingegen wird „*stichprobenartig durch Experimente die Übereinstimmung des Verhaltens eines Moduls mit dem spezifizierten Verhalten*“ gezeigt [18]. Im Gegensatz zur Verifikation ist die Validierung unvollständig. „*Sie beinhaltet ein Experiment, während die Verifikation mit formalen mathematischen Mitteln erfolgt. Aus dem Umfang des Experiments kann bestenfalls auf die Wahrscheinlichkeit der Fehlerfreiheit geschlossen werden*“ [18].

Die Überführung der DBM in eine FSM kann verifiziert werden, indem nachgewiesen wird, dass die generierte FSM eine gültige Lösung der verwendeten DBM darstellt. Hierfür wird aus der generierten FSM wieder eine DBM generiert, die im Gegensatz zur ersten jedoch einen deutlich geringeren Freiheitsgrad aufweist und lediglich eine exakte Lösung im Lösungsraum beschreibt. Werden jetzt beide Matrizen fusioniert und erneut gelöst (wie in Kapitel 4.2.6 beschrieben), so muss sich wieder die gleiche Lösung ergeben. Somit entspricht die generierte FSM genau einer gültigen Lösung der ursprünglichen DBM.

Das Konzept der Co-Prozessor Modellierung und Generierung erlaubt es weiterhin, unterschiedliche Eigenschaften zu verschiedenen Zeitpunkten der Systemgenerierung zu validieren. Hierfür wird teilweise ebenfalls auf die DB-Matrix zurückgegriffen, da dies der zentrale Bestandteil der zeitlichen Modellierung ist.

Es werden drei unterschiedliche Arten an Fehlern für die Validierung unterschieden:

- Fehlerhafte oder fehlende Informationen der Datenquelle, die zur Modellierung des DUT genutzt werden soll

- Fehler im Graphen oder der Optimierung der Difference Bound Matrix
- Fehler im generierten VHDL Code

Diese drei Fehlerarten werden nachfolgend näher beschrieben.

4.5.1. FEHLERHAFTE SOWIE FEHLENDE INFORMATIONEN

Diese Art der Fehler kann automatisch detektiert und gemeldet werden. Hierbei liegt es in der Verantwortung des Testingenieurs, die entsprechenden Maßnahmen zur Korrektur einzuleiten. Im Folgenden werden ein paar Beispiele hierfür gegeben.

- Signale im Waveformdiagramm ohne jegliche Zusammenhänge zu anderen, wie z.B. die Signale nCE und nOE in Abbildung 31, sind in der DB-Matrix daran zu erkennen, dass weder in der zugehörigen Spalte noch in der Zeile irgendeine zeitliche Abhängigkeit vorhanden ist. Diese Signale werden daraufhin dem Testingenieur gemeldet. Er muss entscheiden, wie diese Signale vom Co-Prozessor angesteuert werden sollen oder nicht. Zumeist können diese Signale während des Betriebs des Co-Prozessors statisch auf einen definierten Wert gelegt werden.
- Ereignisse in der DB-Matrix, die zeitliche Randbedingungen zu anderen haben, denen jedoch keine zugehörige Aktion zugeordnet ist, deuten auf fehlende Zuweisungen hin. Der Testingenieur muss dies prüfen und ggf. nop-Aktionen[51] einfügen oder die Einträge in der DBM korrigieren.
- Fehlerhafte Randbedingungen, die zu unlösbaren Konflikten führen, wie diese in Kapitel 4.4.4 beschrieben wurden, können vom Compiler des Testsystems erkannt und gemeldet werden. Der Testingenieur muss die entsprechenden Korrekturen im RTDL Modell durchführen.

4.5.2. FEHLER IM GRAPHEN ODER DER DBM OPTIMIERUNG

Diese Art der Fehler können während der Generierung des ersten Graphen (vgl. Abbildung 36) oder der Optimierung der Difference Bound Matrix erkannt werden. Im Folgenden werden ein paar Beispiele hierfür gegeben.

- Der generierte Graph ist ein zusammenhängender, gerichteter Graph. Daher sind ‚lose' Knoten nicht zulässig. Diese können entstehen, wenn einige der Regeln zur Matrixgenerierung verletzt worden sind und es Ereignisse gibt, die in keinem Zusammenhang mit anderen stehen. Diese Fehler werden dem Testingenieur gemeldet, der entweder die benötigten Zeiten hinzufügen muss oder aber das Signal aus der Liste der relevanten Signale zu entfernen hat.

[51] nop : no-operation

- Für eine Implementierung auf einem FPGA wird der ‚erste' gerichtete Graph aus Abbildung 36 zu einer spezifischen getakteten Implementierung (siehe Abbildung 51) gewandelt und optimiert. Zum Vergleich dieser zwei Graphen können gängige Methoden zum Nachweis der Gleichheit von Graphen unter gewissen Gesichtspunkten durchgeführt werden. Weiterhin ist es auch wieder möglich, den Graphen für die getaktete Realisierung in eine DBM zurück zu überführen und zu verifizieren, dass diese eine gültige Lösung der ursprünglichen DB-Matrix darstellt.

4.5.3. FEHLER IM VHDL CODE

Die dritte Art der Fehler wird nicht mehr vom ROBSY Generator geprüft, sondern vom Synthesetool des FPGA Herstellers. Dieser prüft den VHDL Code syntaktisch und während der Synthese auch auf logische Fehler oder auf Architekturfehler, d.h. auf nicht umsetzbare Forderungen, wie das Zuweisen von zwei Signalen zu einem Pin oder das Nutzen von nicht vorhandenen Ressourcen. Dies könnte z.B. der Fall sein, wenn ein komplexes Testinstrument zu groß ist für den gewählten FPGA.

Das Synthesetool übernimmt auch die Prüfung, ob das gewünschte Timing des Co-Prozessors eingehalten werden kann. Dieses Timing hat nichts mit der zeitlichen Modellierung der DB-Matrix zu tun, sondern gibt Auskunft über die maximale Taktfrequenz, mit der die generierte Struktur des gesamten Testinstruments (also Prozessor und Co-Prozessor) betrieben werden kann. Diese muss mindestens so groß sein wie die für das automatische Scheduling gewählte Taktung.

Wird dieses Timing verletzt, bedeutet dies nicht zwangsläufig einen Fehler in der Modellierung bzw. der Überführung in VHDL Code, sondern kann auch schlicht durch die Grenzen des FPGAs begründet sein. Im Falle einer Verletzung der gesetzten Vorgaben muss ggf. die genutzte Taktfrequenz reduziert und die Generierung einer getakteten Lösung der DB-Matrix erneut durchgeführt werden, bis die Synthese ein Ergebnis ohne Timingfehler liefert. Da die maximal erreichbare Taktfrequenz von vielen Faktoren innerhalb des FPGAs abhängt, kann diese nur nach einer Synthese vom Synthesetool des FPGA Herstellers genau bestimmte werden und nicht vorher.

Falls neben der RTDL Beschreibung des DUTs auch ein Funktionsmodell existiert, kann als letzte Validierung noch eine Simulation auf Register-Transfer-Logik Ebene durchgeführt werden. Funktionsmodelle werden meist von den Herstellern bereitgestellt, wie dies für DDR2 bereits beschrieben worden ist. Je nach Komplexität dieser Modelle können somit funktionale und zeitliche Fehler detektiert werden. Diese Validierung lässt sich bisher aber nicht automatisieren und erfordert umfangreiche VHDL Kenntnisse des Testingenieurs.

4.6. ZUSAMMENFASSUNG

Der in Kapitel 4 vorgestellte konzeptionelle Lösungsansatz für das Generieren von eingebetteten Testinstrumenten beruht auf einer Ebenen-basierten Testsystemarchitektur. Zentrales Element ist die Strukturierung der Testfunktionalität in einem entwickelten Ebenenmodell. Die Ausführung der Algorithmen wird dabei durch eine Kombination aus Test-PC, Testprozessor und Co-Prozessor realisiert, denen die Funktionalität der einzelnen Schichten des Ebenenmodells zugeordnet werden kann. Dies erlaubt eine sehr variable Umsetzung der Algorithmen und macht das Testsystem sehr flexibel. Es ist so möglich, auf die Anforderungen des jeweiligen Testfalls einzugehen und das Testsystem an die verfügbaren Ressourcen des Prüflings bzw. die benötigte Testgeschwindigkeit anzupassen. Diese Anforderungen müssen dabei erst zum Zeitpunkt der Generierung und nicht schon bei der Modellierung bekannt sein.

Dies ist nur möglich, wenn das DUT, zu dem die Verbindungen getestet werden sollen, inklusive aller Zugriffsroutinen und Testalgorithmen vollständig als Modell vorliegt. Hierfür wurden im Rahmen dieser Arbeit unterschiedliche Modellierungssprachen auf ihre Eignung hin untersucht und letztendlich eine eigene Sprache entwickelt, die alle Anforderungen erfüllt. Hierbei stellt insbesondere die Modellierung von zeitlichen Abhängigkeiten einen zentralen Bestandteil dar. Im Modell werden jedoch nicht einzelne Lösungen beschrieben, sondern der gesamte Lösungsraum für eine Zugriffsroutine. Dies ist nötig, um zum späteren Zeitpunkt der Testsystemgenerierung die Ansteuerung an die Randbedingungen wie z.B. den verwendeten Takt des Prüflings anzupassen. Im Rahmen dieser Arbeit ist eine entsprechende Lösung erarbeitet und dargestellt worden.

Weiterhin wird auf die Probleme der Partitionierung eines Testsystems, bestehend aus Test-PC, Testprozessor und Co-Prozessor eingegangen und die Komplexität einer Optimierung mit den gegenseitigen Abhängigkeiten dargestellt.

Der Partitionierung folgt die Generierung, bei der sowohl auf die Erzeugung des Gesamtsystems als auch insbesondere der Co-Prozessoren eingegangen wird. Hierbei werden, ausgehend von unterschiedlichen Eingangsbeschreibungen und darin definierten Randbedingungen, die einzelnen Teile des Testsystems erzeugt. Dies ist in einem Designflow dargestellt worden. Die Generierung der Ebene L1 der Co-Prozessoren basiert auf den Modellierungen der zeitlichen Randbedingungen und beinhaltet eine Optimierung, die eine möglichst schnelle Testausführung für den gewählten Prüfling unter Einhaltung aller Randbedingungen gewährleistet. Der Beschreibung über die Generierung eines Testsystems folgt ein Abschnitt über die Möglichkeiten der Verifikation und Validierung des Systems.

Die Bewertung der Leistungsfähigkeit des Konzepts und der daraus generierten Co-Prozessoren und Testsysteme wird im Rahmen von verschiedenen Untersuchungen im folgenden Kapitel 5 dargestellt.

5. UNTERSUCHUNGEN

In diesem Kapitel werden die durchgeführten Untersuchungen und die sich daraus ergebenden Ergebnisse dargestellt. Die Ergebnisse sind in vier unterschiedliche Bereiche unterteilt:

- die Simulationen von automatisch generierten Co-Prozessoren der Ebene L1 (ohne Prozessor) für verschiedene DUTs werden in Kapitel 5.3 vorgestellt
- experimentelle Untersuchungen von automatisch generierten Co-Prozessoren der Ebene L1 auf physischer Hardware zusammen mit einem Prozessor für die Ausführung von L2 bis L5 in Embedded Software werden in Kapitel 5.4 durchgeführt
- ein Vergleich der Ergebnisse des Testsystems mit dem Testprozessor aus Kapitel 5.4 mit einem ausschließlich in Hardware realisierten Testinstrument erfolgt in Kapitel 5.5
- eine Abschätzung der Leistungsfähigkeit im Vergleich zu den „virtuellen Test-instrumenten“ aus [15] beim Programmieren von Flashspeicher erfolgt in Kapitel 5.6
- der Vergleich verschiedener Partitionierungen manuell generierter Co-Prozessoren mit einer Umsetzung bis zur Ebene L3 ist in Kapitel 5.7 dargestellt

Im Vorfeld zu den Ergebnissen wird in Kapitel 5.1 die Modellierung der für die Untersuchungen relevanten DUTs beschrieben sowie die zugehörige Testsystem-generierung in Kapitel 5.2. Hierbei werden die Möglichkeiten der automatischen Generierung dargestellt und es wird auch auf die nötigen manuellen Einstellungen eingegangen.

Für die Untersuchungen sind drei verschiedene DUTs ausgewählt worden. Diese sind:

- ein SRAM vom Typ IS61LV102416AL in der Konfiguration 1Mx16 von ISSI
- das Display des VarioTAP-Coachs der Firma GÖPEL electronic mit Auflösung von 128x64 Pixel, welches über einen separaten CPLD mit dem FPGA verbunden ist
- ein Flash vom Typ S29GL064N in der Konfiguration 8Mx8 von Spansion

Für die Untersuchungen wurden verschiedene Lese- und Schreibzugriffe realisiert und die erreichten Testzeiten der Simulation bzw. der praktischen Realisierung mit einer optimierten Ansteuerung bzw. anderen Testverfahren verglichen.

5.1. MODELLIERUNG DER DUTS

Die Modellierung der drei betrachteten DUTs basiert ausschließlich auf den aus den Datenblättern entnommenen Daten. Die verwendeten Befehle der einzelnen DUTs sind im Folgenden kurz vorgestellt. Relevante Auszüge der Datenblätter sind den entsprechenden Anhängen zu entnehmen.

5.1.1. SRAM

Das Modell für das in der Untersuchung verwendete SRAM ist in Anhang L dargestellt. Hierbei ist anzumerken, dass die praktische Untersuchung mit dem Typ IS61LV102416AL statt des im Anhang und dem Rest der Arbeit verwendeten IS61LV256416AL durchgeführt worden ist. Dies liegt an dem verwendeten Entwicklungsboard, welches den größeren Speichertyp verwendet. Die Modelle sind bis auf den um zwei Bit größeren Adressbus jedoch gleich. Die Ergebnisse der Simulation aus Kapitel 5.3 sind für beide Modelle identisch, da hier die Adressbusbreite keinen Einfluss auf die Messwerte hat.

Für die Untersuchungen werden folgende Befehle unterstützt:

- CE-basiertes Write
- UB/LB-basiertes Back-to-Back Write
- OE-basiertes Read

5.1.2. DISPLAY

Das verwendete Display befindet sich auf einem von der Firma GÖPEL electronic entworfenen Entwicklungsboard, welches für die Visualisierung und Erprobung von Boundary-Scan basierten Tests entwickelt worden ist. Bei dem Display handelt es sich um ein 128x64 Pixel Grafikdisplay der Firma Electronic Assembly, welches nicht direkt mit dem FPGA verbunden ist, sondern zu dessen Ansteuerung ein CPLD der Firma

Xilinx zwischengeschaltet wurde. Somit ist der verwendete Displaycontroller an dieser Stelle nicht relevant, sondern lediglich die vom CPLD benötigte Ansteuerung. Diese Ansteuerung ist in Anhang Q dokumentiert.

Bei der Modellierung des Displays ist zu beachten, dass es neben den in Abbildung 90 aufgeführten zeitlichen Parametern noch eine im Text definierte Wartezeit von 1µs nach jedem Schreibvorgang gibt. Diese kann entweder durch das Einfügen eines beliebigen Ereignisses ohne Zuweisung in der DBM kodiert werden oder durch das Addieren von Zeitintervallen, wenn die letzte relevante zeitliche Vorgabe auch nur eine Wartezeit ohne zugehörige Aktion ist. Die zweite Variante wurde im Rahmen dieser Arbeit gewählt. Auszüge der entsprechenden Modelle sind in Abbildung 91 und Abbildung 92 im Anhang Q dokumentiert.

Der aktuelle Compiler unterstützt in den DUT-Modellen nur die Definition von Zeitwerten als ganze Nanosekunden. Deshalb müssen bei der Modellierung alle Zeiten entsprechend aufgerundet werden, da es sich in diesem Fall ausschließlich um minimale Zeitvorgaben handelt.

5.1.3. FLASH

Ein Flash hat im Gegensatz zu den bisher beschriebenen DUTs eine umfangreichere Ansteuerung, die sowohl eine größere Anzahl komplexere Befehle umfasst. Weiterhin hängt die erreichbare Leistungsfähigkeit teilweise stark von den verwendeten Befehlen ab. So gibt es zum Beispiel unterschiedliche Möglichkeiten, den Speicher zu löschen oder zu beschreiben. Darauf wird in den Ergebnissen in Kapitel 5.5 näher eingegangen.

Zum Funktionsnachweis der automatischen Co-Prozessorgenerierung und als Grundlage für den Vergleich zu virtuellen Testinstrumenten (siehe Kapitel 5.5) werden folgende Befehle unterstützt: *Reset*, *Program*, *Read* und *Chip Erase*.

Befehle zum selektiven Löschen des Schaltkreises („*Sector erase*“) sowie zum schnelleren Programmieren („*Write to buffer*“ und „*Program buffer to flash*“) werden nicht umgesetzt. Die Auswirkungen einer schnelleren Programmierung werden jedoch als Teil des Kapitels 5.5 betrachtet.

Aufgrund der Ansteuerung des Flash durch Sequenzen von Bytes (siehe hierzu Abbildung 93 in Anhang R) ist für eine optimale Ansteuerung eine hierarchische Modellierung notwendig, wie diese in Kapitel 4.2.7 und Kapitel 4.4.7 beschrieben worden ist. Da dies vom Compiler aktuell noch nicht unterstützt wird, wird diese Funktionalität durch den Prozessor auf Ebene L2 implementiert. Auf die hierdurch zusätzlich benötigte Zeit für die Abarbeitung wird in Kapitel 5.5 eingegangen.

5.2. TESTSYSTEMGENERIERUNG

Im Rahmen dieser Arbeit ist ein Compiler realisiert worden, der als proof-of-concept die Umsetzung der beschriebenen Methode zur automatischen Generierung von Testinstrumenten demonstriert. Neben dem hier beschriebenen Funktionsumfang des Compilers für die Generierung von Co-Prozessoren für Testinstrumente, sind für eine praktische Umsetzung weitere Bestandteile für ein vollständiges Testsystem notwendig. Sie wurden im Rahmen des Projekts ROBSY bereitgestellt bzw. entwickelt, stellen aber keinen Bestandteil dieser Arbeit dar.

Diese Bestandteile sind:

- ein konfigurierbarer Testprozessor
- ein Compiler/Assembler, um die Softwarefunktionen auf dem angepassten Prozessor zu implementieren
- eine graphische Oberfläche für die einfache Handhabung der Testsystemgenerierung, welche die Schnittstelle zu allen benötigten Tools bildet

Für die praktische Umsetzung wurde neben dem beschriebenen Board von GÖPEL electronic das Altera „DE2-115 Development and Education Board“ von Terasic [114] gewählt.

Die Generierung des Testsystems wird für jedes zu testende DUT separat durchgeführt, da die zeitgleiche Implementierung von mehreren Co-Prozessoren bisher vom Softwarecompiler nicht unterstützt wird.

Bevor die Generierung des Co-Prozessors erfolgt, werden durch den Compiler des Prozessors folgende Voraussetzungen geschaffen:

- es wird ein Prozessor mit bestimmten Eigenschaften generiert und dessen Quellen sowie Konfigurationsparameter in vorher definierte Quellenverzeichnisse des Co-Prozessor Compilers abgelegt
- die Registerkonfigurationen für den Co-Prozessor werden ebenfalls in ein vorher definiertes Konfigurationsverzeichnis abgelegt
- die Definition der zu nutzenden Takt- und Resetsignale sind ebenfalls im selben Konfigurationsverzeichnis zu hinterlegen
- gleiches gilt für die zu verwendende Netzliste, die auch im selben Konfigurationsverzeichnis hinterlegt sein muss

Anschließend kann die Generierung des Testsystems erfolgen. Diese erfolgt durch ein in Python 3 geschriebenes Programm, das sich auf der DVD am Ende dieser Arbeit befindet. Die Verzeichnisstruktur dieser DVD ist in Anhang U dargestellt.

Die Dauer einer solchen Testsystemgenerierung hängt von vielen Faktoren ab. Diese sind unter anderem:

- Größe der Netzliste

- Anzahl der zu verwendenden Co-Prozessoren
- Anzahl der Funktionen in den Co-Prozessoren
- Leistungsfähigkeit des verwendeten PCs, auf dem der Compiler ausgeführt wird

Wird die Generierung direkt per Batch-Datei gestartet und nicht aus der Python-Entwicklungsumgebung heraus, so dauert diese in bisherigen Tests[52] bei Verwendung eines Co-Prozessors nicht länger als 1-2 Sekunde. Diese Zeit kann im Vergleich zur ebenfalls notwendigen FPGA Synthese[53] vernachlässigt werden, da diese selbst für kleine FPGAs auf schnellen PCs selten unter einer Minute erfolgt.

Der Designflow einer Testsystemgenerierung ist in Kapitel 4.4 ausführlich beschrieben worden. Auf die Einschränkungen, die der aktuelle Compiler aufweist, wird im folgenden Kapitel 5.2.1 eingegangen.

5.2.1. EINSCHRÄNKUNGEN BEI DER GENERIERUNG DES TESTSYSTEMS

Der aktuelle Stand des Compilers für die Generierung des Testsystems weist einige Einschränkungen in der praktischen Realisierung auf.

- Es können nur Pins als Taktquelle genutzt werden. Der innerhalb des FPGAs genutzte Takt muss somit direkt als externer Takt vorhanden sein. Es ist nicht möglich, den Takt innerhalb des FPGAs z.B. durch den Einsatz von DCMs (digital clock manager) zu verändern.
- Testprozessor und Co-Prozessoren müssen den gleichen Takt verwenden.
- Bisher werden in der praktischen Umsetzung nur ein Prozessor und ein Co-Prozessor unterstützt. Somit entfällt die Notwendigkeit einen oder mehrere Co-Prozessoren einem oder mehreren Prozessoren zuzuordnen. Aktuell werden vom Compiler alle Co-Prozessoren einem Wishbone-Bus und somit einem Prozessor zugeordnet.
- Aktuell wird das Lesen von DUTs nicht zu Zeitpunkten, sondern nur während Zuständen unterstützt. Deshalb muss das zu lesende Signal für einen ganzen Takt stabil gehalten werden und darf sich erst nach Verlassen des betreffenden Zustands ändern. Dies verlängert die Zeitdauer für das Lesen um jeweils einen Takt gegenüber einer optimierten Realisierung.

[52] Diese sind auf einem PC mit folgenden Parametern erfolgt: Intel Core i7-4790 mit 3,6 GHz mit 16 GB RAM und SSD 850 Evo, Betriebssystem Windows 7, 64-Bit, Python 3.4 und yeanpypa (Modifizierter Stand vom 05.05.2010).

[53] Verwendete Software Altera Quartus 14.1 sowie Xilinx ISE 14.7.

Die beschriebenen Einschränkungen stellen keinen prinzipiellen Mangel des Konzepts oder der möglichen Modellierung dar, es fehlt lediglich an der Umsetzung innerhalb des Compilers. Auf die Umsetzung dieser Funktionen wurde verzichtet, weil sie für den Funktionsnachweis und die Darstellung der prinzipiellen Leistungsfähigkeit des Konzepts nicht nötig sind.

5.3. SIMULATIONSUNTERSUCHUNGEN FÜR L1

Bei den durchgeführten Simulationen werden die generierten Befehle mit einer manuell optimierten Ansteuerung verglichen. Ziel ist die zeitlich korrekte Ansteuerung der DUTs. Deshalb wird bei der Umsetzung nur die Realisierung der Ebene L1 betrachtet. Die Generierung wird für unterschiedliche Ansteuerungen verschiedener DUTs durchgeführt. Diese sind in Tabelle 10 dargestellt.

Tabelle 10: Generierte DUTs und Befehle für die Simulationsuntersuchung

Parameter Simulations-nummer	DUT Typ	DUT Bezeichnung	Befehle
S_1	SRAM	IS61LV25616	write CE[54]
S_2	SRAM	IS61LV25616	write LB/UB back-to-back[55]
S_3	SRAM	IS61L25616	read OE[56]
S_4	Display	LC-Display auf VarioTAP Board von GÖPEL electronic	write
S_5	Flash	S29AL064	reset
S_6	Flash	S29AL064	read
S_7	Flash	S29AL064	program
S_8	Flash	S29AL064	chip erase

[54] Schreibzugriff durch das CE (Chip Enable) Signal gesteuert. Entspricht Write Cycle No. 2 im Datenblatt [99].

[55] Zwei direkt aufeinander folgende Schreibzugriffe, die durch die Byte-Select Signale LB und UB gesteuert werden. Entspricht Write Cycle No. 4 im Datenblatt [99].

[56] Lesezugriff mit genutztem OE (Output Enable) Steuersignale. Entspricht Read Cycle No. 2 im Datenblatt [99].

Bei der Generierung wurde die Ebene L1 ausschließlich auf Basis der entsprechenden DUT-Ms erzeugt. Die gemessenen Zeiten beziehen sich auf die Zeit, in der sich die FSM nicht im idle Zustand befindet. Zusätzliche Zeiten, die für die Datenübertragung zum Co-Prozessor über Wishbone sowie für das Starten des Co-Prozessors nötig sind, werden an dieser Stelle nicht berücksichtigt. Näheres hierzu ist in den experimentellen Untersuchungen in Kapitel 5.4 zu finden.

In Tabelle 11 ist eine Übersicht der benötigten Ressourcen für drei verschiedene Testinstrument dargestellt. Hierbei beinhaltet jedes Testinstrument einen Testprozessor und alle in Tabelle 10 dargestellten Befehle für das entsprechende DUT.

Die Angaben zu den benötigten Ressourcen zeigen, dass selbst auf einem älteren FPGA des Typs Spartan3 XC3S1000 nur Look-Up Table (LUT) und Register (FF) im mittleren einstelligen Prozentbereich benötigt werden. Der Ressourcenverbrauch an eingebetteten Speicherressourcen (BRAM) hängt nur vom Prozessor ab und zeigt in der aktuellen Ausbaustufe des Prozessors[57] noch genug freie Ressourcen für die Implementierung weiterer Prozessoren.

Tabelle 11: Übersicht des Ressourcenverbrauchs der Simulationen S_1 bis S_8

Parameter Simulations-nummer	LUT [absolut] [in %]	FF [absolut] [in %]	BRAM [absolut] [in %]
S_1 - S_3	1.279 6%	685 4%	6 25%
S_4	912 5%	469 3%	6 25%
S_5 - S_8	1.339 8%	681 4%	6 25%

Vergleichszahlen zu BScan:

Die Ergebnisse für einen vergleichbaren Test über Boundary-Scan in den folgenden Tabellen basieren für alle durchgeführten Untersuchungen auf einem FPGA mit 384 aktiven Boundary-Scan-Zellen und 10 MHz Testtakt. Für die ermittelten Werte werden nur die nötigen Takte für das Schieben der Daten berücksichtigt. Die Takte für die Synchronisation innerhalb des TAP Controllers sind im Vergleich hierzu zu

[57] Der Prozessor entspricht dem für die experimentellen Untersuchungen verwendeten Prozessor in Kapitel 5.4.

vernachlässigen und verändern das Ergebnis nur geringfügig[58]. Für das logische Verhalten wird ein optimiertes Testinstrument als Referenz für den Boundary-Scan-basierten Test verwendet.

Die Ergebnisse in diesem Kapitel belegen einerseits die erreichbare Leistungsfähigkeit der generierten Co-Prozessoren im Vergleich zu einer manuell optimierten Ansteuerung, dienen andererseits zugleich aber auch als Funktionsnachweis für den Compiler und zeigen dessen generelle Funktionsfähigkeit.

5.3.1. SRAM ERGEBNISSE

Die dargestellten Ergebnisse basieren auf folgenden Quelldaten, die auf dem Datenträger in Anhang U enthalten sind.

- Netzliste: *Xilinx_all_simulation_SRAM.dif*
- Prozessorkonfiguration: *Xilinx_all_simulation_SRAM_proc_config_values.txt*
- Registerkonfiguration: *Xilinx_all_simulation_SRAM_wbReg_configFile.cfg*

Das DUT-M sowie die zur Generierung relevanten Informationen der 3 Zugriffsroutinen, die für die Simulationen S_1 bis S_3 benötigt werden, sind in Anhang L dargestellt. Die Ergebnisse der Simulation sind in Tabelle 12 ersichtlich und wurden bei einer Taktfrequenz von 100 MHz ermittelt. Hierbei wird die Zeit der Prozessausführung im automatisch generierten Co-Prozessor mit der eines optimierten Co-Prozessors verglichen. Bei allen Zeiten wird die Konfiguration des Co-Prozessors durch den Prozessor nicht berücksichtigt.

[58] Für den im Rahmen dieser Arbeit betrachteten FPGA werden 21 zusätzliche Takte für das abwechselnde Senden einer Instruktion und eines Datenwortes benötigt. Dies entspricht ca. 5% der gesamten Taktanzahl. Details zu der Berechnung sind in Anhang T zu finden.

Tabelle 12: Übersicht der Ergebnisse der Simulationen S_1 bis S_3

Parameter / Simulations-nummer	DUT Typ	DUT Bezeichnung	Funktion	Zeitdauer f. Co-Prozessor	Optimierte Ausführungs-zeit	BScan
S_1	SRAM	IS61LV25616	write CE	30 ns	10 ns	76.800 ns
S_2	SRAM	IS61LV25616	write LB/UB back-to-back	60 ns	30 ns	153.600 ns
S_3	SRAM	IS61LV25616	read OE	40 ns	10 ns	76.800 ns

Die einzelnen Ergebnisse der 3 Simulationen inklusive der Zeiten für den Datentransfer über Wishbone sind in der Abbildung 56 bis Abbildung 58 als Waveform dargestellt. Während des Datentransfers über Wishbone befindet sich die Zustandsmaschine des Co-Prozessors (letzte Zeile in den Abbildungen) immer im Zustand *idle*.

Bewertung der Ergebnisse

Bei der Simulation S_1 zeigt sich eine um zwei Takte längere Abarbeitung gegenüber der optimierten Ausführung. Dies hat zwei Ursachen:

- Zum einen werden nach dem Anlegen sämtlicher Adress-, Daten- und Steuerzustände im Zustand q_0 diese im Zustand q_1 explizit wieder zurückgesetzt, bevor die Kontrolle durch die FSM wieder an den Prozessor zurück übergeben wird. Dies verursacht einen zusätzlichen Takt.
- Zum anderen benötigt das Signal Dout eine gewisse Zeit, bis dort gültige Daten verfügbar sein können. Daher muss diese Zeit gewartet werden, bis wieder jede beliebige Art weiterer Befehle zulässig sind.

Die Umsetzung der Funktion *write CE* wäre nach folgenden Kriterien optimierbar:

- Muss am Abschluss einer Aktion weniger als 1 Takt gewartet werden, ohne dass irgendwelche Aktionen auszuführen sind, kann diese Wartezeit unter Umständen auch während der Synchronisation der Ebene L1 mit der nächsthöheren Ebene oder dem Prozessor erfolgen. In diesem Fall muss jedoch sichergestellt werden, dass tatsächlich eine ausreichend lange Wartezeit bei dieser Synchronisation entsteht, da diese nicht mehr in den DBMs enthalten ist.
- Bezieht sich eine Zeitvorgabe auf die Reaktion des DUT und setzt keine Aktion des Co-Prozessors voraus, kann diese ab Beginn des vorgehenden Zustandes gezählt werden, statt, wie sonst üblich, nach dessen Abschluss. Dies gilt sowohl bei der Ansteuerung durch einen Prozessor als auch für Co-Prozessoren.

Bei der Simulation S_2 sind zwei Takte der insgesamt um drei Takte längeren Abarbeitungszeit auf die gleichen Ursachen wie auch bei S_1 zurückzuführen und auf die gleiche Weise optimierbar. Durch die manuell eingefügte Definition, dass DIN_0 nicht vor $DOUT_0$ erfolgen darf, wird ein zusätzlicher dritter Takt Wartezeit generiert, bis gültige Daten an den Datenbus angelegt werden.

Bei der Simulation S_3 ist die Abarbeitungszeit ebenfalls drei Takte länger als dies bei einer optimierten Variante nötig wäre. Während die Zustände q_2 und q_3 wieder auf die gleichen Gründe wie bei den Simulationen S_1 und S_2 zurückzuführen sind, ist der Zustand q_1 der Tatsache geschuldet, dass die Daten während der gesamten Zeitdauer eines Zustands eingelesen werden und nicht nur zum Zeitpunkt seines Beginns. Dies lässt sich aufgrund von internen Signallaufzeiten innerhalb des FPGAs nicht einfach durch die Verschiebung des Übernahmezeitpunkts um einen Zustand nach vorne beheben. Da diese Signallaufzeiten sowohl vom verwendeten FPGA als auch der implementierten Logik zwischen Pin und Register abhängen, also von der Anzahl an Co-Prozessoren und unterstützen Befehlen, muss dies im Einzelfall geprüft werden.

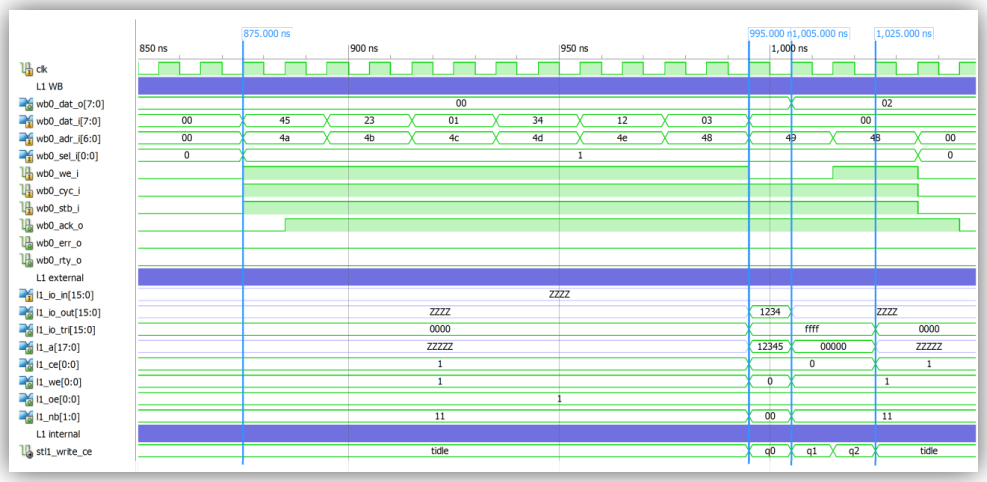


Abbildung 56: Waveform der Simulation S_1

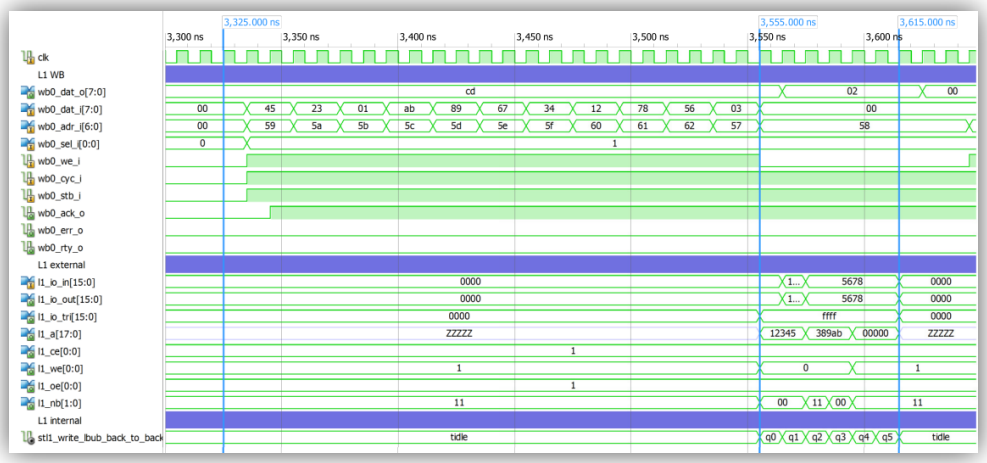


Abbildung 57: Waveform der Simulation S_2

Abbildung 58: Waveform der Simulation S_3

5.3.2. DISPLAY ERGEBNISSE

Die dargestellten Ergebnisse basieren auf folgenden Quelldaten, die auf dem Datenträger im Anhang U enthalten sind.

- Netzliste: *Xilinx_all_simulation_Display.dif*
- Prozessorkonfiguration: *Xilinx_all_simulation_Display_proc_config_values.txt*
- Registerkonfiguration: *Xilinx_all_simulation_Display_wbReg_configFile.cfg*

Das DUT-M sowie die zur Generierung relevanten Informationen der Zugriffsroutinen, die für die Simulationen S_4 benötigt werden, sind in Anhang Q dargestellt. Die Ergebnisse der Simulation sind in Tabelle 13 dargestellt und wurden bei einer Taktfrequenz von 100 MHz ermittelt. Hierbei wird die Zeit der Prozessausführung im automatisch generierten Co-Prozessor mit der einer optimierten Ausführung verglichen. Bei allen Zeiten wird die Konfiguration des Co-Prozessors durch den Prozessor nicht berücksichtigt.

Im Gegensatz zu den bisher gezeigten Simulationen S_1 bis S_3 erfordert die Ansteuerung des Displays im Vergleich zum verwendeten Takt relativ lange Verweilzeiten innerhalb einzelner Zustände. Dies wird durch den Einsatz eines zusätzlichen Zählers erreicht, der die Taktübergänge steuert. Dieser wird mit dem Namenszusatz *_cnt* passend zur Zustandsmaschine generiert und ist in Abbildung 59 in der letzten Zeile dargestellt.

Neben den Zuständen, die der physikalischen Ansteuerung des Displays dienen und in Abbildung 59 dargestellt sind, wird in Abbildung 60 der gesamte Ablauf des Co-Prozessors, inklusive der definierten Wartezeit vor einem nächsten Zugriff gezeigt. Weiterhin wird in dieser Abbildung ein zweiter Schreibzugriff auf das Display dargestellt, der direkt auf den ersten Zugriff folgt.

Tabelle 13: Übersicht der Ergebnisse der Simulation S_4

Parameter / Simulations-nummer	DUT Typ	DUT Bezeichnung	Funktion	Zeitdauer für Co-Prozessor	Optimierte Ausführungs-zeit	BScan
S_4	Display	LC-Display auf VarioTAP Board von GÖPEL electronic	write	1.700 ns	1.690 ns	153.600 ns

Bewertung der Ergebnisse

Bei der automatischen Generierung des Co-Prozessors wird fast die Leistungsfähigkeit der optimierten Realisierung erreicht. Dies liegt daran, dass lediglich ein zusätzlicher Takt für die Beendigung der Zustandsmaschine generiert wird, da diese nach der definierten Wartezeit in q_2 nach q_3 wechselt. In diesem Zustand werden ggf. die letzten Aktionen ausgeführt, bevor die Zustandsmaschine durch den Wechsel nach idle beendet wird. Erfolgen jedoch im letzten Zustand keine Zuweisungen, kann dieser Zustand eingespart werden. Dies entspricht dem Verhalten der Simulationen S_1 bis S_3.

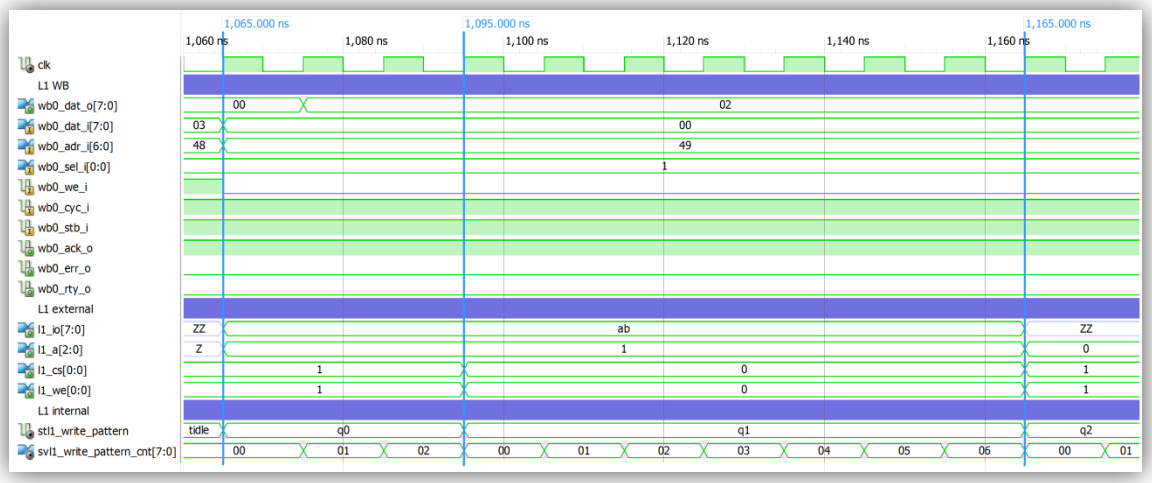


Abbildung 59: Waveform der Simulation S_4 (q_0 bis q_2)

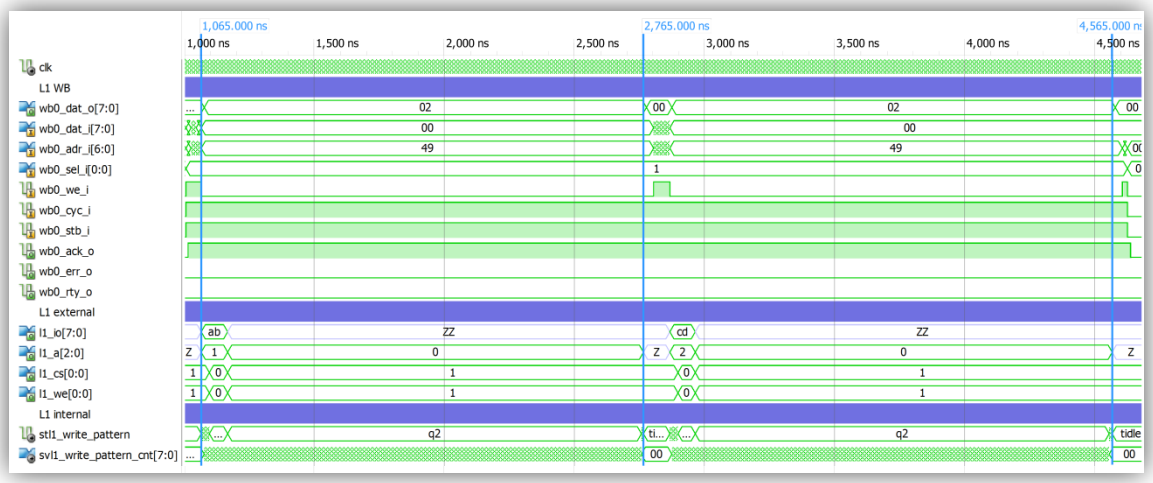


Abbildung 60: Waveform der Simulation S_4 (2 aufeinander folgende Zugriffe)

5.3.3. FLASH ERGEBNISSE

Die dargestellten Ergebnisse basieren auf folgenden Quelldaten, die auf dem Datenträger im Anhang U enthalten sind.

- Netzliste: *Xilinx_all_simulation_Flash.dif*
- Prozessorkonfiguration: *Xilinx_all_simulation_Flash_proc_config_values.txt*
- Registerkonfiguration: *Xilinx_all_simulation_Flash_wbReg_configFile.cfg*

Das DUT-M sowie die zur Generierung relevanten Informationen der 4 Zugriffsroutinen, die für die Simulationen S_5 bis S_8 benötigt werden, sind in Anhang R dargestellt. Die Ergebnisse der Simulation zeigt Tabelle 14, sie wurden bei einer Taktfrequenz von 100 MHz ermittelt. Hierbei wird die Zeit der Prozessausführung im automatisch generierten Co-Prozessor mit der einer optimierten Ausführung verglichen. Bei allen Zeiten wird die Konfiguration des Co-Prozessors durch den Prozessor zu Beginn eines jeden Befehls nicht berücksichtigt.

Tabelle 14: Übersicht der Ergebnisse der Simulation S_5 bis S_8

Parameter / Simulations-nummer	DUT Typ	DUT Bezeichnung	Funktion	Zeitdauer für Co-Prozessor	Optimierte Ausführungs-zeit	BScan
S_5	Flash	S29AL064	reset	560 ns	550 ns	76.800 ns
S_6	Flash	S29AL064	read	150 ns	110 ns	76.800 ns
S_7	Flash	S29AL064	program	760 ns[59]	440 ns	307.200 ns
S_8	Flash	S29AL064	chip erase	-[60]	-[60]	-[60]

Bewertung der Ergebnisse

Bei der Realisierung der *Reset*-Funktion (S_5) werden vom automatisch generierten CoProzessor 10ns mehr Zeit benötigt, als bei einer optimierten Ausführung. Dies hat folgende Gründe:

- 10ns (q_2), für das explizite Freigeben des Datenbusses nach der letzten Wartezeit.

[59] Für den Vergleich des *Program*-Befehls wurde die Wartezeit des Speichers nicht berücksichtigt. Diese ist für beide Implementierungen identisch und im Vergleich zu den aktiven Teilen des Program-Befehls mit typischerweise 60µs sehr lang. Daher werden nur die aktiven Teile verglichen.

[60] Der Vergleich für die Funktion *chip-erase* ist aufgrund der relativ langen fest definierten Wartezeit von 500ms nicht sinnvoll, da diese bei allen Implementierungen erfolgt und im Vergleich zu der restlichen Ausführungszeit von ca. 1µs bei den Co-Prozessoren bzw. 100µs bei Boundary-Scan die gesamte Abarbeitung dominiert.

Bei der Realisierung der *Read*-Funktion (S_6) benötigt der automatisch generierte Co-Prozessor 50ns zusätzliche Zeit im Vergleich zu einer optimierten Ausführung. Dies hat folgende Gründe:

- 10ns (q_1), da Daten während eines Zustands und nicht während eines Ereignisses gelesen werden können
- 20ns (q_2), da der Datenbus auch nach dem Deaktivieren des Output-enable Signals noch eine gewisse Zeit die alten Daten bereithält. In dieser Zeit dürfen keine anderen Daten gelesen werden. Diese Zeit wird im Fall des optimierten Zugriffs während der eigentlichen 110ns Wartezeit des Folgezugriffs durchgeführt
- 10ns (q_3), für das explizite Freigeben des Datenbusses

Bei der *Program*-Funktion (S_7) benötigt der automatisch generierte Co-Prozessor gegenüber einer optimierten Variante 320ns mehr für die Ausführung. Dies hat folgende Gründe:

- für die Programmierung eines einzelnen Datenbytes ist im Vorfeld die Übertragung von drei weiteren Bytes notwendig. Diese sind aktuell als einzelne Funktionen im Co-Prozessor realisiert, werden nacheinander vom Prozessor aufgerufen und hängen nicht direkt voneinander ab. Zwischen den ersten drei Bytes ergibt sich somit zwei Mal eine zusätzliche Zeit von je 70ns, also insgesamt 140ns für die Datenübertragung vom Prozessor.
- zwischen dem dritten und vierten Byte, den eigentlichen Datenbytes, beträgt die Wartezeit aufgrund der vom Prozessor zu programmierenden Register 150ns.
- für das Programmieren jedes der drei ersten Bytes sind jeweils 10ns zusätzlich für die explizite Freigabe des Datenbusses notwendig. Dies sind zusammen 30ns.

Die Waveforms der Ergebnisse sind in Abbildung 113 bis Abbildung 116 im Anhang S zu finden.

5.4. EXPERIMENTELLE UNTERSUCHUNGEN

Bei den durchgeführten praktischen Untersuchungen sind die Ansteuerungen unterschiedlicher DUTs zusammen mit einem Prozessor auf realer Hardware durchgeführt worden. Dabei wurden die Funktionen der Ebene L1 in einem Co-Prozessor realisiert und die restlichen Ebenen in Embedded Software auf einem soft-core Testprozessor innerhalb des FPGAs.

Die Ergebnisse, die in diesem Kapitel vorgestellt werden, hängen nicht nur vom generierten Co-Prozessor und dessen Funktionen ab, sondern auch vom verwendeten Testprozessor. Die Ergebnisse sollen einen Vergleich der prinzipiellen Leistungsfähigkeit eines gesamten Testsystems im Vergleich zu Boundary-Scan erbringen und als prinzipieller Funktionsnachweis des Konzepts der automatisch generierten Testinstrumente dienen. Es ist nicht das Ziel, unterschiedliche Realisierungen des Co-

Prozessors oder des Testprozessors miteinander zu vergleichen. Für solche Untersuchungen sei auf [33] verwiesen. Auf die Komplexität einer entsprechenden Optimierung wurde in Kapitel 4.3 eingegangen.

Bei den praktischen Untersuchungen steht nicht die erreichte Leistungsfähigkeit der Co-Prozessoren im Vordergrund, da diese mit den Ergebnissen der Simulation in Kapitel 5.3 belegt sind und auch bei einer Implementierung in einem FPGA nicht anders ausfallen. Vielmehr geht es um die Leistungsfähigkeit des Gesamtsystems und das Zusammenspiel von Embedded Software auf dem Testprozessor und den in Hardware realisierten Algorithmen im Co-Prozessor. Für diese Zwecke werden für zwei DUTs je eine Testroutine vollständig in RTDL beschrieben und anschließend als vollständiges[61] eingebettetes Testinstrument auf dem FPGA realisiert.

Erfahrungen haben gezeigt, dass der Zugriff auf Boundary-Scan über JTAG oft durch die verwendeten Treiber bzw. die Details der Ansteuerung der Schnittstelle entscheidend beeinflusst werden. Somit ist ein Test nur unter ganz bestimmten Voraussetzungen mit anderen vergleichbar und selbst dann sind die Ergebnisse nur für diesen einen betrachteten Fall korrekt. Eine andere Ansteuerung kann den Vorteil bestimmter Verfahren relativieren bzw. sogar umkehren.

Da die Durchlaufzeiten besonders bei In-line Tests in der Produktion relevant sind, ist davon auszugehen, dass hier optimierte Ansteuerungen über JTAG zum Einsatz kommen, die bessere und auch reproduzierbarere Testzeiten als Standardsysteme liefern. Auf diese Systeme bestand im Rahmen dieser Arbeit jedoch kein Zugriff. Ein Vergleich mit nicht optimierten Standardsystemen würde aber die Ergebnisse der generierten Testinstrumente gegenüber Boundary-Scan ungerechtfertigt verbessern. Daher wird von einem solchen Test abgesehen und eine idealisierte Ansteuerung über JTAG zum Vergleich verwendet. Diese nimmt an, dass keine Verzögerung durch einen PC entsteht, sie betrachtet nur die nötigen Takte der Datenübertragungen. Darüber hinaus wurde eine typische Ansteuerung mit einer festen Verzögerung von 1 ms zwischen jeder JTAG Übertragung betrachtet. Diese Ergebnisse entsprechen einer Abschätzung der Testausführung auf einem nicht optimierten realen System.

Weiterhin ist anzumerken, dass die notwendige Zeit für das Programmieren des FPGAs, in den im Rahmen dieser Arbeit durchgeführten Untersuchungen, nicht berücksichtigt worden ist. Laut [33] hängt diese Zeit stark von der Art der Programmierung und dem verwendeten FPGA ab. Somit lassen sich insbesondere bei den hier betrachteten kurzen Testsequenzen sehr einfach Testfälle erzeugen, bei denen der Geschwindigkeitsvorteil eines eingebetteten Testinstruments im Vergleich zu Boundary-Scan vollständig aufgehoben wird. In der Praxis ist es somit notwendig den

[61] ‚Vollständig' bezieht sich auf die Testalgorithmen. Die Koordinierung des Testablaufs (Ebene L5) verbleibt auf dem Test-PC.

Einsatz der jeweiligen Technologie abzuwägen. Spätestens jedoch wenn at-speed Testen gefordert wird, ist der Einsatz der hier vorgestellten Testinstrumente erforderlich.

Tabelle 15: Ergebnisse der praktischen Untersuchungen

Parameter Praxistest-nummer	DUT Typ	DUT Bezeichnung	Test	Beschreibung
P_1	SRAM	IS61LV102416	walking-1	Der Adress- und Datenbus des SRAM werden durch das Schreiben und anschließende Lesen sowie Vergleichen der Daten getestet.
P_2	Display	LC-Display auf VarioTAP Board von GÖPEL electronic	sequentiell angesteuerte Spalten	Das Display wird durch wiederholtes Ansteuern aller 128 Spalten getestet. Die Prüfung erfolgt visuell.

In beiden Untersuchungen wurde derselbe Prozessortyp verwendet. Dieser hat dabei folgende Parameter:

- kein Pipelining (multicycle Prozessor)
- 8 Bit Registerbreite
- 16x10 Bit Stackspeicher
- 64x8 Bit Datenspeicher
- 1024x19 Bit Programmspeicher
- Aktive Befehle: JMP, JC, JZ, JNZ, CALL, CALL_C, RET, RET_C, STOP, NOP, LOAD, STORE, PUSH; POP, ADDI, SUB, SUBI, AND, ANDI, ORI, SHL, SHLC[62]
- Debug Interface mit 19 Bit Daten- und Adressbusbreite und den unterstützten Befehlen: HALT, CONTINUE, CPU_RESET, DATA_MEM_READ, DATA_MEN_WRITE, CPU_STATE_READ, SFR_READ, SFR_WRITE, ID_READ[62]

Der Test wird vom Test-PC über JTAG gesteuert. Hierfür wurde ein USB-Byteblaster[63] mit einem Testtakt (TCK) von 6 MHz verwendet.

[62] Für Erläuterungen, die genaue Funktionsweise und Eigenschaften der Befehle sei auf [33] verwiesen.

[63] Ein in diesem Fall fest auf dem DE2-115 Developmentboard verbauter Programmieradapter.

Innerhalb des FPGAs existieren zur genauen Bestimmung der Testdauer mehrere Zähler, die vom Prozessor gesteuert und ausgewertet werden. Für die folgenden Untersuchungen ist der *proc_cnt* Zähler relevant. Dieser gibt an, wie lange der Testprozessor und der Co-Prozessor mit der Bearbeitung der eigentlichen Testalgorithmen beschäftigt waren. Dies entspricht somit der Zeit für die Ausführung der Ebenen L1 bis L4. Dies ist der Vergleichswert für die Boundary-Scan-basierten Tests.

Das für die Untersuchungen des SRAMs verwendete RTDL Modell ist in Anhang L und Anhang M dargestellt. Das für die Untersuchungen des Displays verwendete RTDL Modell ist in Anhang U zu finden. Die Ergebnisse beider Tests sind in Tabelle 16 dargestellt. Die genaue Berechnung der benötigten Takte der Boundary-Scan-basierten Tests werden in Anhang T aufgezeigt.

Tabelle 16: Übersicht der Ergebnisse der Untersuchung P_1 und P_2

Parameter Simulations-nummer	proc_cnt [CPU Takte[64] / ms]	BScan (ideal) [TCK Takte[65] / ms]	BScan (typ) [ms]	Beschleunigungsfaktor im Vergleich zu BSCAN (ideal / typ)
P_1	132.130 / 2,64 ms	59.662 / 5,97 ms	78,97 ms	2,26 / 29,91
P_2[66]	23.595.898 / 471,92 ms	53.090.640 / 5.309,1 ms	70.853,1 ms	11,25 / 150,14

Selbst im Vergleich zum theoretischen Bestwert der beschriebenen Tests über Boundary-Scan, ist das Embedded Test Instrument in der aktuellen Realisierung um den Faktor 2,26 für P_1 und 11,25 für P_2 dem Boundary-Scan Test gegenüber überlegen.

In realen Systemen ist jedoch eher mit einer Verzögerung zwischen jedem Boundary-Scan Transfer zu rechnen. Dies liegt an der Ansteuerung der verwendeten Interface-Hardware durch den Test-PC und variiert je nach verwendeter Hardware, Betriebssystem, etc. Für die weiteren Betrachtungen ist von einer einheitlichen Verzögerung von

[64] Die CPU verwendet einen internen Takt von 50 MHz.

[65] JTAG verwendet einen Takt von 10 MHz.

[66] Für das menschliche Auge ist das visuelle Prüfen des Displays unterhalb einer bestimmten Ausführungszeit nicht mehr möglich, da die Ansteuerung zu schnell erfolgt. Mit einer automatischen optischen Inspektion wäre es jedoch möglich, die volle Geschwindigkeit des über ROBSY ausgeführten Tests zu nutzen.

1 ms[67] ausgegangen worden. In diesem Fall erhöhen sich die Testzeiten für P_1 auf 78,97 ms und für P_2 auf 70.853,1 ms. Entsprechend ergibt sich ein größerer Beschleunigungsfaktor beim Einsatz der eingebetteten Testinstrumente von ca. 30 für P_1 bzw. 150 für P_2, da diese vollständig im FPGA realisiert sind und somit weitgehend unabhängig von der verwendeten Hardware zur Ansteuerung der JTAG Schnittstelle und des Test-PCs arbeiten.

Weiterhin soll an dieser Stelle zur Beurteilung der Leistungsfähigkeit des Konzepts der automatisch generierten Testinstrumente angemerkt werden, dass das verwendete Testinstrument aktuell nur die Ebene L1 in Hardware realisiert. Bei einer Realisierung der höheren Schichten ist von einer weiteren Beschleunigung der Testausführung auszugehen. Diese sich potentiell ergebenden Verbesserungen sind in Kapitel 5.7 an einem Beispiel dokumentiert.

In Tabelle 17 ist der Ressourcenverbrauch der beiden Testsysteme dargestellt. Diese beinhalten, wie schon in Kapitel 5.3, sowohl den Testprozessor als auch den vollständigen Co-Prozessor. Das Testsystem wurde auf einem modernen, mittelgroßen FPGA vom Typ EP4CE115F29C7 von Altera realisiert. Der Ressourcenverbrauch sowohl für LUT, FF und internen Speicher beträgt jeweils 1% oder weniger. Es ist zu erwarten, dass bei FPGAs in dieser Leistungsklasse sehr viel Testfunktionalität zeitgleich in einen FPGA integriert werden kann. Insbesondere bei paralleler Testausführung führt dies zu einer weiteren Beschleunigung des Testens.

Tabelle 17: Übersicht des Ressourcenverbrauchs der Untersuchung P_1 und P_2

Parameter Simulations-nummer	LUT [absolut] [in %]	FF [absolut] [in %]	Memory bits [absolut] [in %]
P_1	1.215 1%	705 <1%	20.256 <1%
P_2	931 <1%	548 <1%	20.256 <1%

5.5. SIMULATIONSUNTERSUCHUNGEN FÜR L2 BIS L5

Neben den bisher vorgestellten Ergebnissen zur Realisierung eines Testinstruments mit der Ebene L1 in Hardware und L2 bis L5 in Embedded Software, sollen im aktuellen Kapitel die Ergebnisse einer vollständigen Realisierung in Hardware gezeigt

[67] Der Wert von 1 ms beruht auf eigenen durchgeführten Tests und Aussagen von Testingenieuren und schwankt von Testsystem zu Testsystem leicht. Der Wert ist eine qualifizierte Annahme, der den Einfluss bereits kleiner Verzögerungen auf den Performancevergleich zeigen soll.

werden. Die Generierung des Co-Prozessors erfolgt vollständig automatisch und basiert auf dem gleichen DUT Modell, wie es auch für die bisherigen Untersuchungen verwendet worden ist.

Die Ergebnisse in Tabelle 18 zeigen die Testzeiten für ein vollständig automatisch generiertes Testinstrument, basierend auf dem aktuellen Stand des Compilers (Versuch S_9) und eine Variante, bei der die Ebene L1 gemäß der Möglichkeiten (siehe Kapitel 5.3.1) manuell optimiert worden ist (S_9 optimiert). Die erreichten Testzeiten der vollständigen Hardwareimplementierung liegen hierbei deutlich unterhalb derer einer Embedded Software Lösung. Die Beschleunigungsfaktoren des ROBSY Ansatzes zwischen P_1 und S_9 liegen bei ca. 25, bzw. bei ca. 29 für S_9 optimiert. Im Vergleich zu einer typischen Boundary-Scan-Ansteuerung ergibt sich für S_9 optimiert ein Beschleunigungsfaktor von über 870.

Tabelle 18: Übersicht der Ergebnisse der Untersuchung P_1 und S_9

Parameter / Versuchs-nummer	ROBSY Ansatz [ms]	BScan (ideal) [ms]	BScan (typ) [ms]	Beschleunigungsfaktor im Vergleich zu BSCAN (ideal / typ)	Beschleunigungsfaktor im Vergleich zu P_1
P_1	2,64 ms	5,97 ms	78,97 ms	2,26 / 29,91	1
S_9	105,26 µs	5,97 ms	78,97 ms	56,72 / 750,24	25,08
S_9 optimiert	90,64 µs	5,97 ms	78,97 ms	65,86 / 871,25	29,13

Die Ausführung des Co-Prozessors, beginnend bei L5 bis zu den ersten beiden Aufrufen von L1, ist anhand der Zustandsmaschinen in der oberen Hälfte von Abbildung 61 dargestellt. Weiterhin werden die physikalischen Zugriffe auf das DUT in der unteren Hälfte gezeigt. Die vollständige Simulation ist dem Anhang U zu entnehmen.

Tabelle 19 zeigt den Ressourcenverbrauch des automatisch generierten Co-Prozessors in Versuch S_9 im Vergleich zur Realisierung der gleichen Funktionalität im Testinstrument bei Verwendung eines Testprozessors in P_1 für einen FPGA vom Typ EP4CE115F29C7 von Altera. Im Falle von P_1 wird L5 bis L2 in Embedded Software ausgeführt, während nur L1 in Hardware realisiert wird. Bei S_9 sind alle Ebenen L5 bis L2 in Hardware realisiert.

Die Ergebnisse zeigen deutlich, dass bei modernen FPGAs die Implementierung von Testalgorithmen sowohl Embedded Software wie auch in Hardware nur wenig der vorhandenen Ressourcen belegt.

Tabelle 19: Übersicht des Ressourcenverbrauchs für S_9

Parameter Simulations-nummer	LUT [absolut] [in %]	FF [absolut] [in %]	Memory bits [absolut] [in %]
P_1	1.215 1%	705 <1%	20.256 <1%
S_9	1.305 1%	924 1%	0 0%

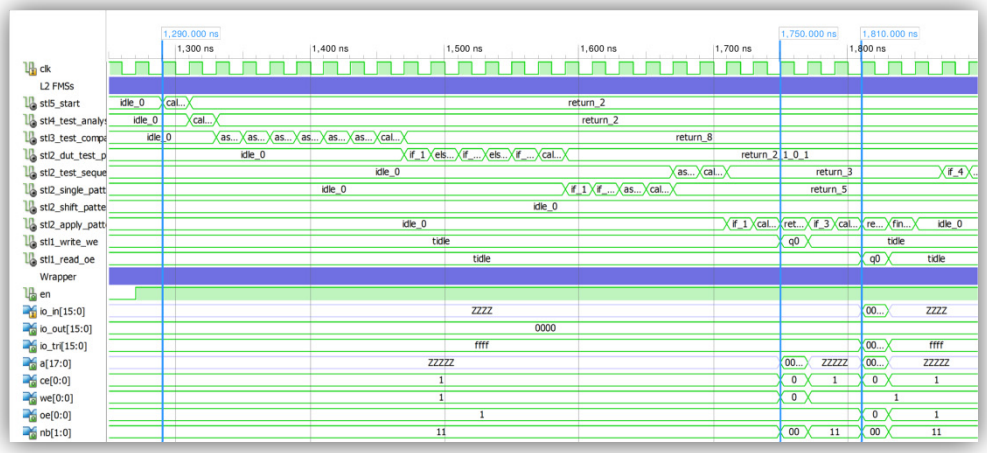


Abbildung 61: Waveform der Simulation S_9 (vollständiger Co-Prozessor)

5.6. VERGLEICH ZU VIRTUELLEN TESTINSTRUMENTEN

Für den Flashbaustein S29AL016 gibt es in [15] Messergebnisse, die die erreichte Beschleunigung beim Programmieren und anschließendem Verifizieren einer 1Mbyte großen Datenmenge bei der Verwendung eines Virtuellen Testinstruments im Vergleich zu Boundary-Scan angibt. Das verwendete Testinstrument nutzt zur Beschleunigung ein *Variable Length Shift Register* (VLSR). Ein entsprechender Auszug der Ergebnisse aus [15] ist in Tabelle 19 dargestellt.

Tabelle 20: Ergebnisse einer Flashprogrammierung mit VLSR Instrumenten

Ergebnisse / Funktionen	Boundary-Scan[68]	VLSR Instrumente	Beschleunigungs-faktor
Program+Verify	827 s	76,3 s	10,8

Der in [15] verwendete Flash ist sehr ähnlich zu dem bereits in Kapitel 5.3 verwendeten Baustein und unterscheidet sich nur in der Größe, nicht aber in der Ansteuerung. Somit lassen sich die erreichten Messergebnisse aufeinander übertragen.

Bei einer Realisierung mit einem L1 Co-Prozessor, welcher über einen 8 Bit Wishbone-Bus an den Prozessor angebunden ist, ergibt sich folgende Ausführungszeit für das Programmieren und Verifizieren. Die Details dieser Berechnung sind im Folgenden genauer erläutert.

Tabelle 21: Ergebnisse einer Flashprogrammierung mit Co-Prozessoren

Ergebnisse / Funktionen	Boundary-Scan	VLSR Instrumente	Co-Prozessor L1	Beschleunigungsfaktor (zu BScan / zu VLSR)	
Program+Verify (ohne Puffer)[69]	827 s	76,3 s	64,17 s	12,89	1,19
Program+Verify (mit Puffer)	827 s	k.A.	8,33 s	99,28	k.A.

Details der Ergebnisse

Aufgrund noch nicht unterstützter hierarchischer Modellierung sind die einzelnen Teile eines Befehls zur Flashprogrammierung separat als Sequenzen der Ebene L1 beschrieben worden und werden vom Prozessor nacheinander aufgerufen. Dadurch verzögert sich die Abarbeitung. Auf die verbesserte Ausführung wird am Ende des Kapitels noch einmal eingegangen und es werden die erreichbaren Programmierzeiten geschätzt.

Das Schreiben eines Steuerbytes zum Flash dauert unabhängig vom Inhalt 120 ns. Hinzu kommen für die Kommunikation zwischen Testprozessor und Co-Prozessor

[68] FPGA: Xilinx Spartan3 xc3s1000 mit 1187 BScan Zellen, JTAG Takt 10 MHz

[69] Es ist nicht genau bekannt, welcher Programmieralgorithmus in [15] verwendet worden ist. Es ist jedoch davon auszugehen, dass die Programmierung ohne Puffer verwendet worden ist.

jeweils $70\,ns$, so dass sich insgesamt $190\,ns$ für ein Byte ergeben. Bevor ein Datenbyte geschrieben werden kann, sind drei Steuerbytes und somit $570\,ns$ notwendig.

Das Schreiben des eigentlichen Datenbytes zum Flash hängt weiterhin von der Reaktionszeit des Flashs ab. Diese beträgt typischerweise $60.000\,ns$ und ergibt somit zusammen mit der Zugriffszeit auf den Co-Prozessor, die in diesem Fall $160\,ns$ beträgt, insgesamt $60.730\,ns$ für das Übertragen eines einzelnen Datenbytes zum Speicher. Für das Verifizieren der geschriebenen Daten fallen noch einmal $310\,ns$ an.

Bei einem JTAG Takt von $50\,MHz$ sind für die optimale[70] Übertragung einer 1 Mbyte großen Datei vom PC zum FPGA ca. $168\,ms$ zu veranschlagen.

Insgesamt ergibt sich somit eine Zeit für die Datenübertragung vom Test-PC zum FPGA und das Programmieren des Flashs sowie das Verifizieren der geschriebenen Daten von ca. $64{,}17\,s$. $(1.048.576 * 61.040\,ns + 168\,ms)$

Vergleicht man die erzielten Ergebnisse für die Programmierdauer durch ein ROBSY Testinstruments mit denen von Boundary-Scan aus Tabelle 19, so ergibt sich eine Beschleunigung um den Faktor von 12,89. Im Vergleich zu VLSR Instrumenten aus [15] ist das generierte Testinstrument um den Faktor 1,19 schneller.

Wird für die Programmierung des Flashs die Möglichkeit des internen Puffers verwendet, kann die benötigte Programmierzeit weiter reduziert werden. In diesem Fall können bis zu 32 Bytes der Reihe nach übertragen werden und so eine deutlich schnellere Abarbeitung erfolgen. Selbst ohne eine optimierte Datenübertragung über Wishbone ergibt sich eine Zugriffszeit für das Schreiben und Verifizieren von 32 Bytes von $(3 * 190\,ns + (32 * 120 + 70) + 240.000$[71] $+ (32 * 140)$[72] $= 248.960\,ns$ für 32 Bytes. Somit ergibt sich die Gesamtrechnung zu $((1.048.576/32) * 248.960\,ns + 168\,ms)$ = ca. 8,33 s.

Dies entspricht einer Beschleunigung um den Faktor von 99,28 im Vergleich zu Boundary-Scan. Der Vergleich zum VLSR Instrument kann an dieser Stelle nicht erfolgen, da keine entsprechende Realisierung bekannt ist.

[70] Wenn die Daten direkt über JTAG in den Speicher des Prozessors abgelegt werden.

[71] Die typische Zeitdauer eines Buffer-writes im Vergleich zu Byte-write erhöht sich von 60 µs auf 240 µs.

[72] Reine Lesezeit. Setzt das Verifizieren der Daten auf L2 voraus, da ansonsten noch die Zeiten des Prozessors für den Datenvergleich dazukommen. Diese sind prozessorabhängig und können variieren.

5.7. VERGLEICH MANUELL GENERIERTER CO-PROZESSOREN

Für die Ansteuerung des SRAM sind manuell generierte Co-Prozessoren mit unterschiedlicher Komplexität betrachtet worden, um das Potential der Umsetzung des Ebenenkonzepts zu verifizieren und den Einfluss der Partitionierung anhand eines Beispiel zu zeigen. Diese Ergebnisse für einen Walking-1 Test des Daten-, Adress- und Steuerbusses wurden in [115] veröffentlicht und sind in Tabelle 21 und Abbildung 63 dargestellt.

Hierbei zeigt sich eine signifikante Beschleunigung der Testausführung, in Abhängigkeit, der in Hardware realisierten Ebenen. Ebenfalls ist die deutliche Beschleunigung gegenüber der ausschließlichen Testdurchführung über Boundary-Scan zu erkennen. Die ermittelten Zahlen sind stark vom DUT und dem verwendeten FPGA sowie dem Boundary-Scan-Takt bzw. Prozessor/Co-Prozessor Takt abhängig.

Auffällig an den vorliegenden Zahlen ist die relativ geringe Beschleunigung bei ausschließlicher Implementierung der Ebene L1 in Hardware. Dies liegt zum größten Teil daran, dass sowohl zum Co-Prozessor als auch zum DUT fast die gleichen Daten bereitgestellt werden müssen; dadurch ergibt sich die größte Zeitdauer für das Bereitstellen der Daten und nicht für deren Anwendung. Dies ist jedoch auch von DUT zu DUT verschieden und hängt ebenfalls vom verwendeten Prozessortakt ab. Der größte Vorteil eines Co-Prozessors mit L1 in Hardware ist eine garantierte zeitliche korrekte Ansteuerung, die jedoch mit den reinen Zahlen der Messwerte nicht ausgedrückt werden kann. Ein signifikanter Geschwindigkeitsvorteil ergibt sich in diesem Beispiel erst bei der Implementierung höherer Ebenen im Co-Prozessor.

Tabelle 22: Testergebnisse für manuell generierte Co-Prozessoren

Eigenschaften Test Szenario [73]	Takte	Beschleunigungs-faktor (zu SW L1-L5)	Testzeit [µs]	Beschleunigungs-faktor (zu BScan [74])
SW: L1-L5 HW: keine	126743	1	2534,86	8,6
SW: L2-L5 HW: L1	124319	1,02	2486,38	8,8
SW: L3-L5 HW: L1-L2	18347	6,91	366,94	59,6
SW: L4-L5 HW: L1-L3	1898	66,78	37,96	576,0

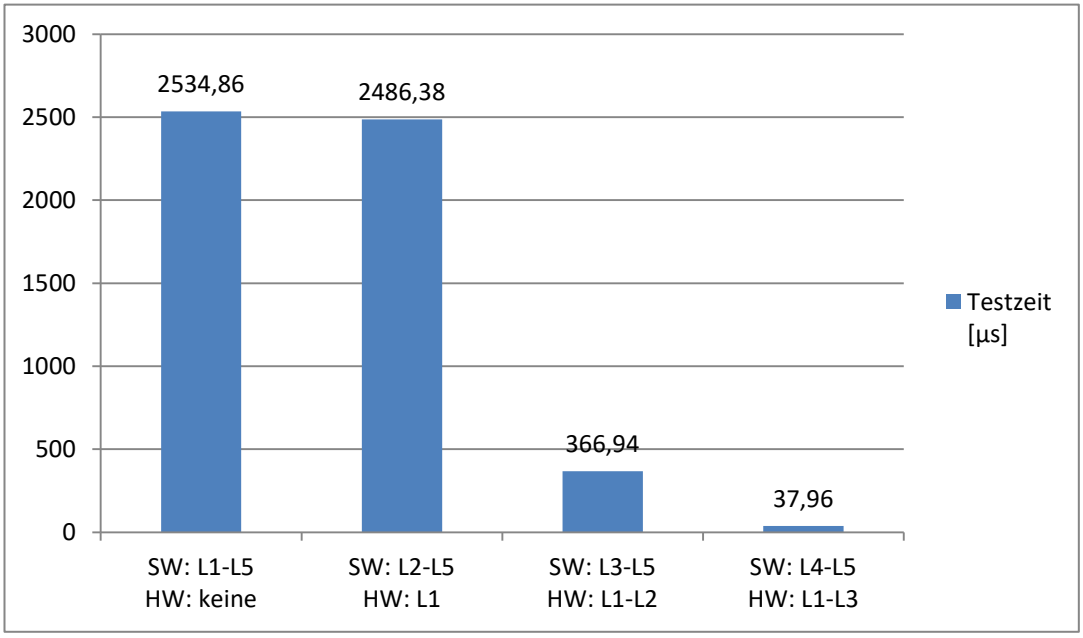


Abbildung 62: Testzeiten für manuell generierte Co-Prozessoren

[73] Testtakt des Prozessors und Co-Prozessors im FPGAs beträgt 50 MHz.

[74] Boundary-Scan mit 384 aktiven Zellen und 10 MHz Testtakt.

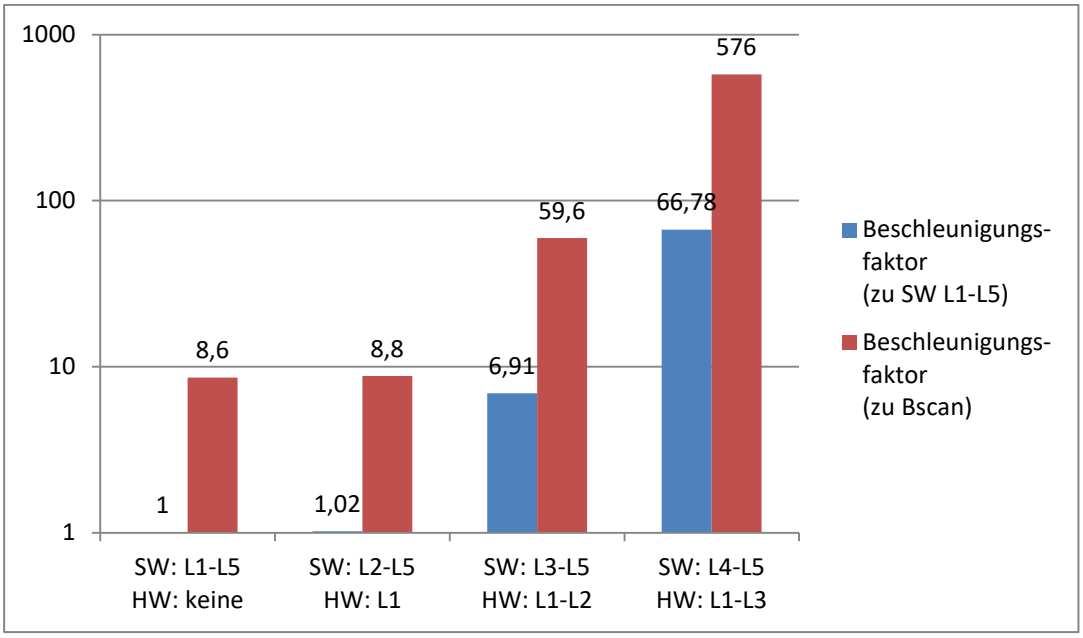


Abbildung 63: Relative Beschleunigungsfaktoren manuell generierter Co-Prozessoren

5.8. BEWERTUNG

Die gezeigten Ergebnisse aus den durchgeführten Simulationen, praktischen Untersuchungen und den Vergleichen mit virtuellen Testinstrumenten und manuell optimierten Co-Prozessoren auf Basis des in dieser Arbeit vorgestellten Ebenenkonzepts lassen folgende Schlussfolgerungen zu:

1. Eine automatische Generierung von vollständigen Testsystemen, bestehend aus Prozessor und Co-Prozessor auf Basis der Netzliste und DUT-Modellen ist möglich.
2. Die Modellierung ermöglicht eine einfache textuelle Abbildung der zeitlichen Abhängigkeiten aus den Datenblättern und erlaubt daraus eine automatische Generierung einer Ansteuerung.
3. At-speed Testen und eine schnelle Testausführung des Gesamttests sind möglich.
4. Die Vergleiche zu manuell optimierten Ansteuerungen zeigen eine geringfügig langsamere Testausführung der automatisch generierten Testinstrumente, jedoch auch eine signifikante Beschleunigung gegenüber einer Boundary-Scan-basierten Umsetzung der Ansteuerungen.
5. Unabhängige Modellierung der Testsystemkomponenten ist möglich. Somit kann die DUT Modellierung ohne Wissen des später verwendeten FPGAs, der Verschaltung auf dem Prüfling, des zu verwendenden Prozessors oder der Partitionierung der Testalgorithmen auf Hardware und Software erfolgen.

6. Die Verwendung von DBMs zur Generierung von Ansteuerungen erlaubt in Abhängigkeit des gewählten Taktes eine automatische und zeitlich korrekte Anpassung des Co-Prozessors an das zu testende System.
7. Eine Umsetzung der höheren Ebenen (L2 bis L5) im Co-Prozessor bringt eine erhebliche Reduzierung der Testzeit.
8. Bei der automatischen Modellierung ist es bisher nicht möglich, kontextbasiert befehlsübergreifend Optimierungen durchzuführen. Insofern muss jede im DUT-M modellierte Funktion in Ebene L1 in sich abgeschlossen sein.
9. Die genauen Beschleunigungsfaktoren, die gegenüber Boundary-Scan oder anderen Verfahren wie den VLSR-basierten Testinstrumenten [15] erreicht werden können, sind von unterschiedlichen Faktoren abhängig und können nur für einzelne Testfälle genau angegeben werden. Tendenziell lässt sich jedoch feststellen, dass der in dieser Arbeit vorgestellte Ansatz teilweise signifikant schneller ist als Boundary-Scan.
10. Der ROBSY Ansatz erlaubt es, beliebige Testmuster zu modellieren und zu generieren. Die Anwendung beliebiger Testmuster erfolgt in Abhängigkeit des gewählten Taktes at-speed. Folglich unterstützt der ROBSY Ansatz den at-speed Test und die damit potentiell bessere Testabdeckung.
11. Durch das vorgestellte Konzept lassen sich die Testzeiten senken bzw. die Testabdeckung um zeitlich korrektes Testen erhöhen. Die genauen Testkosten hängen dabei von den Projektanforderungen ab. Da die Testsystemgenerierung bei verfügbaren DUT-Modellen vollständig automatisch abläuft, ist zu vermuten, dass die Verwendung von ROBSY, insbesondere in Anwendungen, die aktuell lange Testzeiten haben bzw. bei denen es auf zeitlich korrektes Ansteuern der DUTs ankommt, positiven Einfluss auf die Testzeiten und die Testkosten haben wird. Da das Testsystem nur auf bestehende Ressourcen zurückgreift, gibt es keine Erhöhung der Kosten für die eigentliche Leiterplattenherstellung.

6. ZUSAMMENFASSUNG, FAZIT UND AUSBLICK

6.1. ZUSAMMENFASSUNG

Im Rahmen dieser Arbeit ist, ausgehend von den Gegebenheiten moderner Leiterplatten, auf die aktuellen Anforderungen an strukturelle Leiterplattentests eingegangen worden. Es wurden unterschiedliche Testverfahren auf ihre Eignung bezüglich dieser Anforderungen hin betrachtet. Darüber hinaus wurde der aktuelle Stand der Technik verschiedener Testverfahren und innovativer Testansätze untersucht. Als Ergebnis wurden in Kapitel 3.5 unzureichend gelöste Herausforderungen zusammengefasst und daraus eine Zielstellung für die vorliegende Arbeit abgeleitet.

Basierend auf dieser Zielstellung wurde ein konzeptioneller Lösungsansatz erarbeitet. Dessen Kern ist eine Methode, um FPGA-basierte Testinstrumente zu modellieren und automatisch und adaptiv zu generieren. Zentrales Element dieser Methode ist eine flexible Testsystemarchitektur, die es erlaubt, strukturelle Leiterplattentests auf unterschiedliche Weise zu implementieren, um sich somit den jeweils gegebenen Randbedingungen anzupassen. Für die Modellierung der Komponenten des Testsystems wurden unterschiedliche Möglichkeiten betrachtet und letztendlich eine eigene Modellierungssprache entwickelt.

Im Rahmen von mehreren Untersuchungen wurde ein proof-of-concept durchgeführt. Es wurden unterschiedliche Testsysteme generiert und deren Leistungsfähigkeit beurteilt.

6.2. FAZIT

Das vorgestellte Konzept eines FPGA-basierten Testinstruments unterscheidet sich von anderen Ansätzen in wesentlichen Punkten.

- Das Testinstrument kann im Gegensatz zu anderen synthetischen Testinstrumenten wie z.B. [51] [52] automatisch generiert werden und muss nicht manuell erzeugt werden.
- Die Realisierung des Testinstruments im FPGA wird durch eine Kombination aus Prozessor und Co-Prozessor realisiert, die beide automatisch an die jeweiligen Bedürfnisse der Testausführung angepasst werden können.
- Das Testinstrument wird individuell anhand einer Beschreibung der Leiterplatte und des verwendeten FPGAs, sowie der Schaltkreise zu denen die Verbindungen getestet werden soll adaptiert und nicht im Vorfeld universell generiert [15].
- Das Testinstrument basiert auf einer unabhängigen Beschreibung aller Schaltkreise, zu denen die Verbindung getestet werden sollen, sowie einer Beschreibung der Leiterplatte und des verwendeten FPGAs. Im Gegensatz zu [68] werden keine vorgefertigten funktionsspezifischen Beschreibungen verwendet.
- Das Testinstrument besteht aus verschiedenen Ebenen und Interfaces. Dies ermöglicht es verschiedene Varianten der Testinstruments zu erzeugen und die Testalgorithmen auf unterschiedliche Arten zu implementieren (SW, ESW, HW). Eine Partitionierung der Testausführung zum Zeitpunkt der Generierung des Testsystems und nicht zum Zeitpunkt der Modellierung ist in der Literatur bisher nicht beschrieben.

Das Konzept erschließt einen großen Einsatzbereich für den strukturellen Verbindungstests auf Leiterplatten, da es auf alle bestehenden Leiterplatten mit einem FPGA angewendet werden kann. Hierbei stellt das Konzept wenige Voraussetzungen an die Hardware. Die Voraussetzungen sind:

- es muss ein FPGA auf dem Prüfling vorhanden sein
- der FPGA muss über JTAG mit dem Test-PC verbunden sein
- der zu testende Schaltkreis (DUT) muss mit dem FPGA verbunden sein; es können also nur Verbindungen zwischen DUT und FPGA getestet werden

Es entstehen bei der Herstellung der Leiterplatte keine zusätzlichen Kosten, da ausschließlich bestehende Ressourcen verwendet werden, die bereits auf der Leiterplatte vorhanden sind.

Die Schaltkreise (DUTs), zu denen die Verbindung getestet werden soll, werden in separaten, sogenannten DUT-Modellen beschrieben. Diese beinhalten alle nötigen Informationen für die Ansteuerung von DUTs und die Generierung der Testvektoren

anhand algorithmischer Beschreibungen[75]. Hierdurch ist eine systematische Modellierung unabhängig vom späteren Einsatz der DUTs erreicht worden. Durch diese Unabhängigkeit ist es möglich, das Testsystem zum Zeitpunkt der Generierung an die Struktur und Randbedingungen des jeweiligen Prüflings zu adaptieren. So kann ein an die vorhandenen Ressourcen und die benötigte Testgeschwindigkeit angepasstes Testsystem generiert werden. Dieser Prozess ist automatisierbar und erleichtert damit den Einsatz des Testsystems.

Durch die Verwendung eines Ebenenkonzepts ist es möglich, Testfunktionen flexibel auf unterschiedliche Arten auszuführen. Dies kann in Software auf dem Test-PC und somit extern vom FPGA oder in Embedded Software beziehungsweise in Hardware im FPGA erfolgen. Es erlaubt Anpassungen an die verfügbaren Testressourcen sowie an die benötigte Testgeschwindigkeit und verbindet die Vorteile der jeweiligen Ausführungsart. Insbesondere im Vergleich zu Boundary-Scan wird die Testausführung durch die Möglichkeit beschleunigt, Teile der Testfunktionalität innerhalb des FPGAs zu platzieren.

Die Ausführung von Testalgorithmen in Hardware ermöglicht es, diese at-speed und somit unter zeitlich gleichen Bedingungen wie im späteren Einsatz der Leiterplatte durchzuführen. Weiterhin ist ein umfangreicher Testzugriff gegeben, da auf alle Pins des FPGAs in gleicher Weise wie im späteren Einsatzfall zugegriffen werden kann und im FPGA beliebige Funktionen realisiert werden können.

Die Generierung eines Testsystems ist vollständig automatisch durchführbar, so dass im Idealfall von den DUT-Modellen bis hin zum FPGA-Bitfile keine manuellen Eingriffe des Testingenieurs notwendig sind. Dies ermöglicht es, sowohl die Testsystemgenerierung als auch die Testausführung weitgehend zu automatisieren und dadurch die Testkosten zu reduzieren. Dies ist insbesondere bei kurzen Design-Zyklen wichtig, um ein leistungsfähiges Testsystem zu realisieren. Nachteilig gegenüber anderen Ansätzen ist jedoch, dass eine einmalige Generierung (Synthese des FPGA-Bitfiles) für jedes Testinstrument zwingend erforderlich ist. Eine Abwägung der Vor- und Nachteile ist für jeden Einsatzfall einzeln zu prüfen und kann nicht pauschal beantwortet werden.

Durch den modularen Aufbau und die abgeschlossene Modellierung in DUT-Modellen ist weiterhin eine einfache Wiederverwendung dieser Modelle möglich. Dies beschleunigt den Designprozess neuer Testsysteme, da nicht für jeden neuen Prüfling ein neues Testsystem manuell entwickelt werden muss, sondern lediglich neue DUT-Modelle für bisher nicht verwendete Schaltkreise zu erstellen sind.

[75] Testvektoren können in RTDL durch arithmetische Operationen berechnet, durch Schieberegister (LFSR) bestimmt oder in tabellarischer Form abgelegt werden.

Die Untersuchungen in Kapitel 5 haben gezeigt, dass sich die zeitkritische Testfunktionalität von Co-Prozessoren, also die in Hardware realisierte Funktionalität der Ebene L1, automatisch anhand von Difference Bound Matrizen beschreiben lässt. Die hieraus automatisch generierten Ansteuerungen sind zeitlich korrekt, jedoch erreichen sie (noch) nicht die Leistungsfähigkeit manuell optimierter Testinstrumente. Die Funktionen wurden anhand von drei DUTs gezeigt, lassen sich jedoch auf die meisten Schaltkreise überführen, die eine registerbasierte Ansteuerung nutzen.

Im Vergleich zu Testinstrumenten aus [15] zeigen die Ergebnisse dieser Arbeit bereits jetzt ihre konkurrenzfähige Leistungsfähigkeit bei gleichzeitig flexiblerer Implementierung und automatischer Generierung. Im Gegensatz zu anderen Ansätzen wie z.B. in [15], [66] oder [116], ist die Testausführung im FPGA unabhängig von dem verwendeten Boundary-Scan Takt und at-speed fähig. Weiterhin zeigen Versuche mit Co-Prozessoren höherer Komplexität, wie deren Leistungsfähigkeit stark ansteigt, wenn die Implementierung von Funktionen zunehmend in Hardware erfolgt und weniger in Embedded Software auf dem Testprozessor oder in Software auf dem Test-PC.

Diese Arbeit legt den Grundstein für eine sehr leistungsstarke, flexible Testsystemarchitektur, die das automatische Generieren von Testsystemen ermöglicht. Sie erlaubt signifikante Beschleunigungen im Vergleich zu Boundary-Scan und anderen Testinstrumenten [15], ohne dabei auf speziell angepasste und in den meisten Fällen manuell entwickelte Testinstrumente angewiesen zu sein und ermöglicht at-speed Testen.

Das Konzept der modell-basierten Beschreibung von „Synthetischen Testinstrumenten“ unter der speziellen Berücksichtigung der zeitlichen Parameter einer DUT Ansteuerung ist in seiner Art einzigartig und hat ein großes Potential für die Verbesserung struktureller Verbindungstests auf Leiterplatten.

6.3. AUSBLICK

Aufbauend auf den Grundlagen dieser Arbeit und des durchgeführten proof-of-concepts ist es möglich, die bisherige Realisierung weiterzuentwickeln. Im Folgenden werden einige **Modellierungsvarianten** und **Compilererweiterungen** genannt.

Das vorgestellte Konzept setzt für jeden zu testenden Schaltkreis ein DUT Modell voraus. Für einen verbreiteten Einsatz ist eine umfangreiche Bibliothek dieser Modelle notwendig. Hierfür wäre eine Datenbank wie in [80] wünschenswert, wenn diese Modelle in einer geeigneten Modellierung vorliegen, um selbständig daraus Ansteuerungen zu generieren.

Für den Einsatz eines ROBSY Testinstruments in Verbindung mit Testtools unterschiedlicher Hersteller wäre die automatische Erzeugung einer ICL/PDL Beschreibung für jedes generierte Testsystem hilfreich. In dieser Beschreibung würde das Testinstrument den Test-PC über seine Fähigkeiten informieren und diese zur Verfügung stellen.

Zur Unterstützung von Tests für Highspeed Interfaces wie LVDS, PCIe und DDR2 sollte das Einbinden externer IP-Cores unterstützt werden. Die Modellierung in RTDL würde sich dann auf deren Interface beziehen und entsprechend vereinfachen. Insbesondere die extrem kritischen zeitlichen Parameter an den physikalischen Pins des FPGAs und die teilweise komplexen Protokolle dieser Schnittstellen sind nicht einfach und teilweise nur mit speziellen Blöcken innerhalb des FPGAs realisierbar.

Die Modellierung in RTDL sollte so modularisiert werden, dass unterschiedliche Gehäuseformen für ein und denselben Schaltkreis unterstützt werden. Hierbei ist insbesondere die Möglichkeit geringer Abweichungen einzelner Zugriffsmodi zu berücksichtigen, die bei einigen Schaltkreisen nur bei ausgesuchten Gehäuseformen unterstützt werden. Bisher wird jeder Schaltkreis für jeweils genau ein Gehäuse spezifiziert. Bei größeren Datenbeständen wird es zunehmend aufwendiger, einen Schaltkreis in unterschiedlichen Gehäuseformen konsistent zu halten.

Neben der Möglichkeit, Takte durch den Einsatz von Taktmanagern zu verändern und den Anforderungen der DUTs anzupassen, ist auch die Nutzung von fallenden und steigenden Flanken der Taktsignale wünschenswert. Dies würde die Granularität der Ansteuerung der DUTs verfeinern. Dies ist jedoch nicht einfach, da es die Prüfung der Signallaufzeiten innerhalb der Logik des FPGAs verkompliziert. Es ist daher nötig, entsprechende Randbedingungen (Constraints) für die FPGA-Synthese automatisch zu generieren. Weiterhin sollte die Verarbeitung hierarchischer DBMs im Compiler realisiert werden, um eine einfache Modellierung von komplexeren Befehle, Befehlssequenzen und Protokollen zu ermöglichen.

Alle im Ausblick aufgeführten Verbesserungen bei der Generierung von Co-Prozessoren für FPGA-basierte Testsysteme sind nicht durch das Konzept des Testsystems begrenzt, sondern fehlen lediglich in der aktuellen Implementierung.

Neben den bisher genannten direkten Verbesserungen für die Generierung von Co-Prozessoren gibt es noch weiterführende Verbesserungen, die jedoch vorrangig das Gesamtsystem und nicht die Co-Prozessoren direkt betreffen.

Hierzu gehört unter anderem eine **Ressourcenschätzung** sowohl der wahrscheinlich benötigten Ressourcen aller Komponenten eines Testsystems als auch für die zu erwartenden Testzeiten. Während der Ressourcenbedarf für die Ausführungen in Software und Embedded-Software für jede zu untersuchende Partitionierung einfach anhand des generierten Assembler-Codes zu berechnen ist, so ist dies bei der Hardwarebeschreibung aufwendiger. Hierbei muss für jede Partitionierung mindestens die Hardwaresynthese durchlaufen werden, die je nach Größe des zu generierenden Test-Systems mehrere Sekunden, typischerweise jedoch mehrere Minuten dauern kann. Bei solchen Ausführungszeiten ist es nicht möglich, innerhalb einer akzeptablen Zeitspanne ausreichend viele Partitionierungen zu untersuchen und zu vergleichen, um die beste Partitionierung zu bestimmen. Daher ist es nötig, den zu erwartenden Ressourcenverbrauch bereits auf Basis des DUT-Modells oder der VHDL Realisierung

zu schätzen. Auf die Probleme der gegenseitigen Abhängigkeiten wurde in Kapitel 4.3 eingegangen.

Eine weitere zu untersuchende Fragestellung ist, auf welchem Abstraktionslevel Ressourcen sinnvoll geschätzt werden können. Es bieten sich Schätzungen auf sehr hoher Ebene an, indem Möglichkeiten gefunden werden, auf Basis der RTDL Beschreibungssprache zu schätzen. So könnten Ressourcen- und Zeit-Parameter von Funktionsblöcken in einer Bibliothek abgelegt und diese Blöcke dann auf RTDL abgebildet und damit die Ressourcen geschätzt werden.

Eine weitere Möglichkeit liegt in der Generierung von Kontrollfluss- / Datenflussgraphen (CFG/DFG), welche für die Schätzung herangezogen werden können [117] [118]. Es sind geeignete Methoden zu finden, wie man z.B. die Parameter verschiedener Knoten in Bibliotheken beschreibt. Es wäre auch möglich, alle Konstrukte im DUT-M (wie z.B. if, while, shift, add, etc.) als Objekte mit bestimmten Parametern zu verstehen und deren Ressourcenbedarf anhand einfacher Formeln zu schätzen.

Neben der Ressourcenschätzung müssen auch die **Ausführungszeiten** geschätzt werden. Dabei muss die gesamte Struktur des erstellten Systems berücksichtigt werden, also nicht nur die Aufteilung in Hardware, Embedded Software und Software, sondern auch die Prozessorstruktur (Anzahl Register, verfügbare Instruktionen, etc.) sowie die Busarchitektur, die den Datenaustausch zwischen Prozessor und Co-Prozessor realisiert. Alle diese Faktoren haben einen entscheidenden Einfluss auf die Ausführungszeiten. Nur wenn alle Teile korrekt ermittelt bzw. geschätzt werden können, ist eine hinreichend genaue Vorhersage der Ausführungszeit der einzelnen Teilkomponenten und somit eine zuverlässige Optimierung möglich.

THESEN

1. Begrenzter Testzugriff, aufwendige Testsystemgenerierung, kurze Design-Zyklen und ständiger Kostendruck erschweren zunehmend das Testen moderner Leiterplatten.

2. Nicht-invasive strukturelle Leiterplattentests bieten insbesondere bei modernen Leiterplatten mit immer kleiner werdenden Strukturen Vorteile gegenüber anderen Verfahren.

3. Testfunktionalität vom Test-PC auf den Prüfling zu übertragen, kann die Testausführung beschleunigen. Dabei haben sowohl software-basierte als auch hardware-basierte Verfahren ihre eigenen Vorteile. Ein Testsystem muss automatisch generierbar sein und sich flexibel an die zu testende Leiterplatte anpassen lassen.

4. Ein Ebenenkonzept erlaubt die Strukturierung von Testfunktionalität und die wahlweise Ausführung der Algorithmen in Software, Embedded Software oder Hardware und verbindet damit die Vorteile der unterschiedlichen Ausführungsarten miteinander. Difference Bound Matrizen eignen sich zur Modellierung zeitkritischer Zugriffsfunktionen der DUTs innerhalb des Ebenenmodells.

5. Eine automatische Generierung eines FPGA-basierten Testsystems, bestehend aus einem Testprozessor und einem Co-Prozessor mit einer zeitlich korrekten Ansteuerung des DUTs, ist möglich und zeigt im Vergleich zu anderen Verfahren eine beschleunigte Testausführung. Diese steigt, je mehr Ebenen in Hardware realisiert werden.

6. Das ROBSY-Konzept erfüllt die Anforderungen an ein modulares und automatisch generierbares Testsystem und erlaubt eine flexible Anpassung an die Anforderungen des Prüflings.

ERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Bei der Auswahl und Auswertung folgenden Materials haben mir die nachstehend aufgeführten Personen in der jeweils beschriebenen Weise unentgeltlich geholfen:

- Erweiterung der Struktur des DUT-Modells: Dr.-Ing. Heinz-Dietrich Wuttke, Jörg Sachße und Jorge Hernán Meza Escobar (TU Ilmenau).
- Entwicklung der RTDL Sprache der Ebenen L2 bis L5 für die DUT-Modelle: Jörg Sachße und Jorge Hernán Meza Escobar (TU Ilmenau).
- Entwicklung der Timingmodellierung für die DUT-Modelle: Sebastian Richter, Thomas Sasse, Jörg Sachße und Jorge Hernán Meza Escobar (TU Ilmenau).
- Bereitstellung der Grundfunktion eines Parsers in Python für den RTDL Compiler: Markus Brückner

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß § 7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Lilienthal, 29.06.2016 Steffen Ostendorff

LITERATURVERZEICHNIS

[1] „Xilinx.com : Financial Statements“. [Online]. Verfügbar unter: http://investor.xilinx.com/financials-statements.cfm. [Zugegriffen: 29-Feb-2016]

[2] „2014 Form 10-K & Proxy Statement“. [Online]. Verfügbar unter: http://files.shareholder.com/downloads/XLNX/1180608675x0x766014/58CDA6BE-56AE-4AC6-802D-DE27F6E83D47/xilinx-2014_0001.pdf. [Zugegriffen: 16-Dez-2015]

[3] „FPGA Market by Type (High-End, Mid-End, Low-End), Verticals (Telecommunication, Industrial, A&D, Automotive, & Others), Architecture (Sram, Flash, & Antifuse), Technology Node (28nm-10nm, 45/40nm, & Others), and Geography - Forecast to 2022“. [Online]. Verfügbar unter: http://www.researchandmarkets.com/research/xc57px/fpga_market_by. [Zugegriffen: 26-Feb-2016]

[4] S. Eggersglüß, G. Fey, und I. Polian, *Test Digitaler Schaltkreise*. De Gruyter Oldenbourg, 2014.

[5] M. Berger, *Test- und Prüfverfahren in der Elektronikfertigung: Vom Arbeitsprinzip bis zu Design für Test-Regeln*. Berlin: VDE Verlag GmbH, 2012.

[6] T. Wenzel und H. Ehrenberg, „Embedded System Access - White Paper: Der fundamentate Paradigmentwechsel beim elektrischen Test“. 2011.

[7] R. G. Bennets, *Design of Testable Logic Circuits*. Addison-Wesley, 1984.

[8] T. Wenzel und H. Ehrenberg, „Ein neues Test-Zeitalter beginnt: Embedded System Access“, *Elektronik*, Bd. 10/2012.

[9] S. Ostendorff, H.-D. Wuttke, J. Sachße, und Köhler, Steffen, „A new Approach for Adaptive Failure Diagnostics Based on Emulation Test“, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Dresden, 2010, S. 327–330.

[10] M. Berger und R. Hartung, *Vollständige Fehlerabdeckung durch Kombination verschiedener Testverfahren (ICT, MFT, AOI, AXI ...) : unter besonderer Berücksichtigung des Boundary Scan*, Bd. 70–23. Berlin: FED, 2005.

[11] „INGUN Prüfmittelbau GmbH“. [Online]. Verfügbar unter: http://www.ingun.de/. [Zugegriffen: 13-Juni-2016]

[12] „ITOCHU SysTech“. [Online]. Verfügbar unter: http://www.systech-europe.de/. [Zugegriffen: 13-Juni-2016]

[13] „IEEE SA - 1149.1-1990 - IEEE Standard Test Access Port and Boundary-Scan Architecture“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.1-1990.html. [Zugegriffen: 22-Juli-2013]

[14] „IEEE SA - 1149.1-2013 - IEEE Standard for Test Access Port and Boundary-Scan Architecture“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.1-2013.html. [Zugegriffen: 22-Juli-2013]

[15] Aleksejev, Igor, „FPGA-based Embedded Virtual Instrumentation“, Dissertation, Tallinn University of Technology, Tallinn, Estonia, 2013.

[16] P. B. Geiger und S. Butkovich, „Boundary-scan adoption - an industry snapshot with emphasis on the semiconductor industry“, in *IEEE International Test Conference (ITC)*, 2009, S. 1–10.

[17] T. Nirmaier, J. T. Zaguirre, E. Hong, W. Spirkl, A. Rettenberger, und D. Schmitt-Landsiedel, „Efficient High-Speed Interface Verification and Fault Analysis“, in *IEEE International Test Conference (ITC)*, 2008, S. 1–9.

[18] W. Daehn, *Testverfahren in der Mikroelektronik - Methoden und Werkzeuge*. Springer Verlag Berlin / Heidelberg, 1997.

[19] W. J. Hurd, „Efficient Generation of Statistically Good Pseudonoise by Linearly Interconnected Shift Registers“, *IEEE Transactions on Computers*, Bd. C-23, Nr. 2, S. 146–152, Feb. 1974.

[20] H. Wunderlich und S. Hellebrand, „The pseudoexhaustive test of sequential circuits“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Bd. 11, Nr. 1, S. 26–33, Jan. 1992.

[21] J. Rajski und J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems*. Upper Saddle River, NJ: Pearson Education Ltd., 1997.

[22] „IS61LV25616AL 256K x 16 low voltage CMOS static RAM data sheet“. Integrated Silicon Solution, Inc. [Online]. Verfügbar unter: http://www.issi.com/WW/pdf/61LV25616AL.pdf. [Zugegriffen: 10-Juni-2015]

[23] L. Y. Ungar, H. Bleeker, J. E. McDermid, und H. Hulvershorn, „IEEE-1149.x standards: achievements vs. expectations“, in *IEEE Systems Readiness Technology Conference (AUTOTESTCON)*, 2001, S. 188–205.

[24] „IEEE SA - 1149.4-2010 - IEEE Standard for a Mixed-Signal Test Bus“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.4-2010.html. [Zugegriffen: 29-Juli-2014]

[25] „IEEE SA - 1149.5-1995 - IEEE Standard for Module Test and Maintenance Bus (MTM-Bus) Protocol“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.5-1995.html. [Zugegriffen: 29-Juli-2014]

[26] „IEEE SA - P1149.6 - Standard for Boundary-Scan Testing of Advanced Digital Networks“. [Online]. Verfügbar unter: http://standards.ieee.org/develop/project/1149.6.html. [Zugegriffen: 29-Juli-2014]

[27] „IEEE SA - 1149.7-2009 - IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.7-2009.html. [Zugegriffen: 29-Juli-2014]

[28] „IEEE SA - 1149.8.1-2012 - IEEE Standard for Boundary-Scan-Based Stimulus of Interconnections to Passive and/or Active Components“. [Online]. Verfügbar unter: http://standards.ieee.org/findstds/standard/1149.8.1-2012.html. [Zugegriffen: 29-Juli-2014]

[29] „IEEE SA - P1149.10 - High Speed Test Access Port and On-chip Distribution Architecture“. [Online]. Verfügbar unter: http://standards.ieee.org/develop/project/1149.10.html. [Zugegriffen: 29-Juli-2014]

[30] J. Nejedlo und R. Khanna, „Intel IBIST, the full vision realized“, in *IEEE International Test Conference (ITC)*, 2009, S. 1–11.

[31] A. Jutman, „At-speed on-chip diagnosis of board-level interconnect faults“, in *IEEE European Test Symposium (ETS)*, 2004, S. 2–7.

[32] A. Q. Olozabal, M. de los A. C. Chacon, und D. G. Vela, „FPGA-Based Boundary-Scan Bist“, in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2006, S. 1–4.

[33] J. H. Meza Escobar, „Towards an embedded board-level Tester - Study of a specialized Test Processor“, Dissertation, Technische Universität Ilmenau, Ilmenau, unveröffentlicht.

[34] D. Gizopoulos, A. Paschalis, und Y. Zorian, *Embedded Processor-Based Self-Test*. Boston: Kluwer Academic Publishers, 2004.

[35] A. Apostolakis, M. Psarakis, D. Gizopoulos, und A. Paschalis, „Functional Processor-Based Testing of Communication Peripherals in Systems-on-Chip“, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Bd. 15, Nr. 8, S. 971–975, 2007.

[36] L. Chen, S. Ravi, A. Raghunathan, und S. Dey, „A scalable software-based self-test methodology for programmable processors“, in *IEEE Design Automation Conference (DAC)*, 2003, S. 548–553.

[37] H. Ehrenberg und T. Wenzel, „Combining Boundary Scan and JTAG Emulation for Advanced Structural Test and Diagnostics“. GÖPEL electronic, 2009.

[38] D. Bonnett, „Combine Boundary Scan with CPU Emulation to Extend Test-Coverage“. ASSET InterTech, 2004.

[39] J. Webster, B. Fenton, D. Stringer, und B. Bennetts, „On the synergy of boundary scan and emulation board test: a case study“, in *Board Test Workshop (BTW)*, 2003.

[40] S. Devadze, A. Jutman, A. Tsertov, M. Instenberg, und R. Ubar, „Microprocessor-based System Test using Debug Interface“, in *IEEE NORCHIP*, 2008, S. 98–101.

[41] A. Tsertov, A. Jutman, und S. Devadze, „Testing beyond the SoCs in a lego style“, in *East-West Design and Test Symposium (EWDTS)*, 2010, S. 334–338.

[42] A. Tsertov, R. Ubar, A. Jutman, und S. Devadze, „SoC and Board Modeling for Processor-Centric Board Testing“, in *IEEE Euromicro Conference on Digital System Design (DSD)*, 2011, S. 575–582.

[43] A. P. Strole und H. Wunderlich, „TESTCHIP: a chip for weighted random pattern generation, evaluation, and test control“, *IEEE Journal of Solid-State Circuits*, Bd. 26, Nr. 7, S. 1056–1063, 1991.

[44] M. L. Ali, Z. M. Darus, M. A. M. Ali, und I. Ahmed, „Test processor ASIC design“, in *IEEE International Conference on Semiconductor Electronics (ICSE)*, 1996, S. 261–265.

[45] Z. M. Darus, I. Ahmed, und M. L. Ali, „A test processor chip implementing multiple seed, multiple polynomial linear feedback shift register“, in *IEEE Test Symposium (ATS)*, 1997, S. 155–160.

[46] M. A. Kabir und L. Ali, „Design of GLFSR based test processor chip“, in *IEEE Student Conference on Research and Development (SCOReD)*, 2009, S. 234–237.

[47] C. Galke, M. Pflanz, und H. T. Vierhaus, „A test processor concept for systems-on-a-chip“, in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, S. 210–212.

[48] C. Kretzschmar, C. Galke, und H. T. Vierhaus, „A hierarchical self test scheme for SoCs“, in *IEEE International On-Line Testing Symposium (IOLTS)*, 2004, S. 37–42.

[49] R. Frost, D. Rudolph, C. Galke, R. Kothe, und H. T. Vierhaus, „A Configurable Modular Test Processor and Scan Controller Architecture“, in *IEEE International On-Line Testing Symposium (IOLTS)*, 2007, S. 277–284.

[50] S. Zeidler, C. Wolf, M. Krstic, F. Vater, und R. Kraemer, „Design of a Test Processor for Asynchronous Chip Test“, in *IEEE Asian Test Symposium (ATS)*, 2011, S. 244–250.

[51] J. Ferry, J. Scesnak, und S. Shaikh, „A strategy for board level in-system programmable built-in assisted test and built-in self test“, in *IEEE International Test Conference (ITC)*, 2005, S. 798–807.

[52] J. Ferry, „FPGA-based universal embedded digital instrument“, in *IEEE International Test Conference (ITC)*, 2013, S. 1–9.

[53] „IEEE P1687 Working Group“. [Online]. Verfügbar unter: http://grouper.ieee.org/groups/1687/. [Zugegriffen: 08-Feb-2013]

[54] F. G. Zadegan, U. Ingelsson, G. Carlsson, und E. Larsson, „Design automation for IEEE P1687“, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011, S. 1–6.

[55] „International Technology Roadmap for Semiconductors - 2009 Edition“. [Online]. Verfügbar unter: http://efutures.ac.uk/sites/default/files/ITRS_2009.pdf. [Zugegriffen: 24-Juli-2015]

[56] „International Technology Roadmap for Semiconductors - 2011 Edition“. [Online]. Verfügbar unter: http://www.semiconductors.org/clientuploads/directory/DocumentSIA/ITRS_2011ExecSum.pdf. [Zugegriffen: 24-Juli-2015]

[57] C. Giaconia, A. D. Stefano, und G. Capponi, „Reconfigurable digital instrumentation based on FPGA“, in *IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003, S. 120–122.

[58] P. B. Kelly, „A new class of test instrument: The FPGA based module“, in *IEEE AUTOTESTCON*, 2012, S. 269–271.

[59] L. Mostardini, L. Bacciarelli, L. Fanucci, L. Bertini, M. Tonarelli, und M. D. Marinis, „FPGA-based low-cost automatic test equipment for digital integrated circuits“, in *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2009, S. 32–37.

[60] S. D. Carlo, P. Prinetto, A. Scionti, J. Figueras, S. Manich, und R. Rodriguez-Montanes, „A Low-Cost FPGA-Based Test and Diagnosis Architecture for SRAMs“, in *IEEE International Conference on Advances in System Testing and Validation Lifecycle*, 2009, S. 141–146.

[61] H. Chen, D. Chen, J. Ye, W. Cao, und L. Gao, „An integrated Automatic Test Generation and executing system“, in *2011 IEEE AUTOTESTCON*, 2011, S. 383–390.

[62] H. Chen, D. Chen, L. Gao, J. Ye, W. Cao, und K. Liu, „A USB-based Automatic Test Equipment with improved behavior-based Automatic Test generation for complex circuit boards“, in *IEEE AUTOTESTCON*, 2015, S. 440–445.

[63] K. Vanitha und C. A. S. Moorthy, „Implementation of an integrated FPGA based automatic test equipment and test generation for digital circuits“, in *IEEE International Conference on Information Communication and Embedded Systems (ICICES)*, 2013, S. 741–746.

[64] I. Aleksejev, S. Devadze, A. Jutman, und K. Shibin, „Virtual reconfigurable scan-chains on FPGAs for optimized board test“, in *IEEE Latin-American Test Symposium (LATS)*, 2015, S. 1–6.

[65] I. Aleksejev, A. Jutman, S. Devadze, S. Odintsov, und T. Wenzel, „FPGA-based synthetic instrumentation for board test“, in *Proceedings of the IEEE International Test Conference (ITC)*, 2012, S. 1–10.

[66] S. Devadze, A. Jutman, I. Aleksejev, und R. Ubar, „Fast extended test access via JTAG and FPGAs“, in *IEEE International Test Conference (ITC)*, 2009, S. 1–7.

[67] A. Jutman, S. Devadze, I. Aleksejev, und T. Wenzel, „Embedded synthetic instruments for Board-Level testing“, in *IEEE European Test Symposium (ETS)*, 2012, S. 1.

[68] „Elesis Project | European library-based flow of embedded silicon test instruments“. [Online]. Verfügbar unter: http://www.elesis.eu/. [Zugegriffen: 12-Okt-2015]

[69] „BSDL Model for Xilinx Spartan 3 XC3S50A-FT256“. [Online]. Verfügbar unter: http://www.bsdl.info/details.htm?sid=41f5cbf6296a712288ca59b002289d77. [Zugegriffen: 09-Juli-2014]

[70] „Serial Vector Format (SVF) Specification“. [Online]. Verfügbar unter: http://www.asset-intertech.com/Media/en-US/Documents/Whitepapers/Serial_Vector_Format_(SVF)_Specification_Revision_E.pdf. [Zugegriffen: 26-Feb-2014]

[71] „STAPL JEDEC standard“. [Online]. Verfügbar unter: http://www.jedec.org/standards-documents/results/STAPL. [Zugegriffen: 26-Feb-2014]

[72] „Welcome to IJTAG: a no-risk path to IEEE P1687“. [Online]. Verfügbar unter: http://www.techdesignforums.com/practice/technique/using-ijtag-with-jtag-ip-blocks/. [Zugegriffen: 10-Juli-2014]

[73] F. G. Zadegan, U. Ingelsson, E. Larsson, und G. Carlsson, „Reusing and Retargeting On-Chip Instrument Access Procedures in IEEE P1687“, *IEEE Design Test of Computers*, Bd. 29, Nr. 2, S. 79–88, 2012.

[74] T. Taylor und G. A. Maston, „Standard test interface language (STIL) a new language for patterns and waveforms“, in *IEEE International Test Conference (ITC)*, 1996, S. 565–570.

[75] „STIL Language Test Vector Format“. [Online]. Verfügbar unter: http://www.micross.com/pdf/Micross_Technical_Paper-STIL_Language_Test_Vector_Format_Simplified.pdf. [Zugegriffen: 29-Feb-2016]

[76] „IEEE SA - P1450.4 - Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std. 1450-1999) for Test Flow Specification“. [Online]. Verfügbar unter: https://standards.ieee.org/develop/project/1450.4.html. [Zugegriffen: 29-Juni-2015]

[77] R. Kapur, M. Lousberg, T. Taylor, B. Keller, P. Reuter, und D. Kay, „CTL the language for describing core-based test“, in *IEEE International Test Conference (ITC)*, 2001, S. 131–139.

[78] „IEEE Standard for Memory Modeling in Core Test Language“, *IEEE Std 1450.6.2-2014*, S. 1–74, Juni 2014.

[79] C. Hergenröder, „Analyse und Konzeption struktureller Tests von dynamischen high-speed Speichern mittels FPGA“, Masterarbeit, Ilmenau, 2013.

[80] „Free Model Foundry VHDL Models“. [Online]. Verfügbar unter: http://www.freemodelfoundry.com/fmf_VHDL_models.php. [Zugegriffen: 10-Juli-2014]

[81] „IEEE SA - 1850-2010 - IEEE Standard for Property Specification Language (PSL)“. [Online]. Verfügbar unter: https://standards.ieee.org/findstds/standard/1850-2010.html. [Zugegriffen: 28-Juli-2015]

[82] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, und M. Weiglhofer, „Automatic Hardware Synthesis from Specifications: A Case Study“, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007, S. 1–6.

[83] K. Pnueli, Y. Kesten, und A. Pnueli, „Timed and Hybrid Statecharts and their Textual Representation“, 1992, S. 591–620.

[84] „Specification and Description Language - Real Time“. [Online]. Verfügbar unter: http://www.sdl-rt.org/. [Zugegriffen: 28-Aug-2014]

[85] J. A. Stankovic, „Misconceptions about real-time computing: a serious problem for next-generation systems“, *IEEE Computer Journal*, Bd. 21, Nr. 10, S. 10–19, Okt. 1988.

[86] A. Alkhodre, J.-P. Babau, und J.-J. Schwarz, „Modelling of real-time constraints using SDL for embedded systems design“, *IEEE Computing Control Engineering Journal*, Bd. 13, Nr. 4, S. 189–196, Aug. 2002.

[87] „Elektrische und optische Mess- und Testgeräte für den Bereich der Elektrotechnik und Automotive.“, *GOEPEL electronic*. [Online]. Verfügbar unter: http://www.goepel.com/index.html. [Zugegriffen: 28-Juli-2015]

[88] „CASCON GALAXY“. [Online]. Verfügbar unter: http://www.goepel.com/en/jtag-boundary-scan/boundary-scan-instruments/software/cascon-galaxy.html. [Zugegriffen: 28-Juli-2015]

[89] T. Wenzel und H. Ehrenberg, „Embedded System Access - Changing the Paradigm of Electrical Test“. [Online]. Verfügbar unter: http://electronics-eetimes.com/documents/whitepapers/12-03/goepel_wp_embedded_system_access_en.pdf. [Zugegriffen: 27-Juli-2012]

[90] A. L. Crouch, *Design for Test - For Digital IC's and Embedded Core Systems*. Upper Saddle River, New Jersey: Prentice Hall, Inc., 2000.

[91] „Lattice Semiconductor“. [Online]. Verfügbar unter: http://latticesemi.com/. [Zugegriffen: 31-Juli-2015]

[92] S. Ostendorff, H.-D. Wuttke, J. Sachße, und J. H. Meza Escobar, „Test Pattern Dependent FPGA Based System Architecture for JTAG Tests“, in *IEEE International Conference on Systems (ICONS)*, Les Menuires, France, 2010, S. 99–104.

[93] „Wishbone Spezifikation B4“. Open Cores, 2010 [Online]. Verfügbar unter: http://cdn.opencores.org/downloads/wbspec_b4.pdf. [Zugegriffen: 10-Juni-2015]

[94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, und W. Lorensen, *Object-Oriented Modeling and Design*. Perentice-Hall, 1991.

[95] A. Mekki, M. Ghazel, und A. Toguyeni, „Validating Time-constrained Systems Using UML Statecharts Patterns and Timed Automata Observers“, in *IEEE International Conference on Verification and Evaluation of Computer and Communication Systems*, Swinton, UK, 2009, S. 112–124.

[96] A. Muth, „SDL-based design of application specific hardware for hard real-time systems“, Dissertation, 2002 [Online]. Verfügbar unter: http://nbn-resolving.de/urn:nbn:de:bvb:91-diss2002052415643

[97] S. A. Edwards, „The challenges of hardware synthesis from C-like languages“, in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2005, S. 66–67 Vol. 1.

[98] G. Genest, R. Chamberlain, und R. Bruce, „Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C“, in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2007, S. 280–286.

[99] D. Ivosevic und V. Sruk, „Automated modeling of custom processors for DCT algorithm“, in *IEEE International Convention on information and communication technology, electronics and microelectronics (MIPRO)*, 2011, S. 762–767.

[100] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, und Z. Zhang, „High-Level Synthesis for FPGAs: From Prototyping to Deployment“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Bd. 30, Nr. 4, S. 473–491, 2011.

[101] R. Alur und D. L. Dill, „A Theory of Timed Automata“, *Theoretical Computer Science*, Bd. 126, S. 183–235, 1994.

[102] R. Alur, „Timed Automata“, *Theoretical Computer Science*, Bd. 126, S. 183–235, 1999.

[103] R. Alur, C. Courcoubetis, und D. Dill, „Model-Checking in Dense Real-time“, *Information and Computation*, Bd. 104, S. 2–34, 1993.

[104] M. Stoelinga, „Representing timed automata with difference bound matrices“, 23-Sep-2002. [Online]. Verfügbar unter: http://wwwhome.cs.utwente.nl/~marielle/talks/dbm.pdf. [Zugegriffen: 15-Aug-2013]

[105] R. Ehlers, D. Fass, M. Gerke, und H.-J. Peter, „Fully Symbolic Timed Model Checking Using Constraint Matrix Diagrams“, in *IEEE Real-Time Systems Symposium (RTSS)*, 2010, S. 360–371.

[106] L. Ridi, J. Torrini, und E. Vicario, „Developing a Scheduler with Difference-Bound Matrices and the Floyd-Warshall Algorithm“, *IEEE Software*, Bd. 29, Nr. 1, S. 76–83, 2012.

[107] S.-W. Lin und P.-A. Hsiung, „Model Checking Prioritized Timed Systems“, *IEEE Transactions on Computers Journal*, Bd. 61, Nr. 6, S. 843–856, 2012.

[108] K. Ernst und M. Turetschek, „Vergleich von Anforderungen und Möglichkeiten der Timingmodellierung eines DUT-Modells zur Ansteuerung eines DDR2 SDRAM“, Studienarbeit, TU Ilmenau, 2014.

[109] T. Sasse, S. Richter, und S. Ostendorff, „Timing Modelling with Difference Bound Matrix“, Wissenschaftliches Arbeitsdokument, TU Ilmenau, 2013.

[110] S. Ostendorff, J.-H. Meza Escobar, H.-D. Wuttke, T. Sasse, und S. Richter, „Modeling timing constraints for automatic generation of embedded test instruments“, in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2014, S. 201–206.

[111] „STMicroelectronics“. [Online]. Verfügbar unter: http://www.st.com/web/en/home.html. [Zugegriffen: 01-Juli-2015]

[112] „256-Kbit serial I^2C bus EEPROM“. [Online]. Verfügbar unter: http://www.st.com/web/en/resource/technical/document/datasheet/CD00001891.pdf. [Zugegriffen: 01-Juli-2015]

[113] T. M. Schwikal, „Untersuchung hierarchischer Timing-Modellierungen zur Ansteuerung eines seriellen Flash“, Bachelorarbeit, Technische Universität Ilmenau, Ilmenau, 2014.

[114] „Terasic - Altera DE2-115 Development and Education Board“. [Online]. Verfügbar unter: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502. [Zugegriffen: 05-Okt-2015]

[115] J. H. Meza Escobar, J. Sachße, S. Ostendorff, und H.-D. Wuttke, „Automatic generation of an FPGA based embedded test system for printed circuit board testing“, in *Latin American Test Workshop (LATW)*, Quito, Ecuador, 2012, S. 1–6.

[116] S. Devadze, A. Jutman, I. Aleksejev, und R. Ubar, „Turning JTAG inside out for fast extended test access“, in *Test Workshop, 2009. LATW '09. 10th Latin American*, 2009, S. 1–6.

[117] L. M. Chuong, S.-K. Lam, und T. Srikanthan, „Area-Time Estimation of Controller for Porting C-Based Functions onto FPGA“, in *IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, 2009, S. 145–151.

[118] D. Kulkarni, W. A. Najjar, R. Rinker, und F. J. Kurdahi, „Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems“, in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, S. 239–247.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ANHANGSVERZEICHNIS

ANHANG A: EIGENE VERÖFFENTLICHUNGEN ZUM DISSERTATIONS-THEMA

2016

Jorge Hernán Meza Escobar, Steffen Ostendorff, Heinz-Dietrich Wuttke: *A configurable test-processor for board-level testing*. IEEE Euromicro Conference on Digital System Design (DSD), Limassol, Cyprus, August 2016.

2014

Steffen Ostendorff, Jorge Hernán Meza Escobar, Heinz-Dietrich Wuttke: *Modeling Timing Constraints for Automatic Generation of Embedded Test Instruments*. IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Warzaw, Poland, April 2014.

2013

Steffen Ostendorff, Jörg Sachße, Heinz-Dietrich Wuttke, Jorge Hernán Meza Escobar, *Adaptive Test System to Improve PCB Testing in the Automotive Industry*. World Congress & Exhibition (SAE), Detroit, Michigan, United States, April 2013.

Jorge Hernán Meza Escobar, Jörg Sachße, Steffen Ostendorff, Heinz-Dietrich Wuttke, *ISA configurability of an FPGA test-processor used for board-level interconnection testing*. IEEE Latin American Test Workshop (LATW), pp. 1-6 , Cordoba, Argentina, April 2013.

2012

Jorge Hernán Meza Escobar, Jörg Sachße, Steffen Ostendorff, Heinz-Dietrich Wuttke, *Automatic generation of an FPGA based embedded test system for printed circuit board testing*. IEEE Latin American Test Workshop (LATW), pp. 75-80, Quito, Ecuador, April 2012.

2011

Jörg Sachße, Steffen Ostendorff, Heinz-Dietrich Wuttke, Jorge Hernán Meza Escobar, *Adaptive Test System to Speed up PCB Testing*. Nordic Test Forum (NTF), Tuusula, Finland, November 2011.

Jörg Sachße, Heinz-Dietrich Wuttke, Steffen Ostendorff, Jorge Hernán Meza Escobar, *Architecture of an adaptive Test System built on FPGA*. Architecture of Computing Systems (ARCS), pp. 86-97, Como, Italy , Februar 2011.

2010

Jörg Sachße, Steffen Ostendorff, Heinz-Dietrich Wuttke, Jorge Hernán Meza Escobar, *A Holistic Approach of an Architecture for Tests of FPGA Based Systems with Boundary Scan*. GMM/ITG-Fachtagung Zuverlaessigkeit und Entwurf (ZuE), Wildbad Kreuth, September 2010.

Jörg Sachße, Steffen Ostendorff, Heinz-Dietrich Wuttke, Jorge Hernán Meza Escobar, *A Holistic Approach of an Architecture for Tests of FPGA Based Systems with Boundary Scan*. GMM-Fachbericht 66, pp. 51-52 , VDE-Verlag GmbH , Berlin, Offenbach, September 2010.

Steffen Ostendorff, Heinz-Dietrich Wuttke, Jörg Sachße, Jorge Hernán Meza Escobar, *Test Pattern Dependent FPGA Based System Architecture for JTAG Tests*. IEEE International Conference on Systems (ICONS), pp.99-104, Les Menuires, France, April 2010.

Steffen Ostendorff, Heinz-Dietrich Wuttke, Jörg Sachße, Steffen Köhler, *A new approach for adaptive failure diagnostics based on emulation test*, Design, Automation and Test in Europe (DATE), pp: 327-330, Dresden, März 2010.

ANHANG B: EIGENE THEMENFREMDE VERÖFFENTLICHUNGEN

2016

Karsten Henke, Tobias Vietzke, Heinz-Dietrich Wuttke, Steffen Ostendorff, *GOLDi – Grid of Online Lab Devices Ilmenau*. International Journal of Online Engineering (iJOE). ISSN: 1861-2121, Vol 12, No 4 (2016) pp. 11-13, Wien, Mai 2016.

2015

Karsten Henke, Tobias Vietzke, Heinz- Dietrich Wuttke, Steffen Ostendorff, *GOLDi – Grid of Online Lab Devices Ilmenau*, exp.at'15 International Conference, São Miguel Island, Azores, Portugal, Juni 2015 2014.

Karsten Henke, Tobias Vietzke, Heinz- Dietrich Wuttke, Steffen Ostendorff, *Safety in Interactive Hybrid Online Labs*. International Journal of Online Engineering (iJOE). ISSN: 1861-2121, Vol 11, No 3 (2015) pp. 62-67, Vienna, Juni 2015.

2014

Karsten Henke, Heinz- Dietrich Wuttke, Tobias Vietzke, Steffen Ostendorff, *Using Interactive Hybrid Online Labs for Rapid Prototyping of Digital Systems*, International Journal of Online Engineering (iJOE). ISSN: 1861-2121, pp. 57-62. , Wien, Oktober 2014.

Karsten Henke, Heinz- Dietrich Wuttke, Tobias Vietzke, Steffen Ostendorff, *Using Interactive Hybrid Online Labs for Rapid Prototyping of Digital Systems*, International Conference on Remote Engineering and Virtual Instrumentation (REV), pp 61-66, Porto, Portugal, Februar 2014.

2013

Thomas Volkert, Tobias Simon, Karsten Henke, Steffen Ostendorff, *IT Infrastructure for Research and Teaching Combining an Online Lab, a UAV and Audio-Visual Communication*. International Journal of Online Engineering (iJOE), Vol 9 pp. 39-44, September 2013.

Karsten Henke, Steffen Ostendorff, Heinz- Dietrich Wuttke, Tobias Vietzke, Christian Lutze, *Fields of Applications for Hybrid Online Labs*. International Journal of Online Engineering (iJOE), Vol 9 (2013) - Special Issue REV2013; pp. 20-30, Wien, Mai 2013.

Karsten Henke, Steffen Ostendorff, Heinz- Dietrich Wuttke, Stephan Simon, *Fields of Applications for Hybrid Online Labs.* International Conference on Remote Engineering and Virtual Instrumentation (REV), Sydney, Australia, Februar 2013.

2011-2012

Karsten Henke, Steffen Ostendorff, Stefan Vogel, Heinz- Dietrich Wuttke, *A Grid Concept for Reliable, Flexible and Robust Remote Engineering Laboratories*. ISSN: 1861-2121, International Journal of Online Engineering (iJOE), Vol 8, pp. 42-49, Dezember 2012.

Karsten Henke, Steffen Ostendorff, Heinz- Dietrich Wuttke, Stefan Vogel, *A Grid Concept for Reliable, Flexible and Robust Remote Engineering Laboratories*. International Conference on Remote Engineering and Virtual Instrumentation (REV), Bilbao, Spain, Juli 2012.

Karsten Henke, Steffen Ostendorff, Heinz- Dietrich Wuttke, *A Flexible and Scalable Infrastructure for Remote Laboratories - Robustness in Remote Engineering Laboratorie*. The Impact of Virtual, Remote and Real Logistics Labs - ImViReLL2012 in: CCIS 282 pp. 13-24, Springer Verlag, DOI: 10.1007/978-3-642-28816-6_2, Bremen, Berlin, Heidelberg, Februar 2012.

Karsten Henke, Steffen Ostendorff, Heinz- Dietrich Wuttke, *Robustness in Remote Engineering Laboratories*. International Conference on Remote Engineering and Virtual Instrumentation (REV), pp. 57-63, Brasov, Juni 2011.

2005-2010

Karsten Henke, Steffen Ostendorff, Thomas Volkert, Andreas Mitschele-Thiel, *A Universal Communication Framework and Navigation Control Software for Mobile Prototyping Platforms*. International Journal of Online Engineering (iJOE) [Online], Vol. 6, Special Issue REV2010, pp. 19 - 24, ISSN: 1868-1646. , Oktober 2010.

Karsten Henke, Steffen Ostendorff, Thomas Volkert, Andreas Mitschele-Thiel: *A Universal Communication Framework and Navigation Control Software for Mobile Prototyping Platforms*. International Conference on Remote Engineering and Virtual Instrumentation (REV), Stockholm, Sweden, Juli 2010.

Karsten Henke, Steffen Ostendorff, Thomas Volkert, Andreas Mitschele-Thiel, *Flying WiFi Robots as Mobile Research Platforms*. International Conference on Computers and Advanced Technology in Education (CATE 2009), pp. 14-21, St. Thomas, US Virgin Islands, November 2009.

Karsten Henke, Steffen Ostendorff, Thomas Volkert, Andreas Mitschele-Thiel, *Mobile Prototyping Platforms for Remote Engineering Applications*. International Journal of Online Engineering (iJOE) [Online], Vol. 5, Special Issue REV2009, 2009, pp. 35-42, ISSN: 1861-2121, September 2009.

Karsten Henke, Thomas Volkert, Steffen Ostendorff, Andreas Mitschele-Thiel, *Mobile Prototyping Platforms for Remote Engineering Applications*. International Conference on Remote Engineering and Virtual Instrumentation (REV), Bridgeport, CT, USA, Juni 2009.

Karsten Henke, Heinz- Dietrich Wuttke, Steffen Ostendorff, *Far Distance Rapid Prototyping of Embedded Systems*. International Conference on Computers and Advanced Technology in Education (CATE), ACTA Press, ISBN: 0-88986-520-5, pp. 83-88, Oranjestad, Aruba, August 2005.

ANHANG C: KOMMUNIKATIONSARCHITEKTUR ZWISCHEN ESW UND HW (WISHBONE-BUS)

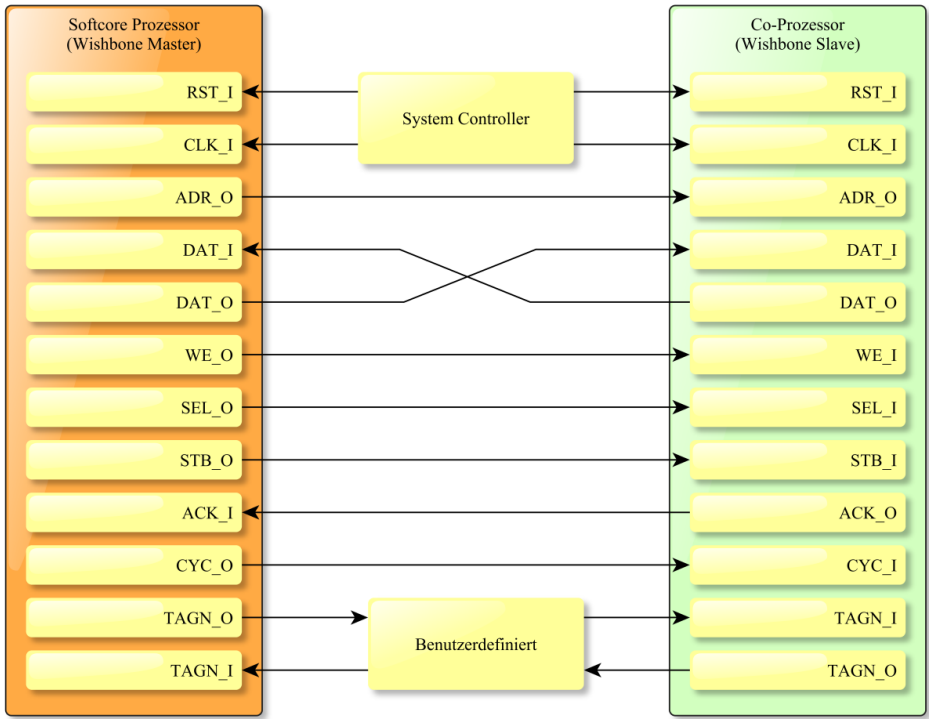


Abbildung 64: Kommunikationsarchitektur zwischen ESW und HW

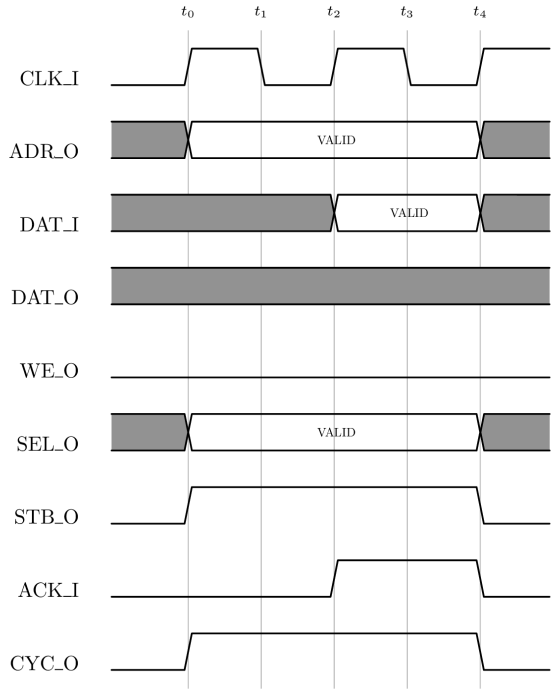


Abbildung 65: Zeitlicher Ablauf eines Lesezugriffs zwischen ESW und HW

ANHANG D: KOMMUNIKATIONSSTRUKTUR ZWISCHEN HW UND HW

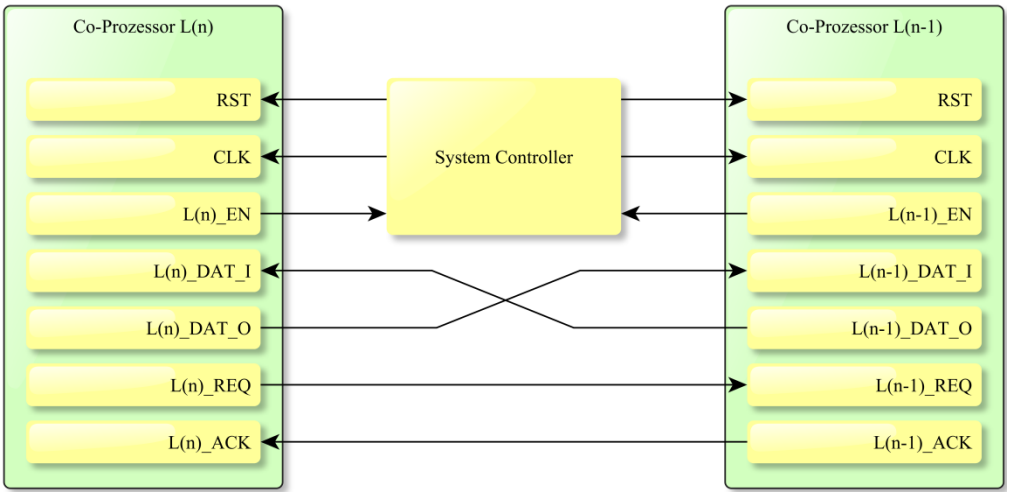


Abbildung 66: Kommunikationsstruktur zwischen HW und HW

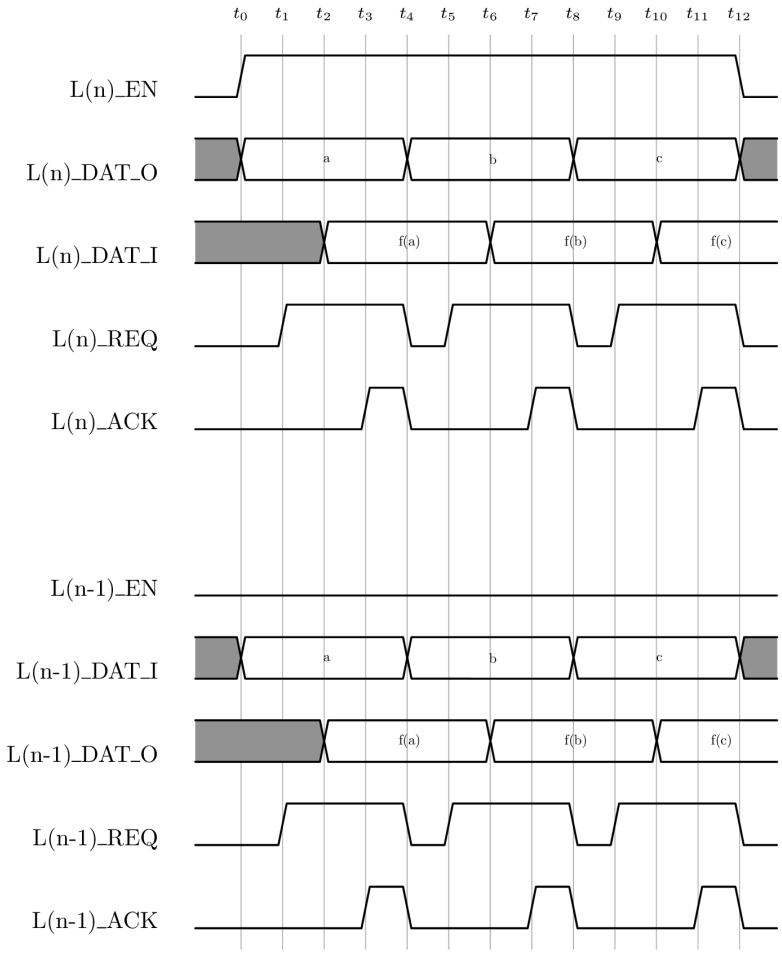


Abbildung 67: Zeitlicher Ablauf der Kommunikation zwischen HW und HW

ANHANG E: ERWEITERTE BACKUS-NAUR-FORM VON RTDL

```
<module> ::= 'MODULE' <ident> <interface_block> <pin_block> <electric_block>
             <sequence_timing_block> <sequence_event_action_block>
             <procedure_block> <program_block> '.'

<ident>  ::= [a-z_]+[a-zA-Z_0-9]*
<int>    ::= [0-9]+
<hex>    ::= '0x'[0-9abcdefABCDEF]+

// ------------------------------ interface block ------------------------------
<interface_block> ::= 'INTERFACE' '(' <data_block> <addr_block> <ctrl_block> ')'
<data-block>      ::= 'DATA' '(' {<ports>} ')'
<addr-block>      ::= 'ADDRESS' '(' {<ports>} ')'
<ctrl-block>      ::= 'CONTROL' '(' {<ports>} ')'

<ports>           ::= <dir> '[' <int> ']' <port_name> '<' <logic_assignment> '>'
                                                  ['{' <ctrl> '}']

<dir>             ::= 'IN' | 'OUT' | 'INOUT'                    // direction of ports
<port_name>       ::= <ident>                                   // port name
<logic>           ::= 'Z' | '0' | '1'                           // valid logic levels
<logic_assignment>::= "'" <logic> {<logic>} "'" | '"' <logic> {<logic>} '"'
                                                                // idle value for port
<ctrl>            ::= '{' <ident> <logic> <logic> '}'           // ctrl settings for signals (optional)

// ------------------------------- pin block -----------------------------------
<pin_block>       ::= 'PIN' '(' <pins> ')'
<pins>            ::= <ident> '[' <int> ']' <ident> ';'

// ---------------------------- electric block ---------------------------------
<electric_block>  ::= <slew_block> <drive_block> <pull_block> <iostandard_block>
<slew_block>      ::= {(<ident> | 'DEFAULT') ('FAST' | 'SLOW') ';'}     // slew rates for pins
<drive_block>     ::= {(<ident> | 'DEFAULT') <int> ';'}                 // driving strength for pins
<pull_block>      ::= {(<ident> | 'DEFAULT') ('PULLUP' | 'PULLDOWN') ';'}  // definition of pullups / pulldowns
<iostandard_block>::= {(<ident> | 'DEFAULT') <stdandard> ';'}           // selection of IO standard
<stdandard>       ::= 'LVCMOS33' | 'LVCMOS25'
```

```
// ---------------------------------- sequence timing block -----------------
<sequence_timing_block> ::= 'SEQUENCE_TIMING' '(' {<ll_timing>} ')'
<ll_timing>             ::= <ident> '[' <time_unit> ']' '(' {<time_def>} ')'
                                                        '(' <time_dbm> ')'
<time_unit>             ::= 'ns'                                          // valid time units
<time_def>              ::= <ident> '=' <int> ';'
<time_dbm>              ::= <dbm_header>
<dbm_header>            ::= '*' <anchor_events> ';'                       // header of difference bound matrix
<anchor_events>         ::= <anchor> {<event>} | <event> <anchor_events>  // at least one anchor and many events
<anchor>                ::= ('_' <event>)                                 // _ marks event as anchor
<event>                 ::= <ident>
<dbm_anchor_row>        ::= <anchor_row> {<row>} | <row> <dbm_anchor_row>
<anchor_row>            ::= <anchor> {<dbm_entry>} ';'                    // if event is an anchor
<row>                   ::= <event> {<dbm_entry>} ';'                     // if event is not an anchor
<dbm_entry>             ::= (['-'] <ident>) | '#' | <int>                 // timing constraint or #

// ---------------------------------- sequence event action block -----------
<sequence_event_action_block> ::= 'SEQUENCE_EVENT_ACTION' '(' {<ll_action>} ')'
<ll_action>             ::= <ident>  '(' <parameter_list> ')' '(' {<action>]} ')'
<parameter_list>        ::= {<trans_parameter>}                           // list can be made of any number of transfer
parameter
<trans_parameter>       ::= <dir> <ident> '[' <int> ']'
<action>                ::= <event> ':' <assignment>                      // assignments at certain events
<assignment>            ::= <ident> ':=' (<ident> | <logic_assignment>) ';'
```

```
// ------------------------------ procedure block ------------------------------
<procedure_block> ::= {<procedure>}                                    // any numbers of procedures
<procedure>       ::= ('L' <int> ':') 'PROCEDURE' <ident>
                      '(' <parameter_list> ')' <proc_block>

<proc_block>      ::= {<var_decl>} <body>
<var_decl>        ::= 'VAR' <variable> {',' <variable>} '[' <int> ']' ';'
<variable>        ::= <ident>
<body>            ::= 'BEGIN' {<statement>} 'END'

<statement>       ::= <assignment> | <report> | <if> | <while> | <proc_call>

<report>          ::= 'REPORT' '(' {<message>} ')'
  <message>       ::= <variable> | <string> | <variable>
    <string>      ::= '"' [^""]* '"'                                   // anything except " (this ends string)
<if>              ::= 'IF' '('<exp>')' {<statement>} 'ENDIF'
                    | 'IF' '('<exp>')' {<statement>} 'ELSE' {<statement>} 'ENDIF'
<while>           ::= 'WHILE' '(' <exp> ')' {<statement>} 'ENDWHILE'

<exp>             ::= <expr> | <relation>
<expr>            ::= <term> | <term> '+' <expr> | <term> '-' <expr>
                    | <term> 'OR' <expr> | <term> 'XOR' <expr>
<term>            ::= <factor> | <factor> '*' <term> | <factor> '/' <term>
                    | <factor> 'SHR' <term> | <factor> 'SHL' <term>
                    | <factor> 'AND' <term>
<factor>          ::= <fac> | '-' <fac> | '~' <fac>
<fac>             ::= <variable> | <constant> | '(' <relation> ')'
                    | '!' '(' <relation> ')' | '(' <expr> ')'
<constant>        ::= <int> | <hex>

<relation>        ::= <expr> <relop> <expr>
<relop>           ::= '==' | '<>' | '!=' | '<' | '>' | '<=' | '>='

<proc_call>       ::= 'CALL' <ident> '(' <parameter> {',' <parameter>} ')'
<parameter>       ::= <variable> '('<variable>')' | <variable> '('<constant>')'

// ------------------------------ program block ------------------------------
<program_block>   ::= 'PROGRAM' <ident> '(' ')' <proc_block>           // content as procedures, but only one allowed
```

Abbildung 68: Erweiterte Backus-Naur-Form von RTDL

ANHANG F: MODELL EINER NETZLISTENBESCHREIBUNG

```
(ADIF_DESCRIPTION
(Name '')                              // ggf. Name der Netzliste und ggf. weitere Beschreibungen

(COMPONENTS                            // beinhaltet alle Komponenten des Prüflings
  (DEV U10
    (TYPE 'IS61LV25616AL_10T'))        // stellt hier das DUT dar
  (DEV U1
    (TYPE 'XC3S1000_4FTG256C'))        // ist der FPGA, der für das testen verwendet werden soll
)

(CONNECTIONS                           // Verbindungen auf der Leiterplatte
  (CLU 'SRAM_ADDR0'                    // CLU definiert sog. Cluster, also Netze in der Netzliste
    (L U10 1)
    (L U1 L5))
  (CLU 'SRAM_ADDR1'                    // Name des Netzes ist 'SRAM_ADDR1'
    (L U10 2)                          // U10 ist mit Pin 2 an diesem Netz verbunden
    (L U1 N3))                         // U1 ist mit Pin 3 an diesem Netz verbunden
  (CLU 'SRAM_ADDR2'
    (L U10 3)
    (L U1 M4))
...                                    // gelöschte Einträge, um die Darstellung zu reduzieren
    (CLU 'SRAM_ADDR15'
    (L U10 42)
    (L U1 K3))
  (CLU 'SRAM_ADDR16'
    (L U10 43)
    (L U1 K5))
  (CLU 'SRAM_ADDR17'                   // Alle Adress-, Daten- und Statusleitungen
    (L U10 44)                         //   des RAMs sind mit dem FPGA verbunden
    (L U1 L3))
  (CLU 'SRAM_DATA0'
    (L U10 7)
    (L U1 N7))
  (CLU 'SRAM_DATA1'
    (L U10 8)
    (L U1 T8))
  (CLU 'SRAM_DATA15'
    (L U10 38)
    (L U1 R1))
...                                    // gelöschte Einträge, um die Darstellung zu reduzieren
  (CLU 'SRAM_OE_N'
    (L U10 41)
    (L U1 K4))
  (CLU 'SRAM_WE_N'
    (L U10 17)
    (L U1 G3))
  (CLU 'SRAM_CE_N'
    (L U10 6)
    (L U1 P7))
  (CLU 'SRAM_LB_N'
    (L U10 39)
    (L U1 P6))
  (CLU 'SRAM_UB_N'
    (L U10 40)
    (L U1 T4))
)
)                                      // Ende der Netzliste
```

Abbildung 69: Gekürzte Netzliste

ANHANG G: FPGA MODELL

```
[VENDOR] = "Xilinx"                    // Hersteller
[DEVICE] = "xc3s1000"                  // Typenbezeichnung
[DEVICE FAMILY] = "Spartan3"           // Familie
[DEVICE SPEED GRADE] = "-4"            // Geschwindigkeitsangabe
[PACKAGE] = "ft256"                    // Gehäuse
[BSCAN] = "2"                          // Anzahl benuzterdefinierte Boundary Scan Register
```

Abbildung 70: Modell eines FPGAs in RTDL

ANHANG H: KOMMENTIERTES BSDL BEISPIEL

Aus Gründen der Übersichtlichkeit ist das folgende Beispiel in Auszügen dargestellt. Für eine vollständige Version der Datei sei auf [69] verwiesen.

Headerbereich mit Kommentaren.

```
-- $ XILINX$RCSfile: xc3s50a_ft256.bsd,v $
-- $ XILINX$Revision: 1.8 $
--
-- BSDL file for device XC3S50A_FT256
```

…

Definition des Namens, Gehäusetyps und Schnittstelle des Schaltkreises.

```
entity XC3S50A_FT256 is

-- Generic Parameter

generic (PHYSICAL_PIN_MAP : string := "FT256");

-- Logical Port Description

port (
    DONE: inout bit;
    GND: linkage bit_vector (1 to 28);
    IO_A10: inout bit; --  PAD26
    IO_A11: inout bit; --  PAD29
    IO_A13: inout bit; --  PAD31
```

…

Definition des Beschreibungsstandards, sowie gesetzter Attribute und des verwendeten Pin-Mappings.

```
use STD_1149_1_2001.all;

-- Component Conformance Statement(s)

attribute COMPONENT_CONFORMANCE of XC3S50A_FT256 : entity is
    "STD_1149_1_2001";

-- Device Package Pin Mappings

attribute PIN_MAP of XC3S50A_FT256 : entity is PHYSICAL_PIN_MAP;

constant FT256: PIN_MAP_STRING:=
    "DONE:T15," &
    "GND:(A1,A16,B7,B11,C3,C14,E5,E12,F2,F6," &
        "G8,G10,G15,H9,J8,K2,K7,K9,L11,L15," &
```

…

Definition der Länge und von Registern der Boundary-Scan-Kette.

```
attribute BOUNDARY_LENGTH of XC3S50A_FT256 : entity is 373;

attribute BOUNDARY_REGISTER of XC3S50A_FT256 : entity is
-- cellnum (type, port, function, safe[, ccell, disval, disrslt])
    "   0 (BC_2, *, controlr, 1)," &
    "   1 (BC_2, IO_D13, output3, X, 0, 1, PULL1)," & --  PAD40
    "   2 (BC_2, IO_D13, input, X)," & --  PAD40
    "   3 (BC_2, *, controlr, 1)," &
    "   4 (BC_2, IO_C13, output3, X, 3, 1, PULL1)," & --  PAD39
    "   5 (BC_2, IO_C13, input, X)," & --  PAD39
    "   6 (BC_2, IPAD38, input, X)," &
    "   7 (BC_2, *, controlr, 1)," &
    "   8 (BC_2, IO_B15, output3, X, 7, 1, PULL1)," & --  PAD37
    "   9 (BC_2, IO_B15, input, X)," & --  PAD37
    "  10 (BC_2, *, controlr, 1)," &
```

…

Definition von Designwarnungen, die vom Testingenieur berücksichtigt werden müssen.

```
attribute DESIGN_WARNING of XC3S50A_FT256 : entity is
    "This is a preliminary BSDL file which has not been verified." &
    "This BSDL file must be modified by the FPGA designer in order to" &
        "reflect post-configuration behavior (if any)." &
    "To avoid losing the current configuration, the PROG_B should be" &
        "kept high.  If the PROG_B pin goes low by any means," &
        "the configuration will be cleared." &
    "PROG_B can only be captured, not updated." &
```

…

Ende der Datei.

```
end XC3S50A_FT256;
```

Abbildung 71: BSDL Beispiel eines FPGAs

ANHANG I: BSDL UND SVF VS. ICL UND PDL

Im folgenden Beispiel wird ein einfaches über JTAG angeschlossenes Testinstrument, in diesem Fall ein Temperatursensor, einmal über BSDL und SVF angesprochen und einmal über ICL und PDL. Diese Gegenüberstellung soll den Vorteil von ICL und PDL verdeutlichen und ist [73] entnommen.

Während Abbildung 72 die schematische Darstellung der Verbindung des Testinstruments zu JTAG zeigt, ist in Abbildung 73 eine partielle Beschreibung der für eine Ansteuerung relevanten Boundary-Scan-Strukturen dargestellt. Hierbei wird insbesondere auf das im Attribute *REGISTER_ACCESS* dargestellte Register *INSTR[4]* hingewiesen, welches für den Datenaustausch genutzt wird und dessen genauer Aufbau in SVF bekannt sein muss. Ein entsprechendes Beispielprogramm ist in Abbildung 74 dargestellt.

Das gleiche Beispieldesign wurde in Abbildung 75 mit Hilfe von ICL beschrieben. Die Darstellung in ICL ist deutlich strukturierter als in BSDL. Die Beschreibung beinhaltet drei Module, die das Interface des Testinstruments beschreiben (Zeile 1-4), die Verbindungen des Testinstruments zu Boundary-Scan (Zeile 6-17), sowie die Instanziierung des Moduls auf der sogenannten Chip Level Ebene (Zeile 19-34). In diesem letzten Abschnitt erfolgt das Verbinden der vorher definierten Module (Zeile 20-26) zu JTAG (Zeile 27-33).

Abbildung 76 zeigt abschließend den PDL Code, der notwendig ist, um auf das Testinstrument zuzugreifen. Dieses kleine Beispiel zeigt bereits deutlich die gestiegene Abstraktion der PDL Darstellung, bei der gegenüber SVF kein direktes Wissen über den internen Aufbau des Testinstruments mehr nötig ist. So wird an dieser Stelle gänzlich ohne Registerbreiten gearbeitet, wie dies in SVF nötig ist.

Auch wenn ICL und PDL bei diesem kleinen Beispiel eine zeilenmäßig größere Beschreibung benötigen, so wird doch deutlich, dass der Code deutlich besser wartbar und einfacher zu benutzen ist als eine vergleichbare Beschreibung in BSDL und SVF. Für ein ausführlicheres Beispiel sei an dieser Stelle auf [73] verwiesen.

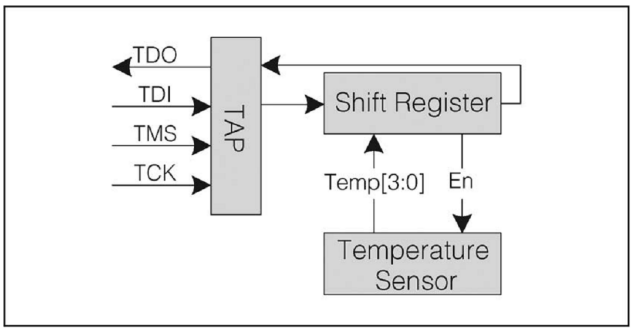


Abbildung 72: Über JTAG angeschlossener Temperatursensor [73]

```
01    entity Chip_BSDL is
02          generic (PHYSICAL_PIN_MAP : string := "DIP22_PACKAGE");
03
04          port (...);
05
06          use STD_1149_1_1990.all;
07
08          attribute PIN_MAP of Chip_BSDL : entity is PHYSICAL_PIN_MAP;
09          constant DIP22_PACKAGE : PIN_MAP_STRING :=      "...";
10
11          attribute TAP_SCAN_IN    of TDI : signal is true;
12          attribute TAP_SCAN_MODE  of TMS : signal is true;
13          attribute TAP_SCAN_OUT   of TDO : signal is true;
14          attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);
15
16          attribute INSTRUCTION_LENGTH of Chip_BSDL : entity is 4;
17          attribute INSTRUCTION_OPCODE of Chip_BSDL : entity is
18                      "BYPASS     (1111, 0000)      , " &
19                      "EXTEST     (0001, 1001)      , " &
20                      "SAMPLE     (0010, 1010)      , " &
21                      "INTEST     (0011, 1011)      , " &
22                      "HIGHZ      (0100, 1100)      , " &
23                      "CLAMP      (0101)      , " &
24                      "READINSTR  (0111, 1000)        ";
25
26          attribute INSTRUCTION_CAPTURE of Chip_BSDL : entity is "0001";
27          attribute INSTRUCTION_DISABLE of Chip_BSDL : entity is "HIGHZ";
28          attribute INSTRUCTION_GUARD   of Chip_BSDL : entity is "CLAMP";
29
30          attribute REGISTER_ACCESS of Chip_BSDL : entity is
31                      "BOUNDARY (EXTEST, INTEST, SAMPLE),"  &
32                      "BYPASS   (BYPASS, HIGHZ, CLAMP)," &
33                      "INSTR[4]   (READINSTR)" ;
34                                      .
35                                      .
36                                      .
37    end Chip_BSDL;
```

Abbildung 73: Partielle BSDL Beschreibung des Beispieldesigns [73]

```
01 SIR 4 TDI (7);                    ! 4-bit IR scan => Loading READINSTR ("0111")
02 SDR 4 TDI (1);                    ! 4-bit DR scan => Activating the sensor ("0001")
03 STATE DRPAUSE;                    ! Going to state DRPAUSE
04 RUNTEST 10 SCK ENDSTATE DRPAUSE;  ! Waiting for 10 system clocks
05 SDR 4 TDI (0);                    ! Shifting out the temperature
06 STATE IDLE;                       ! Going to state Run-Test/Idle
```

Abbildung 74: SVF zum Auslesen des Beispieldesigns [73]

```
01    Module Sensor {
02          DataInPort        en;
03          DataOutPort       temp[3:0];
04          }
05
06    Module TDR {
07          ScanInPort        si;
08          ScanOutPort       so {  Source      SR[0];
09                                  LaunchEdge  Falling; }
10          DataInPort        pi[3:0];
11          DataOutPort       po {  Source SR[0]; }
12
13          SelectPort        en;
14
15          ScanRegister      SR[3:0]{ ScanInSource   si;
16                                     CaptureSource  pi; }
17          }
18
19    Module Chip_ICL {
20          Instance TDR1 Of TDR {
21                InputPort en = Tap1.en_TDR;
22                InputPort pi = Sensor1.temp;
23                }
24          Instance Sensor1 Of Sensor {
25                InputPort en = TDR1.po;
26                }
27          AccessLink Tap1 Of STD_1149_1 {
28                BSDL_Entity Chip_ICL;
29                READINSTR {
30                      ScanPath {  TDR1; }
31                      ActiveSignals { en_TDR; }
32                      }
33                }
34          }
```

Abbildung 75: ICL Beschreibung des Beispieldesigns nach IEEE P1687 [73]

```
01    iPDLLevel 0;
02    iProcsForModule Sensor;
03
04    iProc Read_Temperature{} {
05          iWrite      en    1;
06          iApply;
07
08          iRunLoop    10    -sck;
09          iApply;
10
11          iRead       temp;
12          iWrite      en    0;
13          iApply;
14    }
15
16    iProcsForModule Chip_ICL;
17    iCall Chip_ICL.Sensor1.Read_Temperature();
```

Abbildung 76: PDL Code zum Lesen des Temperatursensors [73]

ANHANG J: KOMMENTIERTES STIL BEISPIEL

Das vorliegende Beispiel ist [74] entnommen. Es stellt ein kurzes, aber vollständiges Beispiel einer STIL Datei dar und beschreibt einen *Octal bus transceiver vom Typ 74LS245*.

Das Beispiel beginnt in Abbildung 77 mit der Definition aller Signale sowie der Zuordnung zu Signalbussen. In Abbildung 78 folgt die Definition von zeitlichen Randbedingungen, jedoch noch ohne Zuweisungen zu bestimmten Ereignissen. Dies folgt im nächsten Abschnitt in Abbildung 79, in denen für die vorher definierten Busse zwei Waveformen beschrieben werden. Im letzten Abschnitt in Abbildung 80 werden abschließend die eigentlichen Testvektoren definiert.

```
STIL 0.0;
Signals {
   DIR In;
   OE_ In;
   A0 InOut; A1 InOut; A2 InOut; A3 InOut;
   A4 InOut; A5 InOut; A6 InOut; A7 InOut;
   B0 InOut; B1 InOut; B2 InOut; B3 InOut;
   B4 InOut; B5 InOut; B6 InOut; B7 InOut;
}

SignalGroups {
     ABUS 'A7 + A6 + A5 + A4 + A3 + A2 + A1 + A0';
     BBUS 'B7 + B6 + B5 + B4 + B3 + B2 + B1 + B0';
     BUSES 'ABUS + BBUS';
     ALL 'DIR + OE_ + BUSES';
}

SignalGroups more {
   ABUS_I 'ABUS' { Base Hex 01; }
   BBUS_I 'BBUS' { Base Hex 01; }
}
```

Abbildung 77: STIL Beispiel Abschnitt 1 [74]

```
Spec tmode_spec {
    Category tmode {
        tplh      { Typ '8.00ns'; Max '12.00ns'; }
        tphl      { Typ '8.00ns'; Max '12.00ns'; }
        tpzl      { Typ '27.00ns'; Max '40.00ns'; }
        tpzh      { Typ '25.00ns'; Max '40.00ns'; }
        tplz      { Typ '15.00ns'; Max '25.00ns'; }
        tphz      { Typ '15.00ns'; Max '25.00ns'; }
        strobe_width '20ns';
    }
}

Selector tmode_typ {
    tplh      Typ;
    tphl      Typ;
    tpzl      Typ;
    tpzh      Typ;
    tplz      Typ;
    tphz      Typ;
}
```

Abbildung 78: STIL Beispiel Abschnitt 2 [74]

```
Timing to_specs {
  WaveformTable pulsed_oe {
    Period '500ns';
    Waveforms {
    DIR{ 01{ '0ns' D/U; }}
    OE_{ 01{ '0ns' U; OE_MARK: '200ns' D/U;
             OE_CLOSE: 'OE_MARK+100ns' U; }}
    BUSES{ 01{ '10ns' D/U; }
        L  { '0ns' Z;'0ns' X; 'OE_MARK+tpzl' l;
             '@+strobe_width' X;}
        H  { '0ns' Z;'0ns' X; 'OE_MARK+tpzh' h;
             '@+strobe_width' X;}
        D  { '0ns' Z;'0ns' X; 'OE_CLOSE+tplz' t;
             '@+strobe_width' X;}
        U  { '0ns' Z;'0ns' X; 'OE_CLOSE+tphz' t;
             '@+strobe_width' X;}
        X  { '0ns' Z;'0ns' X; }}
    }       // end Waveforms
  }         // end WaveformTable pulsed_oe

  WaveformTable const_oe {
    Period '500ns';
    Waveforms {
    DIR{ 01{ '0ns' D/U; }}
    OE_{ 01{ '0ns' D; '480ns' U;}}
    BUSES{ 01{ IN_MARK: '50ns' D/U; }
        L  { '0ns' Z;'0ns' X; 'IN_MARK+tphl' l;
             '@+strobe_width' X;}
        H  { '0ns' Z;'0ns' X; 'IN_MARK+tplh' h;
             '@+strobe_width' X;}
        X  { '0ns' Z;'0ns' X; }}
    }       // end Waveforms
  }         // end WaveformTable const_oe
}           // end Timing to_specs
```

Abbildung 79: STIL Beispiel Abschnitt 3 [74]

```
PatternBurst spec_check_burst {
    spec_check;
}           //end PatternBurst spec_check_burst

PatternExec {
    SignalGroups more;
    Timing to_specs;
    Selector tmode_typ;
    Category tmode;
    spec_check_burst;
}           //end PatternExec


Pattern spec_check {
    W pulsed_oe;
    // the first vector must specify states on all signals.
    V { ALL=00DDDDDDDDXXXXXXXX; }

    // first set of tests check delays from OE_ signal
    // check BBUS tpzl spec
    V { ABUS_I=00; BBUS=LLLLLLLL; }
    //check BBUS tpzh spec
    V { ABUS_I=FF; BBUS=HHHHHHHH; }
    // check BBUS tplz spec
    V { ABUS_I=00; BBUS=DDDDDDDD; }
    // check BBUS tphz spec
    V { ABUS_I=FF; BBUS=UUUUUUUU; }
    V { DIR=1;  ABUS=XXXXXXXX;
    BBUS=DDDDDDDD; }
    // check ABUS tpzl spec
    V { BBUS_I=00; ABUS=LLLLLLLL; }

     // check ABUS tpzh spec
     V { BBUS_I=FF; ABUS=HHHHHHHH; }
     // check ABUS tplz spec
     V { BBUS_I=00; ABUS=DDDDDDDD; }
     // check ABUS tphz spec
     V { BBUS_I=FF; ABUS=UUUUUUUU; }

     W const_oe;
     // second set of tests check data propagation delays
     // check ABUS tphl spec
     V { BBUS_I=00; ABUS=LLLLLLLL; }
     // check ABUS tplh spec
     V { BBUS_I=FF; ABUS=HHHHHHHH; }
     V { DIR=0; BBUS=XXXXXXXX;
     ABUS=DDDDDDDD; }
     V { ABUS_I=00; BBUS=LLLLLLLL; }
     // check BBUS tplh spec
     V { ABUS_I=FF; BBUS=HHHHHHHH; }
     // check BBUS tphl spec
     V { ABUS_I=00; BBUS=LLLLLLLL; }
 }           //end Pattern spec_check
```

Abbildung 80: STIL Beispiel Abschnitt 4 [74]

ANHANG K: DATENQUELLEN FÜR CO-PROZESSOR L1 FÜR IS61LV25616

Im Folgenden sind die verwendeten Datenquellen für den im Anhang L generierten Co-Prozessor dargestellt. Diese sind einer älteren Version des Datenblatts [22] entnommen worden, entsprechen dieser aber qualitativ bis auf einzelne Zeitwerte. Diese Abweichungen haben auf die vorgestellten Ergebnisse keinen Einfluss.

		-8		-10	
Symbol	Parameter	Min.	Max.	Min.	Max.
t_{RC}	Read Cycle Time	8	—	10	—
t_{AA}	Address Access Time	—	8	—	10
t_{OHA}	Output Hold Time	3	—	3	—
t_{ACE}	$\overline{CE}$ Access Time	—	8	—	10
t_{DOE}	$\overline{OE}$ Access Time	—	4	—	5
t_{HZOE}	$\overline{OE}$ to High-Z Output	0	4	—	5
t_{LZOE}	$\overline{OE}$ to Low-Z Output	0	—	0	—
t_{HZCE}	$\overline{CE}$ to High-Z Output	0	4	0	5
t_{LZCE}	$\overline{CE}$ to Low-Z Output	3	—	3	—
t_{BA}	$\overline{LB}$, $\overline{UB}$ Access Time	—	4	—	5
t_{HZB}	$\overline{LB}$, $\overline{UB}$ to High-Z Output	0	4	0	5
t_{LZB}	$\overline{LB}$, $\overline{UB}$ to Low-Z Output	0	—	0	—

Abbildung 81: Zeitliche Parameter für die Read-Befehle (Speedgrade -8 und -10) [22]

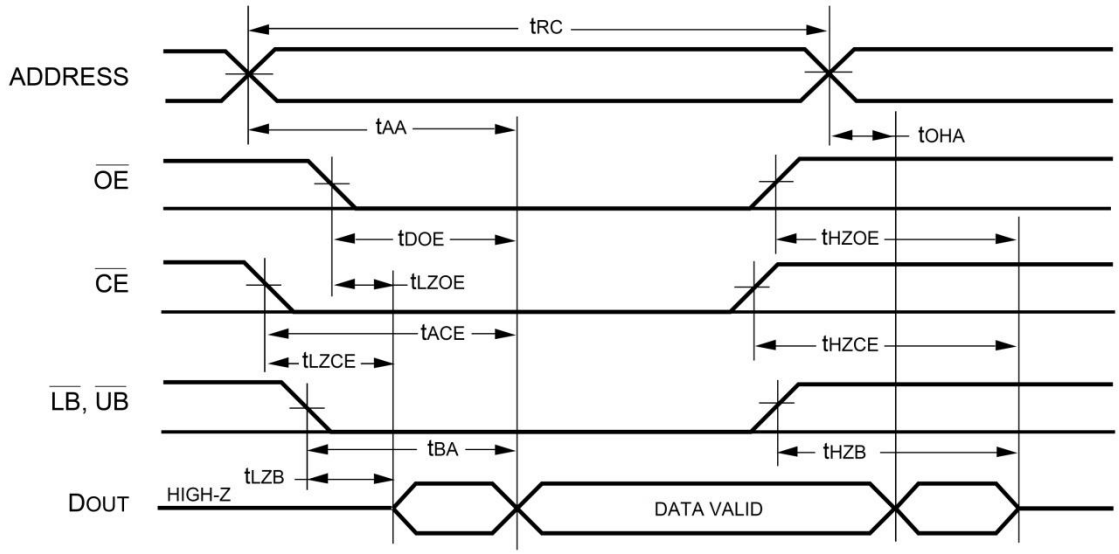


Abbildung 82: Waveform für Read Cycle (nCE und nOE gesteuert) [22]

Symbol	Parameter	-8 Min.	-8 Max.	-10 Min.	-10 Max.
t_{WC}	Write Cycle Time	8	—	10	—
t_{SCE}	$\overline{CE}$ to Write End	7	—	8	—
t_{AW}	Address Setup Time to Write End	7	—	8	—
t_{HA}	Address Hold from Write End	0	—	0	—
t_{SA}	Address Setup Time	0	—	0	—
t_{PWB}	$\overline{LB}$, $\overline{UB}$ Valid to End of Write	7	—	8	—
t_{PWE}	$\overline{WE}$ Pulse Width	7	—	8	—
t_{SD}	Data Setup to Write End	4.5	—	5	—
t_{HD}	Data Hold from Write End	0	—	0	—
t_{HZWE}	$\overline{WE}$ LOW to High-Z Output	—	4	—	5
t_{LZWE}	$\overline{WE}$ HIGH to Low-Z Output	3	—	3	—

Abbildung 83: Zeitliche Parameter für die Write-Befehle (Speedgrade -8 und -10) [22]

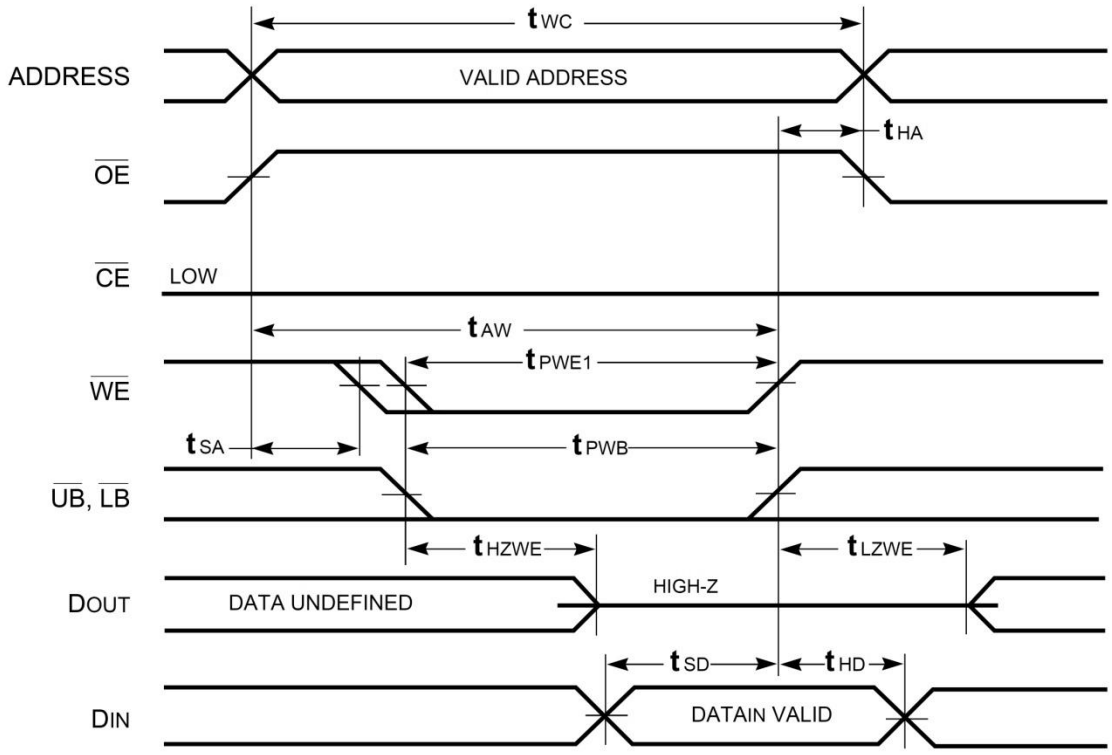


Abbildung 84: Waveform für Write Cycle (nCE gesteuert) [22]

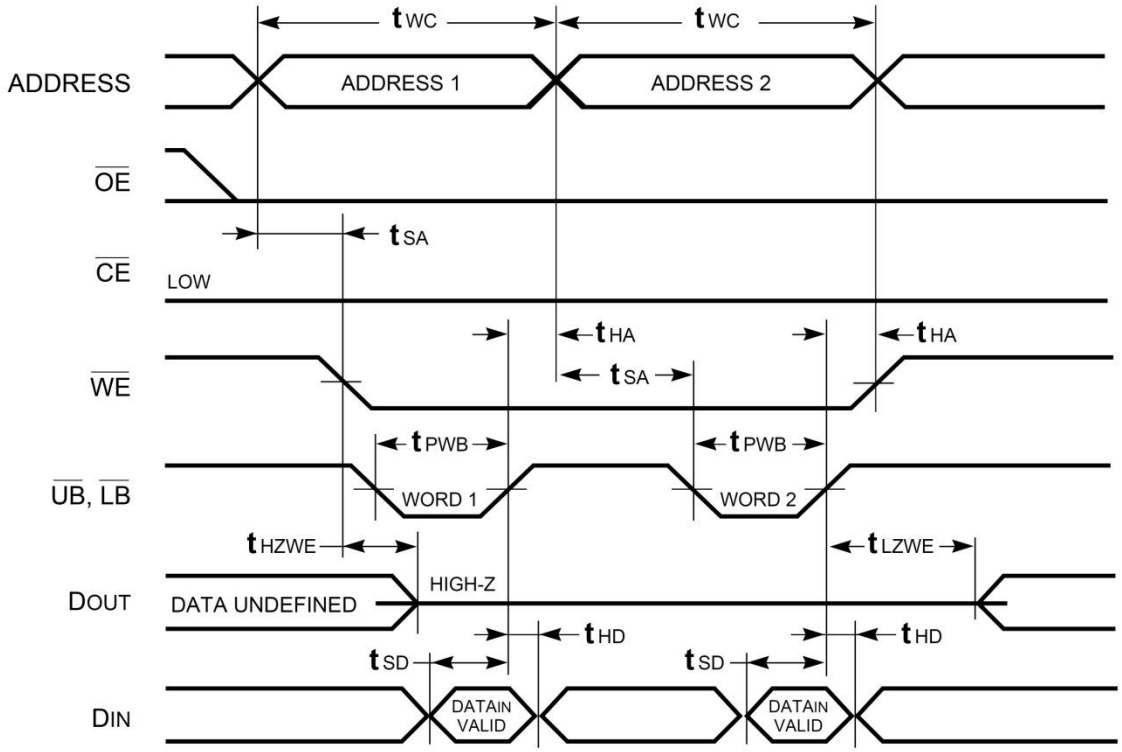


Abbildung 85: Waveform für Back-to-Back-Write Cycle (nUB/nLB gesteuert) [22]

ANHANG L: RTDL MODELL FÜR IS61LV25616 (INTERFACES UND L1)

Im Folgenden sind die relevanten Teile des DUT-M dargestellt, die für eine Generierung eines Co-Prozessors mit der Ebene L1 nötig sind. Dies beinhaltet die Interface Deklaration, sowie die Abbildung der in Anhang K dargestellten Waveforms in Form von DBMs und Zuweisungen. Die Modellierung für die Ebenen L2 bis L5 ist nicht dargestellt. Hierfür wird auf die Quellen in Anhang M verwiesen.

Anmerkung zur Generierung der DBM auf Basis der Daten aus Anhang K:

- Abbildung 82 muss aufgrund von fehlenden Informationen im Datenblatt um einige Randbedingung erweitert werden, so dass die Ereignisse A1, OE1, CE1 und B1 nicht zeitgleich mit DO1 (dem Moment des Lesens der Daten) erfolgen.

```
MODULE IS61LV25616AL

INTERFACE (
   DATA (
      INOUT[16] iO <"0000000000000000"> {oE 1 0};
   )

   ADDRESS (
      IN[18] a <"ZZZZZZZZZZZZZZZZZZ">;
   )

   CONTROL (
      IN[1] cE <'1'> {0};
      IN[1] wE <'1'> {0};
      IN[1] oE <'1'> {0};
      IN[2] nB <"11"> {00};
   )
)
```

```
PIN (
    iO[0] 7;
    iO[1] 8;
    iO[2] 9;
    iO[3] 10;
    iO[4] 13;
    iO[5] 14;
    iO[6] 15;
    iO[7] 16;
    iO[8] 29;
    iO[9] 30;
    iO[10] 31;
    iO[11] 32;
    iO[12] 35;
    iO[13] 36;
    iO[14] 37;
    iO[15] 38;
    a[0] 1;
    a[1] 2;
    a[2] 3;
    a[3] 4;
    a[4] 5;
    a[5] 18;
    a[6] 19;
    a[7] 20;
    a[8] 21;
    a[9] 22;
    a[10] 23;
    a[11] 24;
    a[12] 25;
    a[13] 26;
    a[14] 27;
    a[15] 42;
    a[16] 43;
    a[17] 44;
    cE[0] 6;
    wE[0] 17;
    oE[0] 41;
    nB[0] 39;
    nB[1] 40;
)

ELECTRIC (
    SLEW (
        iO FAST;
        DEFAULT SLOW;
    )
    DRIVE (
        iO 12;
        nB 24;
        DEFAULT 8;
    )
    PULL (
        cE PULLUP;
        wE PULLUP;
        oE PULLUP;
        nB PULLDOWN;
    )
    IOSTANDARD (
        DEFAULT LVCMOS33;
    )
)
```

```
SEQUENCE_TIMING (
   l1_write_CE [ns] (
      tWC =   10 ;
      tAW =    8 ;
      tPWE =   8 ;
      tSA =    0 ;
      tHA =    0 ;
      tPWB =   8 ;
      tHZWE =  5 ;
      tLZWE =  2 ;
      tSD =    6 ;
      tHD =    0 ;
   )
   ( *       _A0  A1  CE0 CE1   WE0    WE1     B0     B1 DO0 DO1    DI0 DI1 ;
     _A0       #   #   #   #      #      #      #      #   #   #      #   # ;
      A1    -tWC   #   #   #      #   -tHA      #   -tHA   #   #      #   # ;
      CE0      #   #   #   #      #      #      #      #   #   #      #   # ;
      CE1      #   #   #   #      #      #      #      #   #   #      #   # ;
      WE0   -tSA   #   #   #      #      #      0      #   #   #      #   # ;
      WE1  -tPWB   #   #   #   -tPWB      #      #      0   #   #   -tSD   # ;
      B0       #   #   #   #      0      #      #      #   #   #      #   # ;
      B1       #   #   #   #      #      0   -tPWB      #   #   #   -tSD   # ;
      DO0      #   #   #   # -tHZWE      # -tHZWE      #   #   #      #   # ;
      DO1      #   #   #   #      # -tLZWE      # -tLZWE   #   #      #   # ;
      DI0      #   #   #   #      #      #      #      #   #   #      #   # ;
      DI1      #   #   #   #      #   -tHD      #   -tHD   #   #      #   # ;
   )

   l1_read_OE [ns] (
      tRC =   10 ;
      tAA =   10 ;
      tOHA =   3 ;
      tDOE =   5 ;
      tLZOE =  0 ;
      tHZOE =  5 ;
      tACE =  10 ;
      tLZCE =  3 ;
      tHZCE =  5 ;
      tLZB =   0 ;
      tBA =    5 ;
      tHZB =   5 ;
   )
   (  *     _A0     A1     OE0     OE1     CE0     CE1     B0     B1 DO0 DO1 DO2 DO3 ;
     _A0      #      #       #       #       #       #      #      #   #   #   #   # ;
      A1   -tRC      #       #       #       #       #      #      #   #  -1   #   # ;
      OE0      #      #       #       #       #       #      #      #   #   #   #   # ;
      OE1      #      #      -1       #       #       #      #      #   #  -1   #   # ;
      CE0      #      #       #       #       #       #      #      #   #   #   #   # ;
      CE1      #      #       #       #      -1       #      #      #   #  -1   #   # ;
      B0       #      #       #       #       #       #      #      #   #   #   #   # ;
      B1       #      #       #       #       #       #     -1      #   #  -1   #   # ;
      DO0      #      # -tLZOE       #  -tLZCE       # -tLZB      #   #   #   #   # ;
      DO1  -tAA      #   -tDOE       #   -tACE       #   -tBA      #   #   #   #   # ;
      DO2      # -tOHA       #       #       #       #      #      #   #   #   #   # ;
      DO3      #      #       # -tHZOE       # -tHZCE       # -tHZB   #   #   #   # ;
   )
```

```
   l1_write_LBUB_back_to_back [ns] (
      tWC =   10 ;
      tSA =    0 ;
      tHA =    0 ;
      tPWB =   8 ;
      tHZWE =  5 ;
      tLZWE =  2 ;
      tSD =    6 ;
      tHD =    0 ;
   )
   (  *    _A0   A1  A2    WE0 WE1    B0   B1    B2     B3 DO0 DO1  DI0 DI1  DI2 DI3 ;
     _A0     #    #   #      #   #     #    #     #      #   #   #    #   #    #   # ;
      A1  -tWC    #   #      #   #     # -tHA     #      #   #   #    #   #    #   # ;
      A2     0 -tWC   #      #   0     #    #     #   -tHA   #   #    #   #    #   # ;
      WE0 -tSA    #   #      #   #     #    #     #      #   #   #    #   #    #   # ;
      WE1    #    #   0      0   #     #    #     #      #   #   #    #   #    #   # ;
      B0  -tSA    #   #      #   #     #    #     #      #   #   #    #   #    #   # ;
      B1     #    #   #      #   # -tPWB    #     #      #   #   # -tSD   #    #   # ;
      B2     # -tHA   #      #   #     0   -1     #      #   #   #    #   #    #   # ;
      B3     #    #   #      #   #     0    0 -tPWB      #   #   #    #   # -tSD   # ;
      DO0    #    #   # -tHZWE   #     #    #     #      #   #   #    #   #    #   # ;
      DO1    #    #   #      #   #     #    #     # -tLZWE   0   #    #   #    #   # ;
      DI0    #    #   #      #   #     #    #     #      #   0   #    #   #    #   # ;
      DI1    #    #   #      #   #     # -tHD     #      #   0   #    0   #    #   # ;
      DI2    #    #   #      #   #     #    #     #      #   0   #    0   0    #   # ;
      DI3    #    #   #      #   #     #    #     #   -tHD   0   #    0   0    0   # ;
   )
)

SEQUENCE_EVENT_ACTION (
   l1_write_CE (IN addr[18], IN data[16]) (
      A0:  a = addr;
      A1:  a = "00000000000000000";
      CE0: cE = "0";
      CE1: cE = "0";
      WE0: wE = "0";
      WE1: wE = "1";
      B0:  nB = "00";
      B1:  nB = "11";
      DI0: iO = data;
   )

   l1_read_oe (IN addr[18], OUT data[16]) (
      A0:  a = addr;
      OE0: oE = "0";
      OE1: oE = "1";
      CE0: cE = "0";
      CE1: cE = "1";
      DO1: data = iO;
   )

   l1_write_lbub_back_to_back (IN addr0[18], IN addr1[18], IN data0[16], IN data1[16]) (
      A0:  a = addr0;
      A1:  a = addr1;
      A2:  a = "00000000000000000";
      WE0:  wE = "0";
      WE1:  wE = "1";
      B0:  nB = "00";
      B1:  nB = "11";
      B2:  nB = "00";
      B3:  nB = "11";
      DI0: iO = data0;
      DI2: iO = data1;
   )
)
```

Abbildung 86: RTDL Modell für IS61LV25616 (Interfaces und L1)

ANHANG M: RTDL MODELL FÜR IS61LV25616 (L2 BIS L5)

```
L2: PROCEDURE l2_apply_pattern(IN wr_rd[2], IN p_data[16], IN p_addr[18], OUT result[16])
VAR dummy[1];
BEGIN
   IF ( (wr_rd AND 1) == 1 )
      CALL awrite(p_data(p_data), p_addr(p_addr));
   ENDIF;
   IF ( (wr_rd AND 2) == 2 )
      CALL aread(p_data(result), p_addr(p_addr));
   ENDIF;
END

L2: PROCEDURE l2_test_sequence_set(IN wr_rd[2], IN ts_set_data[16], IN ts_set_addr[18], IN
ts_set_expect_value[16], OUT l2_error[2])
VAR p_data[16];
VAR p_addr[18];
VAR result[16];
BEGIN
   l2_error := 0;
   p_data := ts_set_data;
   p_addr := ts_set_addr;
   CALL l2_apply_pattern(wr_rd(wr_rd), p_data(p_data), p_addr(p_addr), result(result));
   IF (((wr_rd AND 2) == 2) AND (result != ts_set_expect_value))
      l2_error := 1;
   ENDIF;
END

L2: PROCEDURE l2_single_pattern(IN operation[3], IN data[16], IN address[18], OUT l2_error[2])
VAR   error[2];
VAR   wr_rd[2]; //1-> wr, 2-> rd, 3-> wr+rd
BEGIN
   IF ((operation==0))
      wr_rd := 1; //***single pattern operation WRITE
   ELSE
      IF ((operation==1))
         wr_rd := 2; //***single pattern operation READ
      ELSE
         wr_rd := 3; //***single pattern operation WRITE+READ
      ENDIF;
   ENDIF;
   CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(data), ts_set_addr(address),
   ts_set_expect_value(data), l2_error(error));
   l2_error := error;
END
```

```
L2: PROCEDURE l2_shift_pattern_burst(IN operation[3], IN bus[1], IN data[16], IN
address[18], IN shift_val[18], IN shift_mask[18], OUT l2_error[2])
VAR       error[2];
VAR       wr_rd[1]; //4-> wr+rd (pattern based), 5-> , 6->
BEGIN

   IF (operation==4)                                  //***write and read burst (pattern based)
      wr_rd        := 3;
      WHILE ((error == 0) AND (shift_val > 0))
         IF (bus==0)  //***shift data bus
            CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(shift_val),
            ts_set_addr(address), ts_set_expect_value(shift_val), l2_error(error));
         ELSE
            CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(data),
            ts_set_addr(shift_val), ts_set_expect_value(data), l2_error(error));
         ENDIF;
         shift_val := shift_val SHL 1;
         shift_val := shift_val AND shift_mask;
      ENDWHILE;
   ELSE

      IF(operation==5)                                //***write burst
         wr_rd := 1; //***operation WRITE
         WHILE ((shift_val > 0))
            IF (bus==0)  //***shift data bus
               CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(shift_val),
               ts_set_addr(address), ts_set_expect_value(shift_val), l2_error(error));
            ELSE
               CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(data),
               ts_set_addr(shift_val), ts_set_expect_value(data), l2_error(error));
            ENDIF;
            shift_val := shift_val SHL 1;
            shift_val := shift_val AND shift_mask;
         ENDWHILE;
      ELSE

         IF(operation==6)                             //***read burst
            wr_rd := 2; //***operation read
            WHILE ((error==0) AND (shift_val > 0))
               IF(bus==0)  //***shift data bus
                  CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(shift_val),
                  ts_set_addr(address), ts_set_expect_value(shift_val), l2_error(error));
               ELSE
                  CALL l2_test_sequence_set(wr_rd(wr_rd), ts_set_data(data),
                  ts_set_addr(shift_val), ts_set_expect_value(data), l2_error(error));
               ENDIF;
               shift_val := shift_val SHL 1;
               shift_val := shift_val AND shift_mask;
            ENDWHILE;
         ELSE
            wr_rd := wr_rd;    // some dummy operation needed
                               //***other operations
         ENDIF;
      ENDIF;
   ENDIF;
   l2_error := error;
END
```

```
//***main
L2: PROCEDURE l2_dut_test_primitives(IN operation[3], IN bus[1], IN data[16], IN
address[18], IN shift_val[18], IN shift_mask[18], OUT l2_error[2])
VAR      error[2];
BEGIN
   IF((operation==0))
      //***single pattern operation
      CALL l2_single_pattern(operation(operation), data(data), address(address),
      l2_error(error));
   ELSE
      IF((operation==1))
         //***single pattern operation
         CALL l2_single_pattern(operation(operation), data(data), address(address),
         l2_error(error));
      ELSE
         IF((operation==2))
            //***single pattern operation
            CALL l2_single_pattern(operation(operation), data(data), address(address),
            l2_error(error));
         ELSE
            //***shift pattern burst operation
            CALL l2_shift_pattern_burst(operation(operation), bus(bus), data(data),
            address(address), shift_val(shift_val), shift_mask(shift_mask), l2_error(error));
         ENDIF;
      ENDIF;
   ENDIF;
   l2_error := error;
END

L3: PROCEDURE l3_test_comparator(OUT l3_error[2])
VAR  error[2];
VAR  p_data[16];
VAR  p_address[18];
VAR  p_shift_val[18];
VAR  p_shift_mask[18];
VAR  p_operation[3];
VAR  p_bus[1];
BEGIN
//***data bus test (walking one)***
   //***apply first pattern--> single pattern Data=0, Address=0
   p_operation := 2; //single pattern read+write
   p_bus       := 0; //data bus shifted
   p_shift_val := 0;
   p_shift_mask:= 0;
   p_data      := 0;
   p_address   := 0;
   CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
   address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask), l2_error(error));
   //***apply shifted patterns--> shift pattern burst Data=shl, Address=0
   //***apply shifted patterns and read the others--> Data=shl, Address=0
   IF(error==0)
      p_operation := 4; //shift pattern burst read+write
      p_shift_val := 1;
      p_shift_mask  := 0x0FFFF;//***16 bit data bus
      CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
      address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask), l2_error(error));
   ENDIF;
```

```
   IF(error==0)
//***address bus test (walking one)***
      //***apply single pattern--> Data=0x5A5A, Address=0
      p_operation := 0; //single pattern write
      p_bus       := 1; //address bus shifted
      p_data      := 0x5A5A;
      p_address   := 0;
      CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
      address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask), l2_error(error));

      //***apply shifted patterns--> Data=0x5A5A, Address=shl
      p_operation := 5; //shift pattern burst write
      p_shift_val := 1;
      p_shift_mask:= 0x3FFFF;//***18 bit data bus
      CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
      address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask), l2_error(error));

      //***apply single pattern--> Data=0xA5A5 Address=0, and test the others shl
      //single pattern
      p_operation := 2; //single pattern write+read
      p_data      := 0xA5A5;
      CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
      address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask), l2_error(error));

      //test the other addresses
      IF(error==0)
         p_operation := 6; //shift pattern burst read
         p_data      := 0x5A5A;
         CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
         address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask),
         l2_error(error));

      //***apply single pattern (shifting)-->Data=0xA5A5 Address=shl, and test the others shl
         WHILE((error == 0) AND (p_shift_val > 0) )
            //write+read pattern
            p_operation := 2; //single pattern write+read
            p_data      := 0xA5A5;
            p_address   := p_shift_val;
            CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
            address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask),
            l2_error(error));
            p_shift_val := p_shift_val SHL 1;
            p_shift_val := p_shift_val AND p_shift_mask;

            //read other patterns shl
            IF(error==0)
               p_operation := 6; //shift pattern burst read
               p_data      := 0x5A5A;
               CALL l2_dut_test_primitives(operation(p_operation), bus(p_bus), data(p_data),
               address(p_address), shift_val(p_shift_val), shift_mask(p_shift_mask),
               l2_error(error));
            ENDIF;
         ENDWHILE;
      ENDIF;
```

```
      IF(error==0)
         l3_error := 0;
      ELSE
         l3_error:=3; //***addressBus error
      ENDIF;
   ELSE
      l3_error := 2; //***dataBus error
   ENDIF;
END

L4: PROCEDURE l4_test_analysis(OUT l4_error[2])
VAR error[2];
BEGIN
   CALL l3_test_comparator(l3_error(error));
   l4_error:=error;
END

PROGRAM l5_sram_test()
VAR error[2];
BEGIN
   CALL l4_test_analysis(l4_error(error));
   IF (error == 2)
      REPORT("Data Bus Check failed!");
   ELSE
      IF(error==3)
         REPORT("\nData Bus Check successfully finished!\n");
      ELSE
         REPORT("Data and Address Bus successfully finished!\n");
      ENDIF;
   ENDIF;
END.
```

Abbildung 87: RTDL Modell für IS61LV25616 (L2 BIS L5)

ANHANG N: VERZEICHNISSTRUKTUR UND KONFIGURATION DES COMPILERS

Der RTDL Compiler setzt folgende Verzeichnisstruktur für die Eingaben und Ausgaben voraus:

```
/
├── backup ....Backup fuer automatische Sicherungen per Skript
│
├── compiler ..............................Quellen des Compilers
│   ├── cfg ..................Konfigurationsdateien des Compilers
│   │   ├── _netlist.cfg .....Pfad der zu verarbeitenden Netzliste
│   │   ├── dut_model_wbReg_configFile.cfg . Registerkonfiguration
│   │   │                                    des Co-Prozessors
│   │   └── proc_config_values.txt .........Prozessorkonfiguration
│   ├── doc .................Doxygen Dokumentation des Compilers
│   ├── lib ..............................Bibliotheken fuer RTDL
│   │   ├── _*.rlb ..............  Liste der unterstuetzten DeFT
│   │   │                                    und DUT Modelle
│   │   └── _*.rtdl ....................................DUT Modelle
│   ├── net ..............................Verfuegbare Netzlisten
│   └── src ..............................Quellen des Compilers
│
├── doc ..........Diverse Dokumentation fuer DUTs und anderes
├── software ................Software zur Nutzung des Compilers
│
├── system ............................Generierte Projektdateien
│   ├── cfg ..................................Konfigurationsdateien
│   │   ├── _*.egc ..............Konfiguration fuer GUI (obsolete)
│   │   └── _*.cfg .................Registermap der Co-Prozessoren
│   ├── prc .......................Prozessor bereitgestellt durch
│   │   │                                    Softwarecompiler
│   │   └── src ..............................Quellen des Prozessors
│   ├── prj ................Projektdateien fuer manuelle Synthese
│   ├── sim ..........Simulationsdateien fuer manuelle Simulation
│   └── src ..............Generierte VHDL Dateien des Compilers
```

Abbildung 88: Verzeichnisstruktur RTDL2VHDL Compiler

Hierbei seien besonders folgende Dateien und Verzeichnisse hervorzuheben:

- in \compiler\cfg sind durch den Softwarecompiler die Konfigurationsdateien vorzugeben bzw. müssen hierhin kopiert werden.
- in \system\prc\src müssen die Quellen des Prozessors kopiert werden. Dies ist nur für die manuelle Synthese oder Debugging notwendig.
- in \system\src sind die generierten Quellen des Top Levels und der Co-Prozessoren zu finden.

ANHANG O: EINRICHTEN DES COMPILERS

Der RTDL Compiler führt die Generierung der Co-Prozessoren und des Design Top Levels durch. Die Softwaregenerierung und die Generierung des Prozessors werden vom Softwarecompiler durchgeführt. Die Steuerung der Synthese des Gesamtprojekts erfolgt mit Hilfe eines übergeordneten Tools. Beides ist nicht Teil dieser Arbeit.

Damit der RTDL Compiler die nötigen Schritte ausführen kann, müssen einige Konfigurationen erfolgen, damit Python und PuLP (für die Lineare Optimierung) funktionieren. Diese Konfigurationen sind im Folgenden kurz erläutert und sollten der Reihe nach durchgeführt werden:

- Python 3.4 installieren (hier im Verzeichnis c:\python34)
- Editra Editor installieren
- Editra konfigurieren
    - Codebrowser und Launch aktivieren (Menü Tools → Plugin Manager)
    - Launch konfigurieren
        - File Type: auf python setzen
        - Executables Default: auf python setzen
        - ggf. Alias wie folgt anpassen:
            - pylint: c:\python34\pylint
            - pylinterr: c:\python34\pylint –e
            - python: c:\python34\python –u
- Die Datei ez_setup.py nach c:\python34 kopieren und mit Editra öffnen. Im Menü View → Shelf → Launch starten und anschließend in diesem Fenster mit F5 die Datei ez_setup.py starten. Es sollten sich jetzt die Setup Tools installieren.
- PuLP 1.6.0 entpacken und nach c:\python34 kopieren. (Setup.py sollte somit in c:\python34\PuLP-1.6.0\ liegen.)
- Setup.py aus dem kopierten Verzeichnis in Editra öffnen.
- Im Lauch View bei args: *build* eintragen und mit F5 starten.
- Im Lauch View bei args: *install* eintragen und mit F5 starten.

Falls es keine Fehlermeldungen gibt, sollte die Einrichtung von Python und PuLP damit abgeschlossen sein. Im Verzeichnis c:\python34\PuLP-1.6.0\ sollten unter anderem die Verzeichnisse .\dist und .\build hinzugekommen sein. Die Installation kann durch Öffnen (in Editra) und Ausführen (F5) der Beispieldateien aus c:\python34\PuLP-1.6.0\examples getestet werden. Diese sollten mit dem Exit Code 0 abschließen.

Der RTDL Compiler kann durch Aufruf der Datei RTDL2VHDL_Compiler.py gestartet werden. Diese Datei befindet sich im Unterverzeichnis .\compiler\src\ des RTDL Quellordners.

Alternativ kann auch die Datei RTDL2VHDL.bat aufgerufen werden. Die Ausführung über die Batch-Datei ist um ein vielfaches schneller als im Editra Editor.

ANHANG P: VERWENDETE SOFTWARE

Für die inhaltliche Durchführung der Dissertation sind folgende Programme und Tools verwendet worden:

- Python 3.4.3 als Programmiersprache
- setuptools 18.1 als Toolkit zur Unterstützung von Installationen anderer Toolkits und Pakete in Python
- PuLP 1.6.0 als Toolkit für Lineare Optimierung in Python
- Editra 0.7.20 als Editor für Python
- Notepad++ 6.8.1 als Editor für RTDL, RDM und DIF Modelle
- RTDL Sprachenhighlighting für notepad++ (RTDL.xml, RDM.xml, DIF.xml)
- Xilinx ISE 14.7 als Synthesewerkzeug für VHDL (Xilinx FPGAs)
- Altera Quartus 14.1 als Synthesewerkzeug für VHDL (Altera FPGAs)

Für die Darstellung der Dissertation sind folgende Programme und Tools verwendet worden:

- Word 2010 für die Erstellung des Textdokuments
- Excel 2010 für die Erstellung von Tabellen
- Visio 2010 für die Erstellung von Grafiken
- yEd 3.14.2 für die Erstellung von Grafiken
- TexMaker 4.4.1 zum Editieren von Latex
- MiKTeX 2.9.5105 zum Kompilieren von Latex
- ImageMagick 6.9.1-10 zum Konvertieren von Bildern
- ISE Simulator (Teil der ISE 14.7) für das Erstellen von Waveformausdrucken

ANHANG Q: VARIO-TAP LC-DISPLAY DUT-MODELL UND QUELLDATEN

Im Folgenden sind die relevanten Quelldaten sowie die daraus generierte DB-Matrix des DUT-Modells für den betrachteten Flash dargestellt. Das vollständige Datenblatt ist unter *./compiler/doc/ Ansteuerung des VarioTAP LC-Display.pdf* auf der DVD in Anhang U zu finden.

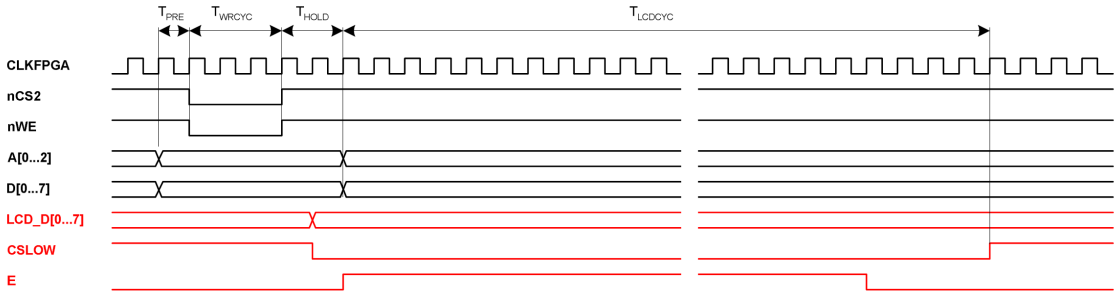


Abbildung 89: Waveform der Vario-TAP LC-Display Ansteuerung

Symbol	Parameter	Min.	Unit
T_{PRE}	Setup time	21	ns
T_{WRCYC}	Bus write cycle time	62.5	ns
T_{HOLD}	Data hold time	0	ns
T_{LCDCYC}	LCD write cycle time	585	ns

Abbildung 90: Zeitliche Parameter der Vario-TAP LC-Display Ansteuerung

Zusätzlich zu den in Abbildung 90 definierten Parametern ist eine zusätzliche Wartezeit vor dem nächsten Befehl von 1000ns notwendig. Diese wurde mit der Zeit T_{LCDCYC} zusammengefasst und als T_{WAIT} bezeichnet.

```
l1_write_pattern [ns] (
   tPRE   =   21 ;
   tWRCYC =   63 ;
   tHOLD  =    0 ;
   tWAIT  = 1585 ;
)
(  *       _A0  A1 D0 D1     wE0    wE1 cS0    cS1 cS2 ;
   _A0      #   #  0  #       #      #   #      #   # ;
   A1       #   #  #  0       # -tHOLD   #      #   # ;
   D0       0   #  #  #       #      #   #      #   # ;
   D1       #   0  #  #       # -tHOLD   #      #   # ;
   wE0 -tPRE    #  #  #       #      #   0      #   # ;
   wE1      #   #  #  # -tWRCYC      #   #      0   # ;
   cS0      #   #  #  #       0      #   #      #   # ;
   cS1      #   #  #  #       #      0   #      #   # ;
   cS2      #   #  #  #       #      #   # -tWAIT   # ;
)
```

Abbildung 91: DBM der Displayansteuerung

```
l1_write_pattern (IN addr[3], IN data[8])
(  A0:  a = addr;
   D0:  iO = data;
   cS0: cS = "0";
   wE0: wE = "0";
   wE1: wE = "1";
   cS1: cS = "1";
   A1: a = "000";
   D1: iO = "ZZZZZZZZ";
)
```

Abbildung 92: Anweisungen aus dem DUT-M, passend zur DBM der Ansteuerung

ANHANG R: FLASH SP29GL064N QUELLDATEN UND DUT-MODELL

Im Folgenden sind die relevanten Quelldaten sowie die das daraus generierte DBMs des DUT-Modells für den betrachteten Flash dargestellt. Das vollständige Datenblatt ist unter *./compiler/doc/S29GL064N.pdf* auf der DVD in Anhang U zu finden.

Command Sequence (Note 1)		Cycles	Bus Cycles (Notes 2–5)											
			First		Second		Third		Fourth		Fifth		Sixth	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)		1	RA	RD										
Reset (Note 7)		1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	4	AAA	AA	555	55	AAA	90	X00	01				
	Device ID (Note 9)	6	AAA	AA	555	55	AAA	90	X02	7E	X1C	(Note 17)	X1E	(Note 17)
	Device ID	4	AAA	AA	555	55	AAA	90	X02	(Note 10)				
	Secured Silicon Sector Factory Protect	4	AAA	AA	555	55	AAA	90	X06					
	Sector Protect Verify (Note 11)	4	AAA	AA	555	55	AAA	90	(SA)X04	00/01				
Enter Secured Silicon Sector Region		3	AAA	AA	555	55	AAA	88						
Exit Secured Silicon Sector Region		4	AAA	AA	555	55	AAA	90	XXX	00				
Program		4	AAA	AA	555	55	AAA	A0	PA	PD				
Write to Buffer (Note 12)		3	AAA	AA	555	55	SA	25	SA	BC	PA	PD	WBL	PD
Program Buffer to Flash		1	SA	29										
Write to Buffer Abort Reset (Note 13)		3	AAA	AA	555	55	AAA	F0						
Chip Erase		6	AAA	AA	555	55	AAA	80	AAA	AA	555	55	AAA	10
Sector Erase		6	AAA	AA	555	55	AAA	80	AAA	AA	555	55	SA	30
Unlock Bypass			AAA	AA	555	55	AAA	20						
Unlock Bypass Program			XXX	A0	PA	PD								
Unlock Bypass RESET			XXX	90	XXX	00								
Program/Erase Suspend (Note 14)		1	XXX	B0										
Program/Erase Resume (Note 15)		1	XXX	30										
CFI Query (Note 16)		1	AA	98										

Abbildung 93: Befehlsübersicht des Flash SP29GL064N

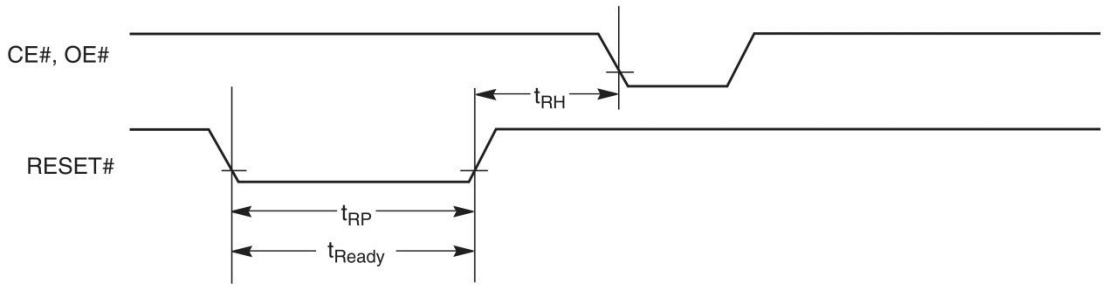


Abbildung 94: Waveform *Reset*-Befehl

Parameter		Description		All Speed Options	Unit
JEDEC	Std.				
	t_{Ready}	RESET# Pin Low (During Embedded Algorithms) to Read Mode (See Note)	Max	20	µs
	t_{Ready}	RESET# Pin Low (NOT During Embedded Algorithms) to Read Mode (See Note)	Max	500	ns
	t_{RP}	RESET# Pulse Width	Min	500	ns
	t_{RH}	Reset High Time Before Read (See Note)	Min	50	ns
	t_{RPD}	RESET# Input Low to Standby Mode (See Note)	Min	20	µs
	t_{RB}	RY/BY# Output High to CE#, OE# pin Low	Min	0	ns

Abbildung 95: Zeitliche Parameter des *Reset*-Befehls

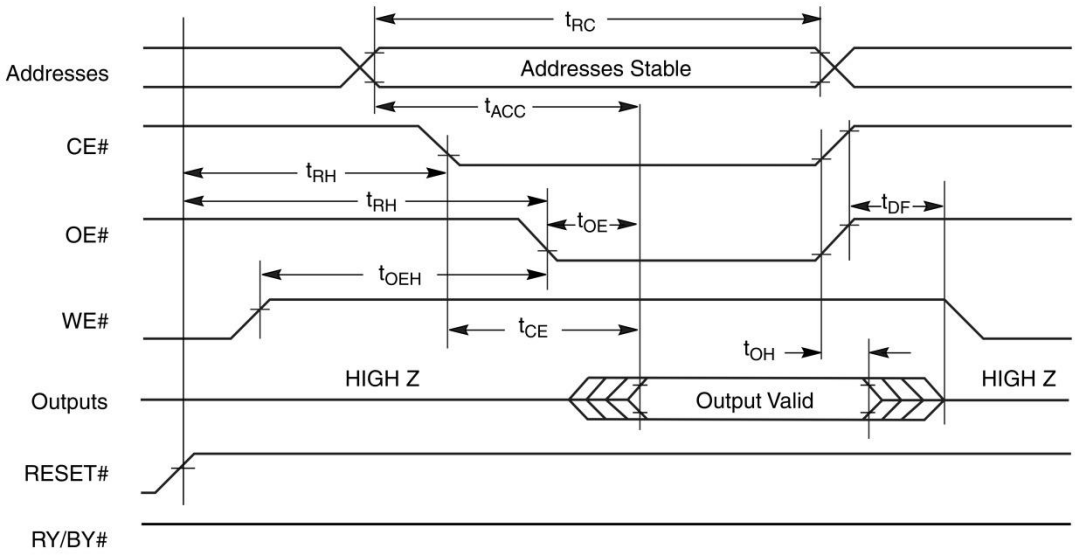


Abbildung 96: Waveform *Read*-Befehl

Parameter		Description		Test Setup		Speed Options		Unit
JEDEC	Std.					90	110	
t_{AVAV}	t_{RC}	Read Cycle Time (Note 1)			Min	90	110	ns
t_{AVQV}	t_{ACC}	Address to Output Delay		CE#, OE# = V_{IL}	Max	90	110	ns
t_{ELQV}	t_{CE}	Chip Enable to Output Delay		OE# = V_{IL}	Max	90	110	ns
	t_{PACC}	Page Access Time		V_{IO} = V_{CC} = 3 V	Max	25	25	ns
				V_{IO} = 1.8 V, V_{CC} = 3 V		—	30	
t_{GLQV}	t_{OE}	Output Enable to Output Delay		V_{IO} = V_{CC} = 3 V	Max	25	25	ns
				V_{IO} = 1.8 V, V_{CC} = 3 V		—	30	
t_{EHQZ}	t_{DF}	Chip Enable to Output High Z (Note 1)			Max	20		ns
t_{GHQZ}	t_{DF}	Output Enable to Output High Z (Note 1)			Max	20		ns
t_{AXQX}	t_{OH}	Output Hold Time From Addresses, CE# or OE#, Whichever Occurs First			Min	0		ns
	t_{OEH}	Output Enable Hold Time (Note 1)	Read		Min	0		ns
			Toggle and Data# Polling		Min	10		ns

Abbildung 97: Zeitliche Parameter des *Read*-Befehls

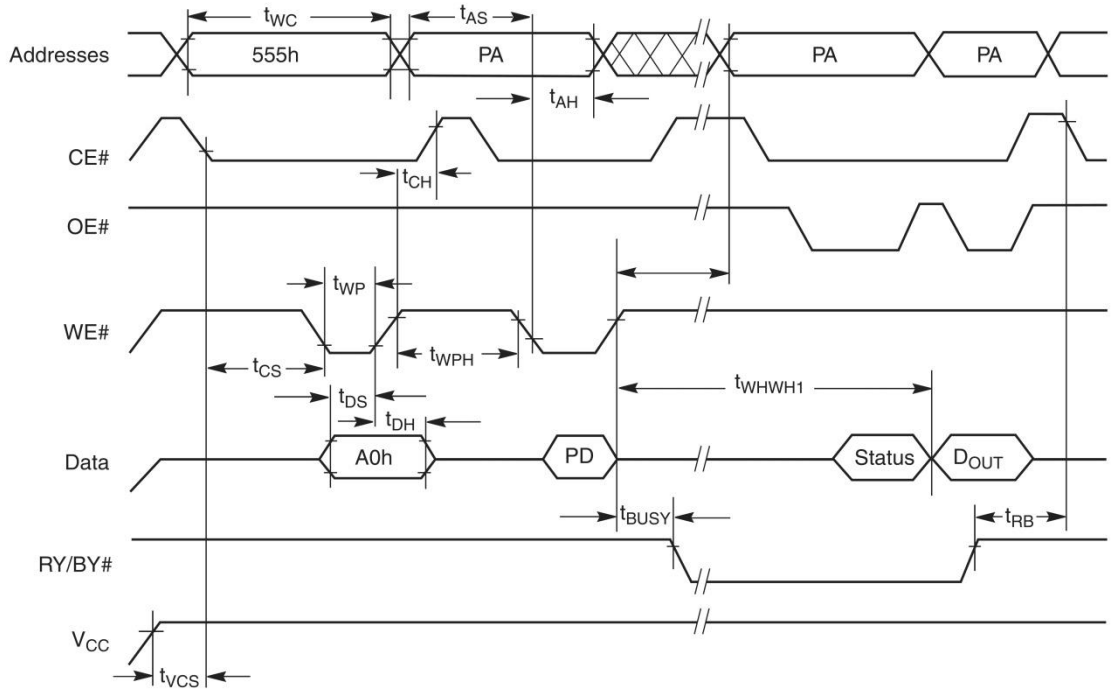


Abbildung 98: Waveform *Program*-Befehl

Parameter		Description		Speed Options		Unit
JEDEC	Std.			90	110	
t_{AVAV}	t_{WC}	Write Cycle Time (Note 1)	Min	90	110	ns
t_{AVWL}	t_{AS}	Address Setup Time	Min	0		ns
	t_{ASO}	Address Setup Time to OE# low during toggle bit polling	Min	15		ns
t_{WLAX}	t_{AH}	Address Hold Time	Min	45		ns
	t_{AHT}	Address Hold Time From CE# or OE# high during toggle bit polling	Min	0		ns
t_{DVWH}	t_{DS}	Data Setup Time	Min	35		ns
t_{WHDX}	t_{DH}	Data Hold Time	Min	0		ns
	t_{CEPH}	CE# High during toggle bit polling	Min	20		ns
	t_{OEPH}	OE# High during toggle bit polling	Min	20		ns
t_{GHWL}	t_{GHWL}	Read Recovery Time Before Write (OE# High to WE# Low)	Min	0		ns
t_{ELWL}	t_{CS}	CE# Setup Time	Min	0		ns
t_{WHEH}	t_{CH}	CE# Hold Time	Min	0		ns
t_{WLWH}	t_{WP}	Write Pulse Width	Min	35		ns
t_{WHDL}	t_{WPH}	Write Pulse Width High	Min	30		ns
t_{WHWH1}	t_{WHWH1}	Write Buffer Program Operation (Notes 2, 3)	Typ	240		µs
		Single Word Program Operation (Note 2)	Typ	60		
		Accelerated Single Word Program Operation (Note 2)	Typ	54		
t_{WHWH2}	t_{WHWH2}	Sector Erase Operation (Note 2)	Typ	0.5		sec
	t_{VHH}	V_{HH} Rise and Fall Time (Note 1)	Min	250		ns
	t_{VCS}	V_{CC} Setup Time (Note 1)	Min	50		µs
	t_{BUSY}	WE# High to RY/BY# Low	Min	90	110	ns

Abbildung 99: Zeitliche Parameter des *Program*-Befehls

```
l1_reset [ns] (
    tRP = 500 ;
    tRH =  50 ;
)
(  *      _rSTn0 rSTn1 oE0 ;
  _rSTn0      #     #    # ;
   rSTn1   -tRP     #    # ;
    oE0        #  -tRH    # ;
)
```

Abbildung 100: DBM des *Reset*-Befehls

```
l1_reset () (
    rSTn0: rSTn = "0" ;
    rSTn1: rSTn = "1" ;
)
```

Abbildung 101: Aktionen des *Reset*-Befehls

```
l1_read [ns] (
    tRC  = 110 ;
    tACC = 110 ;
    tOE  =  30 ;
    tDF  =  20 ;
    tCE  = 110 ;
    tOH  =   0 ;
)
(  *      _a0    a1 cEn0 cEn1 oEn0 oEn1 dQ0  dQ1 ;
  _a0       #     #    #    #    #    #    #    # ;
   a1    -tRC     #    #    #    #    #   -1    # ;
   cEn0     #     #    #    #    #    #    #    # ;
   cEn1     #     0    #    #    #    #    #    # ;
   oEn0     #     #    #    #    #    #    #    # ;
   oEn1     #     0    #    #    #    #    #    # ;
   dQ0 -tACC     # -tCE    # -tOE    #    #    # ;
   dQ1      #     #    # -tDF    # -tDF    #    # ;
)
```

Abbildung 102: DBM des *Read*-Befehls

```
l1_read (IN addr[23], OUT data[8]) (
    a0:     a = addr ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    oEn0: oEn = "0" ;
    oEn1: oEn = "1" ;
    dQ0:  data = dQ ;
)
```

Abbildung 103: Aktionen des *Read*-Befehls

```
l1_programC0[ns] (
    tWC  = 110 ;
    tAS  =   0 ;
    tCH  =   0 ;
    tCS  =   0 ;
    tWP  =  35 ;
    tWPH =  30 ;
    tDS  =  35 ;
    tDH  =   0 ;
)
(  *     _a0   a1 cEn0 cEn1 oEn0 oEn1 wEn0  wEn1 wEn2  dQ0  dQ1 ;
  _a0     #    #    #    #    #    #    #     #    #    #    # ;
   a1   -tWC   #    #    #    #    #    #     #    #    #    # ;
   cEn0    0   #    #    #    #    #    #     #    #    #    # ;
   cEn1    #   #    #    #    #    #    #  -tCH    #    #    # ;
   oEn0    #   #    #    #    #    #    #     #    #    #    # ;
   oEn1    #   #    #    #    #    #    #     #    #    #    # ;
   wEn0 -tAS   # -tCS    #    #    #    #     #    #    #    # ;
   wEn1    #   #    #    #    #    # -tWP     #    # -tDS    # ;
   wEn2    #   #    #    #    #    #    #  -tWPH    #    #    # ;
   dQ0     #   #    #    #    #    #    #     #    #    #    # ;
   dQ1     #   #    #    #    #    #    #  -tDH    #    #    # ;
)
```

Abbildung 104: DBM für *programC0* bis *programC2* des *Program*-Befehls

```
l1_programC0 () (
    a0:     a = "000000000000010101010101" ;   // "555"
    dQ0:   dQ = "10101010" ;                   // "AA"
    dQ1:   dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 105: Aktionen für *programC0* des *Program*-Befehls

```
l1_programC1 () (
    a0:     a = "000000000000001010101010" ;    // "2AA"
    dQ0:   dQ = "01010101" ;                    // "55"
    dQ1:   dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 106: Aktionen für *programC1* des *Program*-Befehls

```
l1_programC2 () (
    a0:     a = "000000000000101010010101" ;    // "555"
    dQ0:   dQ = "10100000" ;                    // "A0"
    dQ1:   dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 107: Aktionen für *programC2* des *Program*-Befehls

```
l1_chip_eraseC2 () (
    a0:     a = "000000000000101010010101" ;    // "555"
    dQ0:   dQ = "10000000" ;                    // "80"
    dQ1:   dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 108: Aktionen für *eraseC2* des *Chip-Erase*-Befehls[76]

[76] Die zeitlichen Randbedingungen entsprechend der Funktion *programC2* des *Program*-Befehls

```
l1_program[ns] (
    tWC   =    110 ;
    tAS   =      0 ;
    tCH   =      0 ;
    tCS   =      0 ;
    tWP   =     35 ;
    tWPH  =     30 ;
    tDS   =     35 ;
    tDH   =      0 ;
    tWHWH = 240000 ;
)
(  *     _a0   a1 cEn0 cEn1 oEn0 oEn1 wEn0  wEn1 wEn2    dQ0  dQ1 ;
  _a0     #    #    #    #    #    #    #     #    #      #    # ;
   a1   -tWC    #    #    #    #    #    #     #    #      #    # ;
   cEn0    0    #    #    #    #    #    #     #    #      #    # ;
   cEn1    #    #    #    #    #    #    #  -tCH    #      #    # ;
   oEn0    #    #    #    #    #    #    #     #    #      #    # ;
   oEn1    #    #    #    #    #    #    #     #    #      #    # ;
   wEn0 -tAS    # -tCS    #    #    #    #     #    #      #    # ;
   wEn1    #    #    #    #    #    # -tWP     #    #   -tDS    # ;
   wEn2    #    #    #    #    #    #    # -tWPH    #      #    # ;
   dQ0     #    #    #    #    #    #    #     #    #      #    # ;
   dQ1     #    #    #    #    #    #    #  -tDH    # -tWHWH    # ;
)
```

Abbildung 109: DBM des vierten Bytes des *Program*-Befehls

```
l1_program (IN addr[23], IN data[8]) (
    a0:    a = addr ;
    dQ0:  dQ = data ;
    dQ1:  dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 110: Aktionen für das vierte Byte des *Program*-Befehls

```
l1_chip_erase[ns] (
    tWC   =       110 ;
    tAS   =         0 ;
    tCH   =         0 ;
    tCS   =         0 ;
    tWP   =        35 ;
    tWPH  =        30 ;
    tDS   =        35 ;
    tDH   =         0 ;
    tWHWH = 500000000 ;
)
(   *     _a0   a1 cEn0 cEn1 oEn0 oEn1 wEn0  wEn1 wEn2    dQ0  dQ1 ;
   _a0     #    #    #    #    #    #    #     #    #      #    # ;
    a1   -tWC    #    #    #    #    #    #     #    #      #    # ;
    cEn0    0    #    #    #    #    #    #     #    #      #    # ;
    cEn1    #    #    #    #    #    #    # -tCH    #      #    # ;
    oEn0    #    #    #    #    #    #    #     #    #      #    # ;
    oEn1    #    #    #    #    #    #    #     #    #      #    # ;
    wEn0 -tAS    # -tCS    #    #    #    #     #    #      #    # ;
    wEn1    #    #    #    #    #    # -tWP     #    #   -tDS    # ;
    wEn2    #    #    #    #    #    #    # -tWPH    #      #    # ;
    dQ0     #    #    #    #    #    #    #     #    #      #    # ;
    dQ1     #    #    #    #    #    #    # -tDH     # -tWHWH    # ;
)
```

Abbildung 111: DBM des sechsten Bytes des *Chip-Erase*-Befehls[77]

```
l1_chip_erase () (
    a0:     a = "000000000000010101010101" ;   // "555"
    dQ0:   dQ = "00010000" ;                   // "10"
    dQ1:   dQ = "ZZZZZZZZ" ;
    cEn0: cEn = "0" ;
    cEn1: cEn = "1" ;
    wEn0: wEn = "0" ;
    wEn1: wEn = "1" ;
)
```

Abbildung 112: Aktionen für das sechste Byte des *Chip-Erase*-Befehls

[77] Die zeitlichen Randbedingungen der ersten fünf Bytes entsprechen der Abbildung 104. Die Reihenfolge der zu sendenden Daten ist Abbildung 93: zu entnehmen. Realisiert wird dies durch das ggf. mehrfache Ausführen der Funktionen *programC0*, *programC1* und *eraseC2*.

ANHANG S: ERGEBNISSE DER ANSTEUERUNG DES FLASHS SP29GL064N

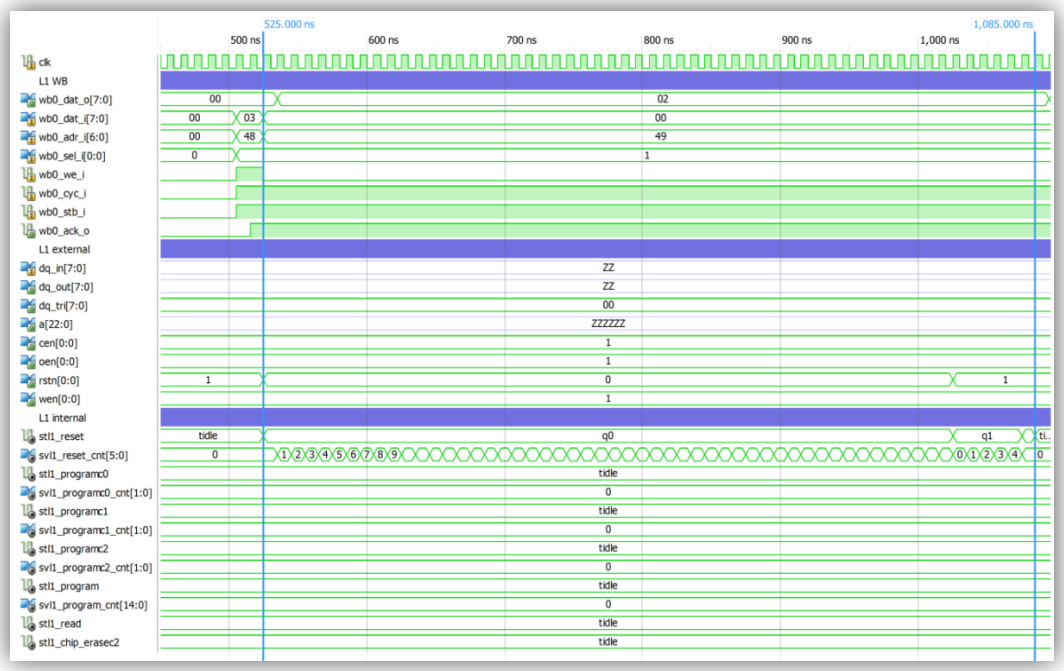


Abbildung 113: Waveform der Simulation S_5 (*Reset*-Funktion)

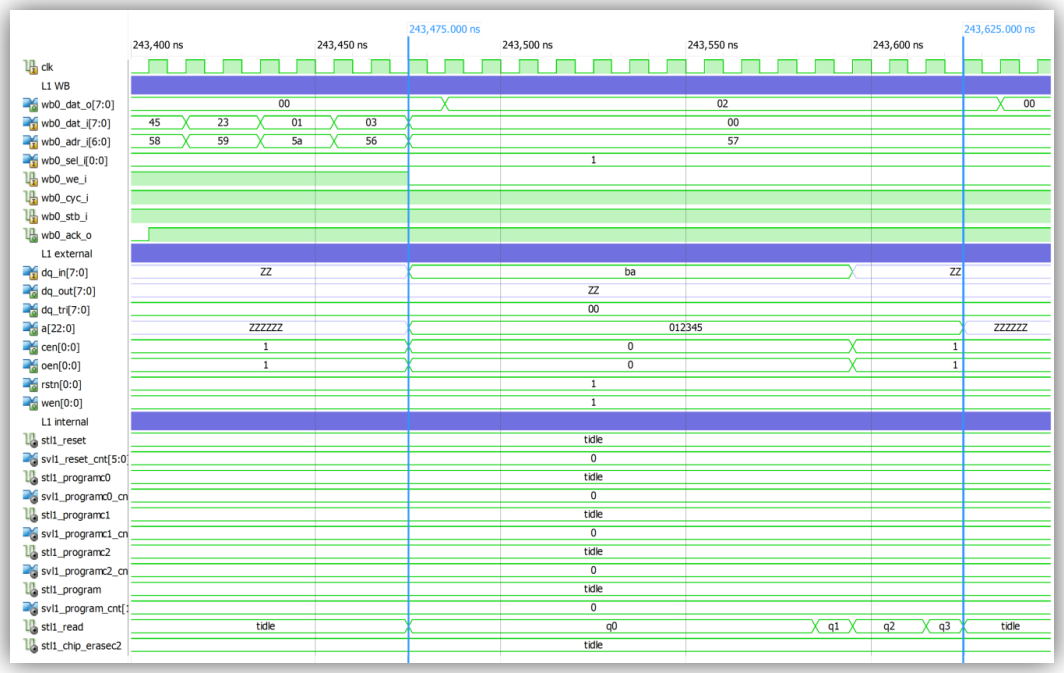


Abbildung 114: Waveform der Simulation S_6 (*Read*-Funktion)

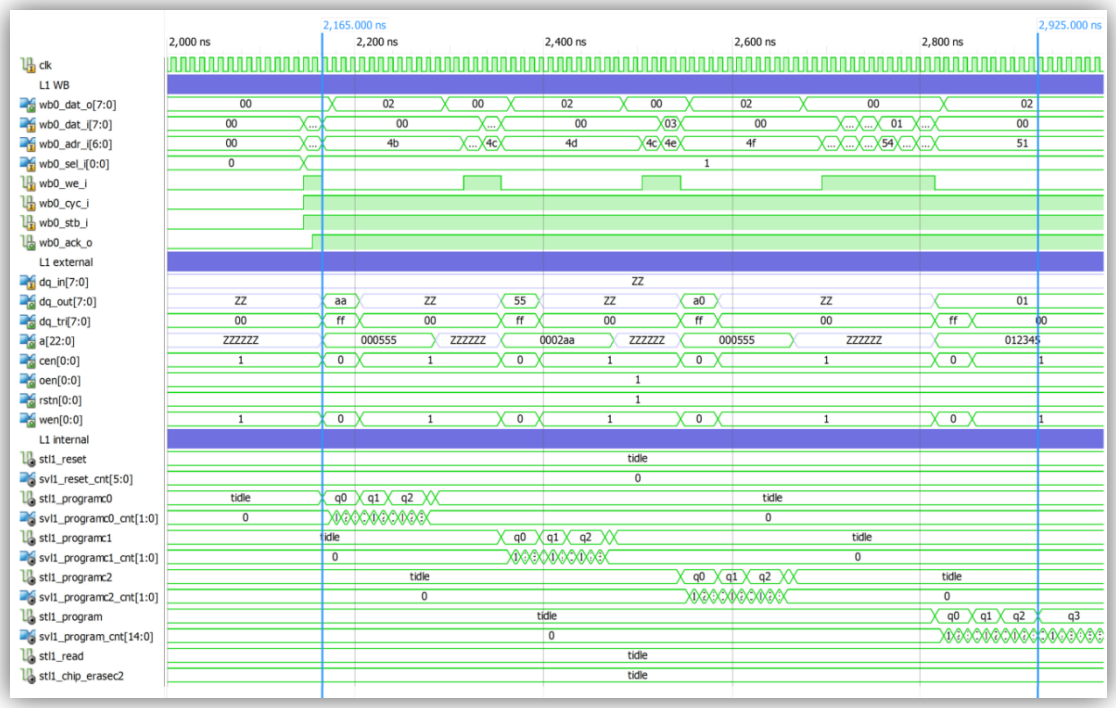


Abbildung 115: Waveform der Simulation S_7 (*Program*-Funktion) (ohne 60us Wartezeit im vorletzten Zustand)

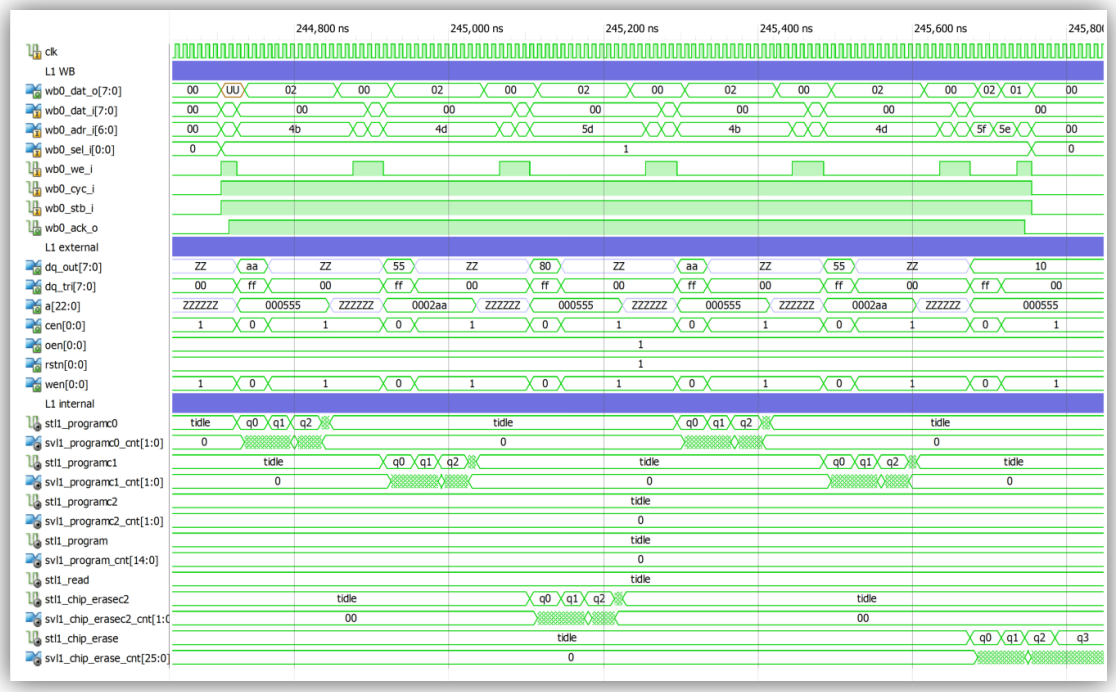


Abbildung 116: Waveform der Simulation S_8 (*Chip-Erase*-Funktion) (ohne 500ms Wartezeit im vorletzten Zustand)

ANHANG T: BERECHNUNGEN FÜR BOUNDARY-SCAN-BASIERTE TESTS

Allgemeines für die Berechnungen

Es wird von einem Boundary-Scan Takt von 10 MHz ausgegangen. Dies entspricht einer mittleren Geschwindigkeit, die sowohl die meisten FPGAs als auch die meisten JTAG Adapter unterstützen. Der verwendete FPGA hat 384 aktive Zellen im Datenregister und die Länge des Instruktionsregisters beträgt 6 Zellen.

Sowohl beim Test P_1 als auch bei P_2 besteht jeder Zugriff auf das DUT aus zwei zu setzenden Mustern innerhalb des Datenregisters. Es wird davon ausgegangen, dass ausgehend vom Zustand Run-Test-Idle innerhalb des TAP-Controllers jeweils abwechselnd ein Datenwort (DR-shift mit 384 Takten) und ein Instruktionswort (IR-shift mit 6 Takten) übertragen werden. Somit ergibt sich für den ganzen Ablauf innerhalb des TAP-Controllers[78] eine Taktanzahl von 405 für ein zu sendendes Muster bzw. 810 für einen abgeschlossenen Zugriff auf das DUT. Dies entspricht im vorliegenden Beispiel einer Zeit von 81µs. Dies entspricht der Ausführungsdauer der Ebene L1.

Berechnung für den Test P_1[79]

Beim Test P_1 müssen neben dem Schreiben auch Daten ausgelesen werden. Das Übertragen der gelesenen Daten über JTAG kann während des folgenden Schreibvorgangs erfolgen. In diesem Fall erhöht sich die Zeit eines vollständigen Datenzugriffs nur um 14 zusätzliche Takte für das Ausführen eines weiteren IR-shifts. Es ergibt sich somit eine Zeit von 82,4µs für einen Lesebefehl auf L1. Der Schreibbefehl bleibt bei den beschriebenen 81µs.

Ein Adressbustest besteht aus einer Initialisierung mit 1+16 Schreibzugriffen für den Datenbus, sowie einem kombinierten Lese- / Schreibzugriff und 20 Lesezugriffen des Adressbusses. Dies ergibt (1+16)*81µs, 1*(81µs+82,4µs) sowie 20*82,4µs also zusammen 3188,4µs für einen vollständigen Adressbustest.

Für den Zugriff auf den Datenbus sind entsprechend 1 kombinierter Lese- / Schreibzugriff für die Initialisierung sowie weitere 16 Lese- / Schreibzugriffe für den eigentlichen Test notwendig. Dies ergibt (1+16)*(81µs+82,4µs) und somit 2777,8µs für den vollständigen Datenbustest. Zusammen also 5966,2µs für beide Tests.

[78] Dies beinhaltet 7 Takte für einen Zyklus von Run-Test-Idle über DR-shift bzw. 8 Takte von Run-Test-Idle über IR-shift und wieder zurück.

[79] Die Berechnungen der auszuführenden Algorithmen für P_1 basieren auf dem Modell in der Datei .\generation_batch\dut_gen\DE2_115\SRAM_detection \dut_model.rtdl

Berechnung für den Test P_2:[80]

Die Ebene L2 besteht aus 8 einzelnen Zugriffen, sowie 64*(8*64) Zugriffen je Displayhälfte. Insgesamt ergibt dies 65.544 Datenwörter für den verwendeten Test. Ben einer Ausführungszeit von 81µs für jeden L1 Zugriff entspricht dies einer gesamten Testzeit von 5,31 Sekunden. Für die Algorithmen auf den Ebenen L3 bis L5 sind keine Übertragungen über Boundary-Scan notwendig.

[80] Die Berechnungen der auszuführenden Algorithmen für P_2 basieren auf dem Modell in der Datei .\generation_batch\dut_gen\DE2_115\LCD_variotap_pattern\dut_model.rtdl

Anhang U: DVD mit sämtlichen Quelldokumenten, Programmen und Ergebnissen

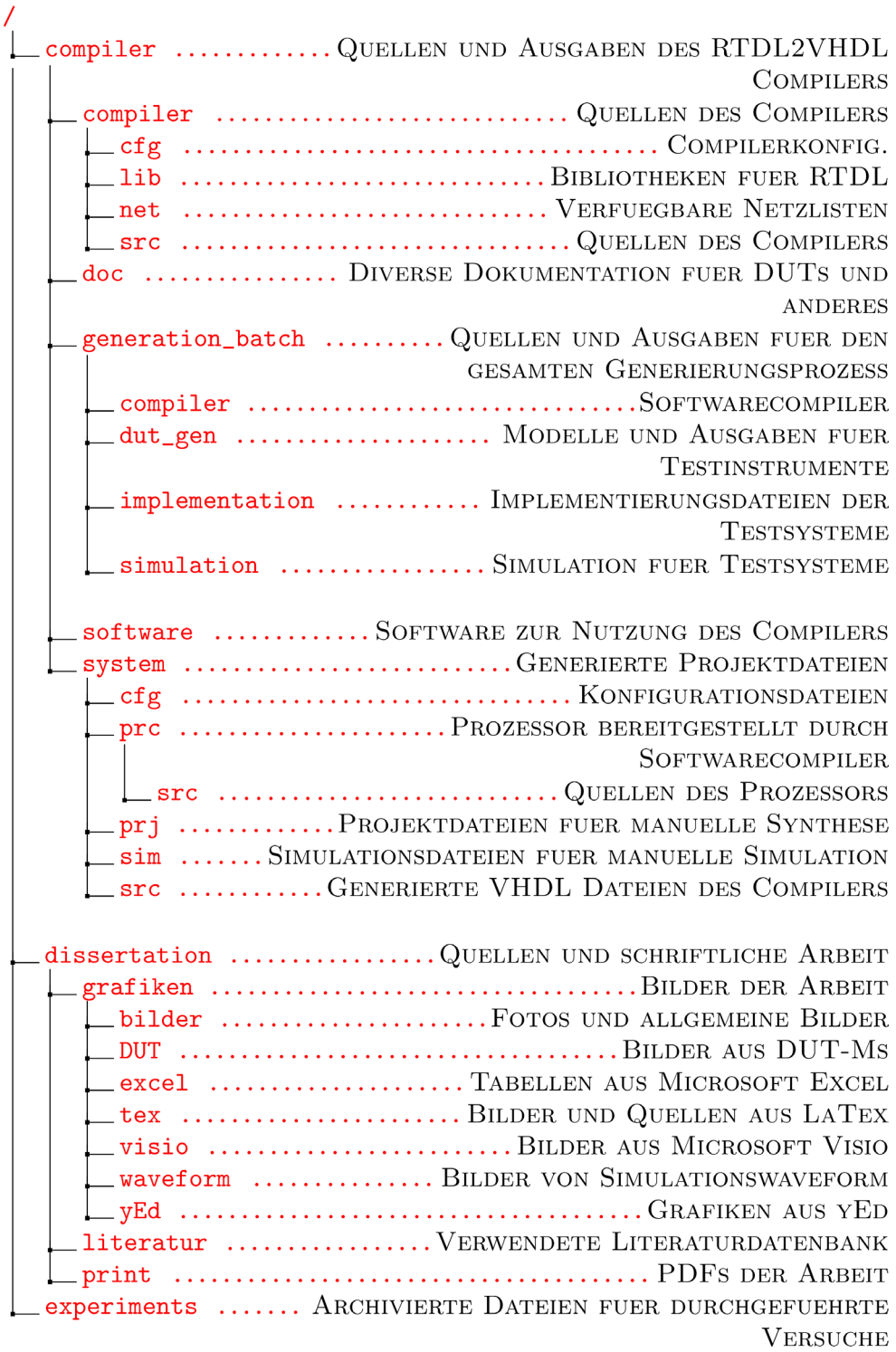


Abbildung 117: DVD Verzeichnisübersicht