Brandenburgische
Technische Universität
Cottbus - Senftenberg

Diagnostischer Test eingebetteter Systeme im Automobil über serielle Standardschnittstellen

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus-Senftenberg

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Christian Gleichner
geboren am 25.08.1980 in Spremberg

Gutachter: Prof. Dr.-Ing. Heinrich T. Vierhaus, BTU Cottbus-Senftenberg

Gutachter: Prof. Dr.-Ing. Sebastian Sattler, FAU Erlangen-Nürnberg

Gutachter: Prof. Dr. Jaan Raik, Tallinn University of Technology

Tag der mündlichen Prüfung: 18.12.2014

Danksagung

Diese Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Technische Informatik der Brandenburgischen Technischen Universität Cottbus-Senftenberg. An erster Stelle möchte ich meinem Doktorvater Professor Vierhaus für seine wissenschaftliche Betreuung während der letzten Jahre danken. Die eingeräumte Freiheit bei Entwicklung und Untersuchung war dabei ein ebenso wichtiger Punkt wie der fachliche Austausch und sein vorhandenes Wissen in den angrenzenden Fachgebieten. Bei Herrn Professor Sattler und Herrn Professor Raik bedanke ich mich für die Begutachtung dieser Dissertationsschrift sowie die daraus hervorgegangenen wertvollen Hinweise.

Die Grundlagen dieser Arbeit entstanden im Rahmen des Forschungsprojekts "Durchgängige Diagnosefähigkeit in Halbleiterbauelementen und übergeordneten Systemen zur Analyse von permanenten und sporadischen Fehlern im Gesamtsystem Automobil" (DIANA) des Bundesministeriums für Bildung und Forschung (BMBF, Förderkennzeichen 16M3188A). Daher möchte ich mich bei den Teilnehmern, insbesondere bei den Projektmitarbeitern der Infineon Technologies AG, für die erfolgreiche Zusammenarbeit und für die gewonnenen Erfahrungen in der anwendungsorientierten Entwicklung bedanken.

Das Gelingen einer solchen Arbeit ist maßgeblich vom kollegialen Umfeld abhängig. Daher gebührt mein besonderer Dank allen Mitarbeitern des Lehrstuhls Technische Informatik der BTU Cottbus-Senftenberg für die tolle Zusammenarbeit und die konstruktiven Diskussionen, die diese Arbeit bereicherten. Darüber hinaus möchte ich Uwe Berger für die Hilfe bei der technischen Umsetzung des für die Untersuchungen erforderlichen Prototypen danken. Außerdem bedanke ich mich bei Tobias Koal, Roberto Urban und Sandro Kollowa für das rasche und dennoch sorgfältige Korrekturlesen.

Allen meinen Freunden und meinem erweiterten Familienkreis bin ich dankbar für die umfangreiche Unterstützung vor und während dieser Arbeit. Vielen Dank auch dafür, dass ihr mich ab und an gedanklich befreien konntet. Meinen Eltern danke ich herzlich für die volle Unterstützung auf meinem bisherigen Bildungs- und Lebensweg.

Von ganzem Herzen danke ich Christin, Leander, Mila und Armon einfach dafür, dass es euch gibt und ihr mein Leben bereichert.

Kurzfassung

Im Automobilbereich nimmt die Komplexität eingebetteter Systeme mit steigenden Ansprüchen bezüglich Qualität und Sicherheit, aber auch aus ökologischen und ökonomischen Aspekten stetig zu. Zum einen wird dies erreicht durch die Anzahl der Steuergeräte und Sensoren und deren Vernetzung, zum anderen durch den rasanten technologischen Fortschritt in der Halbleiterindustrie. Durch die hohe Integrationsdichte und die dadurch erhöhte Sensibilisierung gegenüber potentieller Fehlerquellen während des Produktionsprozesses lassen sich bei der Halbleiterfertigung defekte Chips oder solche mit erhöhter Fehleranfälligkeit nicht vermeiden. Somit werden Ausbeute und Lebensdauer hochintegrierter Schaltungen (IC) reduziert. Da diese vermehrt Einzug in die Automobilindustrie halten, ist es wichtig, einen hohen Qualitätsstandard und, gerade in eingebetteten Prozessoren für sicherheitskritische Anwendungen, eine hohe Fehler- und Ausfallresistenz zu gewährleisten.

In einem solch komplexen elektronischen System wäre es daher wünschenswert einen Fehler möglichst frühzeitig zu erkennen, bevor er zu einer Störung essentieller Funktionen führt. Kommt es zum Teil- oder gar Systemausfall, so ist es darüber hinaus von großer Bedeutung, einen einmal festgestellten Fehler im Nachhinein in der Werkstatt oder als Rückläufer beim Hersteller schnell und eindeutig reproduzieren und diagnostizieren zu können. Die Ursache eines gemeldeten Fehlers in der Fahrzeugelektronik ist aber häufig nicht einwandfrei feststellbar. So besteht im Fehlerfall nur die Möglichkeit, Systemkomponenten anhand der Fehlerbeschreibung auf Verdacht auszutauschen. Eine nachträgliche Fehleranalyse durch den Halbleiterhersteller erfordert hohen Aufwand, da der IC unter anderem erst von der Platine gelöst werden muss.

Im Produktionstest beim Halbleiterhersteller werden hochauflösende strukturorientierte Verfahren angewandt, um fehlerhafte Chips zu identifizieren und auszusortieren. Hierzu werden in den IC eingebrachte Teststrukturen mit separaten Zugangskanälen genutzt, um eine hohe Fehlerüberdeckung in kurzer Testzeit zu garantieren. Der Testzugang zu dieser Produktionstestlogik steht nach dem Packaging und somit nach dem Aufbringen auf die Steuergeräteplatine nicht mehr zu Verfügung.

Die vorliegende Dissertation präsentiert ein Konzept, das einen für die Diagnose eingebetteter Systeme erforderlichen strukturorientierten Test unter Verwendung der schaltungsinternen Produktionstestlogik und serieller Standardschnittstellen realisiert. Somit wird ein Test eines Automotive-ICs mit hoher diagnostischer Auflösung über eine vorhandene Steuergeräteschnittstelle im Zielsystem (Kraftfahrzeug) ohne Demontage des Steuergerätes und der ICs verfügbar gemacht. Der Testzugang setzt dabei die maximale über die genutzte Standardschnittstelle realisierbare Datenrate um. Dem Steuergerätehersteller kann dadurch ein erweiterter Produktionstest bereitgestellt werden, der weit über den Leiterplattentest hinaus eine nachweisbare hohe Prüfschärfe bietet. Kann die Störung einer Systemfunktionalität durch die Diagnosemöglichkeit bereits im Feld, das heißt, während eines Werkstattaufenthalts, eindeutig auf einen fehlerhaften IC zurückgeführt werden, lassen sich wiederholte Fehlersuchen und teure Reparaturen vermeiden. Der Halbleiterhersteller kann zudem die aus der Diagnose eines defekten ICs gewonnenen Informationen nutzen, um während des Fertigungsprozesses und des Fertigungstests entsprechende Maßnahmen zu ergreifen, mit denen die Chipqualität verbessert werden kann.

Neben dem Testzugang umfasst das entworfene Konzept auch einen integrierten Selbsttest, der strukturelle Fehler des ICs bereits vor einer möglichen Auswirkung auf die Funktionalität des Systems identifizieren kann und somit die Systemzuverlässigkeit erhöht.

Abstract

In the automotive industry the complexity of embedded systems increases due to growing demands on quality and safety, but also for economic and environmental reasons. This is achieved by a high amount of electronic control units (ECU), exploiting the rapid advancements of the semiconductor industry. The high level of integration and the increased vulnerability against defects during the semiconductor manufacturing process lead to a higher rate of defective or error-prone chips. This reduces the yield and lifetime of highly integrated circuits (IC). As ICs from advanced nanotechnologies are increasingly being included into automobiles, it is important to maintain a high quality standard and to ensure a high resistance to faults and failures, especially in safety-critical applications.

In such a complex electronic system it would be desirable to detect a fault before it leads to an error or a system failure. In the case of a partial or system failure, it would be very advantageous to have the capability to reproduce and diagnose an identified fault. However, experience has shown that often the actual causes behind errors reported by automotive electronics cannot be identified exactly. This means that in the presence of an error, the repair process relies on replacing various system components until the problem is resolved. In case of malfunction of a control unit, the fault can sometimes be traced back to the responsible faulty IC, but there is no possibility for a precise diagnosis. This complicates the fault analysis done by the semiconductor manufacturer. The subsequent diagnosis of a defective part is associated with great expense and effort, because it may be necessary to remove it from the circuit board.

During production test the semiconductor manufacturer uses fine-grained structure-oriented procedures to identify and discard defective ICs. For this purpose, special test structures are integrated into the chips, which can be accessed through separate test channels, to achieve high fault coverage in short times. After packaging and placing the IC on the printed circuit board, the access to this production test logic is no longer available.

This thesis presents a concept, which implements a structure-oriented test for the diagnosis of embedded systems using the on-chip production test logic in an innovative combination with serial standard-interfaces. Hence, a test of automotive ICs with high diagnostic resolution can be accessed via the interface of an ECU, even in the post-production phase without disassembling the ECU and the ICs. The test access allows the highest data rate according to the interface used. With this test access, the ECU manufacturer is able to implement an extended production test with a provable high testing accuracy, which goes far beyond the printed circuit board test. If an error in the system functionality can be traced back to a defective IC due to the improved diagnostic capability, repair and service cost can be greatly reduced. Moreover, the semiconductor manufacturer can use the collected diagnostic information to take adequate measures during chip production and production test to improve the chip quality.

Beside the test access for error diagnosis, the developed concept also includes a built-in self-test, favourably to be used as system start-up, which can identify a fault in the IC's structure before it affects the system functionality, and therefore increases system reliability.

Inhaltsverzeichnis

1 Kapitel 1
Einleitung

«Das Auto ist jetzt vollkommen. Es bedarf keiner Verbesserung mehr.»
Allgemeine Automobil Zeitung, Berlin, 1921

1.1 Motivation

Die Komplexität eingebetteter Systeme im Automobil nimmt mit steigenden Ansprüchen bezüglich Qualität und Sicherheit, aber auch aus ökologischen und ökonomischen Aspekten stetig zu. Zum einen wird dies erreicht durch die Anzahl der Steuergeräte und Sensoren und deren Vernetzung, zum anderen durch den rasanten technologischen Fortschritt in der Halbleiterindustrie.

Mit Verringerung der Strukturgrößen integrierter Schaltungen kann die Transistoranzahl bezüglich gleicher Chipflächen und somit die Leistung gesteigert werden. Das heißt, es werden kürzere Schaltzeiten bei niedrigerer Versorgungsspannung ermöglicht. Allerdings wird die Herstellung der durch die fortschreitende Miniaturisierung stetig feiner werdenden Chipstrukturen auch immer schwieriger, da die Integrationsdichte an physikalische Grenzen stößt. So steigt bei der Fertigung hochintegrierter Schaltungen (*Integrated Circuit* - IC) die Anfälligkeit gegenüber Ungenauigkeiten und Störeinflüssen wie mikroskopisch kleinste Verunreinigungen. Dies führt neben strukturellen Defekten wie Leitungsunterbrechungen und Kurzschlüsse durch Isolierschichten auch zu statistischen Streuungen, wie beispielsweise die der Dotierungsdichten mit Auswirkung auf die Schwellenspannungen von Transistoren. Durch die hohe Integration und die dadurch erhöhte Sensibilisierung gegenüber potentieller Fehlerquellen während des Produktionsprozesses lassen sich defekte Chips oder solche mit erhöhter Fehleranfälligkeit nicht vermeiden. Somit werden Ausbeute und Lebensdauer der ICs reduziert. Da diese vermehrt Einzug in die Automobilindustrie gehalten haben, ist es wichtig, einen hohen Qualitätsstandard und, gerade in eingebetteten Prozessoren für sicherheitskritische Anwendungen, eine hohe Fehler- und Ausfallresistenz zu gewährleisten.

In der Automobilindustrie sind elektronischen Steuergeräte (*Electronic Control Unit* - ECU) mit eingebetteten Prozessoren schon seit vielen Jahren Stand der Technik. In heutigen Kraftfahrzeugen ist die Elektronik mit durchschnittlich 80 Steuergeräten überaus komplex [IFX14]. In Fahrzeugen der Oberklasse, deren Technik stets mit wenigen Jahren Verzug serienreif in der Mittelklasse zu erwarten ist, sorgen bereits um die hundert ECUs für das entsprechend hohe Qualitäts- und Komfortangebot. In Abbildung 1.1 ist ein komplexes Bordnetz am Beispiel des Audi A8, mit 100 Steuergeräten vernetzt in 7 Bussystemen, dargestellt. Diese Komplexität wird rapide steigen, wenn die Elektronik zukünftig mehr und mehr Basisfunktionen wie Antrieb und Lenkung übernimmt.

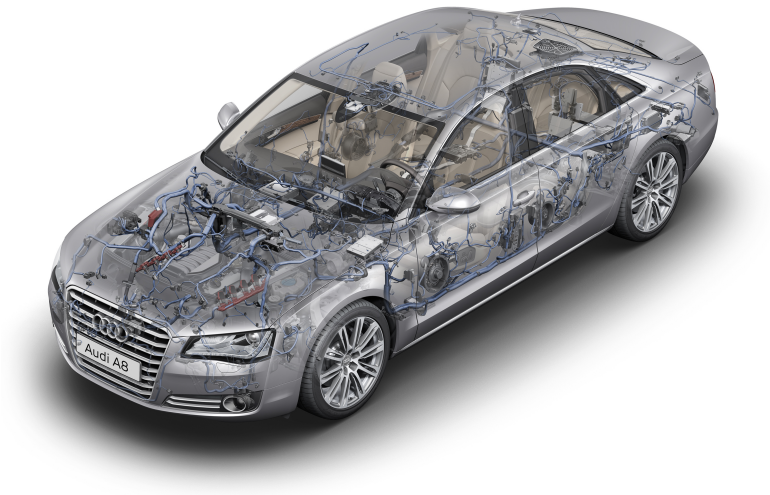


Abbildung 1.1: Bordnetz des Audi A8 [VDI14]

Dabei wird bezüglich der Steuergeräte-ICs aufgrund Qualitätssicherung, aber vor allem aufgrund sicherheitskritischer Anwendungen auf etablierte Technologien zurückgegriffen. Die Strukturgröße der Automotive-Halbleiter näherte sich aber in den letzten Jahren dem ITRS-Technologietrend [ITR11] an, auf Kosten der sogenannten Komfortzone [ALH12], wie in Abbildung 1.2 dargestellt. Das heißt, der Zeitraum in dem sich eine Technologie als ausfallsicher über eine gewisse Laufzeit etabliert hat und allgemein als ausgereift gilt, wird stetig kürzer. Gerade hinsichtlich dieses Trends wird Fehlererkennung und Fehlerdiagnose immer essentieller.

Tritt in einem solch komplexen elektronischen System ein Fehler auf, ist es in erster Linie wichtig diesen zu erkennen, um darauf entsprechend reagieren zu können, sei es durch Warnung an den Nutzer und gegebenenfalls Einschränkung des Funktionsumfangs oder, falls entsprechende Mechanismen vorhanden, durch Selbstreparatur. Kommt es zum Teil- oder gar Systemausfall, ist es darüber hinaus von großer Bedeutung, einen einmal festgestellten Fehler im Nachhinein in der Werkstatt oder als Rückläufer beim Hersteller reproduzieren und diagnostizieren zu können. Erfahrungsgemäß ist die eigent-

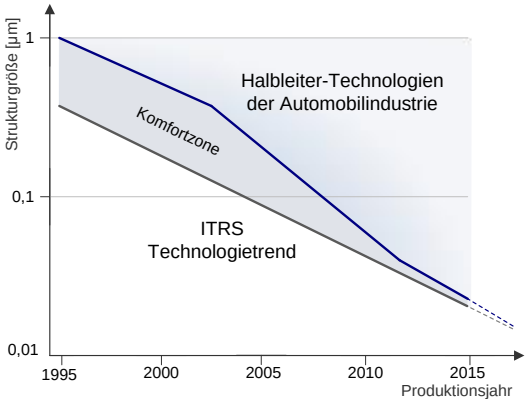


Abbildung 1.2: Technologietrend in der Automobilindustrie

liche Ursache bei bis zu 40 Prozent der gemeldeten Fehler in der Fahrzeugelektronik nicht einwandfrei feststellbar [VSP14].

Häufig bleibt im Fehlerfall nur die Möglichkeit, anhand der Fehlerbeschreibung Systemkomponenten auf Verdacht auszutauschen, was sowohl aus Sicht des Steuergeräteherstellers, des Halbleiterherstellers und vor allem des Automobilherstellers aufgrund eines möglichen Imageschadens höchst unbefriedigend ist.

Es kann im Falle einer Fehlfunktion des Steuergerätes der Fehler, wenn überhaupt, maximal auf den fehlerhaften IC als Verursacher zurückgeführt, nicht aber feingranularer diagnostiziert werden, was natürlich eine anschließende Fehleranalyse durch den Halbleiterhersteller erschwert. Die Diagnose eines defekten Bausteins ist heutzutage mit sehr großem Aufwand verbunden, da dieser möglicherweise erst von der Platine gelöst werden muss. Tritt solch ein Fehler vermehrt innerhalb derselben Baureihe auf, wäre schon ein ungefährer Hinweis auf die interne Fehlerursache durch eine Diagnose, die direkt im Steuergerät stattfindet, wünschenswert. Bezüglich des Gesamtsystems ist eine eindeutige Identifikation der fehlerverursachenden Komponente essentiell für hohe Qualität und Systemzuverlässigkeit.

In der Automobilindustrie werden heutzutage hauptsächlich funktionale Tests angewandt, um mögliche Defekte in Hardware-Komponenten zu erkennen. Diese Tests umfassen aber meist nur Kernfunktionen des Gesamtsystems, ohne auf die spezifische Funktionen der Steuergeräte einzugehen [ACE14]. Das heißt, es wird somit nicht die komplette Funktionalität eingebetteter Systeme abgedeckt. Selbst ein solcher funktionaler Steuergerätetest ist nicht zielführend, da eine strukturelle Fehlerüberdeckung nur schwer nachweisbar ist und typischerweise keine 50 Prozent erreichen würde [MHB00]. Außerdem kann dadurch nur eine *Pass/Fail*-Aussage, also ob ein Fehler vorliegt oder nicht, getroffen werden. Ein strukturorientierter Test hingegen ermöglicht eine Fehlerdiagnose, da erkannte Fehler auf ihre Fehlerorte innerhalb des ICs zurückgeführt werden können.

Ein Testzugang zur Produktionstestlogik über eine Steuergeräteschnittstelle würde diese Diagnosefähigkeit des ICs nachträglich eingebettet im Zielsystem bereitstellen. In dieser Arbeit wird ein Konzept vorgestellt, das ein für die Diagnose eingebetteter Systeme erforderlichen strukturorientierten Test über serielle Standardschnittstellen realisiert.

1.2 Einführung

Um die heutigen komplexen ICs nach der Fertigung effizient testen zu können, werden Teststrukturen in die Architektur eingebracht. Es werden dabei Testzugänge sowie eine Testansteuerung, ein sogenannter *Built-in Self-Test* (BIST), in die Schaltung integriert. Dieser Ansatz des *Design For Testabilty* (DFT) resultiert hauptsächlich aus der Notwendigkeit, einen Zugang zu einem strukturorientierten Test mit hoher Fehlerüberdeckung nach der Fertigung der ICs zur Verfügung zu stellen. Diese BIST-Strukturen werden potenziell auch im Feld für den Selbsttest auf Fehlfunktion des IC-Bausteins, zum Beispiel während des Systemstarts, genutzt. Der Zugang von der Außenwelt zu diesen internen Teststrukturen steht im Zielsystem nicht mehr zur Verfügung.

Im Produktionstest beim Halbleiterhersteller wird die interne Testarchitektur über dedizierte I/O-Kanäle angesteuert. Dabei werden üblicherweise komprimierte Testdaten übertragen, on-chip dekomprimiert und in die Schaltung geladen. Ist die Schaltung durch ein Testmuster komplett initialisiert, wird diese funktional betrieben. Anschließend kann der resultierende Schaltungszustand wieder ausgelesen werden. Anhand der Analyse der Ergebnisdaten können strukturelle Fehler erkannt und auch lokalisiert werden.

Um einen erweiterten Systemtest mit diagnostischen Fähigkeiten im Feld zu realisieren, bietet es sich an, den Zugang zu integrierten Teststrukturen über standardisierte Bussysteme zu ermöglichen. Mittels einer solchen Testschnittstelle können Testroutinen, beispielsweise für das komplette Steuergerät in einem Fahrzeug, durchgeführt werden, welche über den stets unvollständigen funktionalen Test hinaus auch strukturorientierte Tests für integrierte Schaltungen umfassen. Dies ermöglicht also einen Test mit hoher Fehlerüberdeckung auch noch in der Postproduktionsphase.

Der strukturelle Test des eingebetteten Systems verspricht kürzere Fehlersuchzeiten. Im Falle des Automobils bedeutet dies kürzere und, da mit erhöhter Wahrscheinlichkeit direkt auf eine Fehlerursache geschlossen werden kann, die Vermeidung wiederholter Werkstattaufenthalte. Außerdem reduzieren sich der Reparatur- und Servicekosten, wenn durch die Fehlerdiagnose keine fehlerfreien Komponenten mehr ausgetauscht werden.

Die erhöhte Analysetiefe bereits im Zielsystem beschleunigt zudem den Fehlerabstellprozess. Es muss somit im Falle eines fehlerhaften Chips nicht die komplette Lieferkette

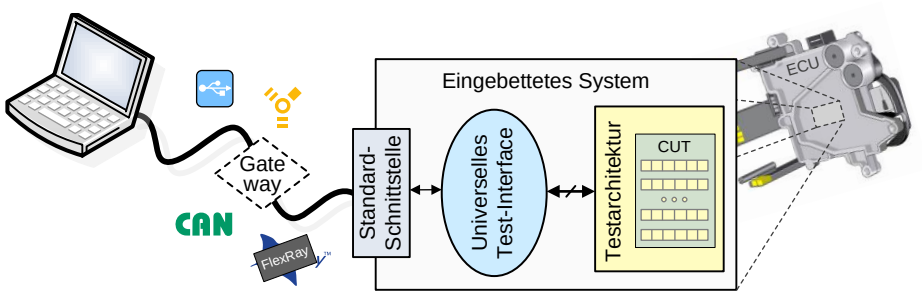


Abbildung 1.3: Diagnostischer Test via Standardschnittstellen

zurückverfolgt werden. Das heißt, kann bei einem Rücklauf der Automobilhersteller die Ursache eines gemeldeten Fehlers nicht auffinden, so muss gegebenenfalls das betroffene Steuergerät, auf das durch die fehlerhafte Funktion zurückgeschlossen wird, dem entsprechenden Zulieferer übergeben werden. Kann dieser wiederum nur auf den IC als fehlerverursachende Komponente schließen, ist der Halbleiterhersteller für eine feingranulare Untersuchung zuständig. Unter Umständen kann auch hier nach aufwendiger Diagnose kein Fehler entdeckt werden, wenn sich dieser nur im Zusammenspiel mit der Peripherie im Steuergerät oder im Betriebszustand des Gesamtsystems auswirkte. Die durch die Diagnose des eingebetteten Systems erhöhte Prüfschärfe erlaubt ein detaillierteres Feedback zum Automobil-, Steuergeräte- oder Halbleiterhersteller. Der Halbleiterhersteller kann zudem die gewonnenen Informationen aus dieser Diagnose nutzen, um während des Fertigungsprozesses und des Fertigungstests entsprechende Maßnahmen zu ergreifen, mit denen die Chipqualität verbessert werden kann [ACE14].

Für die Entwicklung eines seriellen Testzugangs, der diese Postproduktionsdiagnose unterstützt, kommen als Schnittstellen zu etablierten Standardbussystemen USB (Universal Serial Bus) und FireWire/IEEE 1394 in Betracht. In der Automobilelektronik sind andere Bussysteme im Einsatz. Die etablierten Automotive-Bussysteme sind CAN, TT-CAN, FlexRay, LIN, und weitere Hochgeschwindigkeitsbusse, wie MOST, D^2B, Ethernet oder Bluetooth, die aber vor allem multimediale Anwendungen vernetzen.

Steuergeräte sicherheitsrelevanter Anwendungen befinden sich hauptsächlich in den Automobilbordnetzen CAN und FlexRay. So sollte für die zu entwickelnde Testschnittstelle neben USB oder FireWire zumindest auch CAN und FlexRay betrachtet werden. Mittels der Standardschnittstellen wäre es beispielsweise möglich, über ein Diagnosegerät, aber eben auch über ein handelsübliches Endgerät wie PC, Notebook oder Tablet eine eingebettete Schaltung anzusteuern und zu testen (Abbildung 1.3). Dabei könnte die Kommunikation über USB oder FireWire stattfinden, um die Testdaten zum System zu senden und die Ergebnisse wieder auszulesen. Als Schnittstelle zu einem komplexen System kann ein zentrales Steuergerät dienen. In ein Automobilbordnetz integriert, zum Beispiel über ein Gateway, können die Testdaten an die entsprechenden Knoten und deren zu testende Schaltungen weitergeleitet werden.

Das Konzept eines solchen Testzugangs über serielle Standardschnittstellen wird im Kapitel 3 vorgestellt. Hierzu führt das Kapitel 2 die Grundlagen des Tests und der Diagnose integrierter Schaltungen ein. Zudem wird auf verschiedene Standardbussysteme, insbesondere auf USB und FlexRay, eingegangen.

Im Kapitel 4 wird die prototypische Umsetzung des entworfenen Konzepts erläutert. Ferner wird das implementierte Anwendungsprogramm präsentiert, das es ermöglicht, einen Test der auf einem peripheren Gerät verborgenen Schaltung vom PC aus durchzuführen.

Anschließend werden im Kapitel 5 die mittels des entwickelten Testsystems erhaltenen Ergebnisse betrachtet. Das Kapitel 6 gibt eine Zusammenfassung dieser Arbeit wieder.

Kapitel 2

2 Grundlagen & Stand der Technik

Das in dieser Arbeit vorgestellte Konzept umfasst im Allgemeinen den diagnostischen Test eingebetteter Systeme sowie standardisierte Schnittstellen peripherer Geräte als potentielle Testzugänge. Daher wird in diesem Kapitel zunächst auf entsprechende Testverfahren eingegangen. Es werden hierzu unter anderem Fehlermodelle, Testdesigns, Erzeugungs- und Komprimierungsverfahren von Testdaten vorgestellt. Zu diesen Grundlagen werden hier auch State-of-Art-Lösungen betrachtet.

Des Weiteren ist auch die Einführung in Standardschnittstellen Teil dieses Kapitels. Exemplarisch für eine Anwendungsschnittstelle wird der *Universal Serial Bus* herausgegriffen und näher betrachtet. Als Vertreter eines Kommunikationsnetzes im Automobilbereich wird der *FlexRay*-Standard vorgestellt.

2.1 Diagnostischer Test integrierter Schaltungen

2.1.1 Fehler

Bevor auf den Test eingegangen werden kann, muss der Begriff des Fehlers definiert und die für einen Test notwendigen abstrahierten Fehlermodelle erläutert werden.

Ein Fehler liegt in einer integrierten Schaltung vor, wenn sich ein anomaler physikalischer Schaltungszustand, hervorgerufen durch Fertigungsprobleme, Materialermüdung, externe Störungen oder Designfehler, einstellt. Genauer gesagt, bedeutet der Begriff des Fehlers (*Fault*) die Repräsentation des physikalischen Defekts auf abstrahierter Funktionsebene [BA00]. Ein Fehler kann zur Störung (*Error*) führen, falls durch entsprechende Eingangsbelegung die Ausgabe verfälscht wird. Das heißt, es liegt eine Abweichung vom laut Spezifikation erwarteten beziehungsweise ein außerhalb des Toleranzbereichs befindlicher Wert vor. Die Störung ist also schlicht die Auswirkung eines Fehlers. Beeinträchtigt diese Störung die Funktionalität des Systems, wird dies als Ausfall (*Failure*) gedeutet [Pra96].

In der ISO26262 [ISO11], dem aktuellen Standard für sicherheitsrelevante elektrische und/oder elektronische Systeme im Automobilbereich, finden sich äquivalente Definitionen. Die Begriffe bezüglich des Fehlers sind darin wie folgt beschrieben:

Fault Anormaler Zustand, der zum Ausfall eines Bauteils oder einer Einheit führen kann.

Error Abweichung zwischen einem berechneten, beobachteten oder gemessenen Wert oder Zustand und dem richtigen, spezifizierten oder theoretisch korrekten Wert oder Zustand.

Failure Verlust der Fähigkeit eines Bauteils, die erforderliche Funktion auszuführen.

Störungen müssen aber nicht unbedingt durch dauerhafte Schaltungsdefekte hervorgerufen sein. So können Fehler, klassifiziert nach Dauer ihres Auftretens, entweder als permanente Fehler vorliegen oder nur temporär als transiente oder intermittierende Fehler auftauchen. Ein transienter Fehler zeichnet sich dadurch aus, dass dieser zufällig nur eine kurze Zeit vorliegt. Von einem intermittierenden Fehler ist die Rede, wenn dieser sich transient verhält, aber wiederholt auftritt [Pra96]. Transiente und intermittierende Fehler, auch als *Soft Errors* bezeichnet, werden durch Kopplungseffekte, elektromagnetische Interferenzen, Strahlung, Temperaturschwankungen oder Spannungsschwankungen verursacht [URV11]. Der diagnostische Test zielt auf die permanenten Fehler innerhalb einer Schaltung ab. Tritt ein transienter Fehler während des Tests auf, kann dieser ebenso erfasst werden und sogar zur Fehlinterpretation des Testergebnisses führen.

Fehlerdiagnose

Wird in einem System ein Fehler anhand der resultierenden Störung während eines Tests erkannt, so dient eine anschließende Diagnose dazu, diesen auf eine Fehlerursache zurückzuführen. Genauer gesagt, die Aufgabe der Fehlerdiagnose besteht darin, Typ, Ort, Ausmaß und Zeitpunkt des Auftretens des jeweils wahrscheinlichsten Fehlers zu bestimmen [Ise06]. Diagnoseverfahren ermitteln diese Parameter mittels heuristischer Analyse der entdeckten Fehler. Um diagnostizieren zu können, werden Fehlerpunkte bezüglich des zugrundeliegenden Fehlermodells durch entsprechende Testeingaben angeregt, um so mögliche Fehler zu den Testausgängen zu propagieren. Hierzu ist ein Satz an Testeingaben im Umfang entsprechend der angestrebten Fehlerüberdeckung beziehungsweise Diagnosetiefe notwendig.

Fehlerüberdeckung

Die Fehlerüberdeckung ist ein qualitatives Maß für die Effektivität eines Satzes an Testeingaben Schaltungsdefekte zu erkennen. Diese gibt die prozentuale Anzahl der modellierten Fehler an, die Defekte im Schaltungsdesign mittels des Testsatzes in fehlerhaften Ausgaben beobachtbar machen.

Die Fehlerüberdeckung eines Testsatzes ergibt sich also aus der Gesamtzahl der Fehler bezüglich des zugrundeliegenden Fehlermodells und der durch den Testsatz entdeckten Fehler:

$$Fehlerüberdeckung = \frac{Anzahl\,entdeckter\,Fehler}{Gesamtzahl\,der\,Fehler}.$$

Die effektive Fehlerüberdeckung, auch bezeichnet als Testabdeckung, berücksichtigt die nicht entdeckbaren Fehler in einer Schaltung:

$$Effektive\,Fehlerüberdeckung = \frac{Anzahl\,entdeckter\,Fehler}{Gesamtzahl\,der\,Fehler - Anzahl\,unentdeckbarer\,Fehler}.$$

2.1.2 Fehlermodelle

Es ist aufgrund der Vielfältigkeit an Schaltungsfehlern nicht möglich, Tests für spezifische Defekte zu generieren. Um einen Satz an Testvektoren mit hoher Fehlerüberdeckung zu ermitteln, bedarf es einer entsprechenden Modellierung potentieller Fehler. Das Fehlermodell dient der Abbildung physikalischer Fehler auf Fehlerpunkte der abstrahierten Funktionsebene. Im Allgemeinen sollte ein gutes Fehlermodell zwei Kriterien genügen [Wan06]:

1. Korrektes Nachbilden des Verhaltens von Fehlern,
2. Berechnungseffizienz bezüglich der Fehlersimulation und Testmustererzeugung.

Um Testsätze den realen Gegebenheiten, mit vielen potentiellen Fehlern unterschiedlichen Typs, anzupassen und dem Punkt 1 zu genügen, werden Fehlermodelle oft kombiniert.

In einem Fehlermodell existieren n potentielle Fehlerpunkte mit jeweils m verschiedenen Arten des Fehlverhaltens (üblicherweise $m = 2$) [Wan06]. Wird für die Ermittlung von Testvektoren nur ein einzelner Fehler in der Schaltung angenommen, so handelt es sich um ein Einzelfehlermodell. Es werden dabei alle n Fehlerpunkte einzeln betrachtet, so dass die Anzahl der Einzelfehler $s = n * m$ ist. Da es gerade in größeren Schaltungen oft vorkommt, dass mehrere Einzelfehler im selben Testergebnis resultieren, genügt ein Repräsentant aus dieser Menge der sogenannten äquivalenten Fehler. Somit ist die tatsächliche Anzahl der zu betrachtenden (repräsentativen) Einzelfehler $r \ll s$.

Mehrfachfehlermodelle hingegen lassen eine beliebige Anzahl an Fehlern in der Schaltung zu. Hierbei müssen Testvektoren für jede Kombination aller Fehlerpunkte ermittelt werden, um die maximal mögliche Fehlerüberdeckung zu erreichen. Jeder Fehlerpunkt kann entweder fehlerfrei sein, oder einen aus m verschieden Fehlern aufweisen. Werden sämtliche Fehlerpunkte als fehlerfrei angenommen, so bedeutet dies eine fehlerfreie Schaltung. So ist die Anzahl der Mehrfachfehler $k = (m + 1)^n - 1$.

Das Mehrfachfehlermodell bildet die Realität besser ab, ist aber aufgrund der für den erschöpfenden Test enormen Anzahl an zu berücksichtigenden Fehlerkombinationen, höchst ineffizient bezüglich Fehlersimulation und Testvektorerzeugung. Laut praktischer Erfahrungen ist die gegenseitige Maskierung bei Mehrfachfehlern nicht sehr wahrscheinlich. Eine durch ein Einzelfehlermodell erzielte hohe Fehlerüberdeckung wird auch

im Mehrfachfehlermodell hoch ausfallen [BA00]. Daher werden üblicherweise Einzelfehlermodelle für die effizientere Testmustererzeugung angewandt. In den weiteren Erläuterungen wird sich auf die Annahme von Einzelfehlern beschränkt.

Fehlermodelle können auch in statische und dynamische Modelle unterschieden werden. Statische Fehlermodelle werden genutzt, um Defekte wie Kurzschlüsse/Überbrückung zwischen Verbindungsleitungen oder Unterbrechungen der Verbindungsleitungen erkennbar zu machen. Dabei liegt eine Leitungen beziehungsweise ein Gatteranschluss fortwährend auf einen festen Wert (*Stuck-at Fault*). Dynamische Modelle hingegen zielen auf Defekte ab, die in fehlerhaften Transitionen resultieren, sogenannte dynamische Fehler, die beispielsweise durch Signalverzögerungen, Übersprechungen oder auch durch Kurzschlüsse oder Unterbrechungen in Transistoren entstehen.

2.1.2.1 Einzelhaftfehlermodell

Der Einzelhaftfehler ist die gebräuchlichste Modellierung von Fehlern für den Test digitaler Systeme. Dieses Modell bezieht sich auf statische Fehler auf der Gatterebene oder der Registertransferebene (RTL). Das bedeutet, ein Haftfehler beeinflusst den Zustand von Signalleitungen logischer Schaltungen, einschließlich primärer Ein- und Ausgangsports, Ein- und Ausgänge interner Elemente (Gatter, Flipflops) sowie Fanout-Signale. Dabei scheint die Leitung konstant auf logisch 0 (*Stuck-at-0*) oder logisch 1 (*Stuck-at-1*) zu liegen. Jeder Fehlerpunkt kann also einen von zwei Fehlern aufweisen ($m = 2$). Der Haftfehler abstrahiert von der eigentlichen Fehlerursache. Es kann ein physikalischer Defekt wie Leitungsbruch, Kurzschluss, Überbrückungsfehler, Verzögerungsfehler, Transistorfehler, aber auch eine transiente Störung während des Tests zum Fehlverhalten und somit zur Interpretation als Haftfehler führen.

Beispiel In Abbildung 2.1 ist ein Beispiel einer logischen Schaltung mit zwei Eingängen, A und B, und einem Ausgang $Z = \overline{A} \wedge B$ zu sehen. Diese Schaltung besitzt bezüglich des Einzelhaftfehlermodells vier Fehlerpunkte (a, b, c, d), die im Fehlerfall jeweils fest auf '0' oder '1' gesetzt sind. In der Tabelle 2.1 sind die durch die Haftfehler verfälschten Ausgaben (grau unterlegt) dargestellt. Jeder Fehler führt bei mindestens einem der an den Eingängen angelegten Testvektoren '00', '01', '10' und '11' zu einem von der fehlerfreien Ausgabe verschiedenen Wert. Eine Fehlerüberdeckung von 100% wird mit drei Testvektoren erreicht, da mit den Eingaben '00', '01' und '11' jeder der acht Fehlerpunkt mindestens einmal angesprochen und ein möglicher Fehler auch am Ausgang erkennbar gemacht werden kann. Führt der Testvektor '00' zu einer '1' am Ausgang, kann mittels eines weiteren Testvektors, '10' oder '11', genau einer der Fehlerpunkte b oder d (*Stuck-at-1*) diagnostiziert werden. Ist hier für jeden der Testvektoren die Ausgabe stets '0', kann nicht auf einen einzigen Fehlerpunkt, sondern nur auf die Menge der Fehlerpunkte a (*Stuck-at-1*), b, c oder d (jeweils *Stuck-at-0*) geschlossen werden. Es genügt also ein Repräsentant aus dieser Menge, um notwendigen Testvektoren zu ermitteln. Eine feingranularere Diagnose ist in dem Fall ohne Modifikation der Schaltung, wie etwa ein Fanout vom Punkt c, nicht möglich. Dieser Umstand der

zusammenfassbaren Fehler (*Stuck-at Fault Collapsing*) reduziert die Anzahl an zu betrachtenden Fehlern typischerweise um 50 bis 60% [BA00].

Hier ist zu sehen, dass in einem Fall der Fehler bis hinunter aufs einzelne Gatter, im anderen Fall aber nur ein Fehler innerhalb eines Schaltungsbereichs beziehungsweise einer Komponente diagnostiziert werden kann. Dafür ist aber, selbst für die höchste zu erreichende Fehlerüberdeckung, nur eine Teilmenge des erschöpfenden funktionalen Testsatzes nötig.

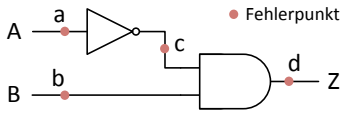


Abbildung 2.1: Beispielschaltung Haftfehler

Testvektor		Fehlerfr.	Ausgabe Z bei Einzelhaftfehler							
A	B	Ausg. Z	a↓0	a↓1	b↓0	b↓1	c↓0	c↓1	d↓0	d↓1
0	0	0	0	0	0	1	0	0	0	1
0	1	1	1	0	0	1	0	1	0	1
1	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	0	0	1	0	1

Tabelle 2.1: Wahrheitstabelle für Einzelhaftfehler der Abbildung 2.1

2.1.2.2 Dynamische Fehlermodelle

Neben Fehlern, die sich als Haftfehler darstellen lassen, gibt es auch Fehler, die nur während des Schaltvorganges, zum Beispiel aufgrund Transistorfehler, Verzögerungen oder Übersprechungen (*Crosstalk*), auftauchen. Da sich der Fehler hier nicht als statischer Wert, sondern als Zustandswechsel, also in Abhängigkeit des vorherigen Zustands, "dynamisch" auswirkt, wird dieser als dynamischer Fehler bezeichnet.

Typische Ursachen für fehlerhafte Transitionen sind Verzögerungsfehler (*Delay Fault*). Hier kann unterschieden werden zwischen der Verzögerung der Umschaltzeiten von Gattern (*Transition Fault*), der Verzögerung der Signallaufzeiten innerhalb des Gatters (*Gate-Delay Fault*), oder der Verzögerungen auf Schaltungspfaden (*Path-Delay Fault*). Zum Fehler kommt es dabei, wenn die Verzögerung einer steigende Flanke (*Slow-to-Rise*) beziehungsweise fallenden Flanke (*Slow-to-Fall*) so groß ist, dass der Sollzustand zum Abtastzeitpunkt am zu lesenden Ausgang noch nicht erreicht ist [Fis03]. Soll beispielsweise wie in Abbildung 2.2 ein Signal aus der Logik ins D-Flipflop übernommen werden, muss es zur steigenden Taktflanke an dessen Eingang stabil anliegen, ansonsten wird der bisherige Zustand gehalten. Hier ist die Transition von logisch 0 auf logisch 1 um δt verzögert, so dass diese nicht rechtzeitig am Eingang D anliegt und somit der neue Wert nicht gespeichert wird.

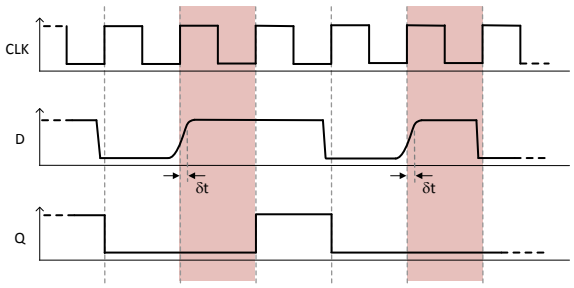


Abbildung 2.2: Verzögerungsfehler Slow-to-Rise

Eine Leitungsunterbrechung (*Stuck-open Fault*) kann zur fehlerhaften Transition führen, wenn diese, die Transistorebene betrachtend, innerhalb eines Gatters liegt. Befindet sich hingegen die Unterbrechung auf einer Verbindung zwischen Gattern, verhält sich diese wie ein Haftfehler.

Beispiel In Abbildung 2.3 ist ein CMOS NOR-Gatter, mit zwei Eingängen A und B und einem Ausgang $Z = \overline{A \vee B}$ dargestellt. Angenommen, es liegt eine Unterbrechung (*Stuck-open*) in Transistor N_2 vor, so ist der Ausgang über diesen Kanal von der Masse getrennt. Wird nun der Testvektor '01' am Eingang AB angelegt, kann Z nicht über N_2 entladen werden und behält seinen vorherigen Zustand. Somit wird der Fehler nur erkannt, falls der Ausgang Z zuvor logisch 1 war. Es muss also zur Fehlererkennung sichergestellt werden, dass das Gatter entsprechend initialisiert ist. Das bedeutet, es sind zwei aufeinanderfolgende Testvektoren notwendig, um einen Transitionsfehler sichtbar zu machen. So kann zunächst mittels des Testvektors '00' der Ausgang $Z = 1$ geladen werden. Mit dem darauffolgenden Testvektor '01', wird ein Umschalten des Ausgangs auf $Z = 0$ erwartet. Mit dem Ausbleiben dieser Transition ist folglich ein Unterbrechungsfehler in N_2 erkannt.

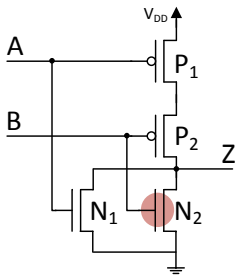


Abbildung 2.3: Beispielschaltung Transistorfehler (Stuck-open)

Testvektor		Z(t)[1]	Ausgabe Z(t) bei Kurzschluss (s) oder Unterbrechung (o) [2]							
A	B		N_1o	N_1s	N_2o	N_2s	P_1o	P_1s	P_2o	P_2s
0	0	1	1	I_{DDQ}	1	I_{DDQ}	Z(t-1)	1	Z(t-1)	1
0	1	0	0	0	Z(t-1)	0	0	0	0	I_{DDQ}
1	0	0	Z(t-1)	0	0	0	0	I_{DDQ}	0	0
1	1	0	0	0	0	0	0	0	0	0

Tabelle 2.2: Wahrheitstabelle für Transitionsfehler der Abbildung 2.3

In Tabelle 2.2 sind die Ausgaben, die eine Erkennung eines Transitionsfehlers zulassen, hellgrau unterlegt. Um eine fehlerhafter '01'-Transition (fallende Flanke) sichtbar zu machen, muss zur Initialisierung zunächst ein Testvektor gewählt werden, der eine Ausgabe $Z(t-1)$ ungleich der erwarteten korrekten Ausgaben $Z(t)$ des darauffolgenden Testvektors erzeugt. Der Transitionsfehler von logisch 0 auf logisch 1 taucht nur in den Fällen, in denen ein Unterbrechungsdefekt (*Stuck-open*) vorliegt, auf.

Ein Kurzschluss (*Stuck-short*) eines Transistors resultiert in einer Spannungsteilung, da bei entsprechender Eingangsbelegung ein Kanal von Masse zu Spannungsversorgung V_{DD} erzeugt wird. Diese Fälle sind in Tabelle 2.2 grau dargestellt (I_{DDQ}). Solch ein Defekt muss sich aber nicht zwangsläufig auf eine fehlerhafte Ausgabe auswirken, da die Ausgangsspannung unter Umständen als korrekter Wert interpretiert werden kann. Um diesen Fehler dennoch zu diagnostizieren, gibt es die Möglichkeit des I_{DDQ}-Tests. Hierbei wird der Versorgungsstrom I_{DDQ} im Betriebszustand überwacht. Es ist zu sehen, dass die in Reihe geschalteten P-Kanal-Transistoren dasselbe Fehlerbild bei einem Unterbrechungsdefekt liefern. Ebenso verhält es sich mit den parallel geschalteten N-Kanal-Transistoren bei einem Kurzschlussdefekt. Das heißt, in diesem Beispiel kann auf 6 verschiedene Fehler getestet werden, wobei eine Unterbrechung in N_1, N_2 oder $P_{1/2}$ anhand der Ausgabe festgestellt werden kann, bei einem Kurzschluss in $N_{1/2}$, P_1 oder P_2 es aber zusätzlicher Komponenten zur Spannungsmessung für den I_{DDQ}-Test bedarf.

Das bedeutet, Fehler in Transistoren, die sich in Transitionsfehlern am Ausgang auswirken, können mittels in den Chip integrierter Teststrukturen erkannt werden. Es kann also auf externe Messvorrichtung verzichtet und so auch noch im Feld bis hinunter auf Transistorebene getestet werden. Soll auch der I_{DDQ}-Test on-chip ermöglicht werden, sind zusätzlich entsprechende analoge Messbauteile zu integrieren. Mit sinkender Versorgungsspannung, steigenden Frequenz und verhältnismäßig hohen Leckströmen und der daraus resultierenden erschwerten Messbarkeit der spezifizierten Schwellwerte verliert der I_{DDQ}-Test aber an Bedeutung [RMC06].

[1] Fehlerfreie Ausgabe zum Zeitpunkt t.

[2] Fehlerbehaftete Ausgabe zum Zeitpunkt t. Die Ausgabe Z(t-1) bezieht sich auf den Zeitpunkt der Eingabe zuvor.

2.1.3 Testmethoden

Es haben sich diverse Testmethoden für verschiedene Anwendungsbereiche und Erfordernisse etabliert (Abbildung 2.4). So wird eine integrierte Schaltung von "außen", etwa mittels aufwendiger Tester (ATE) während der Produktionsphase, funktional, zumeist aber strukturorientiert getestet. Im Zielsystem eingebettet, gibt es Methoden für den Offline- oder Online-Test. Das in dieser Arbeit vorgestellte Konzept eines Testzugangs zielt hauptsächlich auf einen Off-chip-Test zur Diagnose im Zielsystem eingebetteter Systeme ab. Es wird aber auch ein Hardware-Selbsttest (BIST) on-chip realisiert. Zudem ermöglicht das Testsystem auch ein Monitoring-Modus, der quasi ein Off-chip-Test mittels Online-Daten darstellt. Das heißt, im Betriebszustand aufgenommene Daten können ausgelesen und off-chip ausgewertet werden.

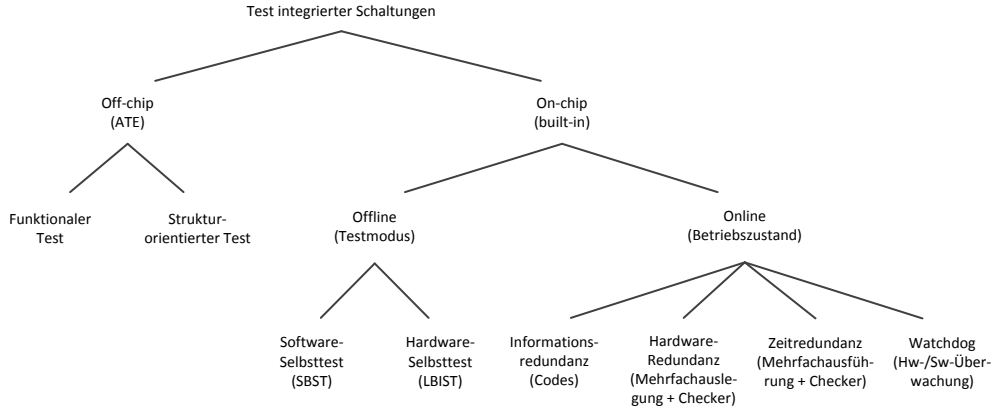


Abbildung 2.4: Testmethoden

2.1.3.1 Design for Testability

Um hochintegrierte Schaltungen qualitativ und effizient testen zu können, wird die Testbarkeit schon im Entwurf berücksichtigt. Wobei die Testbarkeit (*Testability*) die Fähigkeit ist, bestimmte Attribute innerhalb eines Systems zu testen [Pra96]. Entscheidend hierbei ist eine ausreichende Steuerbarkeit und möglichst hohe Beobachtbarkeit bei geringen Testkosten, also der Testqualität angemessenen zusätzlichen Zeit- und Hardware-Aufwand. Die hierzu erforderlichen Techniken werden unter dem Begriff *Design for Testability* (DFT) zusammengefasst.

Typische *Ad hoc*-Maßnahmen, die im DFT-Schaltungsentwurf befolgt werden sollten, sind [Wan06]:

- Vermeidung asynchroner Logik,
- Vermeidung kombinatorischer Rückkopplungen,
- Vermeidung asynchroner Steuersignale (Set/Reset) sequentieller Elemente,
- Partitionierung komplexer Funktionsblöcke in mehrere kleine testbare Module,
- Integration von Testpunkten für bessere Beobachtbarkeit und Steuerbarkeit (Testpads),
- Integration von Multiplexern zur sequentiellen Ausgabe innerer Schaltungszustände auf einen Ausgang oder Testpunkt.

Eine weitere etablierte DFT-Maßnahme ist das Einbringen von Scan-Strukturen in die zu testende Schaltung (*Circuit Under Test* - CUT), um so interne Zustände besser einstellen und beobachten zu können.

2.1.4 Funktionaler Test

Der funktionale Test dient dem Nachweis des korrekten Verhaltens ohne Informationen des strukturellen Schaltungsaufbaus einzubeziehen. Dies geschieht durch das Anlegen im Betrieb vorkommender Testeingaben an die Schaltungseingänge und anschließenden Vergleich mit erwarteten Ausgangszuständen. Die Testeinstellung erfolgt also nur über die primären Eingänge der Schaltung und die Beobachtung nur über die primären Ausgänge. Bei einem solchen rein verhaltensorientierten Test stehen zur Auswertung der Testergebnisse nur Informationen des Schaltungsverhaltens wie die Wahrheitstabelle der Schaltung zur Verfügung. Die für den Test erforderlichen Eingangsbelegungen und die daraus resultierenden Testantworten sind mittels funktionaler Simulation relativ einfach zu erzeugen. Dieser Test ist dennoch nur für kleinere Schaltungen beziehungsweise kleinere Schaltungsmodule erschöpfend durchführbar. Durch die hohe Anzahl an Testeingaben, die exponentiell mit der Anzahl an Eingängen ansteigt, ergibt sich eine hohe Testkomplexität und damit erhebliche Testdauer. Bei n Eingängen einer rein kombinatorischen Schaltung sind bis zu 2^n Eingangskombinationen zu berücksichtigen. Im Falle einer sequentiellen Schaltung sind alle möglichen Schaltungszustände einzubeziehen. Sind in der Schaltung m sequentielle Elemente enthalten, muss jede Eingangsbelegung für 2^m verschiedene Schaltungszustände betrachtet werden. Das heißt, es ergeben sich für einen erschöpfenden Test 2^{n+m} Testfälle. Aufgrund des enormen Testdatenumfangs und der dementsprechend hohen Testzeit ist dies für sequentielle Schaltungen keine adäquate Testmethode.

2.1.5 Strukturorientierter Test

Um den Test komplexer Schaltungen zu beherrschen, werden Strukturinformationen hinzugezogen. Der Test wird auf definierte Fehlerpunkte ausgelegt, um die Schaltungs-

struktur bezüglich eines zugrundeliegenden Fehlermodells zu untersuchen. Das Schaltungsdesign muss hierzu als Netzliste auf Gatter- oder Transistorebene vorliegen, um mittels der Testeingaben direkt Fehlerpunkte zu stimulieren. Es werden hier entgegen des funktionalen Tests nur die bezüglich der angenommenen Fehlerpunkte relevanten Testeingaben betrachtet. Eine Testeingabe kann bereits mehrere Fehlerpunkte überdecken. Testeingaben, die nur bereits überdeckte Fehlerpunkte anregen, können vernachlässigt werden. Somit lässt sich der Testsatz auf Repräsentanten äquivalenter Stimuli von Fehlerpunkten reduzieren. Ziel ist es möglichst viele, selten alle, potentielle Fehlerstellen der Netzliste zu testen. Es gibt Fehlerpunkte, die für den Test nicht einstellbar oder nicht beobachtbar sind. Um den Testdatenumfang und die Testzeit relativ gering zu halten, wird zudem der Test auf eine hinreichend hohe Zahl an Fehlerpunkten beschränkt. Ein qualitatives Maß des diagnostischen Tests ist die durch den ermittelten Testmustersatz zu erreichende Fehlerüberdeckung.

2.1.5.1 Software-basierter Selbsttest

Bei dem software-basierten Selbsttest (*Software-based Self-Test* - SBST) wird eine Prozessoreinheit mittels Software-Routinen, die nur über die primären Eingänge gesteuert werden, getestet. Es werden durch die ablaufende Testroutine Fehlerpunkte bezüglich eines zugrundeliegenden Fehlermodells eingestellt und so statische, aber auch dynamische Fehler observierbar gemacht. Für einen Test auf statische Fehler werden Fehlerpunkte durch Eingaben zur Steuerung der Routine angeregt und mögliche Fehler zum Ausgang propagiert. Testroutinen für dynamische Fehler stellen sich als weitaus komplexer dar, da ein Zustandswechsel an einem Fehlerpunkt durch den Ablauf der Software erzwungen werden muss.

Da die Ausführung der Testroutine *at-speed*, also mit Arbeitsgeschwindigkeit des Prozessors, geschieht, können Fehlerpunkte ohne Weiteres auch auf dynamisches Fehlverhalten, das sich aus Verzögerungsfehlern ergibt, getestet werden. Außerdem erzeugt auf dem Chip ablaufende Software stets funktionale Zustände, so dass das Einstellen illegaler Zustände und somit ein Übertesten der Schaltung nicht berücksichtigt werden muss. Um ein Fehlverhalten der Schaltung zu erkennen, werden Testergebnisse entweder mit on-chip Sollwerten verglichen oder für die Off-chip-Auswertung in den Speicher ausgelagert. Aufgrund Strukturinformationen lässt sich eine verfälschte Testantwort auf eine fehlerhafte Einheit, je nach Granularität auf ein Modul oder sogar bis hinunter auf einen bestimmten Fehlerpunkt zurückführen.

Der Vorteil dieses software-basierten Ansatzes liegt darin, dass das Schaltungsdesign nicht modifiziert werden muss, da die Einstellung des Tests nur über primäre Eingänge und die Beobachtung nur über primäre Ausgänge geschieht. So werden auch strukturorientierte Tests für Schaltungen, die nicht mit zusätzlichen DFT-Elementen ausgestattet sind, ermöglicht. Ferner ist der Testsatz aufgrund der durch den zur Verfügung stehenden Befehlssatz einstellbaren Zustände auf funktionale Testeingaben beschränkt. Allerdings sind aufgrund dieser Beschränkung das Einstellen von Fehlerpunkten und die Berechnung von Testroutinen, gerade für dynamische Fehler, sehr aufwendig. Ein Test mit umfangreichem Testsatz, der unter Umständen für eine hohe Fehlerüberdeckung

benötigt wird, führt zudem zu hohen Speicherbedarf im Programmspeicher und entsprechend lange Testzeit. Ein großer Nachteil ist auch die automatisierte Erzeugung des SBSTs, da diese erfahrungsgemäß keine hohe Fehlerüberdeckung erwarten lässt.

2.1.5.2 Scan-Test

Um den Zugang zu Fehlerpunkten und somit auch die Testmusterermittlung zu erleichtern, wird die Schaltung mittels DFT-Maßnahmen für den strukturorientierten Test ausgelegt. Ein strukturorientierter DFT-Ansatz soll im Allgemeinen den Test einer Schaltung mittels methodischer und systematischer Verfahren verbessern. Das Scan-Design ist die meistgenutzte strukturorientierte DFT-Maßnahme [Wan06]. Die Einstellbarkeit und Beobachtbarkeit der inneren Schaltungszustände wird dadurch erreicht, dass sequentielle Elemente zu Scan-Zellen modifiziert und diese während des Tests zur Initialisierung oder Ergebnisausgabe zu Schieberegistern, sogenannte Scan-Ketten, verschaltet werden. Da diese Scan-Struktur die Schaltung in vielen Teilschaltungen segmentiert und somit die sequentielle Tiefe, im Idealfall bis hin zur reinen Kombinatorik, reduziert, wird die Berechnung der Testmuster und Analyse der Testantworten wesentlich erleichtert. Es existieren diverse Scan-Design-Ansätze bezüglich des Aufbaus der Scan-Zellen und ihrer Clock-Mechanismen. Die bekanntesten Designs sind Muxed-D-Scan, Clocked-Scan und LSSD (*Level-Sensitive Scan Design*) [Okl08, Wan06].

Die im Testmodus erzeugten Schieberegister werden nicht mit dem Takt des Normalbetriebs, sondern mit weitaus langsamerem Takt betrieben. Dies geschieht, um die korrekte Übernahme der Testdaten in die Scan-Zellen zu garantieren. Es muss hier einerseits die nötige Stromaufnahme bei gleichzeitigem Betrieb aller Scan-Zellen gewährleistet, andererseits die Signallaufzeit zwischen aufeinanderfolgenden Scan-Zellen berücksichtigt werden. Zudem könnte während des Testbetriebs ein sich ständig wiederholendes schnelles Umschalten der Scan-Struktur die Schaltung überlasten und sogar zu einem dauerhaften Defekt führen. Es werden deshalb unterschiedliche Clocksignale für Normalbetrieb und Schiebemodus verwendet.

Beim Clocked-Scan werden zwei, beim LSSD drei unabhängige Clocks an jede Scan-Zelle geführt. Hierbei geschieht die Steuerung, das Umschalten zwischen Normal- und Schiebemodus, über die sich nicht überlagernden Clocksignale. Die Daten-Clock lässt die Scan-Zellen im Normalmodus mit Betriebsgeschwindigkeit schalten. Mittels der Scan-Clock beziehungsweise des Scan-Clockpaares des LSSDs wird der langsame Schiebetakt realisiert. Es müssen hierzu aber mehrere parallele Clocknetze in der Schaltung integriert werden. Beim Muxed-D-Scan hingegen wird eine globale Clock dem Testmodus über *Clock-Gating* angepasst. Das Scan-Design ist Grundlage für das in dieser Arbeit vorgestellte Konzept, der Aufbau der einzelnen Scan-Zelle dabei aber irrelevant. Daher wird der Scan-Test im Folgenden anhand des Muxed-D-Scans veranschaulicht.

Dabei werden im Grunde Speicherelemente durch D-Flipflops (DFF) ersetzt und um Signale und Multiplexer erweitert, um diese für den Scan-Modus zu Schieberegister zu verknüpfen. Eine Scan-Zelle, auch als Scan-Flipflop (SFF) bezeichnet, hat einen Dateneingang DI und einen Scan-Eingang SI, die wahlweise mittels eines Steuersignals SE

ans Flipflop angelegt werden (Abbildung 2.5). Im Normalbetrieb ist das Steuersignal SE stets logisch 0. Für den Scan-Test ergeben sich zwei Modi, der für die Schiebefunktion (*Shift*) und der für die Funktion im Normalbetrieb (*Capture*). Ist das Steuersignal SE logisch 0, so befindet sich die Schaltung im Normalmodus und der Wert aus der Kombinatorik am Dateneingang DI wird mit dem Clock-Takt ins D-Flipflop übernommen. Durch das Umschalten des Steuersignals SE auf logisch 1, wird der *Shift*-Modus aktiviert und somit der Wert des Vorgänger-Flipflops des Scan-Pfades geladen. Das erste Scan-Flipflop des Scan-Pfads stellt dabei einen primären Eingang und das letzte Scan-Flipflop einen primären Ausgang der Schaltung dar. So kann mittels der Schiebefunktion jede Scan-Zelle beschrieben, also die gesamte Scan-Struktur der Schaltung mit einem Testmuster initialisiert werden. Die Ausgänge der Scan-Zellen können als Pseudo-Eingänge der CUT betrachtet werden, da sie im Testmodus als ansteuerbare Eingänge innerhalb der Schaltung fungieren.

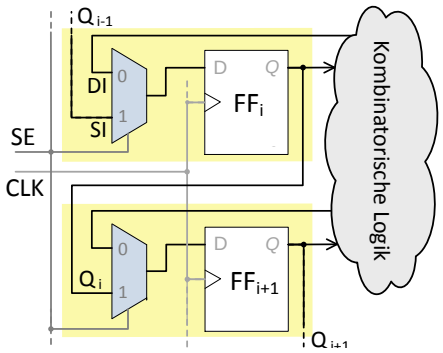


Abbildung 2.5: Zu Schieberegister verknüpfte Scan-Zellen (Muxed-D-Scan)

Sind sämtliche sequentiellen Elemente der zu testenden Schaltung zu einem Scan-Pfad verknüpft, ist dies ein *Full-Serial-Scan*. Bei einem *Partial-Serial-Scan* werden die inneren Zustände einzelner größerer Module der Schaltung nicht betrachtet. Es werden dabei nur die Ein- und Ausgänge dieser Module in die Scan-Struktur integriert.

Um das hinein- und hinausschieben der Testdaten zu beschleunigen, bietet es sich an, statt eines einzigen langen Pfades gleichzeitig mehrere kurze Pfade anzusteuern. Dies wird durch das *Parallel-Scan*-Design realisiert (Abbildung 2.6). Der Scan-Pfad wird quasi in mehrere möglichst gleichlange Pfade aufgebrochen, so dass sich eine Struktur paralleler Scan-Ketten über der Schaltung ergibt. Die Scan-Ketten sind über primäre Eingänge der zu testenden Schaltung parallel ansteuerbar und über primäre Ausgänge parallel auslesbar. Auch hier kann zwischen einem *Full-Scan* und *Partial-Scan* unterschieden werden, je nachdem, ob sämtliche sequentiellen Element oder nur diejenigen, die interne Module umschließen zur Scan-Struktur verknüpft werden. Eine weitere Scan-Architektur ist das *Random-Access-Scan*-Design, bei dem die Scan-Zellen als Matrix angeordnet sind. Jede Scan-Zelle ist direkt über Zeile und Spalte adressierbar, kann also einzeln eingestellt und observiert werden. Da der Schiebemodus und somit das ständige Umschalten der Logik entfällt, wird die Schaltungsbelastung und der Energieverbrauch reduziert. Allerdings fällt aufgrund der zusätzlichen Verbindungsstruktur und Adres-

sierungslogik ein hoher Hardware-Mehraufwand an, und ein Gewinn an Testzeit kann auch nicht erwartet werden [Wan06]. Im weiteren Verlauf dieser Arbeit wird der Scan-Test auf Grundlage des weitverbreiteten *Parallel-Scan* als guter Kompromiss zwischen Hardware-Aufwand und Testzeit betrachtet.

Das Eingabemuster, das während des Tests in die ersten Scan-Zellen der Scan-Ketten geschrieben wird, wird als *Testvektor* bezeichnet. In den Scan-Zellen der Schaltungsausgänge befindet sich folglich die Testantwort. Das heißt, die Datenwortbreite des Testvektors und der Testantwort entspricht der Anzahl der Scan-Ketten, die durch die Anzahl der zur Verfügung stehenden oder für den Test spendierten Ein- und Ausgangsports begrenzt ist. Die Dauer der Phase des Hinein-/Hinausschiebens der Testmuster wird durch die Scan-Tiefe bestimmt. Die Scan-Tiefe ergibt sich aus der Anzahl der Scan-Zellen der längsten Scan-Kette. Daher sollte für eine optimale Testzeit eine Scan-Struktur mit einer möglichst hohen Anzahl an kurzen Scan-Ketten entworfen werden.

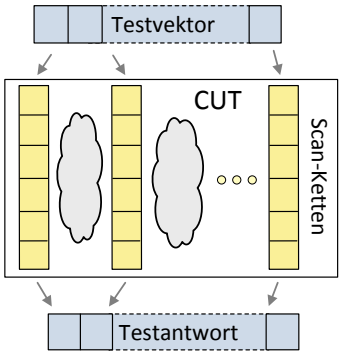


Abbildung 2.6: Parallel-Scan-Design

Statischer Scan-Test

Bei dem Test auf statische Fehler wird die Schaltung mit einem Testmuster initialisiert und mittels eines funktionalen Taktes wie im Normaleinsatz betrieben. Hierzu wird im *Shift*-Modus, der durch das Umschalten des Steuersignal SE auf logisch 1 aktiviert wird, ein Testvektor an die Schaltungseingänge angelegt und mit Scan-Takt in die Scan-Ketten übernommen. Daraufhin folgt der nächste Testvektor. Die bereits in den Scan-Ketten befindlichen Testvektoren werden mit jedem Takt eine Scan-Zelle weiter geschoben, so dass nach m Takten, bei einer Scan-Tiefe m, sämtliche Scan-Zellen initialisiert sind. Das heißt, es befindet sich ein komplettes Testmuster in der zu testenden Schaltung. Nun folgt der sogenannte *Capture*-Modus (Abbildung 2.7). Hierzu wird das Steuersignal SE zurückgesetzt auf logisch 0 und der Inhalt der Scan-Ketten für einen funktionalen Takt durch die zu testende Logik geschaltet. Somit stehen nach diesem Takt die Antwortdaten aus der internen Logik in den Scan-Ketten. Das Steuersignal SE wird wieder auf logisch 1 gesetzt, um im *Shift*-Modus den erhaltenen Schaltungszustand auszulesen. Das heißt, über die primären Eingänge wird das neue Testmuster hinein- und über die primären Ausgänge die aktuellen Testantworten hinausgeschoben.

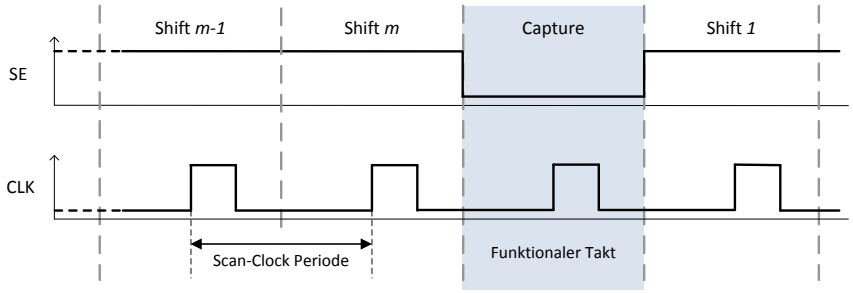


Abbildung 2.7: Statischer Scan-Test

Dynamischer Scan-Test

Der dynamische Scan-Test soll fehlerhafte Transitionen, die während des Normalbetriebs auftreten, aufdecken. Es ist daher entscheidend, dass der Test mit dem Takt des Betriebszustandes (*at-speed*) geschieht. So werden auch Verzögerungsfehler, die sich nur bei diesem schnellen Takt auswirken würden, sichtbar gemacht.

Beim Test auf dynamische Fehler kommt eine weitere Testphase hinzu, die der Initialisierung der Schaltung vor der zu untersuchenden Transition dient. Diese sogenannte *Launch*-Phase wird vor dem *Capture*-Takt ausgeführt. Es wird zwischen drei Testschemata unterschieden: *Launch-on-Capture* (LOC), *Launch-on-Shift* (LOS) und *Enhanced-Scan* [Wan06].

Beim LOC-Schema (auch *Broadside* oder *Double-Capture*) werden Testvektoren im *Shift*-Modus, bei aktiviertem SE-Signal, mit langsamem Takt in die Scan-Ketten geschoben. Das Steuersignal SE wird zurückgesetzt auf logisch 0 und die Schaltung durch zwei in Betriebsfrequenz aufeinanderfolgende Takte (*Launch*- und *Capture*-Takt) auf Transitionsfehler getestet (Abbildung 2.8). Es muss zur Erzeugung der Transition nur ein Testvektor abgespeichert werden, da sich der Folgevektor aus dem *Launch*-Takt, also durch die Logik der Schaltung ergibt.

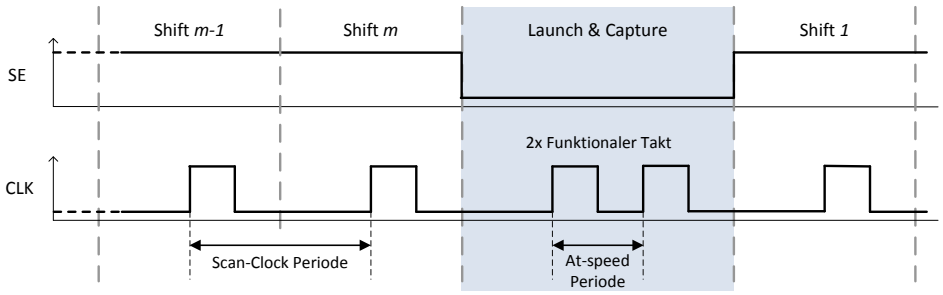


Abbildung 2.8: Launch-on-Capture Scan-Test

Der Scan-Test nach dem LOS-Schema (auch *Skewed-Load*) verzichtet auf einen extra *Launch*-Takt. Die Testvektoren werden mit langsamem *Shift*-Takt in die Scan-Ketten

geschoben. Der letzte *Shift*-Takt entspricht hier dem *Launch* (Abbildung 2.9). Es folgt der schnelle funktionale Takt im *Capture*-Modus. Auch im LOS-Schema muss nur ein Testvektor für die Erzeugung der Transition gespeichert werden, da hier zwei im *Shift*-Modus aufeinanderfolgende Testvektoren das nötige Vektorpaar ergeben.

Im LOS-Design müssen die Scan-Flipflops mittels des SE-Signals schneller umschalten, damit sich die Schaltung zum funktionalen-Takt bereits im *Capture*-Modus befindet. Hierzu bedarf es entsprechend modifizierter Scan-Flipflops, wie in [XS07] vorgestellt. Durch das LOS-Design kann mit weniger Testvektoren eine höhere Fehlerüberdeckung für Verzögerungsfehler als durch das LOC-Design erzielt werden [XS06]. Es wird jedoch auch die Wahrscheinlichkeit des Übertestens erhöht, da viele Verzögerungsfehler erkannt werden, die sich im Normalbetrieb nicht auswirken würden [KK07].

Testvektoren für den dynamischen Test überdecken im Allgemeinen nicht alle Einzelhaftfehler [Fis03]. Daher kann der Testsatz des LOS-Scan-Test um *Stuck-at*-Testvektoren ergänzt werden, um eine hohe Fehlerüberdeckung sowohl für statische als auch dynamische Fehler zu erhalten.

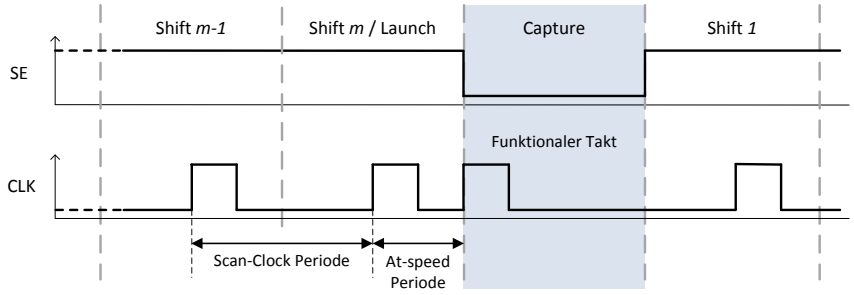


Abbildung 2.9: Launch-on-Shift Scan-Test

Beim *Enhanced-Scan* wird ein Fehlerpunkt mittels eines Vektorpaares getestet. Der erste Testvektor wird in die Schaltung geladen. Daraufhin wird der zweite Testvektor in die Scan-Ketten geschoben, während der erste noch an der Logik anliegt. Hierzu wird ein spezielles *Hold-Scan*-Design verwendet, das in der Lage ist, parallel zu einem an den Logikeingängen gehaltenen Vektor einen zweiten Vektor im *Shift*-Modus zu laden. Mit dem *Capture*-Takt werden die Transitionen vom ersten zum zweiten Testvektor erzeugt. Dieser *Enhanced-Scan* erfordert aufgrund der *Hold-Scan*-Zellen einen höheren Hardware-Aufwand. Dafür fällt der Testsatz für den Test auf dynamische Fehler viel kompakter aus. Um mittels des LOC- oder LOS-Scans die in etwa gleiche Fehlerüberdeckung zu erreichen, ist eine weitaus höhere Anzahl an Testmustern vonnöten [Wan06]. Ferner kommt das *Hold-Scan*-Design einem Low-Power-Ansatz zu Gute, da die zu testende Logik durch die *Hold*-Flipflops von den Scan-Flipflops entkoppelt ist und so während der *Stift*-Phase nicht ständig umschaltet. Aber der *Enhanced-Scan* ist auch anfälliger bezüglich des Übertestens, da Transitionen frei eingestellt werden können und so auf Verzögerungsfehler für den Normalbetrieb nicht relevanter Pfade getestet wird.

2.1.5.3 ATPG

Die Erzeugung umfangreicher Testmuster für die diversen Testmethoden ist aufgrund komplexer und zeitaufwendiger Berechnung nur über automatisierte Verfahren, bezeichnet als *Automatic Test Pattern Generation* (ATPG), beherrschbar. Eine effiziente Testmustererzeugung ist entscheidend für akzeptablen Testaufwand, also möglichst geringe Testkosten für hohe Fehlerüberdeckung in möglichst kurzer Testzeit. In die Testkosten fällt zum einen die Ausführungszeit des Tests, beispielsweise durch ein ATE im Produktionstest, zum anderen auch die Berechnungszeit der Testvektoren durch den ATPG-Prozess selbst. Daher wurde seit den Anfängen des DFT-Entwurfs auch viel Aufwand für die Entwicklung von ATPG-Verfahren betrieben. Heutzutage existieren viele stetig weiterentwickelte Algorithmen und eine Reihe kommerzieller ATPG-Tools von EDA-Herstellern (*Electronic Design Automation*).

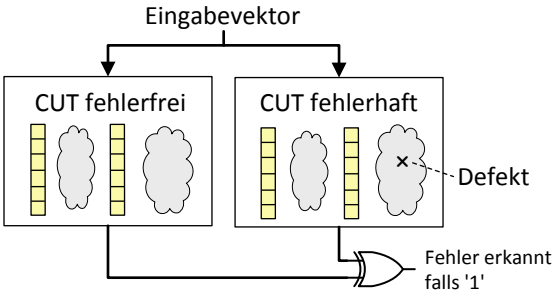


Abbildung 2.10: Automatische Testmustererzeugung (ATPG)

Um einen Testvektor zu ermitteln, wird zunächst ein Fehler in einem Fehlerpunkt angenommen. Dies geschieht mittels Fehlersimulation auf Grundlage eines Fehlermodells. Eingaben werden, je nach ATPG-Algorithmus, systematisch aus einem Satz an potentiellen Testvektoren gewählt. Ist die Ausgabe gegenüber der fehlerfreien Version, dem sogenannten *Golden Device*, verfälscht, wurde der Fehler durch die angelegte Eingabe erkannt (Abbildung 2.10). Somit stellt diese Eingabe einen essentiellen Testvektor dar.

Für den software-basierten Selbsttest ist mittels eines beschränkten Satzes an funktionalen Testeingaben ein Softwareprogramm zu erstellen. Für jeden Fehlerpunkt muss ermittelt werden, durch welche Befehlsabfolge und Operanden dieser einstellbar und observierbar gemacht werden kann. Der Test der zu untersuchenden Fehlerpunkte einer Netzliste wird also auf eine Softwareroutine zurückgerechnet.

Ein direkterer Zugang zu Fehlerpunkten ist durch einen scan-basierten Test gegeben. Dadurch, dass jede Scan-Zelle ansteuerbar ist, kann die Einstellung eines Fehlerpunktes auf eine Teilschaltung begrenzt werden. Das heißt, es muss für einen Fehlerpunkt nur die auf diesen Einfluss nehmenden Scan-Zellen, die dabei die Eingänge der zu betrachtenden Teilschaltung darstellen, für eine Testeingabe berücksichtigt werden. Somit wird die Testmusterermittlung für viele, weniger komplexe Schaltungsteile vereinfacht. Im Falle eines *Partial-Scan*-Design müssen die sequentiellen Elemente in den zu untersuchenden Modulen berücksichtigt werden. Hierzu ist ein sequentielles ATPG-Verfahren, das die

Zustände in den Modulen in die Berechnung mit einbezieht, zu nutzen. Ansonsten, beim *Full-Scan,* kann die Testmusterermittlung auf rein kombinatorische Teilschaltungen zurückgeführt und somit ein kombinatorisches ATPG-Verfahren angewandt werden.

Beispiel Im Beispiel der Abbildung 2.11 liegt der Inverterausgang c fest auf logisch 0. Um die Schaltung für den Test auf den *Stuck-at-0* in diesem Fehlerpunkt einzustellen, muss am Eingang A logisch 0 angelegt werden. Der Fehler muss zudem zum Ausgang propagiert werden. Dies geschieht hier durch den auf logisch 1 gesetzten Eingang B. Der Testvektor '01' würde im fehlerfreien Fall in der Ausgabe $Z = 1$, im fehlerhaften Fall in $Z = 0$ resultieren. Somit wird der Fehler erkannt, ist jedoch nicht eindeutig diagnostizierbar, da Einzelhaftfehler in anderen Fehlerpunkten dieselbe Ausgabe erzeugen würden. In komplexen Schaltungen kann anhand eines entsprechend großen Satzes von Testvektoren im Idealfall bis auf einen einzelnen Fehlerpunkt zurückgeschlossen werden. Eine diagnostische Auflösung bis auf einzelne Gatter ist oft nicht möglich und aufgrund des dazu benötigten enormen Testsatzes nicht praktikabel.

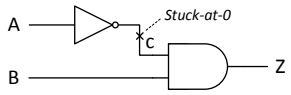


Abbildung 2.11: Beispiel Stuck-at-0

Dieses Verfahren aus Fehlererzeugung und Fehlerpropagierung wird als *Einpfadsensibilisierung* bezeichnet. Fehlererzeugung bedeutet, das Einstellen eines zum Fehler inversen Signalwertes im zu betrachtenden Fehlerpunkt. Unter Fehlerpropagierung ist das Durchschalten des Fehlers, mittels geschickter Eingangsbelegung, zu einem beobachtbaren Ausgang zu verstehen. Bei einem Einzelfehlermodell ist jeder Fehlerpunkt einzeln und bei Mehrfachfehlermodell sind entsprechend mehrere Fehlerpunkte gleichzeitig zu betrachten. Im Allgemeinen ist das Ziel eines ATPG-Algorithmus eine effiziente Bestimmung einer minimalen Anzahl an Testmustern zur Erkennung von Fehlern in einer möglichst hohen Anzahl an spezifizierten Fehlerpunkten. Für die Wahl der "optimalen" Testmuster gibt es eine Vielzahl Methoden, vom rein pseudozufallsbasierten bis hin zum erschöpfenden Ansatz.

Ein erschöpfendes ATPG-Verfahren (*Exhaustive ATPG*) ist für komplexere oder gar sequentielle Schaltungen aufgrund des enormen Datenumfangs nicht durchführbar. Bei n primären Schaltungseingängen und m sequentiellen Elementen sind 2^{m+n} Testeingaben zu betrachten, um 100% der erkennbaren Einzelhaftfehler zu überdecken. Ein erschöpfender Testsatz für dynamische Fehler, also sämtliche Kombinationen von Testvektorpaaren, würde $2^{m+n}(2^{m+n} - 1)$ Testeingaben erfordern.

Beim pseudoerschöpfenden Ansatz (*Pseudo-Exhaustive ATPG*) werden unnötige Testeingaben, die keinen Einfluss auf zu testende Funktionen haben, ausgeschlossen. Sei I die Menge aller Eingänge und O die Menge aller Ausgänge einer kombinatorischen Schaltung. Für jeden Ausgang $o \in O$ ist $I(o) \subset I$ die Menge der Eingaben von denen der Ausgang abhängt. Das pseudoerschöpfende ATPG-Verfahren erzeugt alle mögli-

chen Eingangskombinationen für jede Menge $I(o)$. Anstatt 2^n , mit $n = |I|$, müssen im Idealfall nur 2^w Testvektoren generiert werden, wenn jeder Ausgang nur von maximal w Eingängen abhängt [McC86]. Übersteigt dieser Wert w nicht ein hinnehmbares Limit, ist ein pseudoerschöpfender Test mit der Überdeckung aller erkennbaren Einzelhaftfehler ohne aufwendige Fehlersimulation durchführbar. Ein Lösungsansatz, um mit angemessenen Testsatzumfang einen diagnostischen Test mit sehr hoher Auflösung zu garantieren, ist die Schaltung in separat zu betrachtende Teilschaltungen mit nur wenigen Eingängen, zu unterteilen. Solch ein Verfahren wird als *Partial Pseudo-Exhaustive Test* (P-PET) in [MIW11, CHI12] vorgestellt.

Zufallsbasierte Methoden wie *Pseudo-Random-* oder *Weighted-Random-*Algorithmen reduzieren das Datenaufkommen um ein Vielfaches, aber garantieren keine hohe Fehlerüberdeckung. Da diese Verfahren hier (pseudo-)zufällige Testmuster über die Wahrscheinlichkeit bestimmen, werden diese als probabilistische ATPG-Verfahren bezeichnet. Eine effizientere Herangehensweise sind algorithmische Methoden, sogenannte deterministische ATPG-Verfahren, die mittels strukturierter Suche deterministische Muster bestimmen. Beispiele für etablierte deterministische Methoden sind *D-Algorithmus* (D-ALG), *Path-Oriented Decision Making* (PODEM), *Fan-Out-Oriented Algorithm* (FAN). Deterministische Verfahren haben den Anspruch, die höchstmögliche Fehlerüberdeckung zu erreichen. Eine Kombination aus probabilistischen und deterministischen ATPG, als *Mixed-ATPG* bezeichnet, vereint die Vorteile der verschiedenen Ansätze. Die Idee des *Mixed-ATPG* ist es mittels Pseudozufallsverfahren bereits ein Großteil der Fehlerpunkte zu erfassen. Anschließend wird versucht, die Überdeckung mittels deterministisch ermittelter Testmuster speziell für die nicht einbezogenen Fehlerpunkte zu maximieren.

In Abbildung 2.12 ist eine charakteristische Kurve für die Fehlerüberdeckung mittels Pseudozufallsmuster in Abhängigkeit von der Anzahl der Testvektoren dargestellt. Diese PRPG-Kurve besteht typischerweise aus einer schnell ansteigenden Anfangsphase und einer anschließenden sehr langsam ansteigenden Sättigungsphase. Daran ist zu erkennen, dass mit einem bestimmten Satz an Pseudozufallsmustern eine relativ hohe Fehlerüberdeckung erreicht werden kann. Fehler, die nur mit geringer Wahrscheinlichkeit durch eine Testsatz eingestellt beziehungsweise beobachtbar gemacht werden können, sind durch Pseudozufallsmuster schwer zu überdecken, werden demnach auch als *Random-Pattern Resistant Faults* (*RP-Resistant*) oder *Hard-to-Test Faults* (HTTF) bezeichnet. Die langwierige Sättigungsphase wird hauptsächlich durch diese RP-resistenten Fehler verursacht [JAR11]. Eine höhere Fehlerüberdeckung nahe den 100% wird daher eine zu hohe Anzahl an Zufallsmustern benötigen und kann somit nicht in realistischer Testzeit erreicht werden. Der Umfang des Testsatzes ist abhängig von einem damit zu erreichenden Mindestwert der Fehlerüberdeckung, in erster Linie aber von den Testeigenschaften der CUT selbst. Das heißt, der Testsatz wird bestimmt durch die Komplexität und Testbarkeit der Schaltung und dem integrierten Testdesign. In Schaltungen, die eine Vielzahl RP-resistenter Fehler enthalten, sogenannte *RP-resistente* Schaltungen, kann keine hohe Fehlerüberdeckung mit akzeptablem Testumfang erreicht werden. In diesen Fällen muss, zur Sicherstellung einer hinreichenden Fehlerüberdeckung, ein Kompromiss zwischen einem probabilistischen Ansatz mit

höherer Testzeit und einem deterministischen Ansatz mit umfangreicheren Testsatz gefunden werden.

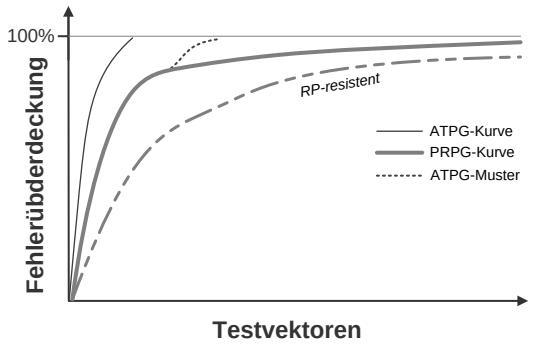


Abbildung 2.12: Fehlerüberdeckung durch (pseudo-)zufällige Testmuster

Ein wichtiger Punkt, der bei der Testmustererzeugung für Scan-Strukturen berücksichtigt werden sollte, ist das Problem des Übertestens (*Overtesting*). Ziel des ATPG ist eine möglichst hohe Fehlerüberdeckung der gesamten Schaltungsstruktur. Eine Schaltung kann aber auch "übertestet" werden, was dazu führt, dass möglicherweise funktionstüchtige Komponenten als fehlerhaft klassifiziert werden. Das liegt daran, dass, anders als beim funktionalen oder software-basierten Test, beim Scan-Test über die Teststrukturen Schaltungszustände eingestellt und beobachtet werden, die für den Normalbetrieb irrelevant oder gar unzulässig sind. Das heißt, es werden Fehlerpunkte auf Fehler getestet, die die Funktionalität nicht beeinträchtigen. Üblicherweise ziehen ATPG-Tools aber sämtliche Fehlerpunkte in die Ermittlung der Testmuster mit ein, da ausschließlich auf Strukturinformationen zurückgegriffen werden kann. Es gibt aber auch Konzepte, die diesem Problem vorbeugen, indem beispielsweise Informationen über funktionale Randbedingungen einbezogen [PF06] oder illegale Zustände vermieden werden [LLH08].

2.1.6 Produktionstest

Die Produktion von Halbleiterbauelementen erfordert eine Vielzahl an Prozessschritten mit vielen potentiellen Fehlerquellen. Daher nutzen Halbleiterhersteller Testvorrichtungen, allgemein bezeichnet als *Automatic Test Equipment* (ATE), um in Fertigungstests die Funktionstüchtigkeit der integrierten Schaltungen, aber auch analoger Bauteile und Leiterplatten (*Printed Circuit Board* - PCB) zu prüfen. Es gibt einfache Apparaturen für parametrische Tests, wie etwa Multimeter zur Spannungs-, Strom- oder Widerstandsmessung. Um automatisierte diagnostische Tests hochintegrierter Schaltungen durchzuführen, bedarf es aber komplexer ATE-Systeme. Durch Automatisierung, hohe Performanz und Parallelisierung sollen Durchsatz, Testzeit und somit Testkosten bei gleichzeitig hoher Fehlerüberdeckung optimiert werden. Die zwei wichtigen Basisfunktionen eines ATEs sind der *Pass/Fail*-Test und die Fehlerdiagnose. Mittels des *Pass/Fail*-Tests wird ermittelt, ob das zu testende Modul korrekt funktioniert oder eben als fehlerhaft zu deklarieren und auszusortieren ist. Das heißt, sobald eine Te-

stantwort ungleich der erwarteten ausgelesen wird, kann der Test des defekten Moduls abgebrochen werden. Wird ein Fehler erkannt, besteht die Möglichkeit, eine anschließende Diagnose durchzuführen, um die fehlerhafte Komponente innerhalb der Schaltung zu lokalisieren. Um in der Produktionsanlaufphase eines neuen Chips, die sogenannte *Ramp-up*-Phase, die Ausbeute zu steigern, ist es entscheidend mit hoher Diagnosetiefe zu analysieren und sogar einzelne Gatter als Fehlerursache auszumachen.

Die Tests werden zu verschiedenen Zeitpunkten des Herstellungsprozesses durchgeführt, um defekte Schaltungen frühestmöglich aus der Produktion zu nehmen. So wird der Chip erstmals nach der Fertigung auf dem Wafer getestet, um so schon vor dem Schneiden fehlerhafte Chips zu identifizieren. Bei diesem Wafertest werden die Schaltungen über Nadelkarten kontaktiert. Nach dem *Packaging*, dem Verpacken des Chips in ein Gehäuse, kann der Package-Test erfolgen. Als Schnittstelle zum ATE dient hier ein sogenanntes *Load Board* mit entsprechendem Sockel, auf dem das zu testende Modul aufgesetzt wird. Auch nach Leiterplattenbestückung werden ATEs eingesetzt, um die Funktionstüchtigkeit des eingebetteten Chips mit seiner Peripherie nachzuweisen.

2.1.7 Eingebauter Selbsttest

Um die Testbarkeit komplexerer ICs zu optimieren und vor allem die Testkosten zu reduzieren, wird ein Großteil des Testverfahrens in das zu testende Modul selbst verlagert. Der eingebaute Selbsttest (*Built-in Self-Test* - BIST) ist eine DFT-Technik, die im Allgemeinen aus einer Steuerungseinheit, einem Mustergenerator und einer Auswertungseinheit besteht (Abbildung 2.13). Die Steuerungseinheit beziehungsweise der BIST-Controller wird von einem ATE oder durch Instruktionen aus einem internen Speicher gesteuert. Das heißt, der BIST lässt sich sowohl durch einen externen Tester, aber auch als eigenständiges Testverfahren komplett on-chip betreiben. Die Testmuster werden durch den integrierten *Test Pattern Generator* (TPG) erstellt, da sowohl die Übertragung zum IC als auch der Speicherbedarf im IC für einen umfangreichen deterministischen Testmustersatz, wie er im Produktionstest angewandt wird, zu aufwendig ist. Ebenso sind die Referenzdaten der erwarteten Testausgaben zu umfangreich. Um diese zu reduzieren, wird üblicherweise eine Kompaktierung vorgenommen, die eine Vielzahl an Testausgaben auf eine Signatur abbildet. So beschränken sich die Referenzdaten auf einen kompakten Satz an Gut-Signaturen. Die Auswertung der Signaturen geschieht durch den *Output Response Analyzer* (ORA), so dass dem ATE nur noch das Ergebnis des *Pass/Fail*-Tests zu übertragen ist. Die Kontaktvorrichtung kann auf nur wenige Kanäle beschränkt und somit auch Störungen vermieden werden.

Hauptmotivation einen BIST einzusetzen ist das kosteneffiziente Testen. Kosten für ATEs können reduziert werden, da diese weniger komplex ausgelegt werden können. Die Testzeit wird durch die erhebliche Reduzierung der zu übertragenen Testdaten und durch hohe Parallelität der Teststrukturen verkürzt. Das bedeutet natürlich eine signifikante Erhöhung des Durchsatzes im Produktionstest. Die komprimierten oder auf BIST-Steuerbefehle reduzierten Testsätze erfordern geringeren Speicherplatz im ATE oder auf dem Chip selbst. Eine bessere Fehlerüberdeckung für statische als auch dyna-

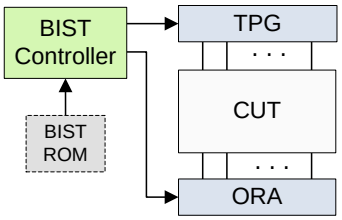


Abbildung 2.13: Schema des Built-in Self-Tests

mische Fehler kann erreicht werden, da die Schaltung um spezielle Teststrukturen wie dem Scan-Design erweitert wird und ein *At-Speed*-Test durchgeführt werden kann. Zudem kommt der BIST der diagnostischen Auflösung kritischer Schaltungskomponenten zu Gute, die keine direkte Verbindung zu externen Ein-/Ausgängen besitzen.

Als Nachteile sind zu erwähnen, die durch Testkomponenten und Teststrukturen erweiterte Chipfläche und die zusätzlichen Pins für den Testzugang. Aufgrund dieses Hardware-Mehraufwands sinkt zudem die Zuverlässigkeit des Gesamtdesigns des ICs. Außerdem wird die Leistung durch das zusätzliche Schaltungsnetz beeinflusst.

Es existiert heutzutage eine Vielzahl an BIST-Verfahren für Online- oder Offline-Tests, für funktionale oder strukturbasierte Tests und für spezifische Tests entsprechend der Art des zu testenden Moduls. Nach Anwendungsbereich klassifiziert, können BIST-Techniken hauptsächlich in Logic-, Memory- und Analog-BIST unterteilt werden. Der *Logic-BIST* (LBIST) zielt auf den Test der Schaltungslogik mittels integrierter DFT-Strukturen ab. Der *Memory-BIST* (MBIST) ist für den Test von Speichereinheiten ausgelegt, besteht also aus spezieller, dem Speicher- und Fehlertypen angepasster Testlogik. Es haben sich verschiedene MBIST-Verfahren wie *March Algorithmus*, *Checkerboard Algorithmus* oder *Varied Pattern Background Algorithmus* als Standard etabliert. Der *Analog-BIST* (ABIST) setzt den Test analoger Schaltungskomponenten um. Da die Grundlage dieser Arbeit LBIST-Strukturen für den strukturbasierten (Offline-)Test sind, wird in weiteren Ausführungen nur auf dieses BIST-Design eingegangen.

Das LBIST-Design enthält typischerweise einen BIST-Controller, einen *Pseudo-Random Pattern Generator* (PRPG) und eine Kompaktierungseinheit als Teil der Auswertungskomponente (ORA) [Wan06]. Der BIST-Controller erzeugt Steuer- und Clock-Signale, um TPG, CUT und ORA zu betreiben. Testantworten werden im ORA zu Signaturen kompaktiert und mit erwarteten Gutwerten, als *Golden Signatures* bezeichnet, verglichen. Bei einem *Pass/Fail*-Test genügt die Aussage, ob ein Fehler während des Tests aufgetreten ist oder nicht. Darüber hinaus kann ein BIST auch eine Fehlerdiagnose umsetzen, um als Basis für Sicherheits- oder Reparaturmaßnahmen zu dienen.

Der industriell am weitesten verbreitete BIST ist die STUMPS-Architektur (*Self-test Using MISR and Parallel Shift Register Sequence Generator*) [BM82]. Diese besteht aus einem *Shift Register Sequence Generator* (SRSG) zur Erzeugung der Pseudozufallsmuster und einem *Multiple Input Signature Register* (MISR) zur Kompaktierung der Testantworten aus den Scan-Ketten. Im ursprünglichen STUMPS-Design sind *Line-*

ar Phase Shifter und *Linear Phase Compactor* nicht enthalten. Sind diese Module Teil des BISTs, so wird dies als STUMPS-basierte Architektur bezeichnet (Abbildung 2.14). Der in dieser Arbeit vorgestellte *Scan-Controller* setzt eine solche STUMPS-basierte Architektur um.

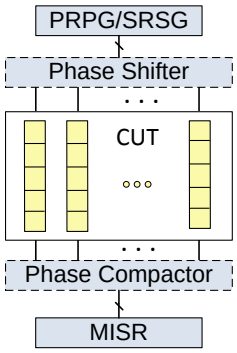


Abbildung 2.14: Schema der STUMPS-basierten Architektur

2.1.8 Komprimierung von Testdaten

Unter der für den BIST angewandten Komprimierung ist die verlustfreie Reduzierung des Testdatenumfangs zu verstehen. Das heißt, aus den komprimierten Informationen kann der komplette Datensatz reproduziert werden [BA00]. Bei der schaltungsinternen Erzeugung von Testmustern handelt es sich um eine Dekomprimierung, da aus wenigen Eingabeinformationen ein umfangreicher Datensatz berechnet wird.

Im ATPG-Prozess werden Testmuster durch pseudozufällige, pseudoerschöpfende oder deterministische Verfahren beziehungsweise durch Mixed-Verfahren erzeugt und für den BIST auf einen Satz an Initialisierungs- und Steuerinformationen komprimiert. Im IC werden während des BISTs die Testmuster nach äquivalenten Verfahren dekomprimiert. Die Testdatenkomprimierung kann in *Code*-basierte, *Linear-Decompression*-basierte und *Broadcast-Scan*-basierte Verfahren unterteilt werden. Im Folgenden soll aber nur das gebräuchliche Verfahren *Linear-Decompression* betrachtet werden, das ausschließlich auf linearen Operationen beruht.

Ein bekannter Ansatz hierfür ist ein Pseudozufallstest, bei dem sich die Testdaten im Grunde nur auf ein paar wenige Initialisierungswerte für den PRPG beschränken. Um aber eine möglichst hohe Fehlerüberdeckung zu garantieren, können deterministische Ansätze auf pseudozufallsbasierte Sequenzen, die gegebenenfalls zusätzliche Modifizierungen beinhalten, abgebildet werden. Dies wird dadurch ermöglicht, dass ein durch ein deterministisches ATPG-Verfahren ermittelter Testvektor üblicherweise eine Vielzahl unspezifizierter Bitstellen, sogenannte *Don't cares* oder X-Werte, enthält. Das bedeutet, der mittels Pseudozufallsverfahren ermittelte Testvektor muss nur in den wenigen spezifizierten Bitstellen mit dem deterministischen Testvektor übereinstimmen.

Da der Mustergenerator eine Vielzahl an Scan-Ketten betreiben soll, muss dieser entweder entsprechend groß ausgelegt sein, so dass die Datenbreite mindestens die Anzahl der Scan-Ketten umfasst, oder durch ein Verfahren ergänzt sein, dass den Testvektor aus einem kompakten Mustergenerator über die Scan-Ketten verteilt. Da der Mustergenerator über den kompakten ressourcensparenden Eingangsport der BIST-Architektur mit einem Startwert initialisiert und gegebenenfalls mehrmals während des Test mit neuem Startwert geladen wird, ist es effizient, diesen entsprechend kompakt mit gleicher Datenbreite auszulegen. So kann der Mustergenerator mit einem Startwert in einem Takt komplett beschrieben werden. Dies wirkt sich positiv auf die Testzeit aus, aber vor allem beansprucht der Satz an Startwerten weitaus weniger Speicherplatz. Die erzeugten Testvektoren müssen auf effiziente Weise entsprechend der Anzahl der Scan-Ketten erweitert werden, ohne aber die gleichmäßige Verteilung im resultierenden Pseudozufallsmuster negativ zu beeinflussen.

Eine etablierte Lösung hierfür ist ein XOR-Netzwerk, auch bekannt als *Linear Phase Shifter*, das den Testvektor auf die nötige Datenbreite expandiert. Hierbei wird jeweils ein weiteres Bit aus zwei oder mehr bereits erzeugten Bits über eine XOR-Verknüpfung erzeugt. Dadurch wird gewährleistet, dass kein fixes Abbildungsschema entsteht und eine gleichmäßige Verteilung der Bits innerhalb eines Testvektors beziehungsweise der durch die Testvektoren repräsentierten Binärwerte über dem Wertebereich erhalten bleibt.

Ein simpleres Verfahren ist das Betreiben mehrerer Scan-Ketten mit demselben Wert, in Anlehnung an den *Multiple-Input Broadcast-Scan* [SP04]. Dieses Verfahren macht sich die Eigenschaft der vielen X-Bits eines deterministischen Testvektors zu Nutze. Die Scan-Ketten sind in Gruppen unterteilt, die jeweils mit einer Bitstelle des Mustergenerators verknüpft sind. Die X-Bits des Testvektors werden dabei durch den Wert einer spezifizierten Bitstelle ihrer Gruppe aufgefüllt. Sind unterschiedliche Werte in derselben Scan-Kettengruppe erfordert, muss das starre Abbildungsschema wieder aufgebrochen werden. Dies kann zum Beispiel durch Rekonfiguration der Broadcast-Verschaltungen oder durch Modifikationen des erweiterten Testvektors geschehen.

Für Testmuster auf Basis von Pseudozufallsverfahren kann auch nach deterministischer Nachbearbeitung oder dem Hinzufügen zusätzlicher deterministischer Muster eine Komprimierung um einen Faktor größer 100, bezüglich eines deterministisch ermittelten Testsatzes, erreicht werden [WWP03a]. Die meistgenutzten Mustergeneratoren sind das *Linear Feedback Shift Register* (LFSR), der zelluläre Automat (*Cellular Automaton* - CA) [Wol83, KA87] und der Ringgenerator [RTK02]. Das LFSR wird im Folgenden näher betrachtet, da es auch die Grundlage für den Scan-Controller bildet.

2.1.8.1 Linear Feedback Shift Register

Das *Linear Feedback Shift Register* (LFSR) ist der gebräuchlichste Mustergenerator, um Testmuster für den pseudozufälligen Test, aber auch erschöpfenden oder pseudoerschöpfenden Test schaltungsintern zu erzeugen [Wan06]. Das LFSR ist ein Schieberegister, das um ein spezielles Rückkopplungsnetzwerk erweitert, eine pseudozufällige Sequenz

von Bitvektoren erzeugen kann. Das bedeutet, es nimmt in einer Sequenz Werte an, die sich durch eine gleichmäßige Verteilung über dem Wertebereich entsprechend der Datenbreite des LFSRs auszeichnen.

Hierzu werden, wie in Abbildung 2.15 dargestellt, einzelne Flipflop-Ausgänge über XOR-Verknüpfungen auf den ersten Flipflop-Eingang beziehungsweise auf mehrere Flipflop-Eingänge zurückgekoppelt. Die Struktur der Rückkopplung, das sogenannte *Feedback*, eines LFSRs der Länge n wird durch ein definiertes *charakteristische Polynom* $p(x)$ vom Grad n, mit den Koeffizienten $c_i \in \{0, 1\}$, bestimmt:

$$p(x) = x^n + c_{n-1}x^{n-1} + ... + c_2x^2 + c_1x + 1.$$

Es kann zwischen zwei LFSR-Schemata unterschieden werden, dem Extern-XOR und Intern-XOR LFSR. Bei dem *Extern-XOR LFSR*, auch als *Standard-LFSR* bezeichnet, sind die XOR-Gatter auf dem externen Feedback-Pfad angeordnet. Befinden sich hingegen die XOR-Gattern zwischen den Flipflops, handelt es sich um eine *Intern-XOR LFSR* oder auch *modulares LFSR*. Beim Standard-LFSR muss eine größere Verzögerung durch die kaskadierte XOR-Struktur berücksichtigt werden. Das modulare LFSR kann schneller betrieben werden, da es nur maximal ein XOR-Gatter auf einem Verbindungspfad zwischen zwei Flipflops besitzt.

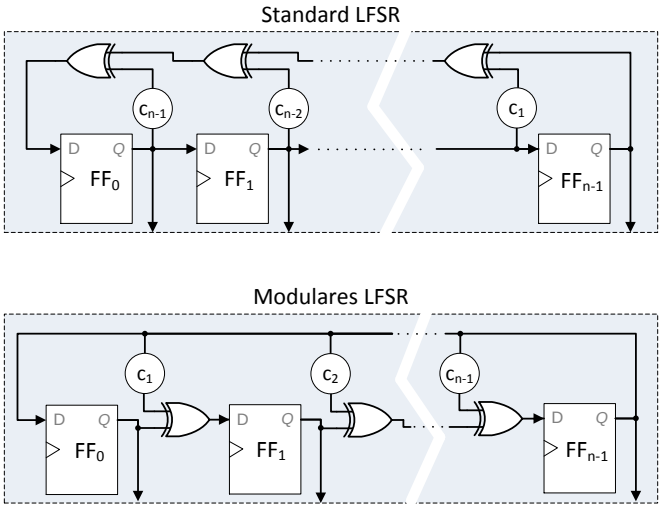


Abbildung 2.15: Aufbauschemata des Linear Feedback Shift Registers

Durch einen Schiebetakt des Registers wird in Abhängigkeit vom aktuellen Inhalt und dem Feedback ein bestimmtes Folgemuster erzeugt. Die Menge aufeinanderfolgender unterschiedlicher Bitvektoren ergeben eine LFSR-Sequenz. Die Sequenz beginnt mit einem definierten Startwert, auch als *Seed* bezeichnet, und durchläuft weitere neu erzeugte Bitmuster bis sich der Startwert wiederholt, also die Sequenz von vorn beginnt. Die Sequenzlänge gibt die in einer Sequenz enthaltenen Bitmuster an. In der maximal möglichen Sequenz sind somit alle $2^n - 1$ Vektoren, ausgenommen des 0-Vektors, enthalten. Der 0-Vektor, in dem alle Bits logisch 0 sind, wird immer auf sich selbst abgebildet, kann somit nicht Teil der maximalen LFSR-Sequenz sein. Es gibt aber auch

Lösungen, die mit Zusatzlogik den 0-Vektor erlauben, und so die maximale Sequenz auf 2^n Vektoren erweitern, um auch als Mustergeneratoren für den (pseudo-) erschöpfenden Test zu fungieren.

Für einen möglichst effizienten Testsatz muss ein Feedback gefunden werden, mit dem eine Sequenz der gewünschten Testmuster generiert werden kann. Durch eine lange Sequenz kann ein großer Testsatz erzeugt werden. Es müssen dabei möglicherweise aber viele unbrauchbare Testvektoren in Kauf genommen werden, die nicht zur Steigerung der Fehlerüberdeckung beitragen. Ein besserer Ansatz ist es mehrere effizientere kurze Sequenzen beziehungsweise Teilsequenzen zu kombinieren. Dies wird durch das sogenannte *Reseeding* erreicht [Kön91, JAR11]. Hierzu werden im ATPG-Prozess Startwerte für günstige Sequenzen, mit vielen relevanten Mustern, berechnet, um das LFSR während des Testlaufs mehrfach zu initialisieren und so die gewünschten Sequenzen einzustellen. Das heißt, ist eine Sequenz komplettiert oder gibt nicht mehr genügend verwertbare Testvektoren her, kann mit einer neuen Sequenz brauchbarer Muster fortgefahren werden. Somit kann durch pseudozufällige Testmuster ein möglichst großer Teil eines entsprechenden deterministischen Testsatzes abgedeckt werden. Auf dem IC muss hierzu nur ein kompakter Satz an Startwerten abgespeichert werden.

Mit einem einzigen charakteristischen Polynom lassen sich möglicherweise nicht genügend günstige Sequenzen erzeugen. Ist neben dem Startwert auch das charakteristische Polynom frei wählbar, steht im ATPG-Prozess ein umfangreicherer Suchraum zur Verfügung, um ein Großteil des deterministischen Testsatzes auf LFSR-Sequenzen abzubilden. Das bedeutet, anstatt einer fest verdrahteten Rückkopplung ist ein konfigurierbares Feedback-Netzwerk zu nutzen. Das *Multiple-Polynomial LFSR* (MP-LFSR) bietet diese Möglichkeit der Rekonfiguration [HTR92]. Jedem Startwert wird hierfür zusätzlich ein Feedback-Wert, der Polynom-Identifikator, zugeordnet, um zur Reinitialisierung des LFSRs die nötige Rekonfiguration durchzuführen.

2.1.8.2 Deterministische Musteroptimierung

Trotz der Rekonfigurations- und Reseeding-Verfahren kann es schwer bis unmöglich sein, Sequenzen zu finden, die bestimmte deterministische Muster für RP-resistente Fehler enthalten. RP-resistente Fehler erfordern für eine effiziente Behandlung deterministische Maßnahmen. Pseudozufällige Testsätze enthalten üblicherweise viele redundante beziehungsweise unbrauchbare Testvektoren, die keine neuen Fehler überdecken. Diese Testvektoren können für deterministische Modifikationen genutzt werden. Das heißt, mittels Zusatzlogik können unbrauchbare Zufallsvektoren in deterministische Testvektoren umgewandelt werden, um so auch RP-resistente Fehler zu entdecken [TM95]. Ein bekanntes Verfahren hierfür ist das sogenannte *Bit-Flipping* [WK96, KW98]. Dabei wird der erzeugte Testvektor, bevor dieser an die Scan-Ketten angelegt wird, mit einem Wert aus einer Bit-Flipping-Einheit über eine XOR-Verknüpfung an spezifizierten Bitpositionen invertiert. Somit können deterministische Muster in pseudozufällige Sequenzen eingebettet werden.

2.1.9 Kompaktierung von Testausgaben

Die im BIST angewandte Kompaktierung ist eine Methode, den Datenumfang der Testausgaben aus der CUT erheblich zu reduzieren. Allerdings ist die Kompaktierung von Daten verlustbehaftet [Wan06]. Das heißt, da Informationen verloren gehen, ist die Kompaktierung ein irreversibles Verfahren. Für den Test bedeutet dies eine Beschränkung der diagnostischen Auflösung.

Kompaktierung kann sequentiell über die Zeit, als *Time Compaction* bezeichnet, oder auch kombinatorisch, als *Space Compaction* bezeichnet, geschehen. Bei *Space Compaction*-Verfahren werden Testantworten aus m Ausgängen der CUT auf n Testausgänge mit $n < m$, in einem Zeitschritt abgebildet. Das bedeutet, diese Kompaktoren bestehen aus rein kombinatorischer Logik. Das bekannteste Verfahren dieser Art ist der XOR-Tree. Dabei werden jeweils mehrere Scan-Ausgänge über kaskadierte XOR-Verknüpfungen auf eine Bitstelle der resultierenden Testantwort abgebildet.

Die *Time Compaction*-Verfahren nutzen sequentielle Elemente, um mehrere Testantworten über einen bestimmten Zeitraum auf eine einzige Testantwort, die sogenannte *Signatur*, abzubilden. Dies kann durch einfache Zähler, wie beim *One Count Testing* (OC) oder *Transition Count Testing* (TC), oder Akkumulatoren geschehen. Beim OC-Verfahren werden die 1-Werte und beim TC-Verfahren die Transitionen des Ausgabestroms eines Testausgangs zu einer Signatur zusammengezählt. In Akkumulatoren werden die Testantworten aus der CUT aufaddiert. Diese einfachen Verfahren sind aber anfällig für die Fehlermaskierung durch Mehrfachfehler. Enthält beispielsweise eine Testantwort ein fehlerhaftes Bit a_i und der aufzuaddierende Wert an derselben Bitposition i ein fehlerhaftes Bit b_i, mit $a_i = \overline{b_i}$, entspricht die Summe dieser fehlerhaften Werte der Summe der fehlerfreien Werte. Fehler in Schaltungen sind typischerweise nicht gleichverteilt, treten also mit erhöhter Wahrscheinlichkeit konzentriert an gleicher Position in den Testausgaben auf. Daher sind einfache Zähler oder Addierer besonders anfällig für diese auf Schaltungsbereiche konzentrierten Mehrfachfehler. Das heißt, die Wahrscheinlichkeit, dass Mehrfachfehler in der korrekten Summe für die Analyse resultieren, kann unbefriedigend hoch ausfallen.

Einen besseren Ansatz bieten Verfahren, die eine Gleichverteilung der Bits der Testausgaben während der Kompaktierung ermöglichen, wie die *Signaturanalyse* basierend auf *Cyclic Redundancy Checking* (CRC) [BCA75]. Der am weitesten verbreitete Kompaktor dieser Art ist das *Multiple Input Signature Register* (MISR) [Wan06].

Ferner gibt es heutzutage auch viele kombinierte Ansätze, sogenannte *Mixed Time and Space Compaction*, die aus mehreren Testantworten erzeugte Signaturen weiter in ihrer Datenbreite reduzieren. Beispiele hierfür sind der OPMISR+ [BBD02] oder der Faltungs-Kompaktor [RTW05].

2.1.9.1 MISR

Das *Multiple Input Signature Register* (MISR) dient der Kompaktierung umfangreicher Testausgaben zu Signaturen für die Testanalyse. Das MISR entspricht üblicherweise einem modularen LFSR, kann demnach durch das *charakteristisches Polynom* $p(x)$ vom Grad n, mit den Koeffizienten $c_i \in \{0, 1\}$, beschrieben werden:

$$p(x) = x^n + c_{n-1}x^{n-1} + ... + c_2x^2 + c_1x + 1.$$

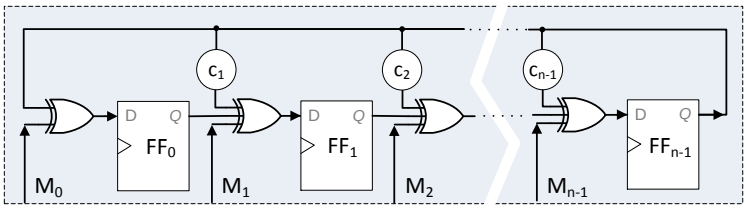


Abbildung 2.16: Aufbau des Multiple Input Signature Registers

Die Testantworten aus den Scan-Ketten werden dabei über die parallelen Eingänge M_0 bis M_{n-1} eingegeben und mit dem Inhalt über XOR-Verknüpfungen kombiniert (Abbildung 2.16). Ein MISR der Länge n kann die Testausgaben von maximal n Scan-Ketten in die zu bildende Signatur aufnehmen. Sollen sämtliche Scan-Ketten einem MISR zugeordnet werden, muss es entsprechend groß ausgelegt werden. Es gibt aber auch Ansätze mit mehreren modularisierten MISR-Einheiten, die jeweils Signaturen für die Testausgaben einer Gruppe von Scan-Ketten erzeugen.

Das MISR kann auf ein *Single-Input Signature Register* (SISR), bei dem nur der Eingang M_0 zur Verfügung steht, zurückgeführt werden [HM84]. Ein SISR stellt einen einfachen CRC-Generator beziehungsweise -Checker dar [BCA75]. Der aus den eingelesenen Eingaben ermittelte CRC-Code, also der Rest der Polynomdivision des Eingabepolynoms $M(x)$ durch das charakteristische Polynom $p(x)$, ist die kompaktierte Signatur. Anstatt eines Eingabestroms kann das MISR mehrere Eingabeströme parallel verarbeiten und so eine Signatur in entsprechend kürzerer Zeit erstellen.

Ein entscheidender Punkt bei der Kompression ist die Betrachtung möglicher Fehlermaskierungen. Das heißt, es muss eine Aussage getroffen werden können, wie wahrscheinlich es ist, dass fehlerhafte Testantworten in einer korrekten Signatur resultieren. Sei m die Anzahl an Scan-Ausgängen und k die Anzahl der m-Bit-Testantworten, die in einer Signatur zu kompaktieren sind, mit $k > n \geq m \geq 2$. Es gibt 2^{mk} Kombinationen von Testantworten aus den Scan-Ketten, die in die Signaturbildung eingehen. Nur eine davon ist die korrekte Testantwortkombination, die aus einer fehlerfreien Schaltung hervorgeht. Das heißt, in $2^{mk} - 1$ Testantworten wird mindestens ein Fehler erkannt. Auf eine n-Bit-Signatur werden aber $\frac{2^{mk}}{2^n}$ Testantwortkombinationen abgebildet. Ausgenommen der korrekten gehen also in die Gutsignatur $\frac{2^{mk}}{2^n} - 1 = 2^{mk-n} - 1$ weitere Testantwortkombinationen ein. Somit ist die Wahrscheinlichkeit P für eine Fehlermaskierung:

$$P(n) = \frac{2^{mk-n}-1}{2^{mk}-1}.$$

Für $k \gg n$ kann vereinfacht werden auf:

$$P(n) \approx 2^{-n}.$$

Daraus folgt, dass für einer Vielzahl an Testantworten aus den Scan-Ketten, die in einer Signatur zu vereinen sind, die Wahrscheinlichkeit der Fehlermaskierung im Grunde von der Größe des MISRs abhängt. Im Allgemeinen ist festzuhalten, dass eine größere Auslegung des MISRs oder auch ein günstigeres charakteristisches Polynom $p(x)$ die Wahrscheinlichkeit der Fehlermaskierung erheblich reduzieren kann [HM84].

2.1.9.2 Maskierung unbekannter Testausgaben

Damit in der Auswertung eines BISTs eine korrekte Aussage durch den Vergleich der Testausgaben mit Gutwerten erfolgen kann, müssen unbekannten Ausgaben aus der Schaltung, sogenannte X-Werte, beachtet und entsprechend behandelt werden. Im Folgenden sind typische Gründe für das Entstehen von X-Werten aufgelistet [CFK04]:

- Es gibt Baugruppen, die sich nicht dem Scan-Test angepasst modellieren lassen. Diese vom Testdesign nicht erfasste Logik oder Black-Boxes geben häufig X-Werte aus.
- Interne Tristate-Logik kann hochohmige Zustände (High-Z) annehmen. Da die in Testantworten eingehenden High-Z-Werte nicht interpretiert werden können, müssen diese als X-Werte aufgefasst werden.
- In DFT-Schaltungen können auch sequentielle Elemente (Flipflops/ Latches) vorkommen, die nicht in die Scan-Struktur integriert sind und vor dem Scan-Test auch nicht initialisiert werden können.
- Eingebettete RAM-Blöcke werden für den BIST normalerweise durch ein RAM-Bypass aus dem Logiktest herausgenommen. Es kann jedoch von Vorteil sein, die RAM-Zugriffe mit deren Verzögerungen in den Test zu integrieren. Die aus dem RAM gelesenen unbekannten Werte fließen aber so in die Testantworten mit ein.
- In Verzögerungstests (*At-Speed Delay Tests*) gibt es häufig Pfade, die das funktionale Timing nicht erfüllen. Gründe dafür sind Verletzungen der Setup- und/oder Hold-Zeiten von Gattern, Multi-Zyklen oder nicht-funktionale Pfade. Da nicht feststellbar ist, ob solch ein verzögerter Wert durch die Testantwort erfasst wird, stellt dieser einen X-Wert dar.

Enthält eine Testantwort solche X-Werte, dürfen diese nicht in die Fehleranalyse mit einfließen, da diese die Aussage verfälschen und, im Falle einer Diagnose, auch keine Rückschlüsse auf eine mögliche Fehlerursache zulassen. Im DFT-Entwurf muss bereits darauf geachtet werden, solche X-Werte in Testausgaben nach Möglichkeit zu vermeiden beziehungsweise auf das nötige Minimum zu reduzieren.

Werden die Testantworten vor der Auswertung kompaktiert, können X-Werte zu ungültigen Signaturen führen. Daher wurden diverse Kompaktierungsverfahren entwickelt, die dieses Problem umgehen. Beim *X-Blocking* [NPR03] wird ein unbekannter Wert in der Schaltung auf dem Pfad zwischen seiner Quelle und den Testausgang mittels integrierter Zusatzlogik blockiert und durch einen definierten Wert ersetzt. *X-Masking*-Verfahren [WWP03b, WWP04] filtern unbekannte Ausgaben mittels Maskierungslogik direkt vor der Kompaktierungseinheit. Das bedeutet, die Bitstellen mit X-Werten in den Testantworten werden auf einen definierten Wert gesetzt. Es können dabei einzelne Bitstellen oder ganze Abschnitte der Testantwort adressiert werden, um diese auszublenden.

Es gibt weitere Verfahren, die sich die Kombinatorik der *Space Compaction* zu Nutze machen. *X-tolerante* Kompaktierungseinheiten wie *X-Compact* [MLM04] sind entsprechend der Schaltung und ihrer X-Quellen ausgelegt, um X-Werte durch geschickte kombinatorische Verschaltung zu filtern. Ferner gibt es auch algorithmische Lösungen, wie *X-Impact* [Wan06], die X-Werte mittels ATPG-Verfahren behandeln.

2.2 Testschnittstellen

2.2.1 JTAG

Die heute als JTAG bekannte Testschnittstelle wurde von der 1985 gegründeten *Joint Test Action Group* (JTAG) entwickelt und im Jahr 1990 schließlich unter der Bezeichnung *IEEE 1149.1 „Standard Test Access Port and Boundary Scan Architecture“* standardisiert. Der aktuell gültige Standard ist die überarbeitete und im Jahr 2001 veröffentlichte Version *IEEE 1149.1-2001* [JTG01]. Das Ziel der JTAG-Entwicklung war es, eine einheitliche und kostengünstige Lösung für den Leiterplattentest (PCB-Test) bereitzustellen. Daher wird der Name der Schnittstelle auch synonym für das *Boundary-Scan*-Verfahren, dem statischen Verbindungstest auf Systemebene, verwendet. JTAG ist heutzutage ein etablierter, weitverbreiteter Standard und wird von den meisten ICs unterstützt [REP05].

Der Aufbau einer JTAG-Schaltung ist in Abbildung 2.17 dargestellt. Die Hauptkomponenten sind:

- *Test Access Port* (TAP),
- TAP-Controller,
- Befehlsregister und
- mehrere Datenregister.

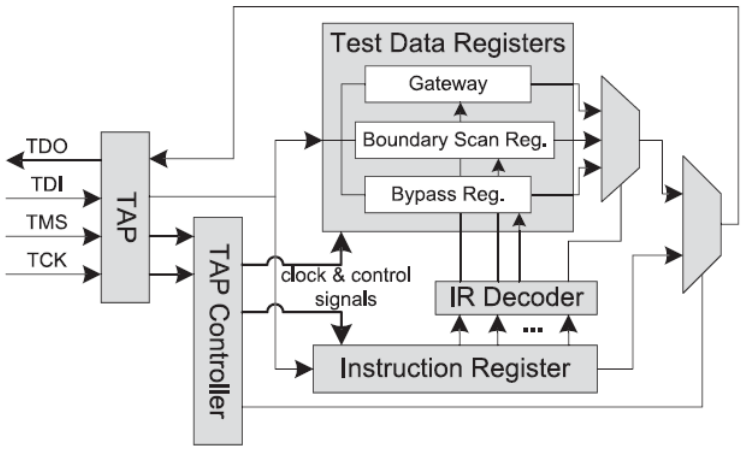


Abbildung 2.17: JTAG - Schaltungslogik [LZ12]

Der TAP ist die Schnittstelle zur schaltungsinternen JTAG-Logik und besteht aus folgenden Ports:

- *Test Clock* (TCK),
- *Test Mode Select* (TMS),
- *Test Data Input* (TDI),
- *Test Data Output* (TDO) und
- *Test Reset* (TRST).

Über den TCK-Port wird die systemweite Test-Clock eingegeben, um während des Testmodus die Teststruktur zu betreiben. Mittels des TMS-Signals wird der TAP-Controller und somit der Testzugang gesteuert. Die Testdaten werden über die seriellen Ports TDI und TDO hinein- beziehungsweise hinausgeschoben. Der TRST-Port ist ein optionaler Eingang, der eine asynchrone Initialisierung der JTAG-Komponenten ermöglicht. In Abbildung 2.18 ist das Zustandsdiagramm des TAP-Controllers zu sehen. Dieser erzeugt in Abhängigkeit vom TMS-Signal die internen Steuerungssignale, um den Test zu konfigurieren und die nötigen Testphasen einzustellen. Die Zustände sind in einen Instruktions- und einen Datenpfad unterteilt, um entweder das Befehlsregister oder das Datenregister anzusteuern.

Die wichtigsten Datenregister sind das Bypass-Register, das nur ein Bit umfasst, und das Boundary-Scan-Register, das den Scan-Pfad für die an die Schaltung anzulegenden Testmuster darstellt. Durch das Befehlsregister wird die Konfiguration der Scan-Struktur und somit der Testmodus bestimmt. Die dafür nötigen Befehle sind `BYPASS`, `SAMPLE/PRELOAD` und `EXTEST`. Im Bypass-Modus wird der Datenpfad von TDI zu TDO direkt über das Bypass-Register geschlossen. Die anderen Modi aktivieren die Testphasen, um das Boundary-Scan-Register seriell über den TDI-Port zu laden oder parallel für einen funktionalen Takt zu betreiben.

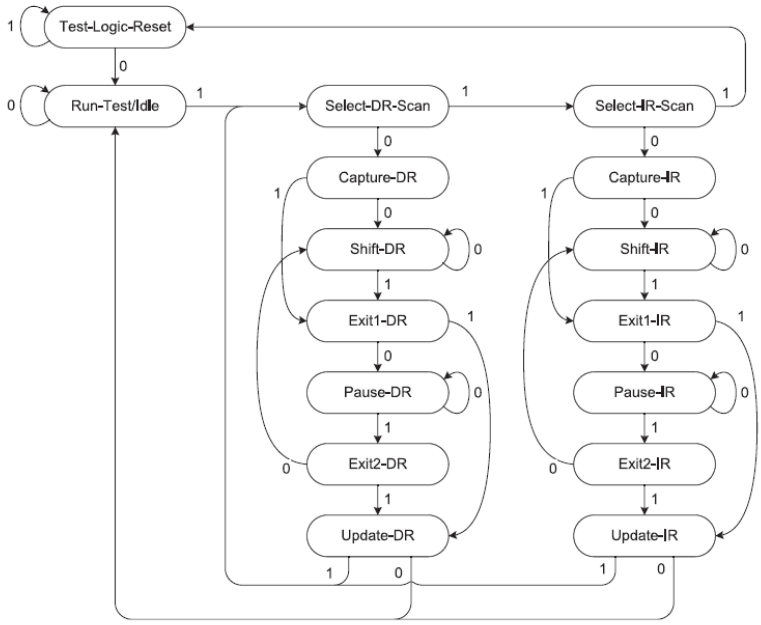


Abbildung 2.18: JTAG - TAP-Zustandsdiagramm [LZ12]

Boundary-Scan-Test

Boundary-Scan bezeichnet das über die JTAG-Schnittstelle realisierte Verfahren zum Testen digitaler und analoger Bausteine auf Systemebene. Im IEEE1149.1-Standard sind der Aufbau der erforderlichen DFT-Erweiterung und die zugehörige Hardware-Beschreibungssprache *Boundary Scan Description Language* (BSDL) definiert. Mit diesem Verfahren sollen im PCB-Test ICs, aber auch weitere Module, wie Treiber, Widerstände, RAM-Blöcke oder Flashspeicher, getestet werden. So kann beispielsweise das korrekte Aufbringen des ICs auf die Leiterplatte sichergestellt werden. Daher wird in erster Linie die Kommunikation zwischen IC und Leiterplatte getestet.

Hierzu sind spezielle *Boundary-Scan-Zellen* zwischen der Schaltungslogik und den Pins integriert. Im Normalbetrieb haben diese Zellen keinen Einfluss auf die Funktionalität. Im Test- beziehungsweise Debug-Modus werden die Zellen umgeschaltet, so dass die Pins des ICs von der Schaltungslogik getrennt sind. Die Boundary-Scan-Zellen sind nun zu einem langen Schiebepfad, dem sogenannten Scan-Pfad, verknüpft. Der Test wird dabei über den TMS-Ports gesteuert, um so mittels des TAP-Controllers die für den Testablauf nötigen Operationen ausführen zu lassen. Testdaten werden in der *Stift*-Phase (Instruktion `SAMPLE/PRELOAD`) über den TDI-Port in den Scan-Pfad geschoben. Es folgt die *Capture*-Phase (Instruktion `EXTEST`) mit einem funktionalen Takt. Anschließend werden die Testantworten wieder über den TDO-Port ausgelesen. Während des Hinausschiebens der Testantworten, werden bereits neue Testdaten über den TDI eingegeben. Auch das Testen mehrerer ICs ist möglich, insofern diese sich im selben Scan-Pfad befinden. In diesem Fall ist das Bypass-Register von entscheidender Bedeu-

tung, um nicht stets den kompletten Scan-Pfad durchlaufen zu müssen. So können Teilpfade überbrückt und einzelne ICs selektiv getestet werden.

Nähere Informationen zum JTAG-Design und zu den dadurch realisierbaren Testverfahren sind in [JTG01, Sto11, Par03] zu finden.

2.2.2 IJTAG

Die Entwicklung eines neuen Standards für den Testzugang, der den gestiegenen Bedürfnissen des Chiptests gerecht wird, ist 2004 durch die *IEEE Test Technology Standards Group* beschlossen worden. Das als *Internal JTAG* (IJTAG) bekannte Konzept befindet sich derzeit in einer Standardisierungsphase unter der Bezeichnung *IEEE P1687 "Draft Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device"*.

Die Idee ist es den bestehenden JTAG-TAP für die Kommunikation mit on-chip Testmodulen zu nutzen, somit also eine Verbindung zwischen der Systemebene und der Komponentenebene zu schaffen. IJATG umfasst eine neue Testschnittstelle zu den internen Testmodulen, sogenannte *Test Instruments* (TI), und die standardisierten Beschreibungssprachen *Instrument Connectivity Language* (ICL) und *Procedure Description Language* (PDL). ICL ist eine Hardware-Beschreibungssprache, die der Spezifizierung der neuen Scan-Strukturen, also der Verknüpfung des Testzugangs mit den internen Testmodulen, dient. Mittels PDL können die durch die Testmodule auszuführenden Operationen beschrieben werden, um einen bestimmten Test einer internen Komponente umsetzen zu lassen.

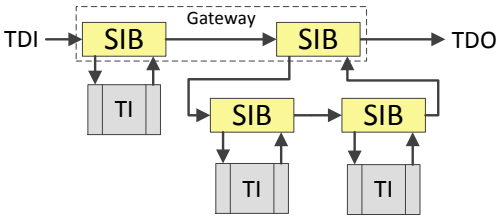


Abbildung 2.19: IJTAG - Scan-Pfad

Das in Abbildung 2.17 enthaltene *Gateway*-Register ist eine Erweiterung der JTAG-Schaltung und stellt die Schnittstelle zu internen IJTAG-Strukturen dar. Dieses Gateway-Register besteht aus sogenannten *Segment Insertion Bits* (SIB), die den Zugang zu den internen Testmodulen realisieren. Im entsprechenden Testmodus bilden die SIBs den Scan-Pfad, um ein selektiertes Testmodul anzusteuern und mit den über TDI seriell hineingeschobenen Testdaten betreiben zu können. In Abbildung 2.19 ist ein Scan-Pfad einer IJTAG-Schaltung dargestellt. Mittels der SIBs kann die Scan-Struktur auch hierarchisch ausgelegt werden, indem ein SIB statt eines Testmoduls einen weiteren Scan-Pfad einbindet. So muss für den Zugang zu einem Testmodul höherer Priorität nicht der

komplette Scan-Pfad durchlaufen werden, da Teilpfade unterer Hierarchieebenen über einen Bypass ausgeschlossen werden können. Das Gateway-Register besteht in diesem Fall nur aus den SIBs der obersten Hierarchieebene.

Ausführliche Informationen zu diesem neuen Standard sind unter anderem in [IJT13, Sto11, REP05, LZ12] zu finden.

2.3 Standardbussysteme

2.3.1 Überblick und Vergleich

Die heute meistgenutzten Übertragungssysteme in der elektronischen Datenverarbeitung sind USB und FireWire. Da es standardisierte seriellen Bussysteme sind, werden diese hier für den Testzugang in Betracht gezogen. In Tabelle 2.3 sind die etablierten Versionen USB 2.0, USB 3.0 und USB 3.1, sowie FireWire IEEE1394a und IEEE1394b gegenübergestellt.

Der Vorteil von FireWire gegenüber USB sollte die höhere Übertragungsrate sein. FireWire war ursprünglich für die schnelle Verbindung von Digitalkameras und Video-Schnittrechner ausgelegt. Der Schwerpunkt lag bei Audio- und Video-Anwendungen. Es stellte sich heraus, dass genauso gut auch Massenspeicher per FireWire an PCs und Notebooks anzuschließen sind. Die Datentransferrate des FireWire S400 wurde durch die USB HighSpeed-Version übertroffen.

Mit den neuen Protokollen S800 (IEEE1394b, 800 MBit/s) und S3200 (IEEE1394-2008, 3,2 GBit/s) erzielen FireWire-Verbindungen eine enorme Durchsatzsteigerung. FireWire IEEE1394b ist ein neu entwickeltes Protokoll, das auf IEEE1394a basiert, in dem unter anderem ein neues Arbitrierungsverfahren definiert, außerdem andere Signalkodierung und Signalpegel genutzt werden. Mit FireWire IEEE1394-2008 wurde eine vollständig überarbeitete Version als neuer Standard definiert. Abwärtskompatibilität der neuen Version zu ihren Vorgängern kann nur durch bilinguale Chips gewährleistet werden. Der neue SuperSpeed-Modus des USB 3.0-Protokolls, mit einer Datenrate von bis zu 5 GBit/s, holt aber auch das Geschwindigkeitsdefizit wieder auf. Per Spezifikation sind auch alle USB-Versionen abwärtskompatibel gehalten [USB00, USB08]. Mit der Weiterentwicklung zu USB 3.1 (Superspeed+) wurde die Datenrate sogar verdoppelt.

Im Allgemeinen divergieren die beiden Bussysteme bezüglich der Übertragungsparameter nur in wenigen Punkten. USB stellt die kostengünstigere Lösung dar und ist heute als Schnittstelle zu DV-Systemen weitaus verbreiteter [Axe07]. Für die Umsetzung des Konzepts eines Testzugangs über Standardschnittstellen wird deshalb die USB- einer FireWire-Schnittstelle vorgezogen.

2.3.2 Universal Serial Bus

Der *Universal Serial Bus* (USB) wurde als Standardschnittstelle für den EDV-Bereich konzipiert und entworfen. Viele ältere externe PC-Schnittstellen, sowohl serielle (RS-232, PS/2), parallele als auch analoge, sollten durch diesen neuen Standard ersetzt werden. USB eignet sich für viele Geräte wie Massenspeicher, Drucker, Scanner, Webcams, Eingabegeräte, aber auch Grafikkarten und Monitore. Mittlerweile hat USB auch PCMCIA-Slots und externe SCSI-Schnittstellen weitgehend verdrängt. USB kann für Geräte mit geringem Stromverbrauch wie Mäuse, Tastaturen, aber auch 2,5″-Festplatten die Stromversorgung übernehmen.

	USB 2.0	**USB 3.0**	**USB 3.1**	**IEEE1394a FireWire/ i.Link**	**IEEE1394b Firewire-800**
Topologie	Stern (kaskadierbar), Host-basiert	Stern (kaskadierbar), Host-basiert	Stern (kaskadierbar), Host-basiert	Peer-to-Peer	Peer-to-Peer
Datenübertragungsraten	LowSpeed: 1.5 MBit/s FullSpeed: 12 MBit/s HighSpeed: 480 MBit/s	5 GBit/s	10 GBit/s	S100: 100 MBit/s S200: 200 MBit/s S400: 400 MBit/s	800 MBit/s
Buspower	5 V / 500 mA (100 mA im LowPower-Mode)	5 V / 900 mA (150 mA im LowPower-Mode)	5 V / 900 mA (150 mA im LowPower-Mode)	8-30 V / 1,5 A	8-30 V / 1,5 A
Max. Anzahl der Geräte	127	127	127	63	63
Hot plug	ja	ja	ja	ja	ja
Max. Kabellänge zwischen zwei Geräten	5 Meter	3 Meter	1 Meter	4.5 Meter (bei S400)	4.5 Meter (Kupfer), 100 Meter (Glasfaser)
Übertragungsarten	asynchron/ isochron	asynchron/ isochron	asynchron/ isochron	asynchron/ isochron	asynchron/ isochron

Tabelle 2.3: Vergleich USB und FireWire

Die Entwicklung ist von namhaften Firmen wie Apple, Hewlett-Packard, NEC, Microsoft und Intel im Rahmen der dafür neu gegründeten Organisation *USB Implementers Forum Inc.* durchgeführt worden. Diese Organisation stellt sämtliche Spezifikationen frei zur Verfügung. Begonnen hat die Freigabe damit, dass im Januar 1996 die erste Version USB 1.0 nach einer mehrjährigen Entwicklung veröffentlicht wurde. Im September 1998 folgte die Version 1.1 mit einigen Nachbesserungen. Der nächste große Schritt für USB war die Version 2.0. Das Hauptziel bestand in der Erhöhung der Datenrate unter Berücksichtigung einer vollständigen Abwärtskompatibilität zu den vorherigen Versionen. Die USB 2.0 Spezifikation wurde im April 2000 veröffentlicht.

USB ist durch seine Konzeption deutlich leistungsfähiger, aber auch komplizierter als die traditionellen seriellen oder parallelen Computerschnittstellen. So gibt es verschiedene Übertragungsmodi, zum Teil mit automatischer Fehlerkorrektur, und verschiedene Datenkanäle mit konfigurierbaren Pufferspeichern.

Um die Entwicklung von Treibern erheblich zu vereinfachen, wurde das USB Klassenkonzept entwickelt. Hierzu werden USB Geräte ihren Eigenschaften nach in sogenannte Geräteklassen kategorisiert. Betriebssysteme haben so die Möglichkeit, diese Geräte zu identifizieren und die nötigen Standardtreiber zu laden, auch bekannt als *Hot-Plug*-Verfahren. Gebräuchliche Klassen sind zum Beispiel die *Human Interface Device Class* (HID), *Communication Device Class* (CDC), *Audio Device Class* (ADC) und *Mass Storage Class* (MSC). Die HID-Klasse wird beispielsweise benutzt, um Tastaturen oder auch Mäuse anzusteuern, CDC repräsentiert Daten von Kommunikationsschnittstellen, ADC bezieht sich auf Geräte mit Audio-, Sprach- oder Sound-Funktionalität, und MSC wird für Massenspeicher wie Festplatten oder USB-Sticks eingesetzt.

Jedes USB-Gerät besitzt einen zweiteiligen Identifikator, bestehend aus *Vendor-ID* (VID) und *Produkt-ID* (PID), der es erlaubt das Gerät eindeutig zu identifizieren. Somit wird die Zuordnung des entsprechenden Treibers durch das Betriebssystem ermöglicht. Für USB-Geräte mit derselben VID-PID-Kombination wird also auch derselbe Treiber installiert. Die VID, eine 16-Bit-Nummer, muss über das *USB Implementers Forum* erworben werden und gilt damit als weltweit registriert. Die PID, ebenso 16-Bit, kann durch den Hersteller des Endgerätes festgelegt werden.

Im Folgenden soll ein Überblick über Aufbau und Funktionsweise gegeben und wichtige Begrifflichkeiten erläutert werden. Da für die prototypische Umsetzung des Konzepts des Testzugangs eine USB FullSpeed-Schnittstelle implementiert wurde, bezieht sich die Einführung hauptsächlich auf diese Version. Detaillierte Informationen finden sich in den Spezifikationen [USB00, USB08] oder in [Axe05, Axe07, Gre09].

2.3.2.1 USB-Struktur

Trotz seines Namens handelt es sich bei USB nicht um einen physischen Datenbus. Bei einem solchen werden mehrere Geräte parallel an eine Leitung angeschlossen. Die Bezeichnung „Bus“ bezieht sich hier auf die logische Vernetzung, die tatsächliche elektrische Umsetzung erfolgt nur mit *Punkt-zu-Punkt*-Verbindungen. Ein zentraler Host

(*Master*) übernimmt die Koordination der angeschlossenen Peripheriegeräte (*Slaves*). Es können theoretisch durch entsprechende Hubs bis zu 127 Geräte an einen Host angeschlossen werden. Da über einem USB-Port nur ein USB-Gerät verbunden werden kann, muss im Falle mehrerer anzuschließenden Geräte ein Hub für deren Kopplung sorgen.

Topologisch existieren für USB zwei Busmodelle. Das physikalische, welches die Struktur als Baum darstellt, und das logische, welches die Struktur als Stern-Architektur abbildet. Beim physikalischen Modell bildet der Host die Wurzel des Baumes. Die Hubs entsprechen den Verzweigungsknoten und die Endgeräte den Blättern. Im logischen Modell bildet der Host den zentralen Mittelpunkt, an dem jedes Endgerät direkt angeschlossen ist.

Die Schnittstelle zum Host-Computer-System wird als Host-Controller bezeichnet, welcher normalerweise als Kombination von Hardware, Firmware und Software implementiert wird. Ein sogenannter Root-Hub ist im Host-System integriert, um schon Anschlussmöglichkeiten für ein oder mehr Endgeräte zu ermöglichen. Ein Endgerät ist normalerweise ein separates Peripheriegerät, das mit einem spezifikationskonformen Kabel über einen Hub angeschlossen wird. Es ist jedoch auch möglich, ein physikalisches Gerät mit mehreren Funktionen zu versehen, die sonst in verschiedenen Endgeräten realisiert würden.

Endgeräte müssen eine standardisierte USB-Schnittstelle mit folgenden Merkmalen zur Verfügung stellen:

- Unterstützung des USB-Protokolls,
- Reaktion auf standardisierte USB-Operationen wie Konfiguration oder Reset,
- Bereitstellung von Informationen über ihre implementierten Funktionen.

Da es sich bei USB um einen Single-Master-Bus handelt, muss der Host jegliche Kommunikation initiieren und steuern. Dieser kann Daten in einen Pufferspeicher des Endgeräts schreiben, und Daten aus einem Pufferspeicher auslesen. Das Endgerät dagegen kann nicht selbständig dem Host melden, dass Daten zur Verfügung stehen oder dass Daten erwartet werden.

USB überträgt elektrische Signale und die Betriebsspannung über ein vieradriges Kabel. Die Signalübertragung erfolgt differentiell über 2 dieser Adern, D+ und D-. Der Datenfluss vom Host zu den Geräten wird als *Downstream*, von den Geräten zum Host als *Upstream* bezeichnet.

USB unterstützt laut der jeweiligen Spezifikationen folgende Datenraten:

- *LowSpeed*: 1.5 MBit/s, USB 1.0,
- *FullSpeed*: 12 MBit/s ab USB 1.1,
- *HighSpeed*: 480 MBit/s, ab USB 2.0,
- *SuperSpeed*: 5 GBit/s, ab USB 3.0,
- *SuperSpeed+*: 10 GBit/s, ab USB 3.1.

2.3.2.2 Endpunkte

Ein Endpunkte stellt das Ende eines Kommunikationskanals zwischen Host und einer USB-Funktion dar, ist somit eine Datenquelle beziehungsweise Datensenke des USB-Gerätes. Ein USB-Gerät besitzt unabhängige, eindeutig nummerierte Endpunkte. Endpunkte verfügen über Eigenschaften wie die Transferrichtung (*Downstream*/*Upstream*), die Transferart, die benötigte Bandbreite, die Zugriffsfrequenz, die benötigte Fehlerbehandlungsroutinen und die maximale Paketgröße.

Den Endpunkt *EP0* muss jedes USB-Gerät unterstützen, da dieser für Kontroll- und Statusanfragen vom Host zuständig ist. Als einziger Endpunkt ist *EP0* bidirektional ausgelegt, um Lese- als auch Schreibzugriff zu ermöglichen. Weiter Endpunkte sind stets unidirektional und werden ja nach Bedarf als Empfangspuffer oder Sendepuffer für eine Funktion des Endgerätes benötigt. Der virtuelle Kommunikationskanal zwischen einem Endpunkt und der Host-Software wird als *Pipe* bezeichnet.

2.3.2.3 Transaktion

Die Transaktion erfolgt in Datenpaketen nach einem *Handshake*-Verfahren. Hierzu ist die Kommunikation in Phasen, für Token-, Daten- und Handshake-Pakete, unterteilt. Der Aufbau dieser Pakete ist in Abbildung 2.20 dargestellt. Jedes zu übertragende Paket wird mit dem *Sync*-Feld, das der Synchronisierung dient, eingeleitet. Umschlossen wird das Paket von dem *Start-of-Packet*- (SOP) und dem *End-of-Packet*-Feld (EOP). Das SOP-Feld ist Teil des Sync-Feldes und das EOP-Feld markiert das Ende der Übertragung eines Pakets. Dem SOP-Feld folgt stets der Paket-Identifikator (PID), wie in Tabelle 2.4 aufgelistet. Wobei das PID-Feld die 4-Bit-PID und, aus Gründen der Erkennung von Übertragungsfehlern, zudem ihren invertierten Wert enthält.

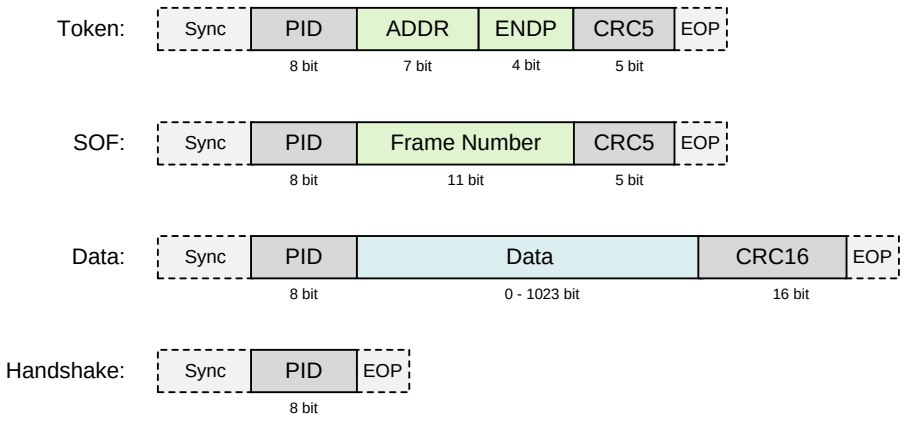


Abbildung 2.20: USB - Paketaufbau

PID-Typ	PID	Beschreibung
Token	SETUP	Setup-Transaktion zur Konfiguration des Interfaces/Endpunktes
	OUT	Transaktion vom Host zum Gerät
	IN	Transaktion vom Gerät zum Host
	SOF	*Start-of-Frame*-Paket markiert zeitliche Busabschnitte
Daten	DATA0	Datenpaket gerader Parität
	DATA1	Datenpaket ungerader Parität
Handshake	ACK	Fehlerfreier Empfang
	NAK	Anfrage bzw. Datenpaket kann noch nicht akzeptiert werden
	STALL	Gerätefehler

Tabelle 2.4: USB - Paket-Identifikator

In einem Token-Paket sind der Pakettyp durch die PID und der Empfänger angegeben. Der Empfänger des Paketes wird durch die jedem Gerät eindeutig zugeordnete Adresse und dem jeweiligen Geräte-Endpunkt bestimmt. Die korrekte Paketreihenfolge der Daten-Pakete wird durch eine abwechselnde Daten-ID (*DATA0 / DATA1*) gewährleistet. Ein Token-Paket wird durch eine 5-Bit-CRC und ein Datenpaket durch eine 16-Bit-CRC gesichert. Durch das Handshake-Paket wird eine der PIDs *ACK*, *NAK* oder *STALL* übermittelt. Korrekt empfangene Daten werden durch ein *ACK*, verfrühte Anfragen, wenn das Gerät nicht betriebsbereit ist oder die angefragten Daten fehlen, durch ein *NAK* und ein Gerätefehler durch ein *STALL* quittiert.

Jeder Endpunkt ist für eine Transferart ausgelegt. Das USB-Protokoll unterstützt die folgenden vier Arten des Datenflusses:

- *Control-Transfer*,
- *Bulk-Transfer,*
- *Interrupt-Transfer,*
- *Isochron-Transfer.*

Control-Transfer

Kontroll-Daten werden von der USB-Software benötigt, um ein Gerät zu konfigurieren oder Statusinformationen anzuzeigen. Die Datenübertragung erfolgt dabei verlustfrei und fehlerkorrigiert. Ein Control-Transfer ist in drei Kommunikationsstufen unterteilt, wie in Abbildung 2.21 dargestellt. Dieser wird eingeleitet durch eine Setup-Transaktion (*Setup Stage*) gefolgt von mehreren IN- oder OUT-Transaktionen (*Data Stage*). Abgeschlossen wird der Transfer von einer Status-Transaktion (*Status Stage*), um die korrekte Übertragung zu bestätigen.

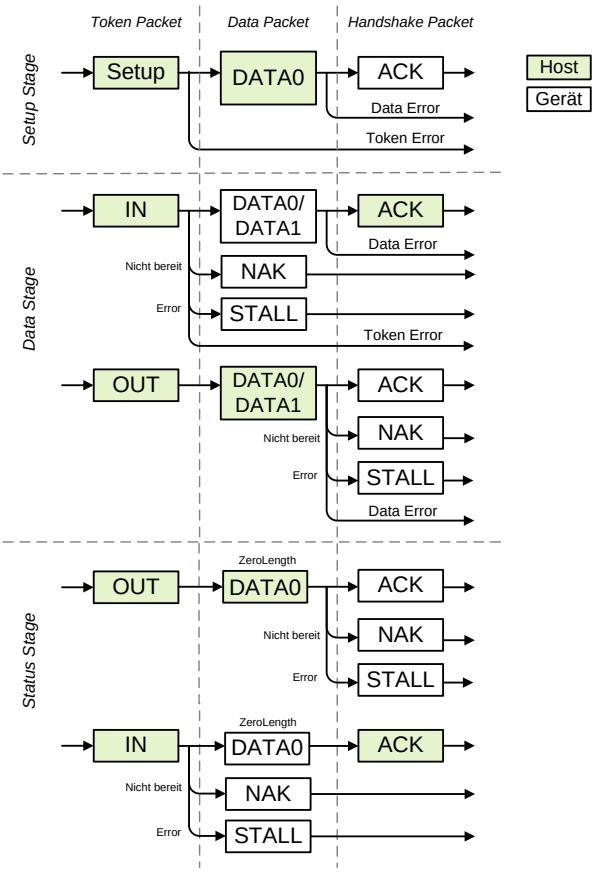


Abbildung 2.21: USB - Control-Transfer

Interrupt-Transfer

Für kleine aperiodisch auftretende Daten wird diese Art der Übertragung verwendet. Es findet ein verlustfreier, fehlerkorrigierter Transfer mit garantierter Latenzzeit, aufgrund periodischen *Pollings* statt. Das Zeitintervall, in dem der Host Daten anfragt, wird durch den Interrupt-Endpunkt vorgegeben. Die Daten werden dabei mit der vollen Geschwindigkeit übertragen, die das Gerät unterstützt. Interrupt-Transfers finden normalerweise in Verbindung mit auftretenden Ereignissen Verwendung. Der Interrupt-Transfer besteht im Grunde nur aus mehreren aufeinanderfolgenden IN-Transaktionen (Abbildung 2.22). Das heißt, der Datenfluss währende des Interrupt-Transfer ist unidirektional.

Bulk-Transfer

Mittels Bulk-Transfer werden typischerweise große zeitunkritische Datenmengen verlustfrei übertragen. Ordnungsgemäßer Datenaustausch wird auf Hardware-Ebene durch Fehlererkennung sichergestellt. Im Bedarfsfall können auch begrenzt Wiederholungen

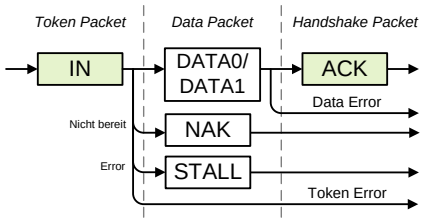


Abbildung 2.22: USB - Interrupt-Transfer

einer Transaktion ausgeführt werden. Eine feste Bandbreite ist für diese Art der Datenübertragung nicht vorgesehen. Es wird keine feste Latenzzeit garantiert, das heißt, ist der Bus voll ausgelastet, so wird mit der Übertragung solange gewartet, bis wieder Kapazitäten vorhanden sind. Der Bulk-Transfer besteht aus mehreren IN- und/oder OUT-Transaktionen (Abbildung 2.23).

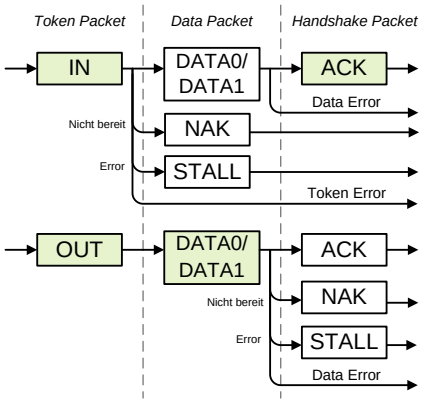


Abbildung 2.23: USB - Bulk-Transfer

Isochron-Transfer

Die mit der isochronen Transferart übertragenen Daten sind fortlaufend und werden in Echtzeit erstellt, übermittelt und empfangen. Timing-Informationen werden dabei mit übertragen. Ein isochroner Datenstrom muss mit der Rate geliefert werden, mit der dieser empfangen werden soll. Die dabei für die jeweilige *Pipe* benötigte Bandbreite ist abhängig von der Sampling-Rate des USB-Gerätes. Mögliche Latenzzeiten während der Übertragung hängen von der Pufferung an den entsprechenden Endpunkten ab.

Die zeitlich korrekte Übertragung wird bei isochronen Transfers durch das potentielle Auslassen bestimmter Datenteile erreicht. Das heißt, es wird hier auf ein Handshake-

Verfahren verzichtet (Abbildung 2.24). Wenn ein Fehler bei der Übertragung auftritt, werden somit auch keine Korrekturmechanismen, wie beispielsweise wiederholte Anfragen, angewandt. In der Praxis wird der Fall des Datenverlustes jedoch nur selten auftreten, da beim isochronen Transfer die ausreichende Bandbreite für die Übertragung reserviert wird, um sicherzustellen, dass die Daten mit der gewünschten Rate geliefert werden können. Außerdem ist USB für minimale Datenverzögerungen bei der Übertragung konfiguriert.

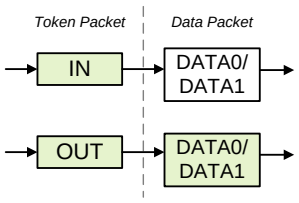


Abbildung 2.24: USB - Isochron-Transfer

2.3.2.4 Datenübertragung

Die Datenübertragung auf physikalischer Ebene geschieht nach NRZI-Kodierung (*Non Return to Zero Invert*). Hierbei wird eine logische 0 durch einen Wechsel des Signallevels repräsentiert. Das heißt, jeder 0-Wert im Bitstrom des zu übertragenden Pakets bewirkt einen Zustandswechsel, die 1-Werte hingegen nicht. Vor der Kodierung wird allerdings noch das sogenannte *Bit-Stuffing*-Verfahren angewandt. Mittels des Bit-Stuffings wird ein regelmäßiger Zustandswechsel erzwungen, indem nach sechs aufeinanderfolgenden 1-Werten eine logische 0 in den Bitstrom eingefügt wird. Der Empfänger decodiert den übertragenen Bitstrom und entfernt die eingefügten 0-Werte wieder.

2.3.2.5 Bus-Enumeration

Nach Anschluss eines USB-Gerätes an den Root-Hub des USB-Host wird dieses enumeriert und konfiguriert. Dies geschieht nach folgenden Punkten:

1. Der Root-Hub informiert den Host über ein neues Gerät,
2. der Host erfragt den Port,
3. aktiviert den Port und verschickt ein Reset-Signal.
4. Der Host liest den *Device-Descriptor* des Gerätes über Adresse 0 aus und
5. weist dem Gerät eine neue eindeutige Adresse zu.
6. Nun liest der Host sämtliche Konfigurationen des Gerätes aus und
7. aktiviert schließlich eine der möglichen Konfigurationen.

Das Auslesen von Deskriptoren, die Zuweisung der Adresse und die Konfiguration der Funktionen und Endpunkte erfolgt im Control-Transfer über die Setup-Pakete. Dabei wird über spezifizierte Standard-Requests die Anfrage beziehungsweise Konfigurationsanweisungen an EP0 gesendet.

2.3.2.6 Deskriptoren

In den Deskriptoren sind die Eigenschaften eines Gerätes festgelegt. In Abbildung 2.25 ist eine mögliche Struktur der Deskriptor-Hierarchie zu sehen. Der *Device-Descriptor* dient der Identifikation eines Gerätes. Es kann mehre Konfigurationen geben, die jeweils durch einen *Configuration-Descriptor* beschrieben sind. Nur eine dieser zur Auswahl stehenden Konfiguration wird vom Host aktiviert. Jede Konfiguration besitzt eine oder mehrere *Interfaces* mit jeweils zugeordneten Endpunkten. Ein *Interface* stellt eine Funktion des USB-Gerätes dar und wird über den *Interface-Descriptor* beschrieben. Ferner können in optionalen *String*-Deskriptoren weitere lesbare Zusatzinformationen in Unicode abgelegt sein.

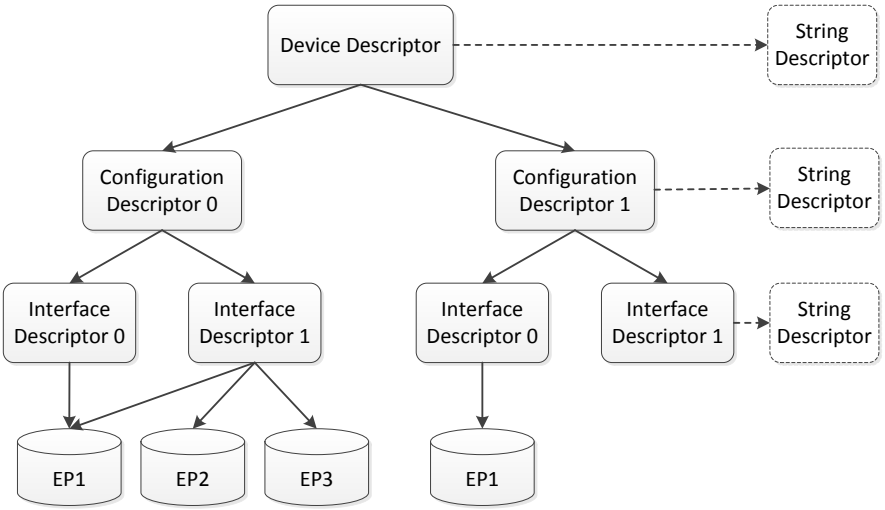


Abbildung 2.25: USB - Deskriptoren

2.4 Automotive-Bussysteme

Im Gegensatz zu Standardbussystemen stehen die Busse im Automobil nicht in direkter Konkurrenz. So haben die etablierten Netzwerke ihre Vorteile in bestimmten Anwendungsbereichen und können nebenher in separaten Bereichen des Automobils verbaut sein. Entscheidende Kriterien sind dabei die benötigte Datenübertragungsrate, die Datensicherheit und vor allem die Kosten, die sich aus der Komplexität der Kommunikations-Controller (CC) und der Verbindungsleitungen ergeben. In Abbildung 2.26 sind die gebräuchlichsten Automobilbussysteme bezüglich Datenrate und Kosten gegenübergestellt.

2.4.1 Überblick

Der CAN-Bus ist das meist genutzte Protokoll der Automobilelektronik und dient aufgrund seiner hohen Datensicherheit der Vernetzung von Steuergeräten und Sensoreinheiten in sicherheitsrelevanteren Bereichen. Das CAN-Protokoll unterstützt eine ereignisgesteuerte Kommunikation nach dem CSMA/CA-Verfahren (*Carrier Sense Multiple Access/ Collision Avoidance*) [Eng02, Nol09, Law11]. Aus dem CAN-Protokoll haben sich zwei grundlegende Versionen herausgebildet. Zum einen CAN-B (*LowSpeed-CAN*) mit einer Datenübertragungsrate von bis zu 125 KBit/s, das zur Kommunikation von Innenraum- oder Karosserie-Steuergeräten eingesetzt wird. Zum anderen CAN-C (*HighSpeed-CAN*) bis zu 1 MBit/s, wobei in realen Anwendungen üblicherweise nur eine Bitrate von 500 KBit/s genutzt wird. Die Motor-Fahrwerk-Steuergeräte sind seit Anfang der 1990er mit dem schnellen CAN-C vernetzt, der sich seitdem für diese Anwendung zum Standard entwickelt hat [Law11]. Auch die Vernetzung der beiden Bussysteme CAN-B und CAN-C kann über ein Gateway realisiert werden [Eng02].

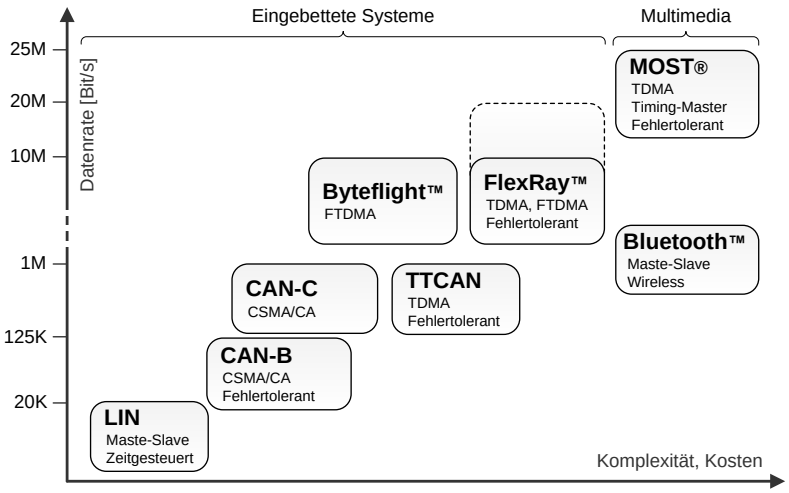


Abbildung 2.26: Technischer Überblick Automotive-Bussysteme

Der LIN-Bus (*Local Interconnect Network*) wurde entwickelt, um eine kostengünstige Alternative zu den bereits etablierten LowSpeed-CAN-Bussystemen für einfache Sensor-Aktor-Anwendungen zu bieten [ZS07, TR09]. LIN ist ein einfacher Master-Slave-Bus mit einer Single-Wire-Verbindung und einer Datenübertragungsrate von 1 bis 20 KBit/s. Es ist somit eine kostengünstige Lösung für kleine lokale Sensornetze in Bereichen, in denen die hohe Bandbreite und Flexibilität von CAN oder FlexRay nicht benötigt wird. Typische Anwendungen für den LIN-Bus sind lokale Netzwerke für die Steuerung des Spiegels oder Fensterhebers innerhalb der Tür, für das Steuerungspanel des Lenkrades, für die Regulierung der Klimaanlage oder für Regen- oder Lichtsensoren.

Aufgrund der Überführung von immer mehr Funktionalität des Automobils von der mechanischen hin zur elektronischen Steuerung haben die Komplexität der Netzwerktopologie, die Anzahl der Gateways zwischen den Netzen und die Bedürfnisse nach Flexibilität und somit Portabilität zwischen Anwendungsbereichen stetig zugenommen. Aus den gestiegenen Anforderungen bildet sich der Bedarf einer deterministischen (und gegebenenfalls redundanten) Kommunikation mit hoher Bandbreite für die Echtzeitfähigkeit und/oder für erhöhte Fehlertoleranz heraus, was durch bisherige Protokolle nur unzureichend abgedeckt wird [PR07].

FlexRay ist ein neueres Protokoll, das einen weitaus besseren Datendurchsatz von bis zu 10 MBit/s mit hoher Datensicherheit bietet, und es ist zudem durch die zugrundeliegende TDMA-Medienzugriffsstrategie echtzeitfähig. Im FlexRay-Netz stehen jedem Teilnehmer in einem Kommunikationszyklus ein oder mehrere Zeitschlitze zur Verfügung, in denen dieser exklusiv senden kann. Voraussetzung für dieses Zugriffsverfahren ist eine synchronisierte Zeitbasis, die dezentral von jedem Teilnehmer ermittelt wird. Neben dem statischen Segment eines Kommunikationszyklus gibt es ein optionales dynamisches Segment für ereignisgesteuerte Nachrichten. Den Kommunikationsteilnehmern wird im dynamischen Segment der exklusive Buszugriff jeweils nur für ein kurzes Zeitintervall ermöglicht. Nur wenn innerhalb dieses Intervalls ein Buszugriff erfolgt, wird der Slot um die benötigte Zeit verlängert. So wird für dynamische Nachrichten nur die tatsächlich benötigte Bandbreite verbraucht. FlexRay unterstützt neben allen gängigen Netztopologien, mehreren Kommunikationsarten und Nachrichtenaustauschprinzipien eine flexible Netzwerkkonfiguration durch einen umfangreichen Parametersatz. Dadurch kann das Netz ideal skaliert der jeweiligen Automobilanwendung angepasst werden. Hinsichtlich steigender Komplexität der Steuerelektronik und deren schnelle und fehlertolerante Vernetzung gewinnt FlexRay immer mehr an Bedeutung. So wird CAN-C gerade im Bereich sicherheitskritischer Anwendungen durch FlexRay abgelöst [ZS07, Rau07].

Um auch für das CAN-Protokoll einen deterministischen Ansatz zu unterstützen, wurde TTCAN (*Time-Triggered CAN*) entwickelt. Hierfür wurde das CAN-Protokoll um eine extra Protokollebene entsprechend des OSI-Modells erweitert mit dem Ziel, die Latenzzeit der CAN-Nachricht, unabhängig von der jeweils aktuellen Buslast auf einen festen Wert zu setzen [PR07, TR09]. Um ein globales Zeitsystem zu realisieren, wird ein Master-Knoten (gegebenenfalls mehrere) genutzt, der die Referenzzeit für das Netzwerk vorgibt [TR09]. TTCAN kann zwischen CAN und FlexRay eingeordnet werden.

Festzuhalten bleibt, dass die heute wichtigsten Bussysteme der Steuergerätekommunikation CAN und FlexRay sind beziehungsweise die Hauptansätze der Automotive-Kommunikation in den zugrundeliegenden Protokollen spezifiziert sind. Eingebettete Systeme in FlexRay- oder CAN-Netzwerkknoten haben also aufgrund sicherheitsrelevanter Anwendungen einen hohen Bedarf an einer Testschnittstelle für einen strukturorientierten Test.

Die Netzwerke multimedialer Anwendungen im Automobilbereich wie MOST (*Media Oriented Systems Transport*), D²B (*Domestic Digital Bus*), Bluetooth oder auch Ethernet stellen für den Ansatz der Testschnittstelle zu eingebetteten Systeme keine Alternative dar und werden daher auch nicht in Betracht gezogen.

Die prototypische Umsetzung des Konzepts des Testzugangs enthält als Anwendungsbeispiel einer Automotive-Schnittstelle neben FlexRay auch CAN, mit jeweils protokollkonformen Kommunikations-Controller und zugehörigem Transceiver. Da in dieser Arbeit der Testzugang über CAN aber nicht Teil experimenteller Untersuchungen ist, wird auf weitere Erläuterungen verzichtet. Bei näherem Interesse sei auf [CAN91, Eng02, Law11, Nol09] verwiesen. Exemplarisch für die Vernetzung sicherheitskritischer Steuergeräte im Automobilbereich soll im Folgenden FlexRay näher betrachtet werden.

2.4.2 FlexRay

In heutigen Fahrzeugen erfordert die hohe Anzahl an elektronischen Steuergeräten, Sensoren und Aktoren und somit deren Vernetzung eine hohe und schnelle Datenübertragung. Hinsichtlich sicherheitskritischer Anwendungen, wie etwa die Motorsteuerung, Getriebesteuerung, Bremssystemsteuerung, ist zudem eine hohe Fehlertoleranz unabdingbar. Gerade die X-by-Wire Systeme wie Drive-by-Wire oder Brake-by-Wire erfordern ein hochverfügbares, sicherheitsrelevantes Bussystem. Der heutzutage in der Automobilindustrie weit verbreitete CAN-Bus kann den wachsenden Anforderungen an Datenrate und Datensicherheit in Zukunft nicht mehr gerecht werden.

Um einen offenen standardisierten System-Bus für Hochgeschwindigkeits- und Echtzeitanforderungen zu entwickeln, schlossen sich im Jahr 2000 Automobil-, Steuergeräte- und Halbleiterhersteller zusammen und gründeten das *FlexRay Konsortium*. Die Kernpartner des Konsortiums (*Core-Members*) waren BMW, Daimler-Crysler, Freescale, Philips/NXP Semiconductors, General Motors, Bosch und VW/Audi. Unter den Premiumpartnern (*Premium Associated Members*) befanden sich weitere namhafte Hersteller wie Ford, Nissan, Renault, Peugeot, Toyota, Continental und Fujitsu. Hinzu kamen assoziierte Partner (*Associated Members*) und einige hundert weitere Partner aus dem Bereich der Software- und Hardware-Entwicklung (*Development Members*).

Es wurde schließlich mit FlexRay ein zeitgesteuertes Bussystem hoher Bandbreite definiert. Die bis heute gültige Spezifikation 2.1A [FPS05] wurde 2005 veröffentlicht. Nach Fertigstellung der FlexRay-Spezifikation 3.0 hat sich das Konsortium 2009 aufgelöst. 2010 wurde begonnen FlexRay in einen ISO-Standard zu überführen. Der neue Stan-

dard *ISO 10681-1/2:2010 "Road vehicles - Communication on FlexRay"* ist bereits in einer zweiteiligen Vorabversion [ISO10a, ISO10b] verfügbar.

Der FlexRay-Bus bietet eine schnelle, ausfallsichere und verzögerungsfreie Datenübertragung mit deterministischem Verhalten. Der Bus basiert auf einer erweiterten TDMA-Medienzugriffsstrategie (*Time Division Multiple Access*) und verfügt neben einem statischen auch über ein optionales dynamisches Segment für ereignisgesteuerte Nachrichten. Die Kommunikation erfolgt über zwei physikalisch getrennte Kanäle mit jeweils einer Übertragungsrate von 10 MBit/s. Durch in der Spezifikation festgelegte Zweikanalauslegung kann die Performance oder die Fehlertoleranz des Systems erhöht werden. Das heißt, ein Kommunikationsteilnehmer kann den zweiten Kanal entweder zur Verdopplung der Datenrate oder zur redundanten Übertragung für erhöhte Datensicherheit nutzen.

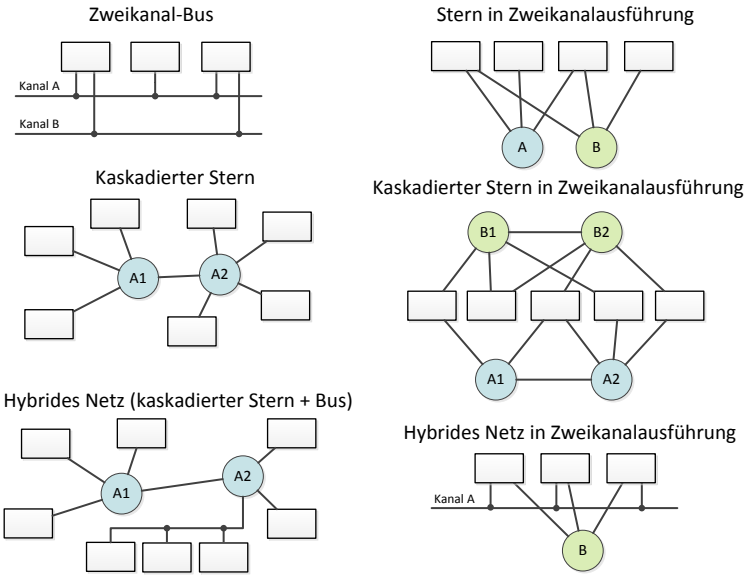


Abbildung 2.27: FlexRay - Netztopologien

FlexRay ist als ein sehr flexibles, robustes Bussystem konzipiert. Alle gängigen Topologien wie passiver Bus, aktiver Stern, kaskadierter Stern, sowie hybride Netze (Mischform von Linien- und Stern-Struktur) sind ohne qualitative Einschränkungen umsetzbar (Abbildung 2.27). FlexRay unterstützt neben den Netztopologien auch mehrere Kommunikationsarten und Nachrichtenaustauschprinzipien. Aus der sich daraus ergebenen Vielzahl an Möglichkeiten kann in der Entwicklung die Alternative gewählt werden, mit der die größtmögliche Skalierbarkeit und Flexibilität für die elektronische Architektur in Automobilanwendungen erreicht wird.

Die wesentlichen Punkte der FlexRay-Spezifikation, die den Aufbau der Kommunikationsinfrastruktur beschreiben, sind:

- Nachrichtenaustausch basierend auf einer deterministischen zyklusbasierten Übertragung,
- Synchronisation auf eine gemeinsame Zeitbasis für alle Knoten,
- Inbetriebnahme mittels einer autonomen Aufstartprozedur,
- Weckfunktion, die den Anforderungen des Power Managements entspricht,
- Fehlerverwaltung mit Fehlerbehandlung und Fehlermeldung.

2.4.2.1 Nachrichtenformat

Eine zu übermittelnde Nachricht wird in einen *Frame*, wie in Abbildung 2.28 dargestellt, verpackt. Dieser besteht aus einem Header Segment, in dem die nötigen protokollspezifischen Informationen untergebracht sind, einem Payload Segment mit der eigentlichen Nachricht und einem Trailer Segment mit der Prüfsumme des Frames.

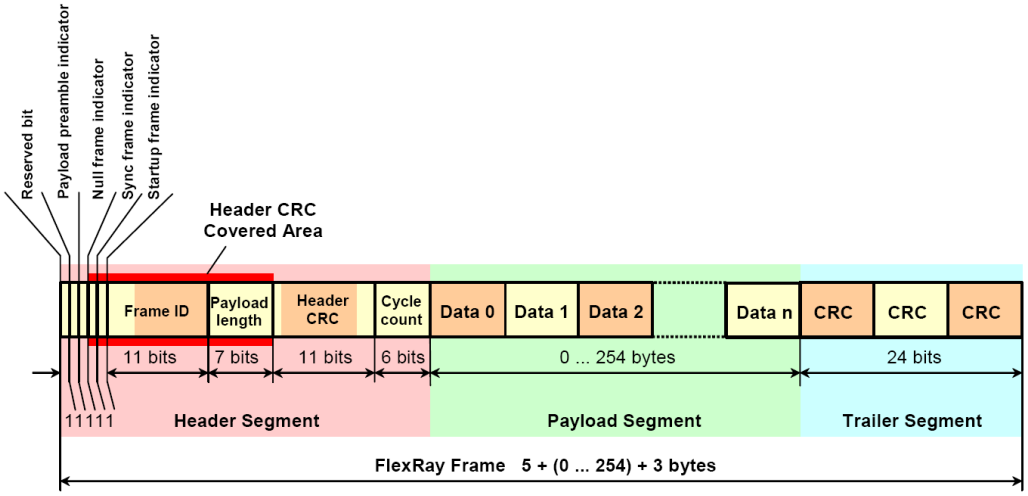


Abbildung 2.28: FlexRay - Frame-Aufbau [FPS05]

Im *Header* befinden sich die Frame-Nummer, die Nachrichtenlänge, die Zyklusnummer, eine Prüfsumme und weitere Indikatoren. Die *Frame-ID* ist genau einem Kommunikationsteilnehmer beziehungsweise Netzwerkknoten zugeordnet und entspricht dem Slot, in dem dieser einen Frame senden darf. So kann es durch die exklusiv vergebenen IDs beziehungsweise Slots nicht zu Kollisionen kommen. An einen Teilnehmer können auch mehrere Frame-IDs vergeben sein, um so die Übertragung in mehreren Slots zu erlauben. In dem *Payloadlength*-Feld steht die Länge des Payload Segments als Anzahl an 2-Byte-Wörtern. Im statischen Segment sind alle Frames gleich groß, im dynamischen Segment kann die Nachrichtenlänge variieren. Die *Header-CRC* schützt den vorderen Teil des Headers, ausgenommen der ersten drei Bits, mit einer Hamming-Distanz von

6. Das bedeutet, dass bis zu 5 bei der Übertragung veränderten Bits pro Frame sicher erkannt werden. Im Feld *Cycle-Count* ist der aktuelle Zyklus, in dem der Frame versendet wird, angegeben. Der *Payload Preamble Indikator* zeigt an, ob in der Nachricht ein *Netzwerk Management Vektor* im statischen Segment beziehungsweise eine *Message-ID* im dynamischen Segment enthalten ist. Der *Null Frame Indikator* gibt an, ob der Frame gültige Daten enthält. Der *Startup Frame Indikator* wird für Frames zum Aufstarten des Netzwerks verwendet. Der *Sync-Frame Indikator* weißt Frames aus, die von den Kommunikationsteilnehmern für die Uhrensynchronisation benötigt werden.

Das *Payload-Segment* enthält die zu übertragene Nachricht, die durch die in dem *Payloadlength*-Feld spezifizierte Größe begrenzt wird. Die Größe statischer Frames wird offline im Entwurf des FlexRay-Netzes festgelegt und kann während des Netzbetriebes nicht verändert werden. Der Datenabschnitt kann optional eine Nachrichtenkennung, den *Netzwerk Management Vektor* beziehungsweise die *Message-ID*, enthalten, die die Art der im Frame übertragenen Information kennzeichnet. Der Frame wird durch die Prüfsumme im Trailer Segment abgeschlossen. Die *Frame-CRC* wird über Header- und Payload-Segment berechnet und hat ebenso eine Hamming-Distanz von 6.

Frame-Sequenzen

Vor dem Versenden eines Frames werden Kodierungssequenzen eingefügt, die den Frame begrenzen und in Abschnitte unterteilen, wie in Abbildung 2.29 dargestellt. Vor jedem Frame wird eine *Transmission Start Sequence* (TSS) übertragen, die verhindert, dass der Beginn eines Frames durch einen Repeater beziehungsweise aktiven Sternkoppler abgeschnitten wird. Der Frame beginnt mit der *Frame Start Sequence* (FSS), eine logische 1. Jedes Byte des Frames wird durch eine *Byte Start Sequence* (BSS) gekennzeichnet. Die BSS besteht aus einer '10'-Folge, erzeugt also eine fallende Flanke, die als Synchronisationspunkt für den Empfänger dient. Die *Frame End Sequence* (FES), eine '01'-Folge, zeigt schließlich das Ende der Übertragung an. Für dynamische Frames wird zusätzlich eine *Dynamic Trailing Sequence* (DTS) verwendet, um nach Frame-Ende den restlichen Slot aufzufüllen und so eine Kollision, also eine weitere Übertragung im selben Slot zu vermeiden. Der Empfänger des Frames erkennt die Kodierungssequenzen und kann sie direkt aus dem ankommenden Bitstrom wieder entfernen.

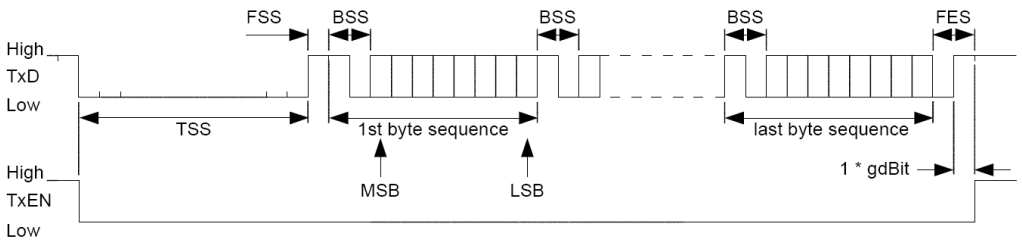


Abbildung 2.29: FlexRay - Sequenzen eines statischen Frames [FPS05]

Sampling des Datenstroms

Auf Empfängerseite wird das ankommende Bussignal mit einem höheren Takt als der Bitfrequenz abgetastet. Das FlexRay-Protokoll sieht eine 8-fache Bitabtastung vor, spezifiziert in der Konstante *cSamplesPerBit*. Dadurch wird die Unterdrückung von Busstörungen wie Glitches und Spikes ermöglicht. In Abbildung 2.30 ist das sogenannte *Voting Window* dargestellt. Das Eingangssignal RxD ist der über den Bus übertragene Signalpegel. Mit der *Sample Clock* wird dieses Signal abgetastet und in das *Voting Window*, das die jeweils letzten 5 Samples enthält, geschoben. Mittels eines Mehrheitsentscheids wird der Bitwert (*zVotedVal*) ermittelt. Das heißt, erst wenn ein neuer Signalpegel über 3 Samples stabil anliegt, wird dieser auch als Transition im Bitstrom interpretiert. Um den Datenstrom einer Nachricht zu erhalten, wird das Signal *zVotedVal* jeweils in der Bitmitte gelesen. Hierzu dient ein Sample-Counter, der jeweils nach 5 der 8 Samples pro Bit ein Abtastsignal erzeugt. Mittels der in den Frame eingefügten BSS wird der Sample-Counter zurückgesetzt und so das Abtastsignal synchronisiert.

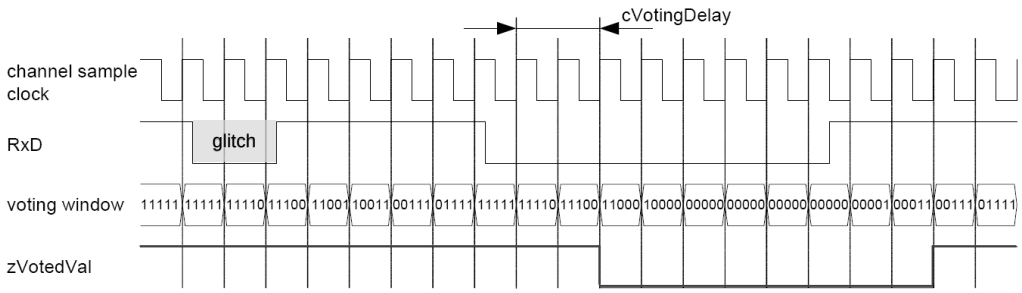


Abbildung 2.30: FlexRay - Abtastung des Bitstroms [FPS05]

2.4.2.2 FlexRay TDMA-Struktur

Da FlexRay ein TDMA-Verfahren umsetzt, sind den Netzwerkknoten feste Zeitschlitze beziehungsweise Slots zugewiesen, in denen sie einen exklusiven Zugriff auf den Bus haben. Die Slots wiederholen sich in einem festgelegten Zyklus. Der Zeitpunkt, zu dem ein Frame über den Bus übertragen wird, kann exakt vorausgesagt werden, der Buszugriff erfolgt daher deterministisch.

Die feste Zuordnung der Busbandbreite auf die Komponenten und deren Frames durch die Slots hat aber den Nachteil, dass die Bandbreite nicht optimal ausgenutzt wird. Deshalb unterteilt FlexRay den Zyklus in ein statisches und ein dynamisches Segment (Abbildung 2.31). Statische Slots bilden stets das erste Segment des Zyklus. Im darauffolgenden optionalen dynamischen Segment erfolgt eine ereignisgesteuerte Kommunikation. Den Segmenten folgt ein optionales Symbolfenster und bis zum Ende des Zyklus eine Leerlaufzeit.

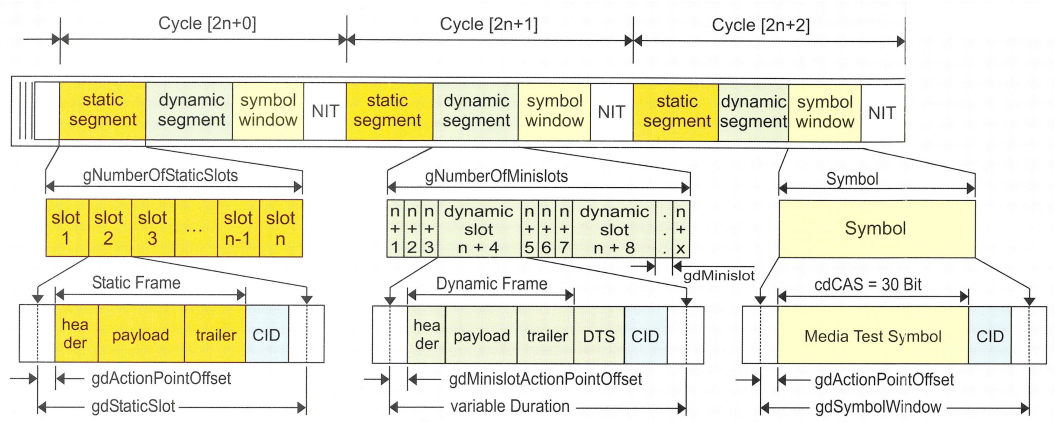


Abbildung 2.31: FlexRay - TDMA-Struktur [VCI14]

Da das statische Segment und die Leerlaufzeit obligatorisch, während das dynamische Segment und das Symbolfenster optional sind, werden grundsätzlich drei verschiedene Konfigurationen des Kommunikationszyklus ermöglicht:

- eine rein statische Konfiguration mit ausschließlich statischen Slots für die Übertragung,
- eine rein dynamische Konfiguration, wobei die gesamte Bandbreite der dynamischen Kommunikation zugeordnet ist, und
- eine gemischte Konfiguration, bestehend aus einem statischen und einem dynamischen Segment, wobei statische und dynamische Bandbreite variieren.

Statisches Segment

Das statische Segment erfüllt durch das deterministische Zeitverhalten in der Kommunikation die wichtigen Anforderungen, wie Latenzzeit und Jitter. Da FlexRay einem TDMA-Modell mit gleich großen Slots folgt, ist der Zeitpunkt genau bekannt, wann ein Frame auf dem Kanal übertragen wird. Außerdem hat jeder Netzwerkknoten mindestens einen eigenen exklusiven Slot, der die Übertragung seiner Daten sicherstellt. Die Konfigurationsparameter des FlexRay-Netzes, wie in Abbildung 2.31 dargestellt (*gNumberOfStaticSlots*, *gdStaticSlot* und viele mehr), bestimmen die Anzahl und Dauer der Slots und stellen somit einen bindenden Rahmen dar. Da Slotzuordnung und Zeitplanung offline während der Systemplanung stattfinden, muss nicht zur Laufzeit um den Zugriff auf das Kommunikationsmedium konkurriert werden. Das *Scheduling*, die Auslegung des Kommunikationsplans, wird also in den Entwurfsprozess des FlexRay-Systems verlagert. Solch ein Kommunikationsplan gibt die Abbildung 2.32 beispielhaft wieder.

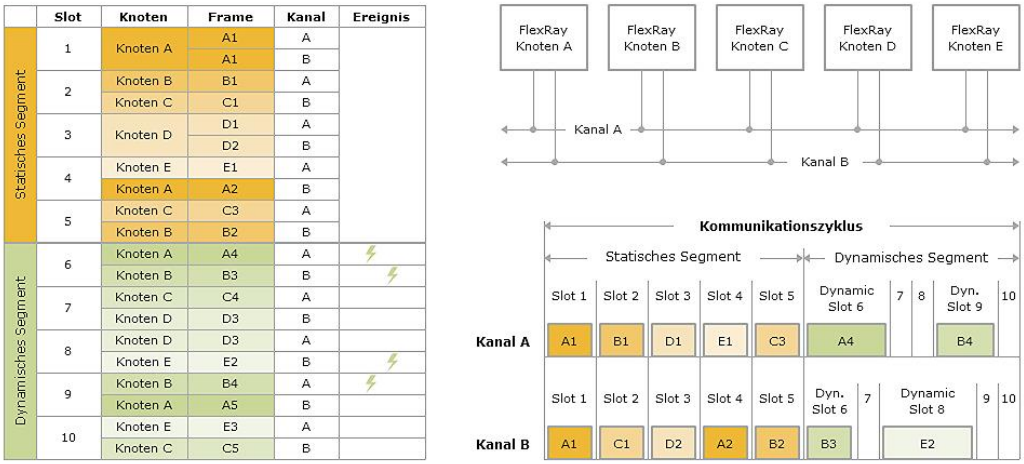


Abbildung 2.32: FlexRay - Kommunikationsplan [VCI14]

Dynamisches Segment

Ereignisgesteuerte Kommunikationsanforderungen werden im dynamischen Segment eines Zyklus abgearbeitet. In diesem Segment wird ein dynamisches Minislot-Modell nach dem sogenannten FTDMA-Verfahren (*Flexible Time Division Multiple Access*) umgesetzt [FPS05, Rau07]. Dieses Verfahren wurde dem vor FlexRay entwickelten Byteflight-Protokoll entnommen. Hierbei ist die Kommunikation in kleine Zeitintervalle beziehungsweise Minislots eingeteilt, wobei die Priorität der Nachricht den jeweiligen Minislot bestimmt.

Eine weitere wichtige Grundlage bildet das anforderungsgesteuerte Zugriffsmodell, mit dem Bandbreite für die dynamische Kommunikation bestmöglich genutzt werden soll. Sendet das Steuergerät eine Nachricht, verlängert sich das Zeitfenster entsprechend der zu übertragenden Nachricht. Das heißt, wird eine Nachricht in einem Minislot begonnen, so wird dieser um die für die Nachrichtenlänge nötige Zeitspanne erweitert und die darauffolgenden Minislots überschrieben. Muss keine Nachricht gesendet werden, läuft der Minislot als kurze Leerlaufperiode ungenutzt ab. Alle Steuergeräte im FlexRay-Netz behalten deshalb durchgängig einen Überblick über den aktiven Minislot. In der Konfiguration des dynamischen Segments und der Zuordnung der Minislots muss ein Kompromiss zwischen Flexibilität und Leistung getroffen werden.

Entgegen einer Slot-ID des statischen Segments kann eine Slot-ID des dynamischen Segments auf mehrere Knoten im Netz aufgeteilt werden. Dies geschieht durch das sogenannte *Slot-Multiplexing*, wobei Knoten mit demselben dynamischen Slot unterschiedliche Zyklen zugewiesen bekommen, um einen dynamischen Frame zu versenden.

Durch das Slot-Multiplexing kann die Bandbreite effizienter genutzt und die Latenz langsameren Knoten angepasst werden. Fallen in einem Netzwerkknoten Daten an, die

nur alle paar Zyklen versendet werden sollen, muss der zugeordnete dynamische Slot nicht in jedem Zyklus belegt werden. Solch einen Minislot können sich mehrere Knoten teilen, wenn diese durch fest zugewiesene Zyklen differenziert werden. Die Zuweisung der Zyklen geschieht durch einen Basiswert b von 0 bis 63, der den Startzyklus angibt, und einer Wiederholungsrate $r = 2^x$ mit $0 \leq x \leq 7$. Das bedeutet, ein Frame kann in diesem Fall nur alle r Zyklen versendet werden.

Symbolfenster und Leerlaufzeit

Neben dem statischen und dynamischen Segment besteht ein Kommunikationszyklus aus dem optionalen *Symbol Window* (SW) und der *Network Idle Time* (NIT). Das Symbolfenster kann für das Versenden spezieller Symbole, wie Weck-, Inbetriebnahme- und Alarmsymbole, genutzt werden. Die konfigurierbare NIT schließt den Kommunikationszyklus ab und ist grundsätzlich kommunikationsfrei. Sie wird hauptsächlich zur Ermittlung der gemeinsamen Zeitbasis benötigt, um die Synchronisation für den darauffolgenden Zyklus zu gewährleisten. In der Konfiguration des Netzwerks muss sichergestellt werden, dass die verbleibende Länge der NIT für die Durchführung von Berechnungen von Uhrkorrekturwerten, Fehlerbehandlungen oder Aktualisierung von Fehleranzeigen und Zähler ausreicht.

2.4.2.3 Synchronisation

Dezentrale Kommunikationssysteme hängen von einem gemeinsamen und konsistenten Zeitplan ab. In einem Netzwerk können die Uhren der Controller jedoch durch die Temperatur- und Spannungsschwankungen oder Produktionstoleranzen, zum Beispiel ein von der Sollfrequenz abweichender Oszillator, beeinflusst werden. Die Netzwerkknoten müssen sich aufeinander abstimmen und die lokale Zeitbasis der globalen Zeit des Netzwerkes anpassen.

Das FlexRay-Protokoll verwendet einen dezentralen fehlertoleranten Synchronisationsmechanismus, der auf der in Abbildung 2.33 dargestellten Zeithierarchie basiert. Ein *Microtick* (µT) ist die vom Oszillator abhängige lokale Zeiteinheit. Eine definierte Anzahl an Microticks ergibt ein *Macrotick* (MT). Ein Zyklus setzt sich wiederum aus einer bestimmten Anzahl Macroticks zusammen.

Jeder Knoten synchronisiert sich individuell mit dem Netzwerk, indem dieser den Takt der von anderen Knoten übertragenen Synchronisationsnachrichten (*Sync-Frames*) überwacht. Der dadurch ermittelte Referenztakt wird zur Ausgleichskorrektur der Phasen- und Frequenzabweichungen, die sogenannte *Offset-Rate-Correction*, genutzt.

Zur Korrektur der Frequenz beziehungsweise der Rate wird die Abweichung über die Zeit ermittelt. Hierzu wird die Zeit in Microticks zwischen zwei Sync-Frames derselben Frame-ID vermessen, um so die Zykluslänge des Knotens, dem diese Frame-ID zugeordnet ist, zu erhalten. Dies geschieht mit allen Sync-Frames des Netzwerkes. Ein fehlertoleranter Mittelwertalgorithmus errechnet die mittlere Abweichung zur eigenen

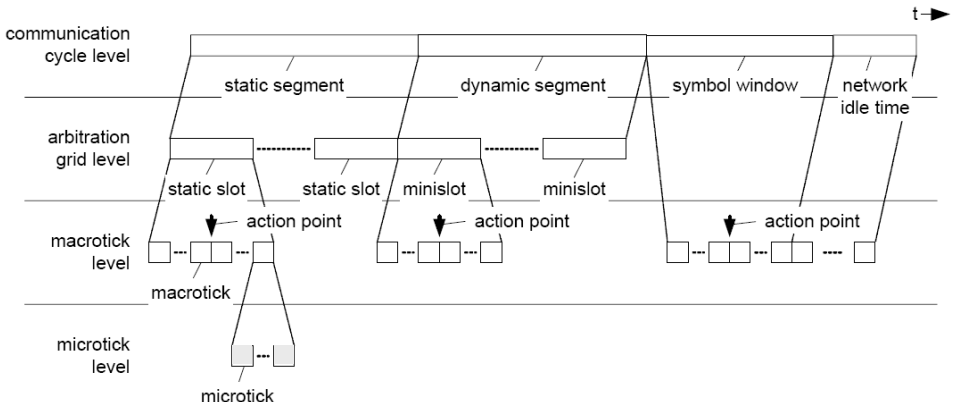


Abbildung 2.33: FlexRay - Zeitbasis [FPS05]

Zykluslänge. Durch die Anpassung der Anzahl an Microticks pro Macrotick kann die Frequenz der lokalen Uhr der globalen Uhr angeglichen werden. Da die Berechnungen zur Ratenkorrektur auf den Werten zweier aufeinanderfolgender Zyklen basieren, finden diese nur in ungeraden Zyklen statt.

Für die Phasenkorrektur werden die erwarteten Empfangszeitpunkte mit den tatsächlichen Zeitpunkten des Eintreffens der Sync-Frames verglichen. Der Mittelwertalgorithmus ergibt nun ein positives oder negatives Offset zum eigenen Startzeitpunkt des folgenden Zyklus. Durch entsprechendes Einfügen beziehungsweise Entfernen von Microticks während der NIT werden die Knoten auf einen gemeinsamen Zyklusbeginn geeicht. Die Offset-Korrektur wird in jedem Zyklus berechnet, aber nur am Ende ungerader Zyklen angewandt.

Kapitel 3

3 Konzept der Testanbindung

Der Ansatz der Verknüpfung der Produktionstestlogik mit den Standardschnittstellen eines eingebetteten Systems wird durch die Notwendigkeit motiviert, einen hochauflösenden Test im Zielsystem durchführen zu können. Somit wird im Falle eines erkannten Fehlers auch eine Diagnose des eingebetteten Systems im Feld ermöglicht. Der Begriff "im Feld" bedeutet, dass sich ein System in seinem vorgesehenen Funktionalitätskontext befindet. Für den in ein Automobilsteuergerät eingebetteten IC trifft dies zu, sobald das betriebsbereite Fahrzeug das Fabrikgelände des Automobilherstellers verlässt. Ab dem Zeitpunkt steht das System gemäß der Spezifikation für seine reale Anwendung bereit.

Es wurde ein Konzept entworfen das den Testzugang über HighSpeed-Standardschnittstellen, wie in Abbildung 3.1 dargestellt, realisiert. Über eine als Testzugang spezifizierte Schnittstelle werden Testdaten übertragen, in ein definiertes einheitliches Zwischenformat, welches der zugrundeliegenden Testtechnologie angepasst ist, empfangen und an den integrierten Test-Controller weitergeleitet. Diese Testdaten können komplette Testmuster für einen direkten Scan-Test, alternativ aber auch Konfigurationsdaten für einen Selbsttest sein. Da die Grenze zwischen konfigurierbaren Ansätzen für den Selbsttest einerseits und Verfahren des Scan-Tests mit hochgradiger Kompaktierung andererseits fließend ist, sind hier flexible Ansätze möglich.

Die Testkonfigurationen, wie beispielsweise der Testtyp, die Testanalyse oder das Ausgabeformat, werden über die im Testsatz enthaltenen Metadaten bestimmt. Testdaten werden entsprechend dem spezifizierten Verfahren in die ICs geladen, um diese strukturell testen zu können. Mittels zusätzlich übertragener Referenzdaten kann eine schaltungsinterne Analyse der Testergebnisse erfolgen. Zudem können die Testantworten auch direkt oder kompaktiert zu Signaturen und gegebenenfalls zusätzlich gefiltert über die Standardschnittstelle für anschließende diagnostische Auswertungen ausgelesen werden. Das Konzept wurde bereits in [GVE12a, GVE12b, DIA13] vorgestellt.

Um integrierte Schaltungen auch nach dem Fertigungstest noch testen zu können, steht heutzutage bereits die etablierte JTAG-Schnittstelle zur Verfügung. Diese ist hauptsächlich für den Boundary-Scan zum Test der Verbindungen zwischen IC und Lei-

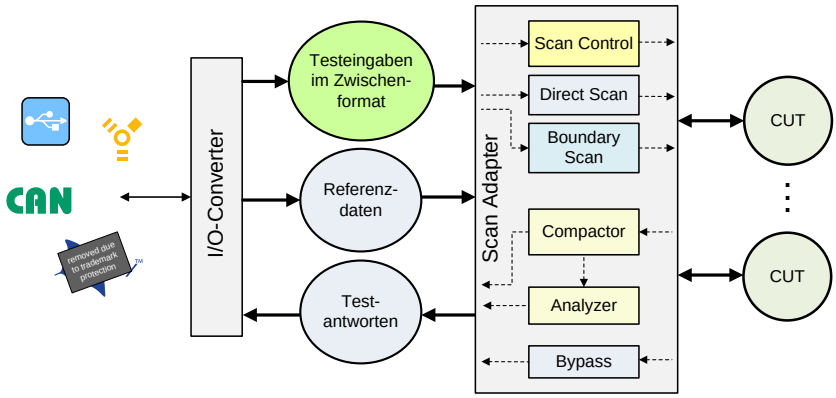


Abbildung 3.1: Konzept der Testschnittstelle

terplatte vorgesehen. Darüber hinaus ist es mittels JTAG möglich, Testvektoren an die primären Eingänge der internen Schaltungslogik anzulegen, um diese erschöpfend, pseudoerschöpfend oder funktional zu testen. Ist der interne Schaltungsaufbau bekannt, kann auch ein strukturorientierter Test realisiert werden. Hierzu müssen die Testdaten durch ein entsprechendes ATPG-Verfahren für einen software-basierten Selbsttest erstellt werden.

Die neue IJTAG-Schnittstelle soll einen standardisierten und kostengünstigen Zugang zu Testmodulen integrierter Schaltungen auch noch nach dem Produktionstest ermöglichen. Diese internen Testmodule können etwa MBIST-Einheiten zum Test der Speicherbausteine oder LBIST-Einheiten zur Ansteuerung der integrierten Scan-Strukturen sein. Somit macht es IJATG ebenso möglich, die in den IC integrierte Produktionstestlogik für einen diagnostischen Test zu nutzen. Das in dieser Arbeit vorgestellte Konzept stellt eine Alternative zur IJTAG-Lösung dar.

IJTAG nutzt den JTAG-TAP als Zugang zum *Gateway*-Pfad und zu den daran angeschlossenen Testmodulen. JTAG wurde für Offline-Tests oder Debugging-Verfahren als serielle Low-Speed-Schnittstelle konzipiert, deren obere Grenze der Transfergeschwindigkeit weit unter 100 MHz liegt [Sto11]. Da der JTAG-TAP und der Scan-Pfad üblicherweise mit 5 bis 30 MHz betrieben werden, ist auch die Datenrate für den IJATG-Test entsprechend beschränkt. Da die Testeingaben direkt mit jedem Takt seriell hineingeschoben werden, bedeutet eine JTAG-Clock von 30 MHz eine Datenrate von 30 MBit/s.

Der Testzugang über standardisierte Anwendungsschnittstellen kann eine kürzere Testzeit ermöglichen, wenn ein Bussystem mit hoher Datenrate die Grundlage bildet. Eine exklusiv genutzte USB-Verbindung beispielsweise erlaubt in der HighSpeed-Version eine theoretische Datenrate von bis zu 480 MBit/s oder in der SuperSpeed-Version gar bis zu 5 GBit/s. Bei einem Automotive-Steuergerät, bei dem üblicherweise CAN oder FlexRay zur Wahl stehen, kommt dieser Vorteil nicht zum Tragen. Setzt sich Ethernet, wie es beispielsweise bereits in der 100 MBit/s-Version für Kameranetze von Fahreras-

sistenzsystemen zum Einsatz kommt, vermehrt im Automobilbereich durch, bietet sich auch dieser Zugang für die Diagnose mit umfangreichen Testdatensätzen an.

Ein in ein Fahrzeug verbautes Steuergerät gestattet außerdem keinen Zugang über die JTAG-Schnittstelle, da diese nicht nach außen geführt wird. Eine effiziente Diagnose im Feld wird nur durch das Nutzen der Standardbordnetze als Testzugang zu eingebetteten Systemen ermöglicht.

3.1 Aufbau der Testschnittstelle

Die Anbindung an die Scan-Testarchitektur geschieht hauptsächlich über eine Steuerungseinheit und einen oder mehrere Zwischenspeicher für die zu übermittelnden Testdaten. In Abbildung 3.2 ist der konzeptionelle Aufbau der Testschnittstelle dargestellt. Die Testdaten können in reservierte Speicherbereiche eines Hauptmoduls abgelegt werden. Im Folgenden werden diese Zwischenspeicher aber als separate Module nach dem FIFO-Prinzip dargestellt. Der Speicher für die Testeingaben ist das *TestIn-FIFO*. Für die Referenzdaten, die zur Auswertung der Testergebnisse benötigt werden, ist hier ein separater Speicher vorgesehen. Dieser wird als *TestRef-FIFO* bezeichnet. In dem Ausgangsspeicher *TestOut-FIFO* werden die Testantworten abgelegt. Die Testantworten des Scan-Tests können entweder direkt der Scan-Struktur entnommen oder kompaktiert zu Signaturen von der Testarchitektur abgegriffen werden. Die Analysekomponente setzt die optionale Auswertung on-chip um und kann, im Falle einer anschließenden diagnostischen Auswertung, für die Filterung einer umfangreichen Ausgabe auf fehlerhafte Testantworten sorgen. Um den Testzugang zu steuern, muss eine zusätzliche Controller-Einheit implementiert werden. Diese wertet die empfangenen Metadaten aus, koordiniert somit Schreib- und Lesezugriffe und konfiguriert den Scan-Test. Mittels zusätzlicher Kontrolllogik soll auch ein schaltungsinterner Selbsttest umgesetzt werden, der die Testarchitektur mit kompakten Testinformationen, die auf dem IC untergebracht sind, betreibt.

In der Abbildung 3.2 ist zudem ein optionales JTAG-FIFO zu sehen, das die über eine Standardschnittstelle empfangenen JTAG-Eingangssignale TDI und TMS paketweise zwischenspeichert und für den JTAG-TAP bereitstellen kann. Die Ausgangsdaten aus dem TDO-Signal werden wieder zwischengespeichert, um diese paketweise auslesen zu können. Die Test-Clock TCK und das Reset-Signal TRST werden dabei on-chip erzeugt. Es kann somit auch ein alternativer JTAG-Zugang über eine Standardschnittstelle realisiert werden.

Das Gesamtkonzept soll diverse Test- und Analysemodi unterstützen und somit verschiedenen Anwendungsszenarien dienen. Um die universelle Einsatzmöglichkeit zu un-

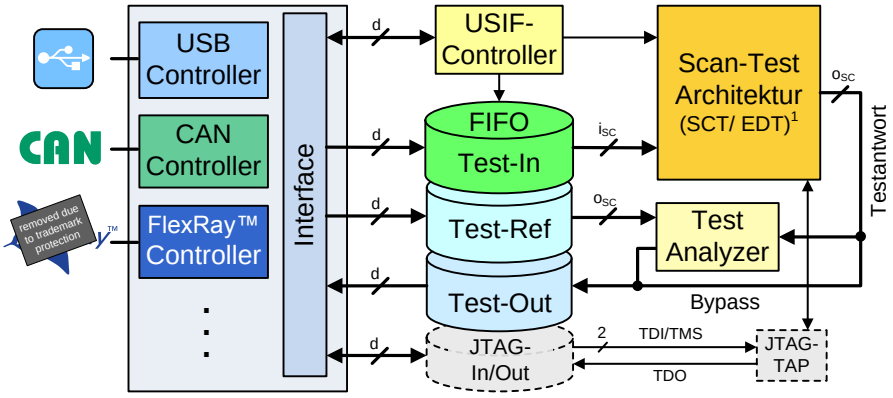


Abbildung 3.2: Universal Scan-Test Interface

terstreichen, wird das entworfene Design als *Universal Scan-Test Interface* (USIF) bezeichnet.

3.1.1 Zentrale Teststeuerung

Kern der zentralen Steuerungseinheit, im Folgenden auch als *USIF-Controller* bezeichnet, ist der Zustandsautomat (FSM) aus Abbildung 3.3. Gesteuert wird dieser durch Befehle, die mittels spezieller Instruktionswörter übertragen werden. Der Aufbau des USIF-Instruktionswortes, der Befehlssatz des USIF-Controllers und ein erweitertes Zustandsdiagramm mit einer ausführlichen Beschreibung, die wichtige Randbedingungen mit einbezieht, sind im Kapitel 4 zu finden. Im Folgenden soll ein kurzer Überblick über die Funktion der zentralen Steuerungseinheit gegeben werden.

Für den Testzugang durch das USIF wird von einem im speziellen Testmodus befindlichen System ausgegangen. Das heißt, sämtliche von der Standardschnittstelle empfangenen Daten werden als Testdaten, also USIF-Instruktionen beziehungsweise Testmuster für die Testarchitektur, interpretiert.

Vom Ausgangszustand Idle wird ein Test durch entsprechende Anweisung gestartet. Hierzu wird der Empfangspuffer des Kommunikations-Controllers ausgelesen, sobald dieser verfügbare Daten enthält. Soll ein interner Selbsttest ausgeführt werden, so wird dieser über den Zustand RunBIST initiiert. Im Falle des Zugangs zur internen Testlogik ist zunächst eine Authentifizierung erforderlich. Das in der Abbildung eingerahmte Teildiagramm soll den vor unbefugtem Zugriff geschützten Bereich darstellen.

Im zentralen Zustand TestControl findet die Dekodierung der empfangenen USIF-Instruktionswörter statt, um in den entsprechenden Folgezustand überzugehen. Um eine

[1] Die Produktionstestlogik wird in dieser Arbeit anhand des SCT (Scan-Controller-based Test) sowie des EDT (Embedded Deterministic Test) veranschaulicht.

Konfiguration, also ein Schreibzugriff auf ein adressiertes Register vorzunehmen, wird der Zustand WriteConfig genutzt. Ein Lesezugriff auf ein Konfigurationsregister kann durch den Zustand ReadConfig geschehen. Hierbei wird der Registerinhalt angefragt und im darauffolgenden Takt, nachdem der Wert geladen ist, im Zustand ForwardConfig in den Sendepuffer des Kommunikations-Controllers geschrieben. Die gestrichelten Pfeile in der Darstellung des Zustandsdiagramms sollen den Zugriff beziehungsweise den Einfluss eines Zustandes auf die entsprechenden Komponenten verdeutlichen.

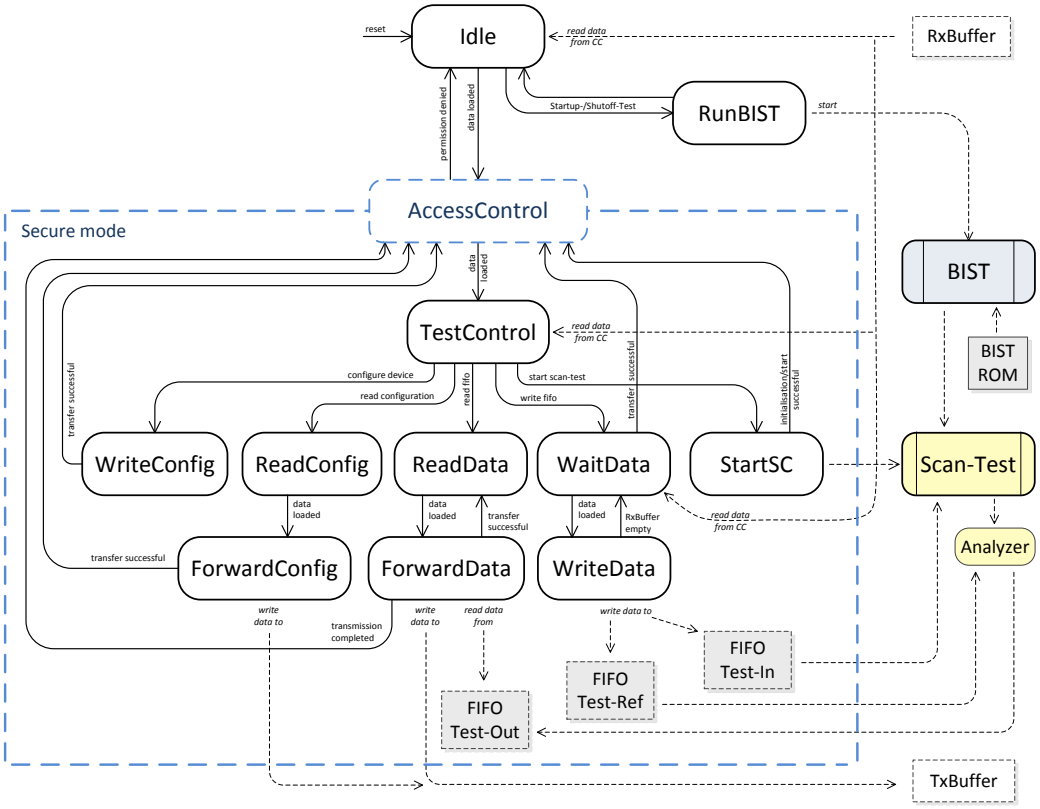


Abbildung 3.3: Vereinfachtes USIF-Zustandsdiagramm

Ein Lesezugriff auf den Testausgabespeicher wird durch die Zustände ReadData und ForwardData realisiert. Es wird jeweils ein Datenwort aus dem adressierten Speicherblock entnommen und an den Sendepuffer des Kommunikations-Controllers weitergereicht. Bei einem Schreibzugriff wird zunächst in den Zustand WaitData gewechselt, um auf den Empfang eines Testdatenpaketes mit einem im Instruktionswort spezifizierten Umfang zu warten. Meldet der Kommunikations-Controller verfügbare Daten, wird eine Leseanfrage gestellt, um das erste Datenwort im Zustand WriteData in den adressierten Testspeicher zu schreiben. Hier wird in jedem weiteren Takt ein Datenwort gelesen und an den Testspeicher weitergeleitet. Dies wird wiederholt, bis das komplette Datenpaket in den Testspeicher übertragen ist.

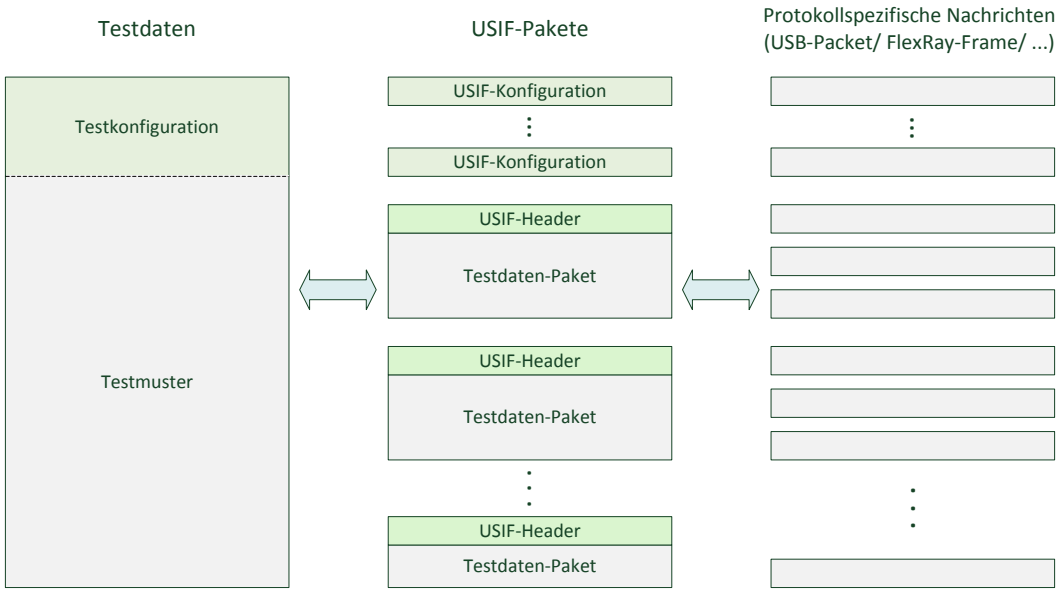


Abbildung 3.4: Paketweiser Datenaustausch

Durch den Zustand StartSC wird ein Scan-Test initiiert. Hierfür werden die Testkomponenten initialisiert und der Test-Controller gestartet. Nachdem der Scan-Test aktiviert ist, wird direkt wieder in den Zustand TestControl gewechselt, um auf den Empfang von Datenpaketen als Testeingaben für den Scan-Test zu warten. Der Test-Controller entnimmt die Testeingaben dem Testspeicher *TestIn-FIFO*. Die Testantworten werden nach optionaler Filterung durch die Analysekomponente im Testausgabespeicher *TestOut-FIFO* gesammelt.

Die Übermittlung der Testdaten erfolgt paketweise, um den Scan-Test in Zyklen, je nach verfügbaren Testdaten in den Testspeichern, auszuführen. Das heißt, der Testmustersatz wird auf Anwenderseite in Pakete spezifizierter Größe aufgeteilt. Für die eigentliche Übertragung auf dem Buskanal werden die Pakete in Nachrichten dem zugrundeliegenden Busprotokoll entsprechend unterteilt (USB-Pakete, FlexRay-Frames). Dieses Schema ist in Abbildung 3.4 veranschaulicht.

Die Übertragung der Testantworten erfolgt auf gleiche Weise. Enthält der Testausgabespeicher ein Datenpaket in entsprechenden Umfang, so wird dieses ausgelesen, um Speicherplatz für weitere Testausgaben freizugeben. Das Datenpaket wird wieder im entsprechenden Nachrichtenformat über den Bus übertragen. Auf Anwenderseite können die Datenpakete entweder direkt ausgewertet oder zu einem kompletten Testantwortsatz zusammengefügt werden.

3.1.2 Zugangsschutz

Wie bereits in Abbildung 3.3 veranschaulicht, bedarf die Integration eines Zugangs zu internen Komponenten des ICs Schutzmaßnahmen vor unautorisiertem Zugriff. Gerade das Nutzen eines von außen zugänglichen Standardbussystems erfordert entsprechend hohe Sicherheitsmaßnahmen. Der Zustand AccessControl soll dabei eine Authentifizierung für sämtliche Zugriffe als auch einen Schutz der zu übermittelnden Daten darstellen. Für einen angemessenen Schutz sollten zumindest Zugangsdaten und Steuerdaten verschlüsselt übertragen werden. Wichtig dabei ist neben der Wahl eines sicheren Verschlüsselungsalgorithmus vor allem ein angemessenes Sicherheitsverfahren, um das System auch vor unautorisierter Kommunikation (*Man-in-the-Middle Attack*) oder auch vor Seitenkanalattacken (*Side Channel Attacks*) zu schützen.

Es gibt eine Reihe etablierter und industriell genutzter Sicherheitsverfahren. Speziell für den Automobilbereich wurde der Kryptografiestandard *Secure Hardware Extension* (SHE) [SHE09] durch das HIS-Konsortium (Herstellerinitiative Software), dem die Automobilhersteller Audi, BMW, Daimler, Porsche und Volkswagen angehören, spezifiziert. Durch den SHE-Standard wird eine kompakte Hardware-Erweiterung definiert, um wichtige Sicherheitsfunktionalitäten innerhalb eines ICs verfügbar zu machen. Damit wird es Automobilzulieferern und Automobilherstellern ermöglicht, Gegenmaßnahmen zur Software-Manipulation in elektronischen Steuerungen zu entwickeln und den unbefugten Zugriff auf Steuergeräte zu verhindern. Klassische Anwendungsfälle der kryptographischen Verfahren sind die elektronische Wegfahrsperre, schlüssellose Zugangssysteme sowie Aktivierungssysteme für Ferndiagnosen und Software-Updates.

Da bewährte Standards zumeist in herstellerspezifischen Verfahren umgesetzt werden, wird an dieser Stelle von der Betrachtung einer konkreten Lösung für den entworfenen Testzugang abgesehen.

3.1.3 Test der Testschnittstelle

Der Testmodus sollte mit dem Selbsttest der Komponenten des Testzugangs beginnen. Dies kann durch einen rein funktionalen Test geschehen. Hierzu wird eine Loopback-Funktion vorgesehen, die es ermöglicht die empfangenen Testdaten aus dem *TestIn-FIFO* direkt in das *TestOut-FIFO* zu schreiben. Die Daten aus dem *TestOut-FIFO* können wieder über die Schnittstelle ausgelesen werden. Entsprechen die vom USIF empfangenen Daten den zuvor gesendeten, kann von einer korrekten Funktion der Empfangs- und Sendekomponente des Kommunikations-Controllers und des Zugriffs auf die Testdatenspeicher geschlossen werden.

Um auch die Analysekomponente, den Referenzspeicher und das Verbindungsnetz zwischen den Testkomponenten in den funktionalen Test einzubeziehen, kann der sogenannte Chain-Test, der die korrekte Funktion der Testkomponenten und der Scan-Struktur sicherstellt, um entsprechende Testmuster erweitert werden. Somit können bereits vor der Ausführung des Scan-Tests gegebenenfalls Fehler in USIF-Komponenten erkannt werden.

3.2 Scan-Controller

Die industriell genutzten Testtechnologien sind in der Regel Eigentum spezieller EDA-Hersteller, wie zum Beispiel Mentor Graphics, Synopsys oder Cadence, und damit nicht frei verfügbar. Eine der meistgenutzten Methoden ist der *Embedded Deterministic Test* (EDT) von Mentor Graphics. Die Entwicklung der USIF-Architektur erfordert an dieser Stelle aber eine Lösung, welche frei und ohne Restriktionen verwendet werden kann. Hierzu wird der an der BTU Cottbus entwickelte Scan-Controller als interne Testumgebung in die Architektur eingebunden. Im Kapitel 4 wird beschrieben, wie auch der EDT in der USIF-Architektur integriert und angewandt werden kann.

Die Basisarchitektur des Scan-Controllers ist in Kooperation mit Infineon Technologies AG im BMBF-Verbundprojekt AZTEKE [AZT05] entstanden. Dieser wurde bis heute kontinuierlich weiterentwickelt und beherrscht den strukturorientierten Test sowohl auf statische als auch auf dynamische Fehler. Der Test mittels des Scan-Controllers, im Folgenden auch als SCT (*Scan-Controller-based Test*) bezeichnet, wird hier kurz erläutert und anschließend wird auf ein paar Erweiterungen eingegangen. Detailliertere Ausführungen zum Aufbau und Funktionsweise des Scan-Controllers sind in den Vorarbeiten [GGV04, KGV04, KGV05, GGV06, FRG07] zu finden.

Die Grundlage des Scan-Controllers bildet eine STUMPS-basierte Architektur. Die Erzeugung der Testmuster beruht dabei auf Pseudozufallssequenzen eines *Standard-LFSRs* mit nachgeschalteter Musteroptimierung (Abbildung 3.6). Die Rückkopplung des LFSR ist nicht fest verdrahtet, sondern kann innerhalb der Testinitialisierungsphase, aber auch während des Tests, über den Feedback-Parameter konfiguriert werden. Es handelt sich um ein sogenanntes *Multiple-Polynomial LFSR* (MP-LFSR) [HTR92]. Wie in Abbildung 3.5 dargestellt, findet die Rückkopplung in Abhängigkeit des im Feedback-Register eingestellten Wertes statt. Somit kann neben dem Startwert auch das charakteristische Polynom abgeändert werden, um möglichst günstige Sequenzen erzeugen zu lassen.

Der Testvektor aus dem kompakten LFSR wird auf die Datenbreite entsprechend der Anzahl der Scan-Ketten durch simples *Broadcast*-Verfahren erweitert. Das bedeutet, dass ein Bit aus dem LFSR auf mehrere Bitpositionen des erweiterten LFSR-Vektors

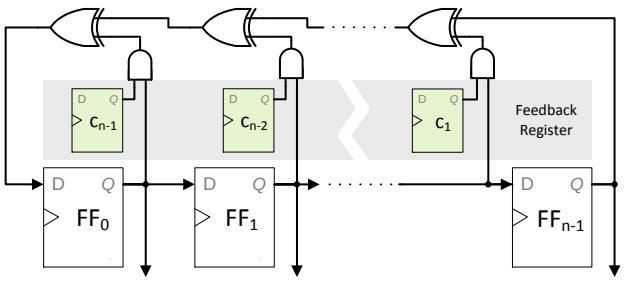


Abbildung 3.5: MP-LFSR mit konfigurierbarem Feedback

(*Extended LFSR* - ELFSR) abgebildet wird. Bei einer Datenbreite des LFSRs d_{LFSR} und einer Kettenanzahl n_{Chain} wird ein Bit auf jeweils $i = \lceil n_{\mathrm{Chain}}/d_{\mathrm{LFSR}} \rceil$ Positionen verteilt[2], gegebenenfalls mit Ausnahme des letzten Bits, das den restlichen $r \leq i$ Positionen zugeordnet ist (mit $r = n_{\mathrm{Chain}} - i\,(d_{\mathrm{LFSR}} - 1)$). Auf eine komplexere Verteilung, wie etwa durch ein XOR-Netzwerk eines klassischen *Linear Phase Shifters*, wird hier verzichtet, da ohnehin eine nachfolgende Modifikation des Testvektors vorgesehen ist. Die statische Vervielfachung des LFSR-Vektors zu Bitblöcken geht zunächst zu Lasten der Pseudozufallsverteilung der erzeugten Werte. Das heißt, eine gleichmäßige Verteilung über den gesamten Wertebereich $[0, 2^{n_{\mathrm{Chain}}}]$ ist für die durch die Erweiterung erzeugten Werte nicht mehr gegeben, da das ELFSR nur noch Binärwerte entsprechend des Blockschemas annimmt. Zum einen kann durch die in Tabelle 3.1 aufgelisteten Operationen wieder eine ausgeglichene Verteilung der resultierenden Testvektoren erreicht werden. Zum anderen sind aufgrund des hohen Anteils an unbestimmten Bits (*Don't Care Bit* - DCB) nur wenige Bitpositionen eines Testvektors für die Fehlerstimulation und -propagierung relevant. Es zeigt sich, dass der Anteil an DCBs in Testsätzen komplexer Schaltungen bei 80-90% liegt (siehe Tabelle 5.1).

Neben dem vollständig programmierbaren MP-LFSR umfasst der Dekompaktor eine ALU und ein SFR (*Scan Feed Register*) mit Bitflipping-Funktionalität. Im SFR, das in der Datenbreite den n_{Chain} Scan-Ketten entspricht, werden die nach Erweiterung und Musteroptimierung erhaltenen Testvektoren gespeichert. Mittels des *Bitflipping* kann, nach der Modifikation durch eine ALU-Operation, eine bit-genaue Anpassung an deterministische Vorgaben realisiert werden. Hierzu werden Bitflip-Adressen eingelesen und dekodiert, um den Testvektor im SFR an den spezifizierten Positionen zu invertieren. Die Kombination aus LFSR und optionaler Musteroptimierung ermöglicht die Erzeugung von pseudozufälligen, aber auch von deterministischen Mustern.

Die erzeugten Testvektoren werden während der *Shift-Phase* mit Scan-Takt in die parallelen Scan-Ketten hineingeschoben. Ist die komplette Scan-Struktur durch ein Testmuster initialisiert, erfolgt im Falle eines statischen Tests ein funktionaler Takt in der *Capture*-Phase, um die Testeingaben durch die Logik der CUT zu schalten. In den Scan-Ketten befinden sich daraufhin die Testantworten, die mit jedem weiteren Scan-Takt der folgenden Shift-Phase wieder herausgeschoben werden. Der dynamische Test erfolgt hier nach dem im LOC-Schema. Dieses Schema wurde umgesetzt, da es im Vergleich zum LOS eine simplere Scan-Struktur, mit dem geringeren Hardware-Mehraufwand, erfordert und zudem ein geringeres Risiko des Übertestens bezüglich Verzögerungsfehler besteht [KK07].

Die Kompaktierung der Testausgaben aus der CUT geschieht mittels eines MISRs, das mit einer Datenbreite $d_{\mathrm{MISR}} = n_{\mathrm{Chain}}$ sämtliche Scan-Ketten umfasst. Es findet hier also eine sequentielle Kompaktierung der Testausgaben statt. Eine adressgesteuerte Bitfix-Einheit ist ebenfalls Bestandteil der Kompaktierungseinheit. Unbekannte Schaltungszustände während des Scan-Tests, sogenannte X-Werte, beeinflussen die Testantworten und können so zu ungültigen Signaturen führen. Daher werden durch die Bitfix-Einheit

[2]Die Funktion $\lceil \cdot \rceil : \mathbb{R} \rightarrow \mathbb{Z}$ stellt die Aufrundungsfunktion dar, mit $\lceil x \rceil = \min \{k \epsilon \mathbb{Z} \mid k \geq x\}$

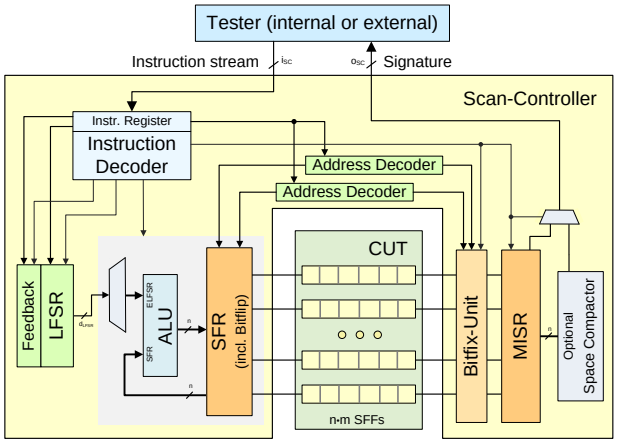


Abbildung 3.6: Aufbau des Scan-Controllers

diese X-Werte am Ausgang der Scan-Ketten maskiert, bevor das MISR die Ausgangskompaktierung durch das Abbilden mehrerer aufeinanderfolgender Testantworten auf eine Signatur realisiert.

Die SCT-Architektur kann auch modularisiert ausgelegt werden. Hierdurch wird es ermöglicht, mehrere CUTs parallel oder auch ausgewählte CUTs separat zu betreiben [FRG07]. Der Testmustergenerator wird dabei nicht unterteilt, erzeugt also einen globalen Testvektor für sämtliche Scan-Ketten der angeschlossenen CUTs. Das heißt, die Datenbreite des Testvektors entspricht der Summe der Scan-Ketten aller CUTs. Der Kompaktor wird in diesem Fall aber modularisiert, um individuelle Signaturen für die CUTs zu erhalten.

Der Scan-Controller wurde für den Low-Power-Betrieb neu konzipiert [KV08, KV10]. Das heißt, dieser ist dahingehend optimiert, den Energiebedarf und die Verlustleistung während des Tests zu senken. Während des strukturorientierten Tests wird eine Vielzahl von Testmustern durch die komplette DFT-Architektur geschoben. Durch das ständige Umschalten der Scan-Flipflops und der angeschlossenen Logik wird die Schaltung einem enormen Stress ausgesetzt. Der Scan-Test stellt somit für eine Schaltung eine weitaus höhere Belastung dar als der Normalbetrieb, in dem nur einige wenige Einheiten beziehungsweise Schaltungsteile gleichzeitig arbeiten und einzelne Einheiten gar nur höchst selten angesprochen werden.

Der Low-Power-Ansatz zur Verringerung des Energiebedarfs und des Stresses basiert auf der Reduktion der Transitionsrate. Hierzu wurde die SCT-Architektur einem neu konzipierten ATPG-Prozess, der auf dem *MT-Filling*-Verfahren (*Minimum Transition*) basiert, angepasst. Es wird dabei nach Möglichkeit nicht für jeden Schiebetakt ein komplett neuer Testvektor generiert, sondern der vorangegangene Vektor durch geschickte Operation nur an den nötigen Stellen modifiziert. Durch dieses sogenannte *Overlapping* aufeinanderfolgender Testvektoren lassen sich die Transitionen auf den Scan-Ketten beim Hineinschieben der Testmuster um bis zu 80% reduzieren, ohne die Fehlerüber-

deckung negativ zu beeinflussen [KV10]. Diese Low-Power-Version des SCTs wurde als Grundlage für die Testumgebung des USIF-Konzepts herangezogen.

3.2.1 Steuerung des Scan-Controllers

In der Tabelle 3.1 sind die verfügbaren Befehle des Scan-Controllers für die Testmustergenerierung zusammengefasst. Dieser Satz an ALU-Operationen resultiert aus der Low-Power-Optimierung des SCTs und besteht im Grunde aus Konjunktion und Disjunktion des Testvektors aus dem SFR mit dem erweiterten LFSR-Vektor.

Befehls-code	**Mnemonik**	**ALU-Operation**	**Beschreibung**
1111	*Start/ Reseed*	-	Einleiten der Konfigurationsphase; während des Test bedeutet der Befehl Reseeding und ggf. Rekonfiguration des LFSRs
1110	*Stop*	-	Ende der Testmustergenerierung
0111	*Phaseshift*	$SFR \Leftarrow rol(SFR) \oplus ELFSR$	Kontravalenz mit zyklisch links-rotiertem SFR
0110	*NOP*	$SFR \Leftarrow SFR$	Beibehalten der aktuellen Scan-Scheibe
0100	*AND*	$SFR \Leftarrow SFR \wedge ELFSR$	Konjunktion mit (inv.) erweitertem LFSR-Wert
0101	*InvAND*	$SFR \Leftarrow SFR \wedge \neg ELFSR$	
1100	*OR*	$SFR \Leftarrow SFR \vee ELFSR$	Disjunktion mit (inv.) erweitertem LFSR-Wert
1101	*InvOR*	$SFR \Leftarrow SFR \vee \neg ELFSR$	
0010	*RorAND*	$SFR \Leftarrow SFR \wedge ror(ELFSR)$	Konjunktion/ Disjunktion mit zyklisch rechts-rotiertem (inv.) erweitertem LFSR-Wert
0011	*RorInvAND*	$SFR \Leftarrow SFR \wedge ror(\neg ELFSR)$	
1010	*RorOR*	$SFR \Leftarrow SFR \vee ror(ELFSR)$	
1011	*RorInvOR*	$SFR \Leftarrow SFR \vee ror(\neg ELFSR)$	
0000	*RolAND*	$SFR \Leftarrow SFR \wedge rol(ELFSR)$	Konjunktion/ Disjunktion mit zyklisch links-rotiertem (inv.) erweitertem LFSR-Wert
0001	*RolInvAND*	$SFR \Leftarrow SFR \wedge rol(\neg ELFSR)$	
1000	*RolOR*	$SFR \Leftarrow SFR \vee rol(ELFSR)$	
1001	*RolInvOR*	$SFR \Leftarrow SFR \vee rol(\neg ELFSR)$	

Tabelle 3.1: Befehlssatz für die Testmusteroptimierung

Der Scan-Controller liest in jedem Takt ein neues Datum ein, um die Initialisierung und Konfiguration des LFSRs, die nötigen Modifikationen auf dem zu konstruierenden Testvektor oder der Bitmaskierung der Testantwort aus der CUT durchzuführen. Jeder zu erzeugende Testvektor wird durch eine Instruktion initiiert. Ist der Testvektor durch die spezifizierte Operation komplettiert, wird die nächste Instruktion für einen neuen Testvektor erwartet. Ansonsten wird eine spezifizierte Anzahl an Adressen eingelesen, um die nötigen Bitflips auf dem Testvektor durchzuführen. Falls Bitmaskierungen in-

nerhalb der aktuellen Testantwort am Ausgang der Scan-Ketten notwendig sind, so werden auch diese Adressen vor der nächsten Instruktion eingelesen.

Der Scan-Controller wird mit der Instruktion *Start* initialisiert. Das heißt, das LFSR wird konfiguriert und mit einem ersten Startwert geladen, das SFR zurückgesetzt und der Mustergenerator gestartet. Zudem wird diese Instruktion auch dazu benutzt, ein *Reseeding* des LFSRs während des Tests durchzuführen. In diesem Fall wird ein weiteres Bit des Instruktionswortes ausgewertet, um gegebenenfalls eine Rekonfiguration des LFSRs zu ermöglichen. Der Seed- und gegebenenfalls der Feedback-Wert werden daraufhin eingelesen.

Durch die Instruktion *Stop* wird der Scan-Test beendet. Es werden hierzu die Testmustergenerierung und die Kompaktierung terminiert und anschließend die Signatur aus dem MISR ausgegeben. Die *NOP*-Instruktion wird genutzt, um einen Testvektor ohne zusätzliche Modifikation durch die ALU zu erhalten. Diese Instruktion kann beispielsweise eingesetzt werden, wenn der Testvektor nur durch einige wenige Bitflips abzuändern ist. Außerdem sollten die Testeingaben für das letzte Testmuster ausschließlich aus *NOP*-Instruktionen bestehen, um beim Herausschieben der letzten Testantworten eines Testlaufs unnötige Schaltvorgänge in den Scan-Ketten zu vermeiden.

Die weiteren Befehle stellen die ALU-Operationen dar und setzen sich folgendermaßen zusammen:

- Bit 0 des Befehlscodes bestimmt die logische Verknüpfung, also Konjunktion (`'0'`) oder Disjunktion (`'1'`),
- Bit 1 gibt an, ob der ELFSR-Vektor um 1 Bit verschoben wird,
- Bit 2 bestimmt gegebenenfalls die Schieberichtung, also eine Links- (`'0'`) oder Rechtsrotation (`'1'`) des ELFSR-Vektors,
- Bit 3 gibt an, ob zusätzlich eine Invertierung des ELFSR-Vektors stattfindet.

3.2.2 Erweiterung der Dekomprimierung

Da der SCT für den Low-Power-Betrieb ausgelegt wurde, ist die Basis für die Testmustergenerierung das SFR, das in jedem Scan-Takt nur wenig modifiziert wird. Um einen Selbsttest zu realisieren, ist es aber günstiger, Pseudozufallswerte aus einem freilaufenden LFSR zu generieren. Somit wird ein hoher Speicheraufwand für den schaltungsinternen Testsatz vermieden. Es müssen nicht für jeden Testvektor die Instruktionen, sondern nur wenige LFSR-Startwerte und gegebenenfalls Feedback-Werte für Rekonfigurationen abgelegt werden, um das Einstellen neuer LFSR-Sequenzen on-chip zu ermöglichen. Das direkte Betreiben der Scan-Ketten mit dem erweiterten LFSR-Vektor ist auch nicht zielführend, da aufgrund des fixen Blockschemas, das durch die Verteilung der LFSR-Bits nach simplen *Broadcasting*-Verfahren entsteht, keine gleichmäßige Verteilung im Testmuster erreicht und somit eine wichtige Eigenschaft des Pseudozufallstest nicht erfüllt werden kann.

Die SCT-ALU wird für diese Anwendung deshalb um eine *Phaseshift*-Funktion ergänzt (siehe Tabelle 3.1). Die Hardware-Erweiterung besteht neben dem zusätzlichen Schaltnetz der Befehlsdekodierung im Grunde nur aus der XOR-Funktion. Es ist zudem ein Rotieren des SFR-Vektors notwendig, um die Blockstruktur des ELFSR-Vektors zu durchbrechen. Für die Rotier-Operation (*rol*) werden aber keine weiteren Elemente benötigt. Da die XOR-Funktion von der *Phaseshift*-Instruktion exklusiv genutzt wird, kann auf Multiplexer für das optionale Umschalten zwischen verschiedenen Eingabevektoren verzichtet werden. Der SFR-Vektor wird hier direkt links-rotiert, also um ein Bit verschoben, an den XOR-Eingang angelegt. Im Gegensatz zur Umsetzung des kombinatorischen Netzwerks eines zusätzlichen *Linear Phase Shifters*, ist hier also nur eine Erweiterung um n_{Chain} XOR-Gatter (mit je zwei Eingängen) erforderlich. Um diesen Modus der Testmustererzeugung von außen über die Testeingaben initiieren zu können, wird die Instruktion *Phaseshift* definiert.

In Tabelle 3.2 ist beispielhaft ein Auszug einer Pseudozufallssequenz für einen Selbsttest dargestellt. Aus einem 3-Bit-LFSR und einer 3-fach-Erweiterung wird hier ein 12 Bit umfassender Vektor erstellt. In der ersten Spalte ist eine beliebige (maximale) LFSR-Sequenz zu sehen. Die nächsten beiden Spalten, der erweiterte LFSR-Vektor (*ELFSR*) und der links-rotierte SFR-Vektor, bilden die Eingabewerte für die *Phaseshift*-Funktion der ALU. Der Ausgabewert, nach einer bitweisen XOR-Verknüpfung, wird wieder ins SFR geschrieben, um als pseudozufälliger Testvektor für den Scan-Test zu fungieren.

$LFSR$	$ELFSR$	$rol(SFR_{t-1})$	$SFR_t \Leftarrow ELFSR \oplus rol(SFR_{t-1})$
101	1111 0000 1111	0000 0000 0000	**1111 0000 1111**
110	1111 1111 0000	1110 0001 1111	**0001 1110 1111**
001	0000 0000 1111	0011 1101 1110	**0011 1101 0001**
011	0000 1111 1111	0111 1010 0010	**0111 0101 1101**
111	1111 1111 1111	1110 1011 1010	**0001 0100 0101**
010	0000 1111 0000	0010 1000 1010	**0010 0111 1010**
011	0000 1111 1111	0100 1111 0100	**0100 0000 1011**
100	1111 0000 0000	1000 0001 0110	**0111 0001 0110**
101	1111 0000 1111	1110 0010 1100	**0001 0010 0011**
110	1111 1111 0000	0010 0100 0110	**1101 1011 0110**
001	0000 0000 1111	1011 0110 1101	**1011 0110 0010**
011	0000 1111 1111	0110 1100 0101	**0110 0011 1010**
111	1111 1111 1111	1100 0111 0100	**0011 1000 1011**
010	0000 1111 0000	0111 0001 0110	**0111 1110 0110**
011	0000 1111 1111	1111 1100 1100	**1111 0011 0011**
100	1111 0000 0000	1110 0110 0111	**0001 0110 0111**

Tabelle 3.2: Beispiel einer erzeugten Pseudozufallssequenz für den Selbsttest

Da ein Testvektor hier nicht nur vom LFSR-Wert, sondern auch von seinem Vorgängerwert abhängt, kann eine LFSR-Sequenz wiederholt durchlaufen werden, ohne eine Wiederholung in der resultierenden Testvektorsequenz zu verursachen. Es ist zu sehen,

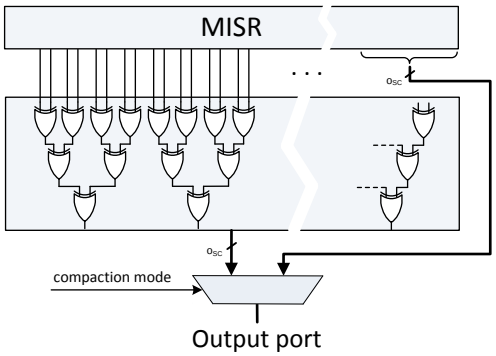


Abbildung 3.7: Kompaktor des Scan-Controllers

dass nach einer kurzen Einschwingphase der Testvektor keine Struktur von Bitblöcken mehr aufweist. Bei Bitblöcken der Länge $i = \lceil n_{\text{Chain}}/d_{\text{LFSR}} \rceil$ bedarf es anfänglich i Takte, um das SFR um einen Block zu verschieben und so die Bits zu verteilen. Durch diese zusätzliche *Phaseshift*-Funktion kann eine gleichmäßige Verteilung der erzeugten Werte im Testsatz erhalten und somit in der vorgegebenen SCT-Architektur auch ein Pseudozufallstest on-chip realisiert werden.

3.2.3 Erweiterung der Kompaktierung

Die USIF-Architektur soll nicht nur einer speziellen Testarchitektur angepasst, sondern für diverse Konzepte ausgelegt werden. Daher werden auch verschiedene Kompaktierungsverfahren unterstützt. Die Kompaktierung im SCT erfolgt ursprünglich über ein MISR, welches die Testantworten eines kompletten Testsatzes in einer Signatur zusammenfasst und dementsprechend nur einmal am Ende des Tests auszulesen ist. Im Falle einer kombinatorischen Kompaktierung, wie beim EDT, muss beim Herausschieben der Testantworten in jedem Scan-Takt eine kompaktierte Signatur ausgelesen und weiterverarbeitet werden können.

Die SCT-Architektur soll ebenso die Möglichkeit bieten, zu beliebigen Zeitpunkten während des Testlaufs Signaturen entnehmen zu können. Daher wird die Kompaktierungseinheit um einen zusätzlichen *Space Compactor* erweitert, der optional eine in Datenbreite kompaktierte Zwischensignatur am Ausgangsport des Scan-Controllers bereitstellt (siehe Abbildung 3.6). Somit kann die SCT-Architektur als Anwendungsfall sowohl für ein sequentielles als auch für ein kombinatorisches Kompaktierungsverfahren dienen.

Diese kombinatorische Kompaktierung wird durch einen das komplette MISR umfassenden XOR-Baum entsprechend der Abbildung 3.7 umgesetzt. Dieser Ansatz der Kompaktierung (*Mixed Time and Space Compaction*) entspricht dem des OPMISR+ [BBD02]. Um die Signatur eines MISRs der Datenbreite d_{MISR} auf die Datenbreite o_{SC} des Ausgangsports zu kompaktieren, wird ein XOR-Baum mit einer Stufenanzahl

$$s = \lceil \mathrm{ld}(d_{\mathrm{MISR}}) \rceil - \lceil \mathrm{ld}(o_{\mathrm{SC}}) \rceil$$

benötigt[3]. Daraus ergibt sich ein Hardware-Mehraufwand an zusätzlichen XOR-Gattern von

$$n_{\mathrm{Gate}} = \sum_{i=1}^{s} \left\lceil \frac{d_{\mathrm{MISR}}}{2^i} \right\rceil < d_{\mathrm{MISR}}.$$

Im Falle eines modularisierten Scan-Controllers mit mehreren MISR-Komponenten, wird der XOR-Baum über alle Module gespannt. Das heißt, es werden die MISR-Signaturen zur weiteren XOR-Kompaktierung wie eine einzelne Signatur behandelt.

Durch die Erweiterung wird es ermöglicht, auch optional Signaturen während des Testlaufs in jedem Takt oder jeweils nach einer spezifizierten Anzahl an Takten am Ausgangsport auszulesen. So kann zum Beispiel für jedes Testmuster, also nachdem dessen Testantworten im MISR zu einer Signatur akkumuliert wurden, die zugehörige Zwischensignatur ausgewertet werden. Durch ein frühzeitiges Erkennen eines Fehlers mittels einer Zwischensignatur kann ein langer Testlauf vermieden werden. Ein umfangreicherer Satz an Analysedaten mit vielen Zwischensignaturen kann auch der Diagnose zu Gute kommen, wenn ein Fehler dadurch auf wenige Fehlerpunkte begrenzt wird.

Am Ende des Testlaufs wird stets die komplette MISR-Signatur in $k = \lceil d_{\mathrm{MISR}}/o_{\mathrm{SC}} \rceil$ Takten aus dem MISR geschoben.

3.3 Analyseverfahren

Für die Auswertung der Testantworten wurde die Analysekomponente entworfen und in die USIF-Architektur integriert. Es werden folgende Modi der Testanalyse unterstützt:

- *LogAll*, die Ausgabe aller Testergebnisse,
- *LogFaults*, die Ausgabe der fehlerhaften Testergebnisse inklusive zugehöriger Zeitstempel,
- *PassFail*, die on-chip Auswertung der Testergebnisse mittels off-chip Referenzdaten, und
- *BIST*, die on-chip Auswertung der Testergebnisse mittels on-chip Referenzdaten.

[3] Mit der Funktion ld (logarithmus dualis) wird der Logarithmus zur Basis 2 ermittelt, mit $\mathrm{ld}\,(x) = \log_2(x)$.

3.3.1 Analysefilter

Die Analysekomponente enthält eine Filterfunktion, um die Möglichkeit zu bieten, nicht sämtliche Testantworten auswerten zu müssen und somit den Referenzdatensatz und, im Falle der *Log*-Modi, den Datensatz an zu speichernden Testantworten zu reduzieren. Unabhängig vom gewählten Auswertungsmodus sorgt dieser Filter stets für eine spezifizierbare Vorauswahl der Testantworten.

Bevor ein Scan-Test gestartet wird, kann eine Sample-Rate a_{Rate} und eine Basis a_{Base} konfiguriert werden. Während des Tests wird ein Modulo-(a_{Rate})-Zähler mit jedem Scan-Takt der Shift-Phase, also mit jeder neuen Testantwort, inkrementiert. Dieser Zähler, im Folgenden auch als Shift-Counter bezeichnet, enthält also die Anzahl der im Testlauf erfolgten Schiebetakte. Dieser kann beispielsweise nach jedem funktionalen Takt zurückgesetzt werden, um nur die Anzahl an Schiebetakten pro Testmuster zu zählen. Die Basis a_{Base}, mit $a_{\text{Base}} < a_{\text{Rate}}$, gibt den Startwert an, ab welcher Testantwort mit der Auswertung begonnen werden soll. Eine neue Testantwort wird nur geladen, wenn der Shift-Counter s den nächsten gültigen Wert erreicht, so dass $s \mod a_{\text{Rate}} = a_{\text{Base}}$ erfüllt ist. Es findet also vor der Speicherung oder gegebenenfalls vor der Analyse eine Filterung auf jede (a_{Rate})-te Testantwort statt.

Dieses $(a_{\text{Rate}}, a_{\text{Base}})$-Tupel ist während der Initialisierungsphase eines Tests zu spezifizieren. Es muss dabei eine Rate $a_{\text{Rate}} \geq 1$ angegeben werden. Bei einem spezifizierten Basiswert $b \geq a_{\text{Rate}}$, wird der entsprechend kleinste Repräsentant $a_{\text{Base}} = b \mod a_{\text{Rate}}$ übernommen. Die Voreinstellung im USIF-Controller ist $(a_{\text{Rate}}, a_{\text{Base}}) = (1, 0)$, so dass, begonnen bei der ersten Testausgabe, jede weitere zur Auswertung ausgelesen wird, so also keine Vorfilterung für den Analyzer stattfindet.

Im Falle einer kombinatorischen Kompaktierung, die durch den EDT umsetzt wird, ist jede Testausgabe auszuwerten, da ansonsten eine gegebenenfalls verfälschte Antwort übersehen werden könnte. Bei einer sequentiellen Kompaktierung muss nicht jede Testausgabe in die Auswertung mit einfließen. Da mehrere Testantworten aus den Scan-Ketten zu einer Signatur akkumuliert werden und sich ein Fehler durch die Rückkopplungen innerhalb des MISRs fortpflanzt, genügt es, Signaturen nach jeweils einer bestimmten Anzahl an Scan-Takten zu entnehmen. Im Falle des SCTs können, bei einer Scan-Tiefe m_{Chain}, die Werte auf $(a_{\text{Rate}}, a_{\text{Base}}) = (m_{\text{Chain}}, m_{\text{Chain}} - 1)$ gesetzt werden, um nach jeder Shift-Phase eine räumlich kompaktierte MISR-Signatur, also eine Zwischensignatur pro Testmuster, auszulesen.

3.3.2 Komplettaufzeichnung

In diesem Modus, bezeichnet als *LogAll*, werden sämtliche Ergebnisse der Testarchitektur, die den konfigurierten Filter passieren, im TestOut-FIFO aufgezeichnet. Hierbei muss das TestOut-FIFO regelmäßig ausgelesen werden. Der Scan-Test wird durch den USIF-Controller angehalten, sobald kein Speicherplatz im TestOut-FIFO vorhanden ist, und kann erst wieder mit freigegebenem Speicher fortgesetzt werden.

3.3.3 Fehlerfilter

Sollen nur die fehlerhaften Testantworten gespeichert werden, ist der Modus *LogFaults* zu wählen. Es findet somit bereits eine Filterung auf fehlerhafte Ausgaben on-chip statt. Hierzu müssen zusätzlich zu den Testeingaben die Referenzdaten zu den Testantworten über den Testzugang übertragen und im TestRef-FIFO abgelegt werden. Ein Fehler in der zu testenden Schaltung ist erkannt, wenn eine Testantwort nicht dem zugehörigem Referenzwert, der zeitgleich aus dem TestRef-FIFO geladen wird, entspricht. Zu jeder abweichenden Testantwort muss zusätzlich ein Zeitstempel gespeichert werden. Der Zeitstempel setzt sich aus der Pattern-Nummer p, die der Anzahl bisheriger Testmuster eines Testlaufs entspricht, und der Shift-Nummer s, die sich aus der aktuellen Anzahl an Schiebetakten der aktuellen Shift-Phase ergibt, zusammen. Hierzu wird ein Shift-Counter mit jeder innerhalb der Shift-Phase herausgeschobenen Testantwort inkrementiert und vor Beginn der nächsten Shift-Phase, also in der Capture-Phase, zurückgesetzt. Zeitgleich mit dem Reset des Shift-Counters wird der Pattern-Counter inkrementiert.

Zu jedem Fehler ist somit ersichtlich, in welcher Testantwort welchen Testmusters dieser erkannt wurde. Die Zeitstempel werden in dem separaten Speicher *TS-FIFO*, der synchron zum TestOut-FIFO betrieben wird, geschrieben. Das heißt, im *LogFaults*-Modus wird mit jedem Schreibzugriff auf das TestOut-FIFO zeitgleich das Wertepaar (p, s) in das TS-FIFO übernommen. Die beiden Speicher werden separat ausgelesen. Eine Testantwort und ein Zeitstempel sind dabei aufgrund der gleichen Speicheradresse, die sich durch das FIFO-Prinzip aus dem relativen Zeitpunkt des Schreib- beziehungsweise Lesezugriffs ergibt, eindeutig einander zuordenbar.

3.3.4 Pass/Fail-Test

Im *PassFail*-Modus wird der Test abgebrochen und ein entsprechendes Fehlerbit im USIF-Statusregister gesetzt, sobald ein Fehler erkannt wird. Zudem wird die verfälschte Testantwort mit zugehörigem Zeitstempel gespeichert, um diese optional auslesen zu können. Diese Information kann für eine darauffolgende optimierte Diagnose genutzt werden, in der der Suchraum bereits eingeschränkt werden kann.

Der *BIST*-Modus entspricht dem *PassFail*-Modus, nur das hier die on-chip verfügbaren Testeingabe- und Referenzdaten zum Betreiben der Testarchitektur beziehungsweise zur Auswertung der Testantworten genutzt werden.

3.4 Testszenarien

Das Ziel dieser Entwicklung ist es, mit derselben Testumgebung optional die folgenden Anwendungsfälle für den Offline-Test zu unterstützen:

- den Produktionstest
  - nach Packaging beim IC-Hersteller,
  - nach Leiterplattenbestückung beim ECU-Hersteller,
- den on-chip Selbsttest im Zielsystem,
- die Fehleranalyse im Zielsystem und
- das Monitoring während des Normalbetriebs.

Die folgenden Szenarien werden anhand der SCT-Architektur als exemplarische Testumgebung betrachtet. Diese sind aber ohne Weiteres auch auf andere Testarchitekturen adaptierbar.

3.4.1 Produktionstest

Das hier als Produktionstest bezeichnete Szenario bezieht sich vor allem auf den Test des bereits auf der Leiterplatte aufgebrachten ICs. Das USIF-Konzept ermöglicht es, über eine vorhandene Standardschnittstelle diesen IC mit hoher Fehlerüberdeckung strukturorientiert zu testen. Ein Produktionstest nach dem Packaging kann ebenfalls durch das USIF unterstützt werden. Hierbei werden die Testinformationen direkt über das Load Board des Testers übermittelt. Im Falle eines on-chip Bus-Controllers kann der Test über die entsprechende serielle Schnittstelle durchgeführt werden.

Für den Produktionstest ist es von entscheidender Bedeutung, die Testzeit aufgrund eines hohen Testdurchsatz und somit niedrigeren Testkosten so gering wie möglich zu halten. Es werden deshalb vom ATE über die serielle Standardschnittstelle hochgradig komprimierte Testinformationen übertragen, welche mittels des Scan-Controllers zur Steuerung des LFSR-basierten Mustergenerators dienen (Abbildung 3.8). Hierzu wird der durch die eingestellte Rückkopplung erzeugte Vektor dem LFSR entnommen, durch Bitvervielfachung, entsprechend der Anzahl der parallelen Scan-Ketten, erweitert und mittels ALU- und Bitflip-Operationen optimiert. Da es sich hierbei um einen Pass/Fail-Test handelt, genügt es am Ende des Testlaufs die finale Signatur aus dem MISR auszuwerten. Darüber hinaus ist es aber auch möglich Zwischensignaturen abzugreifen und den Test, im Falle eines erkannten Fehlers, zeitiger abzubrechen.

Da die SCT-Architektur einen Analyzer enthält, könnte die Auswertung hierbei auch in den IC verlagert werden. Es werden also, anstatt kompaktierte Signaturen auszulesen, die zugehörigen Referenzsignaturen übertragen. Der Tester empfängt dann über das USIF nur die Information, ob ein Fehler erkannt wurde oder nicht.

Angenommen die eingebettete Teststruktur des zu testenden ICs besitzt 16 Eingangskanäle für die komprimierten Testdaten und wird mit einem Scan-Takt von 20 MHz

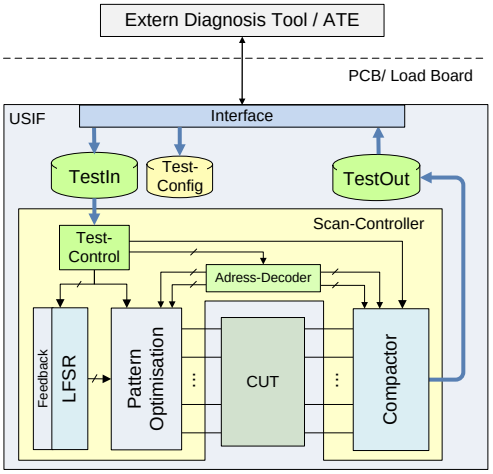


Abbildung 3.8: Szenario des Produktionstests

betrieben. Dann beträgt die maximale Datenrate $r = 20\,\mu s^{-1} \cdot 16$ Bit $= 320$ MBit/s. Das bedeutet, um eine Testzeit zu erreichen, die ein ATE im IC-Fertigungstest benötigt, bedarf es einer seriellen Schnittstelle mit entsprechend hohem Datendurchsatz. Dies wäre durch den USB-Standard ab der HighSpeed-Version, mit Übertragungsraten von bis zu 480 MBit/s, gewährleistet. Stehen nur Automotive-Schnittstellen zur Verfügung, so muss entweder eine hohe Testzeit für einen umfangreichen Testsatz in Kauf genommen oder der Testumfang auf Kosten einer hohen Fehlerüberdeckung reduziert werden.

3.4.2 Eingebauter Selbsttest

Selbsttests werden angewandt, um eingebettete Systeme im Feld auf Hardware-Defekte zu testen und so die Systemverfügbarkeit sicherzustellen. Die Selbsttests in ICs der Automobilindustrie beruhen traditionellerweise auf funktionale Tests [ACE14]. Es werden zumeist Watchdogs eingesetzt, um funktionale Ausgaben eines Prozessors auszuwerten. Im Gegensatz dazu ermöglicht ein Selbsttest unter Verwendung der Produktionstestlogik einen strukturorientierten Test, der auch in begrenztem Zeitrahmen den heutigen Anforderungen an hoher und aussagekräftiger Fehlerüberdeckung gerecht wird.

Lässt sich aufgrund der durch den strukturellen Test erhöhten Prüfschärfe ein Fehler frühzeitig erkennen, so kann darauf entsprechend reagiert werden, womöglich bevor sich dieser auf die Funktionalität auswirkt. Somit kann einem Fehlverhalten vorgebeugt und die Systemverfügbarkeit erhöht werden. Gerade für sicherheitskritische Anwendungen ist vor Inbetriebnahme eines Steuergerätes ein solcher Test, der die Wahrscheinlichkeit einer anschließenden Fehlfunktion senkt, von essentieller Bedeutung.

Ein struktureller Test muss offline in Phasen, in denen die Anwendungen eines Systems inaktiv sind, durchgeführt werden. In einem Fahrzeug sind dies die *Startup-*

und die *Shutdown*-Phase. Startup bezeichnet die Phase zwischen dem Anschalten des Fahrzeugs und dem betriebsbereiten Zustand. Die Shutdown-Phase ist der Zeitraum vom Ausschalten des Fahrzeugs durch den Fahrer bis zum tatsächlichen Ruhezustand (*Power-down Mode*). Die Startup-Phase umfasst nur wenige Millisekunden, da das System aus Qualitätsgründen dem Nutzer schnellstmöglich betriebsbereit zur Verfügung stehen muss. Für das Herunterfahren ist ein weitaus größeres Zeitfenster vorgesehen. Doch selbst in diesem kann nur ein eher begrenzter Test untergebracht werden. In [ACE14] ist dargestellt, wie solch ein Selbsttest in die Shutdown-Phase eines Fahrzeugs integriert werden kann.

Eine weitere wesentliche Anforderung an den Test ist ein möglichst minimaler Mehraufwand im Design, um zusätzliche Kosten gering zu halten. Dies wird erfüllt durch die Wiederverwendung der bereits im Design integrierten Testlogik, die im IC-Fertigungstest angewandt wird. Außerdem darf der vorzuhaltende Testsatz nur einen geringen zusätzlichen Speicherplatz beanspruchen. Hierzu wird der Test auf die schaltungsinterne Generierung von Pseudozufallsmustern ausgelegt. Die in begrenzter Zeit dadurch zu erreichende Fehlerüberdeckung kann je nach Testeigenschaften der CUT bereits relativ hoch ausfallen. Um diese gegebenenfalls auf einen erforderlichen Mindestwert zu erhöhen, gibt es die Möglichkeit, zusätzlich einen geringen Satz an deterministischen Mustern on-chip zu speichern.

Im Falle des SCTs können mittels des freilaufenden LFSRs Pseudozufallsmuster erzeugt werden. Hierzu wird, wie in Abbildung 3.9 verdeutlicht, das LFSR mit einer günstigen Anfangsbelegung gestartet und im Bedarfsfall mit neuen Seed- und Feedback-Werten versehen. Um einen Testvektor aus dem LFSR auf die Scan-Ketten zu verteilen, wird die *Phaseshift*-Funktionalität genutzt. Eine komplexe Aufbereitung der erzeugten Testvektoren entfällt. Im internen Speicher muss im Falle eines rein pseudozufallsbasierten Tests nur der Satz an Seed/Feedback-Paaren abgelegt werden. Die Testantworten werden im MISR zu einer Signatur kompaktiert und am Ende des Tests mit einem abgespeicherten Referenzwert verglichen. Da dieser Modus einen Pass/Fail-Test darstellt, enthält die Testausgabe nur die Information, ob ein Fehler erkannt wurde oder nicht. Die Übermittlung des Testergebnisses (*Pass/ Fail*) und gegebenenfalls auch die Initiierung des Tests geschehen über die zur Verfügung stehende Automotive-Schnittstelle.

Da hier in jedem Scan-Takt ein Testvektor in die Scan-Ketten geschoben wird, ist die Testzeit die Summe über die Laufzeiten der LFSR-Sequenzen. Dieser Wert entspricht einer Anzahl an Testmustern n_{Pat} mal der Scan-Tiefe m_{Chain}, die durch die Scan-Kette mit der höchsten Anzahl an Flipflops bestimmt ist. Hinzu kommt ein weiterer Takt pro Testmuster für die Capture-Phase. Für diese Phase, in der die Scan-Struktur funktional mit der Clock der CUT betrieben wird, ist ebenso der Zeitraum eines Scan-Taktes, mit der Frequenz f_{SC}, vorgesehen. Somit ergibt sich eine Testzeit von:

$$t = n_{\mathrm{Pat}} * \frac{m_{\mathrm{Chain}}+1}{f_{\mathrm{SC}}}.$$

Sind Scan-Struktur und Scan-Takt durch das Testdesign vorgegeben, hängt die Zeit also von der Anzahl der für den Test zu spendierenden Testmuster ab. Angenommen die Scan-Tiefe beträgt $m_{\mathrm{Chain}} = 99$ und die Scan-Frequenz $f_{\mathrm{SC}} = 20$ MHz. Dann wird eine Zeit von $5\,\mu s$ pro Testmuster benötigt. Bei einem vorgegebenen Zeitfenster für

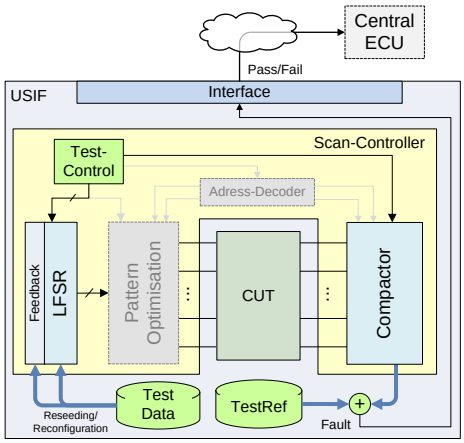


Abbildung 3.9: Szenario des eingebetteten Selbsttests

einen Startup-Test von beispielsweise 10 ms darf der erzeugte Testsatz nur bis zu 2000 Testmuster umfassen. Eine hohe Fehlerüberdeckung kann hierbei nur mit zusätzlichen deterministischen Testinformationen garantiert werden. Steht in der Shutdown-Phase eine Testzeit im Sekundenbereich zur Verfügung, kann auch mit rein pseudozufallsbasierten Mustern nahezu 100% Fehlerüberdeckung erreicht werden [ACE14].

3.4.3 Fehleranalyse

Durch den Testzugang über serielle Standardschnittstellen wird ein neuer Testmodus in der Postproduktionsphase bereitgestellt. Dieser Test entspricht dem Ablauf des zuvor vorgestellten Szenarios des Produktionstests. Nur das hier zusätzlich eine diagnostische Auswertung der erhaltenen Testausgaben erfolgt. Außerdem müssen keine strengen Vorgaben der Testdauer eingehalten werden.

Ein im Feld durchzuführender diagnostischer Test, etwa während eines Werkstattaufenthalts, wendet einen durch den Halbleiterhersteller zur Verfügung gestellten Testsatz an. Für Zwecke der erweiterten Fehlererkennung mit beschränkten Diagnosefähigkeiten ist hier ein Testmodus, der ohne die volle Verfügbarkeit der Testinformationen aus dem Fertigungstest auskommt, hinreichend. Es wird hier eine Ausgabe kompaktierter Testantworten vorgesehen, welche Aussagen über Fehlerdichten und eventuell fehlerhafte Baugruppen erlaubt.

Im Falle eines Rückläufers zum Automobilhersteller oder nach Weiterreichung zum Steuergerätehersteller wird ein möglicherweise als störanfällig gemeldetes Steuergerät einer umfangreicheren Diagnose unterzogen. Der Testumfang und somit die Diagnosetiefe unterliegt dabei der Freigabe durch den Halbleiterhersteller. Spätestens ein zur detaillierten Untersuchung zum Halbleiterhersteller zurückgeführtes Modul kann im vollen Umfang ohne restriktive Einschränkungen analysiert werden. Die Auswertung des umfangreichen Tests erlaubt eine feingranulare Diagnose.

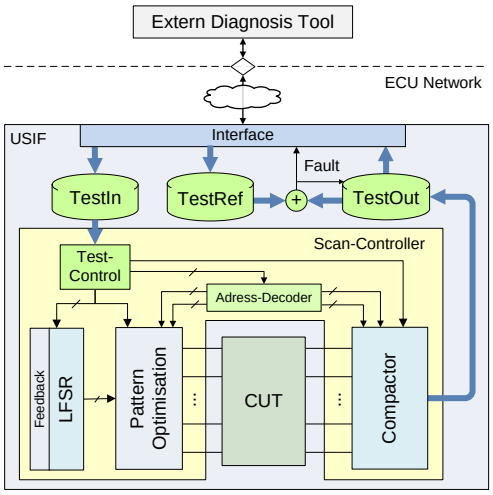


Abbildung 3.10: Szenario des diagnostischen Tests

Für dieses Testszenario steht der volle Funktionsumfang des USIFs zur Verfügung (Abbildung 3.10). Das heißt, die Testkonfiguration und die Wahl des Scan- und Analysemodus sind den spezifischen Bedürfnissen anpassbar. Entscheidend sind hier die Analysemodi *LogAll* und *LogFaults*, da dieser Test auf das Auslesen der Testausgaben abzielt. Die diagnostische Auswertung geschieht off-chip auf speziellem Testequipment oder auf anwendernahen EDV-Systemen. Die Übermittlung der Testdaten erfolgt bei einem Automotive-Steuergerät über eine vorhandene Schnittstelle, typischerweise zu einem CAN- oder FlexRay-Bordnetz.

Diagnostische Auswertung

Für eine Diagnose müssen die Testantworten ausgewertet und interpretiert werden. Um von einem verfälschten Bit in der Ausgabe der Scan-Ketten direkt auf einen Fehler zu schließen, müssten sämtliche Testantworten unkompaktiert abgespeichert und ausgelesen werden. Dies ist grundsätzlich durch den in vielen Testumgebungen angebotenen Bypass-Modus möglich, aber bezüglich des Kosten-Nutzen-Verhältnisses nicht unbedingt zielführend. Das heißt, das erhöhte Datenaufkommen und demzufolge die lange Testzeit garantieren keine erheblich verbesserte Diagnosetiefe oder, umgekehrt gesagt, eine relative hohe Diagnosetiefe kann auch durch hochkompaktierte Ausgaben erreicht werden. Daher ist es oft hinreichend, auch zu Diagnosezwecken Kompaktierungsverfahren anzuwenden. Da eine wichtige Eigenschaft der Kompaktierung die sehr geringe Wahrscheinlichkeit einer Fehlermaskierung ist, spiegeln sich Fehler auch mit hoher Sicherheit in einer verfälschten Signatur wieder.

Die Komprimierung der Testantworten geht mit einem Informationsverlust einher, was ein Zurückrechnen einer Signatur auf eine konkrete Fehlerursache nicht mehr möglich macht. Bei einer diagnostischen Auswertung von komprimierten Signaturen wird daher

ein zugehöriges Fehlerwörterbuch (*Fault Dictionary*) angewandt. Solch ein Wörterbuch ordnet jedem Fehler, gegebenenfalls auch Mehrfachfehler, eine Signatur zu.

Hierzu muss in der Fehlersimulation, die Teil des ATPG-Prozesses ist, für die Fehler jedes Fehlerpunktes die erzeugte Signatur abgespeichert werden. Jeder Einzelfehler oder, im Falle von Mehrfachfehlern, jede Kombination von Fehlern erzeugt eine eindeutige Signatur. Umgekehrt kann eine Signatur aber aus verschiedenen Fehlern resultieren. Da es sich also um eine surjektive Abbildung handelt, lässt sich bei einer vom fehlerfreien Referenzwert abweichende Signatur nicht unbedingt auf einzelne Fehler, aber zumindest auf eine Klasse von Fehlern schließen. Es kann aber auch zur Fehlinterpretation kommen, wenn beispielsweise eine Signatur, die aus einem Mehrfachfehler resultiert, auf einen Einzelfehler verweist. Es besteht aber die Möglichkeit, mittels der gewonnenen Informationen feingranulare Diagnosen durchzuführen, die auf jeweils angenommene Fehlerursachen abzielen.

3.4.4 Monitoring

Während des Normalbetriebs kann es zu Fehlern kommen, die in einer darauffolgenden Diagnose nicht reproduzierbar sind. Ein Fehler der nur im Betriebszustand, beispielsweise im Zusammenspiel mehrerer Bauteile, auftritt, kann auch nur während diesem beobachtet werden. Daher kann die Überwachung des eingebetteten ICs im Betriebszustand von großer Bedeutung sein. Spezielle Zugriffspunkte innerhalb des ICs werden mittels eines Test-Controllers kontinuierlich ausgelesen und in einem internen Speicher abgelegt. Es befindet sich dabei immer der aktuelle Ausschnitt des Datenstroms im Speicher. Ein registrierter Fehler führt schließlich dazu, dass die Aufnahme sofort oder nach einer bestimmten Anzahl an Folgezuständen angehalten und zusätzlich ein zugehöriger Zeitstempel abgespeichert wird. Anschließend kann eine Auswertung der vor und gegebenenfalls nach dem Auftritt der Fehlfunktion gesammelten Systemzustände erfolgen. So kann die Signalkombination, die zum Fehler führte, nachvollzogen werden. Hierbei steht in Abhängigkeit der zu beobachtenden Signale, des daraus folgenden Datenumfangs und des verfügbaren Speicherplatzes nur ein sehr begrenztes Zeitfenster zur Verfügung. Daher sollte die Untersuchung auf jeweils eine Teilfunktion fokussiert und die Aufnahme auf wenige primäre Ausgänge einzelner Module beschränkt werden.

Um auch dedizierte Scan-Zellen beobachten zu können, muss deren Inhalt parallel zum Normalbetrieb aus den Scan-Ketten herausgeschoben und in einem Testspeicher abgelegt werden können. Ein bekannter DFT-Ansatz, der solch einen Modus unterstützt, ist das sogenannte *Snapshot Scan*-Design, das erstmals in [Ste78] vorgestellt wurde. Mittels des *Snapshot Scan* können interne Zustände der sequentiellen Elemente der zu untersuchenden Schaltung im Betriebszustand ausgelesen werden, ohne die Funktionsausführung zu beeinflussen [Wan06]. Hierzu werden die zu beobachtenden Speicherelemente um zusätzliche Scan-Zellen erweitert. Für diese Scan-Zellen werden ein oder mehrere zusätzliche Clock-Signale benötigt, je nachdem, ob das Scan-Design als *D-Muxed-Scan*, *Clocked-Scan* oder *LSSD* umgesetzt wird. Das bedeutet, dieses Scan-Design stellt ein separates Schaltungsnetz dar, das an dedizierte Speicherelemente geknüpft ist.

Um diesen Modus zu realisieren, sind also entsprechende Ressourcen vorzusehen, wie der für das Zeitfenster zu reservierende Speicherplatz, zusätzliche Signalleitungen und gegebenenfalls Scan-Zellen als interne Beobachtungspunkte. Da das Auslesen solcher Monitoring-Daten nicht Teil der Kommunikation im Betriebszustand ist, muss dies in Ruhe-Phasen der zu beobachtenden Anwendung geschehen. Die dadurch gewonnenen Momentaufnahmen ermöglichen eine verbesserte Echtzeitdiagnose im Betriebszustand.

3.5 Zusammenfassung

In diesem Kapitel wurde das Konzept des Zugangs zur schaltungsinternen Testarchitektur vorgestellt. Das konzeptionelle Design und die nötige Steuerungseinheit für die Zugriffe auf Speicher und Testarchitektur wurde betrachtet. Als repräsentative Testarchitektur wurde hier der Scan-Controller herangezogen. Daher wurde eine Einführung in dessen Aufbau und Funktionsweise gegeben.

Der Scan-Controller musste erweitert werden, um diesen auch für einen Selbsttest mit zufallsbasierten Testmustern auszulegen. Hierzu wurde die *Phaseshift*-Funktion hinzugefügt, die für eine gleichmäßige Verteilung der LFSR-Muster über eine hohe Anzahl von Scan-Ketten sorgt. Damit der Scan-Controller verschiedene Analysemodi unterstützt, war ebenso eine Erweiterung der Kompaktierungseinheit vonnöten. Somit wird neben der sequentiellen auch eine kombinatorische Kompaktierung von Testausgaben ermöglicht. Diese zusätzliche Kompaktierung der Signaturen bietet die Option des kontinuierlichen Auslesens von Testantworten und somit erweiterte Analysemöglichkeiten.

Die präsentierten Auswertungsverfahren werden durch die entworfene Analysekomponente realisiert. Diese unterstützt entweder einen Pass/Fail-Test oder eine Fehlerdiagnose durch die Aufnahme sämtlicher oder nur fehlerhafter Testantworten. Des Weiteren ist eine Filterfunktion Teil der Analyse, um eine Vorauswahl der auszuwertenden Testantworten umzusetzen.

Es konnten mehrere Anwendungsfälle, die sich durch dieses Testkonzept ermöglichen lassen, aufgezeigt werden. Diese sind vor allem der erweiterte Produktionstest beim Steuergerätehersteller, der eingebaute Selbsttest, die Fehlerdiagnose im Feld und die Fehleranalyse beim Steuergeräte- oder Halbleiterhersteller.

Kapitel 4

4 Umsetzung des Universal Scan-Test Interface

Das USIF-Konzept ist prototypisch auf Basis programmierbarer Logik umgesetzt. Hierzu wurde das USIF-Design als komplett synthetisierbare VHDL-Beschreibung implementiert und auf einem FPGA (*Field Programmable Gate Array*) als Testumgebung einer CUT realisiert. Als Standardschnittstellen wurden ein USB-, ein FlexRay- und ein CAN-Controller integriert, die jeweils für die protokollkonforme Übersetzung der Testdaten sorgen.

Die Architektur wurde generisch ausgelegt. Somit sind zum einen die einzelnen Module individuell integrierbar und zum anderen in ihrer Auslegung konfigurierbar. Hierzu ist in einer separaten USIF-Bibliothek eine Vielzahl an Konfigurationsparametern zusammengefasst. Der Umfang sämtlicher im USIF oder in Untermodulen enthaltener Speicherblöcke und Register kann an dieser Stelle spezifiziert werden. Die Kommunikations-Controller können dem Design je nach Bedarf hinzugefügt werden. Ferner sind SCT und EDT nach Belieben austauschbar. Das Design kann somit einem spezifischen Anwendungsfall angepasst werden.

Als Beispiel hierfür sei der Testausgabespeicher betrachtet. Dieser kann im Falle nur wenig zu speichernder oder auszuwertender Testantworten entsprechend kompakt ausgelegt werden. Der Speicher der Referenzdaten ist dabei gleichermaßen anzupassen. Ist nur eine geringe Anzahl an MISR-Signaturen vorgesehen, kann so unnötiger Speicheraufwand vermieden werden. Der Umfang von Registern kann ebenso entsprechend der spezifischen Anwendung konfiguriert werden. Sind die Scan-Struktur und ein Maximalwert für die Testmusteranzahl bekannt, kann zum Beispiel die maximal benötigte Datenbreite der Zähler für Analysefilter und Zeitstempel festgelegt werden.

4.1 Kommunikations-Controller

4.1.1 USB-Controller

Mittels der USB-Schnittstelle lässt sich von einer Anwendung auf einem Diagnosegerät oder PC über den USB-Host auf das zu testende periphere System zugreifen. Der USB-Controller des USIF setzt die Protokollspezifikation 2.0, ausgenommen der HighSpeed-Funktionalität, um und wurde komplett in Hardware implementiert. Somit ist ein Großteil des USB-Stacks im USB-Controller integriert. Der Controller wird dabei auf die für das USIF nötige Funktion des Datenaustausches beschränkt. So kann beispielsweise auf die Datenübertragungsart des isochronen Transfers verzichtet werden, da die Endpunkte für die verlustfreie, fehlerkorrigierte Übertragung des Bulk- oder Interrupt-Transfers auszulegen sind. Nichtsdestotrotz soll der Controller sämtliche Standardanfragen der USB Spezifikation 2.0 beherrschen und darauf entsprechend reagieren.

Da der implementierte USB-Controller als Teil der USIF-Architektur auf einem FPGA-Entwicklungsboard realisiert wird, sind durch die verfügbare Hardware seine Eigenschaften bezüglich Geschwindigkeit und Übertragungsmodi begrenzt. Daher kann der Controller hier nur als LowSpeed- oder FullSpeed-Version konfiguriert und betrieben werden. Der USB-Transceiver, der für die prototypische Umsetzung genutzt wird, unterstützt ebenso die Datenraten LowSpeed und FullSpeed. Der für den USB-Controller benötigte Takt von 48 MHz wird mittels einer PLL (*Phase-Locked Loop*) aus einem 50 MHz-Takt generiert.

Die für die Umsetzung der USB-Schnittstelle genutzten Ressourcen sind:

- Altera Cyclone II *EP2C70F896C6N*,
- Quarz-Oszillator 50 MHz, mittels PLL auf 48 MHz heruntergetaktet,
- USB Transceiver *Texas Instruments TUSB1106*.

Die Konfiguration erfolgt durch die Deskriptoren, die im Anhang A.4 angegeben sind.

4.1.1.1 Endpunkte

Im USB-Controller wurden neben dem für Standardanfragen erforderlichen Endpunkt *EP0* drei weitere Endpunkte umgesetzt, der Empfangspuffer *EPOut*, der Sendepuffer *EPIn* und ein separater Endpunkt für zu übermittelnde Statusinformationen. Die Bezeichner der Daten-Endpunkte, EPIn und EPOut, beziehen sich auf die Transferrichtung aus Sicht des USB-Hosts. Die Daten-Endpunkte wurden als *Dual-Clock Dual-Port FIFOs* realisiert (siehe Abschnitt 4.2).

In das EPIn-FIFO werden die vom USIF zu versendenden Daten geschrieben. Die notwendigen Transferinformationen, PID und CRC16, werden beim Versenden durch den USB-Controller ermittelt und dem USB-Paket hinzugefügt.

Statuswert	Beschreibung
`EPIn_usedw[]`	Der Füllstand des Sende-FIFOs (bzgl. des Leseports), d.h. die Anzahl an lesbaren Adressen. (8 Bit)
`EPOut_usedw[]`	Der Füllstand des Empfangs-FIFOs (bzgl. des Schreibports), d.h. die Anzahl an bereits belegten Adressen. (8 Bit)
`EPOut_locked`	Signalisiert ein gesperrtes Empfangs-FIFO des Endpunktes EPOut. EPOut wird bereits gesperrt, sobald nicht genügend freier Speicherplatz für ein maximales USB-Paket (64 Byte) vorhanden ist.
`USIFStatus[]`	Die Statusinformationen des USIF-Controllers (12 Bit, siehe Abschnitt 4.4.4). Hierdurch wird der aktuelle USIF-Status an die Applikation auf Anwenderseite weitergeleitet.
`SyncApp`	Synchronisationsbit bzgl. des Lese-/Schreibzugriffs. Kann durch die Applikation genutzt werden, um den abgeschlossenen (erfolgreichen) Datentransfer zwischen Host und Gerät zu registrieren. Dieses Bit wird mit jeder versendeten Statusnachricht auf initialen Wert 0 gesetzt. Sobald ein Paket korrekt empfangen wird oder die Bestätigung (ACK) eines korrekt versendeten Pakets eintrifft, schaltet das Synchronisationsbit auf 1 um, wird mit der nächsten Statusnachricht versendet und zurückgesetzt.

Tabelle 4.1: USB - Status-Endpunkt

Im EPOut-FIFO werden die über den Bus korrekt empfangenen Daten, die im *Data*-Feld des USB-Pakets enthalten sind, abgespeichert. Um ein Überlauf zu vermeiden, signalisiert der Endpunkt EPOut ein gesperrtes FIFO (`EPOut_locked`), sobald der freie Speicherplatz für ein maximales USB-Paket (64 Byte) unterschritten wird. Somit kann, nachdem über den Status-Endpunkt ein (fast) voller Empfangspuffer übermittelt wurde, noch eine in der EPOut-Pipe befindliche Nachricht ohne Datenverlust empfangen werden.

4.1.1.2 Status-Endpunkt

Um Statusinformationen des USB-Controllers und des USIF-Controllers auf Anwenderseite kontinuierlich zu aktualisieren, wurde ein zusätzlicher Interrupt-Endpunkt integriert. Da es sich bei USB um einen Master-Slave-Bus handelt, kann das Gerät nicht selbständig verfügbare Daten melden, sondern kann nur auf entsprechende Anfrage reagieren. Hierzu liest der USB-Host mittels Polling im Intervall von 1 ms den Status-Endpunkt aus. Dieser Endpunkt besteht aus dem in Tabelle 4.1 beschriebenen Statusregister.

Durch die Statuswerte `EPIn_usedw` und `EPOut_usedw` ist auf Host-Seite der in den Datenspeichern belegte beziehungsweise freie Speicherplatz bekannt. Somit muss keine unnötige Leseanfrage an ein leeres EPIn-FIFO beziehungsweise unnötig viele Anfragen an EPIn, mit nur jeweils geringem Füllstand, gestellt werden.

4.1.2 FlexRay-Controller

Der FlexRay-Controller setzt die aktuelle Protokollspezifikation Version 2.1, Revision A um. Es ist jedoch nur ein Kanal implementiert, da dieser für die prototypische Umsetzung hinreichend ist und so auch der Hardware aufwand reduziert werden kann. Grundsätzlich ist der Controller auch durch äquivalente Implementierung der Empfangs-/Sendemodule auf zwei Kanäle erweiterbar.

Die Konfiguration des FlexRay-Controllers und somit die Auslegung des FlexRay-Netzes erfolgte nach dem Parametersatz in Anhang A.5. Die für die Umsetzung der FlexRay-Schnittstelle genutzten Ressourcen sind:

- Altera Cyclone II *EP2C70F896C6N*
- Quarz-Oszillator 80 MHz
- FlexRay Transceiver *NXP Semiconductors TJA1080A*

4.1.2.1 Datenpuffer

Für die Datenübermittlung wurden ein Empfangs- und ein Sendepuffer im FlexRay-Controller integriert. Diese wurden als *Dual-Clock Dual-Port FIFOs* umgesetzt (siehe Abschnitt 4.2). Im Sendepuffer werden die zu übermittelnden Nachrichten zwischengespeichert. Den Nachrichten sind zudem die nötigen Header-Informationen (*Indicator Bits*, *Frame ID*, *Cycle Code*) zugeordnet. Die Header- und Frame-CRC werden durch den Controller berechnet und ebenso wie die Kodierungssequenzen (TSS, FSS, BSS, FES, DTS) beim Versenden eingefügt.

Der Empfangspuffer enthält die korrekt empfangenen Nachrichten der ihm zugeordneten Slots. Hierzu werden die Daten des *Payload*-Segments (ohne Kodierungssequenzen) sowie die für eine Zuordnung nötigen Werte *Frame ID* und *Cycle Code* abgespeichert. Damit nicht sämtliche Slots aufgenommen werden, ist eine entsprechende Auswahl zugeordneter Slots vor Kommunikationsbeginn zu spezifizieren. Der Frame-Header innerhalb eines dieser Slots wird stets empfangen und dekodiert. Um einen korrekt übertragenen Header zu identifizieren, werden die Felder des Headers auf ihre Gültigkeit geprüft und der *Cyclic Redundancy Check* durchgeführt. Ist der Header fehlerfrei, wird mit dem Empfang des *Payload*- und *Trailer*-Segments fortgefahren. Ansonsten kann jede weitere Busaktivität im aktuellen Slot ignoriert werden. Nur wenn nach Empfang des kompletten Frames auch die Frame-CRC korrekt ist, werden die Nutzdaten im Empfangspuffer abgelegt. Wird hier eine fehlerhafte Übertragung festgestellt, muss der Frame verworfen und ein entsprechendes Error-Flag gesetzt werden.

Der Empfangspuffer signalisiert, dass dieser für den Empfang weiterer Daten gesperrt ist (`RxB_locked`), sobald kein freier Speicherplatz für die Nutzdaten eines maximalen Frames vorhanden ist (254 Byte). So kann noch mindestens ein Frame eines Knotens, der den *Lock*-Status noch nicht registrieren konnte, empfangen werden. Da im Testaufbau nur ein weiterer Knoten vorgesehen ist, genügt hier der im Lock-Status reservierte freie Speicherplatz.

Statuswert	Beschreibung
`USIFStatus[]`	Die Statusinformationen des USIF-Controllers (12 Bit, siehe Abschnitt 4.4.4). Hierdurch wird der aktuelle USIF-Status über ein Gateway an die Applikation auf Anwenderseite weitergeleitet.
`FwdReset`	Reset-Signal, das durch die Applikation auf Anwenderseite initiiert wird. Ein Reset wird hiermit über ein Gateway an den angesprochenen FlexRay-Knoten weitergeleitet.
`RxB_locked`	Zeigt an, dass der Empfangspuffer gesperrt ist. Verhindert somit, dass weitere Testdaten an diesen Knoten verschickt werden.

Tabelle 4.2: FlexRay - Statuswort

4.1.2.2 Statuswort

Die Statusinformationen des USIF-Controllers müssen über den FlexRay-Bus an die Applikation auf der Anwenderseite übermittelt werden. Im Testaufbau wurde hierfür ein Gateway implementiert, das die Daten vom FlexRay-Bus über die USB-Verbindung an die Applikation weiterleitet. Hierfür wird ein Statuswort (2 Byte) innerhalb des *Payload*-Segments eines Frames übertragen. Sind Nutzdaten vorhanden, wird das Statuswort diesen vorangestellt. Ansonsten wird das Statuswort dem leeren *Payload*-Segment hinzugefügt und zudem der *Null Frame Indikator* im Header zurückgesetzt. Die Gegenseite, also das Gateway, kann so das erste Datenwort des entsprechenden Slots stets als Statusinformationen interpretieren. Da dieses Statuswort einem bestimmten Slot, gegebenenfalls auch mehreren, zugeordnet ist, findet eine regelmäßige Übermittlung des aktuellen Status statt.

Das Statuswort setzt sich aus den in Tabelle 4.2 aufgelisteten Werten zusammen. Neben dem USIF-Status wird der Gegenseite mitgeteilt, ob der Empfangspuffer des FlexRay-Controllers für weitere Daten gesperrt ist (`RxB_locked`). Mit dem Reset-Signal `FwdReset` kann die USIF-Architektur durch die Applikation zurückgesetzt werden.

4.2 Clock Domain Crossing

Das USIF-Design bildet mit den verschiedenen unterschiedlich getakteten Komponenten wie USIF-Controller, Test-Controller und Kommunikations-Controller ein Multiple-Clock-System. Es kann als sogenanntes GALS-System (*Globally asynchronous locally synchronous*) betrachtet werden, welches das Problem der Kommunikation und des Datenaustausches zwischen asynchronen Einheiten in sich birgt. Bei solch einem Design mit unabhängigen Subsystemen mit eigenen Clock-Domains ist es unabdingbar, diese bei Interaktion zu synchronisieren. *Clock Domain Crossing* (CDC) bedeutet, dass ein

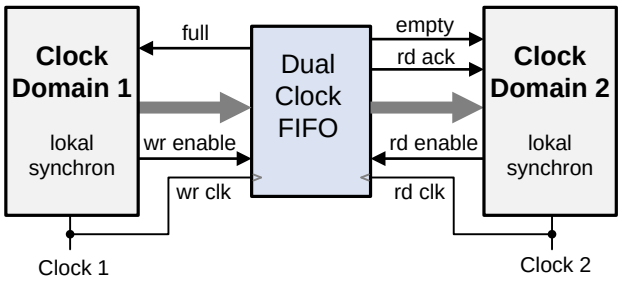


Abbildung 4.1: Clock Domain Crossing mittels FIFO-Speicher

Signal zwischen zwei Einheiten unterschiedlicher Taktung ausgetauscht wird. Am Eingang der Zieleinheit erscheint das Signal asynchron, falls es nicht lange genug oder zu lange anliegt. Das heißt, das auszutauschende Signal kann entweder nicht registriert oder, im Falle eines Events, mehr als einmal verarbeitet werden.

Der Datenaustausch kann über Zwischenspeicher synchronisiert werden. Hierzu sind die Datenspeicher der Controller als *Dual-Clock Dual-Port FIFOs* ausgelegt (Abbildung 4.1). Da Schreib- und Leseport durch den jeweils angeschlossenen Controller getaktet sind, ist das Timing des Schreib- und Lesezugriffs unabhängig voneinander. Somit ist keine weitere Administration vonnöten.

Die FIFOs sind zudem mit unterschiedlich großen Lese- und Schreibports ausgestattet. Dadurch kann die Datenübergabe zwischen Komponenten unterschiedlicher Datenbreite ohne zusätzlichen Aufwand realisiert werden. So werden beispielsweise die Testeingaben von der Schnittstelle als 32-Bit-Wörter in das TestIn-FIFO abgelegt. Der Test-Controller liest pro Scan-Takt aber nur ein Datenwort entsprechend der Datenbreite seines Eingangsports aus.

Der Kontrollpfad muss auch synchronisiert werden, um den korrekten Empfang von Kontrollsignalen oder den Datenaustausch zwischen Registern der unterschiedlichen Clock-Domains sicherstellen. Das kann durch *Handshake*-Verfahren geschehen. Das 2-Phasen-Handshake-Protokoll ist schneller, beruht aber auf einer komplexeren Schaltung. Das 4-Phasen-Handshake-Protokoll hat zwei redundante Phasen, wurde aber bevorzugt, da es durch eine simplere Schaltung umgesetzt werden kann und stets einen definierten Ausgangszustand besitzt.

Die Kommunikation findet zwischen *Talker* und *Listener* im sogenannten *Push-Pull-System* statt (Abbildung 4.2). Im 4-Phasen-Handshake-Protokoll dienen die ersten beiden Phasen der Synchronisation und die dritte und vierte Phase des Zurücksetzens der Handshake-Signale auf Talker- und Listener-Seite (Abbildung 4.3). Der Talker signalisiert mit einer Anfrage ($req = 1$), dass Daten übergeben werden sollen (*Push*) oder entgegengenommen werden können (*Pull*). Sobald der Listener die Anfrage quittiert ($ack = 1$), werden die Daten transferiert. Nach abgeschlossener Datenübertragung können Anfrage- und daraufhin auch Bestätigungssignal wieder in den Ausgangszu-

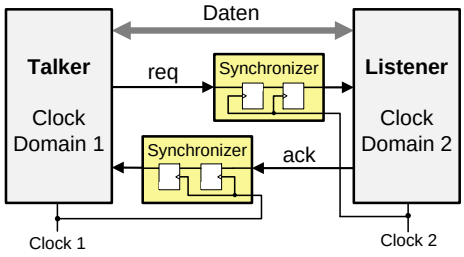


Abbildung 4.2: Push-Pull-System mit Synchronizer

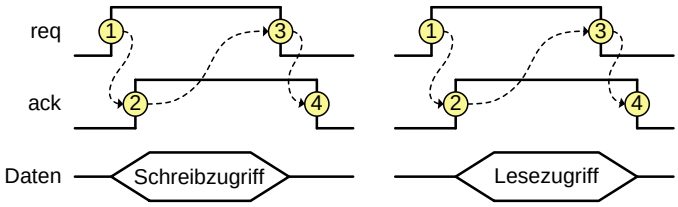


Abbildung 4.3: 4-Phasen-Handshake

stand gesetzt werden ($req = 0 \rightarrow ack = 0$). Da das Handshake-Signal *req* Lese- oder Schreibanfrage bedeuten kann, ist von einem bidirektionalen Datentransfer die Rede.

Mittels Synchronizer werden metastabile Zustände vermieden, indem die Handshake-Signale aus der anderen Clock-Domain über Flipflop-Stufen der eigenen Clock angepasst werden. Um eine Aussage über die nötige Anzahl an Flipflops des Synchronizers zu treffen, kann der Erwartungswert MTTF (*Mean Time To Failure*), der unter anderem von der Ausbreitungsverzögerung in der Kombinatorik, der Flipflop-Ladezeiten und der Clock abhängig ist, herangezogen werden [Cum08]. Für ein Synchronizer, bestehend aus einem Flipflop, bewegt sich der MTTF-Wert im Sekundenbereich für große kombinatorische Schaltungen und steigt bis zu mehreren hundert Jahren für kleine Schaltungen. Mit einem zweistufigen Synchronizer steigt der MTTF auf mehrere tausend Jahre und mit einer dritten Flipflop-Stufe sogar auf 10^4 bis 10^{18} Jahre. Der am häufigsten von Schaltungsdesignern verwendete Synchronizer ist der mit zwei Flipflop-Stufen [Cum08]. Für die Synchronisation der Steuersignale innerhalb der USIF-Architektur ist solch ein zweistufiger Synchronizer, wie in Abbildung 4.2 zu sehen, völlig hinreichend.

Da in der prototypischen Umsetzung die Kommunikations-Controller in der USIF-Architektur integriert sind, werden diese Synchronisationskonzepte auch für den Signal- und Datenaustausch mit dem jeweiligen Controller angewandt.

4.3 Speicherung der Testdaten

Für die Zwischenspeicherung der Testdaten wurden folgende generische Module implementiert:

- *TestIn-FIFO*, zur Speicherung der empfangenen Testdaten,
- *TestRef-FIFO*, zur Speicherung der empfangenen Referenzdaten,
- *TestOut-FIFO*, zur Speicherung der Testantworten, und
- *TS-FIFO*, zur Speicherung der zugehörigen Zeitstempel.

Die Testdatenspeicher wurden, wie in Abschnitt 4.2 beschrieben, als *Dual-Clock Dual-Port FIFOs* umgesetzt. Die Größe dieser Speicherblöcke kann unabhängig von der Datenrate des jeweiligen Kommunikationskanals festgelegt werden, da durch Synchronisation über Statusinformationen (siehe Abschnitt 4.4.4) ein Datenverlust vermieden wird. Als untere Grenze ist aber der Datenumfang der Nutzdaten einer über den jeweiligen Bus zu übertragenen maximalen Nachricht (USB-Paket, FlexRay-Frame) vorzusehen. So kann dem Empfangspuffer des Kommunikations-Controllers jeweils mindestens eine komplette Nachricht entnommen werden. Das heißt, es wird direkt der minimal benötigte Speicherplatz im Empfangspuffer freigegeben und somit nicht der Empfang eines möglicherweise folgenden Nachrichtenpakets blockiert.

Um sicherzustellen, dass kein Datenstau entsteht, erfolgt die Übermittlung der Testdaten paketweise. Die Größe dieser USIF-Pakete wird dabei dem für die Testdaten zur Verfügung stehenden Speicherplatz angepasst. Ein Testdatenpaket wird nur in dem Umfang versendet, in dem es auch durch das TestIn- beziehungsweise TestRef-FIFO aufgenommen werden kann. Enthält das TestOut-FIFO ein Paket spezifizierter Größe, wird es ausgelesen, um Speicherplatz für weitere Testantworten freizugeben. Je Größer die FIFOs und die Pakete ausgelegt werden, umso seltener müssen Schreib- und Leseanfragen erfolgen. Im Falle einer USB-Verbindung vergeht zwischen jeder dieser Anfragen die für das Polling-Intervall spezifizierte Zeit (mindestens 1 ms). In der FlexRay-Kommunikation sollten Slots durch die spezifizierten USIF-Pakete möglichst komplett genutzt werden. Sind bei einer zu gering ausgelegten Paketgröße Slots nur teilweise belegt, verstreicht potentielle Sendezeit ungenutzt. Um weitere Daten zu versenden, muss auf den nächsten verfügbaren Slot gewartet werden. Das bedeutet, eine geringere Auslegung der Testdatenspeicher hat eine erhöhte Testdauer zur Folge.

4.3.1 Speicherung der Testeingaben

Die über die Kommunikationsschnittstelle empfangenen Testeingabedaten werden im TestIn-FIFO zwischengespeichert. Das FIFO befindet sich in der USIF-Architektur zwischen dem Empfangspuffer des jeweiligen als Testschnittstelle genutzten Kommunikations-Controllers und der integrierten Testarchitektur (SCT oder EDT). Wie in Abbildung 4.4 dargestellt, wird es mit den aus dem Empfangspuffer der Schnittstelle entnommenen Daten über einen Schreibport der Datenbreite d_{USIF} beschrieben. Diese

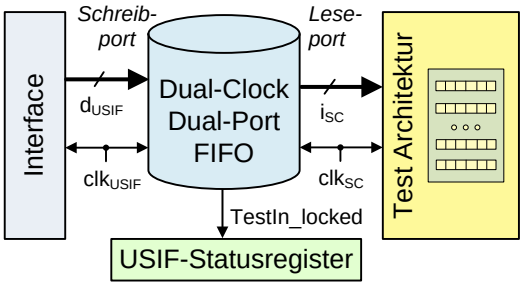


Abbildung 4.4: Speicher der Testeingabedaten

Datenbreite d_{USIF} wird durch die Auslegung der USIF-Architektur bestimmt und wurde für die Umsetzung auf 32 Bit festgelegt. Die Administration der Datenpuffer bei einem Zugriff über die Testschnittstelle wird durch den USIF-Controller übernommen, daher auch durch diesen mit dem Takt der USIF-Clock clk_{USIF} betrieben. Der Lesezugriff erfolgt durch den Test-Controller. Hierbei wird mit Scan-Takt clk_{SC} jeweils ein Datenwort entsprechend der Datenbreite des Eingangsportes i_{SC} der Testarchitektur ausgelesen. In der Abbildung ist zudem das FIFO-Signal `TestIn_locked` als Teil des USIF-Statusregisters zu sehen. Hierdurch soll rechtzeitig signalisiert werden, dass das FIFO für den Schreibzugriff über die Testschnittstelle gesperrt ist, um so das System vor einen Deadlock-Zustand zu bewahren.

Es sei folgendes Beispiel betrachtet. Über die Schnittstelle werden Testeingabedaten für das TestIn-FIFO versendet. Nun läuft das TestIn-FIFO voll und übermittelt über die Statusinformationen ein entsprechendes *Full*-Signal. Die Gegenseite stellt daraufhin das Senden weiterer Daten ein. Aber im Empfangspuffer des Kommunikations-Controllers befinden sich noch zuvor übermittelte Testeingabedaten, die durch das TestIn-FIFO nicht mehr entgegengenommen werden können. Dem TestIn-FIFO wiederum können keine weiteren Daten entnommen werden, da der Test-Controller aufgrund eines gesperrten TestOut-FIFOs sich im Wartezustand befindet. Um das TestOut-FIFO auszulesen und wieder Speicherplatz freizugeben, wird eine Leseanfrage durch das entsprechende USIF-Instruktionswort versendet, welches sich aber im Empfangspuffer hinter den noch nicht ausgelesenen Testeingabedaten für das TestIn-FIFO befindet. Hier werden Steuerinformationen durch Daten blockiert. Somit liegt ein Deadlock vor, der im Nachhinein nicht mehr ohne Datenverlust aufzulösen ist.

Um dies zu vermeiden, signalisiert das TestIn-FIFO bereits den Lock-Status, sobald es halb voll ist (`TestIn_locked`). Somit muss das Versenden der nächsten Datenpakete unterbrochen werden. Die bereits im Empfangspuffer des Kommunikations-Controllers befindlichen Daten können noch in die freie reservierte Speicherhälfte des TestIn-FIFOs geschrieben werden. Die Datenanfrage, um das TestOut-FIFO auszulesen, wird so nicht durch verbliebene Testdaten im Empfangspuffer blockiert.

Die Applikation auf Anwenderseite prüft die regelmäßig aktualisierten Statusinformationen (siehe Abschnitt 4.4.4) und versendet Testeingaben nur, wenn freier Speicher-

platz in entsprechendem Umfang garantiert ist. Die Applikation hat dabei sicherzustellen, dass zwischen zwei aufeinanderfolgenden Testeingabepaketen die Statusinformationen aktualisiert werden. Bei der USB-Verbindung muss also entsprechend des Polling-Intervalls des Status-Endpunktes gewartet werden. Liegt kein Lock-Status vor, kann in diesem Zeitfenster garantiert werden, dass die Eingabedaten vom TestIn-FIFO aufgenommen werden.

4.3.2 Speicherung der Referenzdaten

Im Falle der Analysemodi *LogFaults* oder *PassFail* speichert das TestRef-FIFO die über den Testzugang empfangenen Referenzdaten. Dabei handelt es sich um die aus der Testarchitektur erwarteten Ausgaben, die im ATPG-Prozess den ermittelten Testeingaben zugeordnet werden. Der Schreibzugriff auf das TestRef-FIFO entspricht dem des zuvor beschriebenen TestIn-FIFOs.

Gelesen wird der Referenzspeicher durch die Analysekomponente, wie in Abbildung 4.5 aufgezeigt. Da diese die Schnittstelle zwischen dem TestRef-FIFO und der Testarchitektur darstellt, wird ein Referenzwert entsprechend der Datenbreite des Testausgangsports o_{SC} mit Scan-Takt clk_{SC} entnommen.

Der Speicherumfang des TestRef-FIFOs kann unabhängig von den anderen Testspeichern spezifiziert werden. Bei nur wenigen relevanten Signaturen wird das TestRef-FIFO in entsprechend geringen Umfang ausgelegt. Wird zum Beispiel ausschließlich eine Zeitkompaktierung der Testantworten umgesetzt, so muss auch nur ein kleiner Satz an Signaturen oder gar nur eine finale Signatur, die am Ende des Testlaufs auszuwerten ist, zwischengespeichert werden. Ist hingegen zu jeder Testeingabe eine Testantwort zu erwarten, sollte die Größe des TestRef-FIFOs der des TestIn-FIFOs entsprechen. So muss der Test-Controller nicht wiederholt aufgrund fehlender Referenzdaten im Wartezustand verharren.

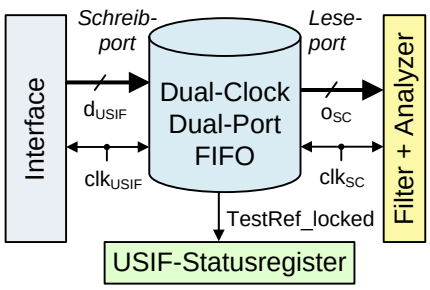


Abbildung 4.5: Speicher der Referenzdaten

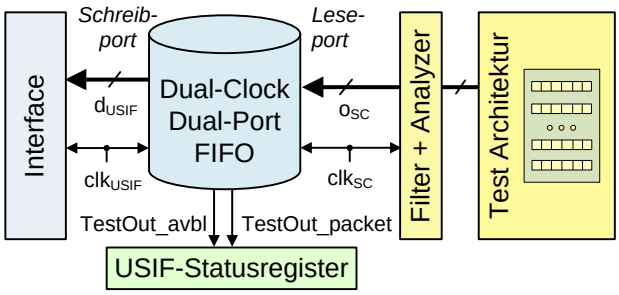


Abbildung 4.6: Speicher der Testausgabedaten

4.3.3 Speicherung der Testausgaben

Die für eine externe Auswertung auszulesenden Testausgaben werden im TestOut-FIFO zwischengespeichert. Daher wird dieses, wie in Abbildung 4.6 zu sehen, mit Scan-Takt clk_{SC} über einen Schreibport der Datenbreite o_{SC} beschrieben. Im Falle des Analysemodus *LogAll* werden die Testausgaben aufgenommen, die den entsprechend konfigurierten Analysefilter passieren. Ist hingegen der *LogFaults*-Modus eingestellt, wird mittels der Referenzwerte zusätzlich auf fehlerhafte Testausgaben gefiltert. Im *PassFail*-Modus wird der Test unmittelbar nach einem erkannten Fehler abgebrochen, aber dennoch die verfälschte Signatur im Speicher abgelegt, um gegebenenfalls bereits als Ergebnis für eine Auswertung oder als Ansatz für eine anschließende feingranulare Diagnose zu dienen.

Um im Falle vieler Ausgabedaten nicht direkt mit jeweils den ersten verfügbaren Daten das Auslesen zu initiieren, wird durch das Signal `TestOut_packet` ein erreichter Mindestfüllstand gemeldet. Dieser Schwellwert wird in der Konfigurationsphase des Tests durch den Parameter `TestOutPacket` (siehe Anhang A.3) spezifiziert. Dadurch werden erst ab einem bestimmten Kontingent Testausgaben übermittelt und so ein paketweiser Austausch garantiert. Kommen nach einer bestimmten Zeit oder am Ende eines Testlaufs keine Testdaten im Umfang des spezifizierten Paketes zusammen, wird das Signal `TestOut_avbl` benötigt, um die wenigen verfügbaren Daten dennoch zu registrieren.

Speicherung der Zeitstempel

Im Falle des Analysemodus *LogFaults* sind im TestOut-FIFO nur verfälschte Testantworten enthalten. Um im Nachhinein eine Auswertung zu ermöglichen, ist zusätzlich zu jeder Testausgabe die Information nötig, durch welche Testeingaben diese erzeugt wurde. Daher erfolgt durch die Zeitstempel eine zeitliche Zuordnung im Testlauf. Der Zeitstempel, der sich aus der Pattern-Nummer und der Shift-Nummer zusammensetzt, wird synchron zum Schreibzugriff auf das TestOut-FIFO in einem separaten TS-FIFO abgespeichert. Im *PassFail*-Modus wird zu der gespeicherten Fehlersignatur ebenfalls der Zeitstempel aufgenommen, um über den Zeitpunkt des Fehlerauftretens die Signa-

tur einer Testeingabe zuordnen zu können. Für den *LogAll*-Modus werden keine Zeitstempel benötigt, da sämtliche Ausgaben übermittelt werden und so die Zuordnung auf der Anwenderseite trivial ist. Auch bei einer Vorauswahl durch den Analysefilter kann durch die auf Anwenderseite bekannten Werte für die Analyserate und die Analysebasis die Zuordnung durch äquivalenten Filterprozess wieder hergestellt werden.

Die Datenbreite des Zeitstempels und somit des FIFOs sollte für die potentiell maximale Anzahl an abzuspeichernden Zeitstempeln ausgelegt sein. Ansonsten kann es zum Werteüberlauf kommen. Ein Überlauf des Zählerwertes kann nur erkannt werden, wenn eine Vielzahl an gleichmäßig verteilten Zeitstempeln enthalten ist. Genauer gesagt, es müsste in einem Zeitfenster zwischen zwei Zählerüberlaufen mindestens ein Zeitstempel aufgenommen werden. Die Applikation auf der Anwenderseite kann im Falle eines auf ein Zeitstempel a nachfolgenden kleineren Zeitstempels b diesen wieder hochrechnen. Das heißt, wird festgestellt, dass $a > b$ ist, lautet der korrekte Zeitstempel $b' = 2^{\lceil \mathrm{ld}(a) \rceil} + b$. Um für diese Methode einen regelmäßig aufgenommenen Zeitstempel sicherzustellen, könnte beispielsweise die Capture-Phase des Scan-Test mitgezählt werden. In regelmäßigen Abständen wird einer dieser Capture-Zeitstempel mit abgespeichert, um auf Anwenderseite den Zeitverlauf zu registrieren. Anschließend wird dieser zusätzliche Zeitstempel wieder aus den Testausgabedaten beseitigt. In der prototypischen Umsetzung wird auf solch zusätzliche Zeitstempel verzichtet, da durch entsprechend große Auslegung des Zählers sichergestellt werden kann, dass während eines Tests ein Überlauf vermieden wird.

4.4 Steuerungsschnittstelle

4.4.1 USIF-Datenformat

Für die USIF-Architektur wurde ein konsistentes Datenformat für Instruktionswörter zur Konfiguration von Controller-Einheiten oder Übertragung von Testdatenpakete definiert. Ein Instruktionswort besteht aus einem Befehlsfeld, einem (unterteilten) Adressfeld und einem optionalen Datenfeld. Die Größe eines Instruktionswortes entspricht der Datenbreite der USIF-Architektur. Die Datenpuffer der Kommunikations-Controller sind entsprechend der Gesamtarchitektur ausgelegt, so dass in einem Takt ein komplettes Instruktionswort aus dem Empfangspuffer ausgelesen und verarbeitet werden kann.

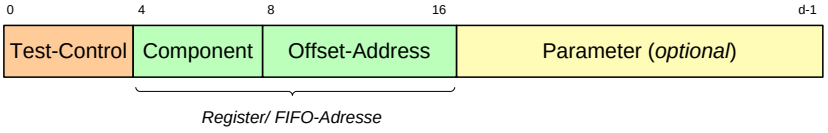


Abbildung 4.7: USIF-Instruktionswort

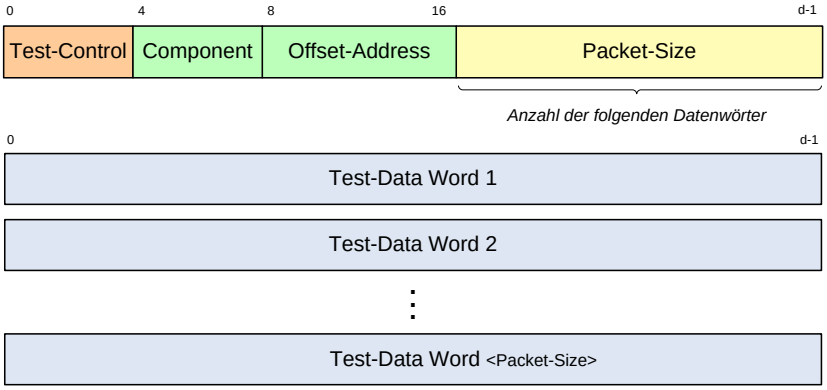


Abbildung 4.8: USIF-Datenformat zur Datenübertragung

Die Instruktionswörter des USIF-Controllers sind, wie in Abbildung 4.7 zu sehen, codiert. Das Feld *Test-Control* (TC) enthält den Befehlscode (4 Bit) für die auszuführende Aktion (siehe Tabelle 4.3). Hiermit geschieht die Steuerung des USIF-Zustandsautomaten. Das *Component-Address*-Feld (4 Bit) gibt die Controller-Einheit an, in der ein Schreib- beziehungsweise Lesezugriff erfolgen soll. Das *Offset-Address*-Feld (8 Bit) enthält die interne Adresse der spezifizierten Komponente, um auf ein Konfigurationsregister oder einen Speicherblock zuzugreifen.

Falls ein Konfigurationszugriff (`ConfigRead`, `ConfigWrite`) erfolgt, enthält das *Parameter*-Feld den Wert für das durch *Component* und *Offset* adressierte Register. In der 32-Bit-Architektur umfasst dieses Feld 16 Bit. Im Falle mehrerer Register geringer Datenbreite kann das *Parameter*-Feld auch eine Kombination mehrerer in einer Adresse "gebündelter" Konfigurationswerte enthalten. Übersteigt hingegen die Datenbreite eines Konfigurationsregisters die des *Parameter*-Feldes, muss es auf mehrere Adressen aufgeteilt werden. Dementsprechend würden auch mehrere Schreib-/ Lesezugriffe zur Konfiguration benötigt.

Im Falle von zu übertragenden Testdaten ist *Test-Control* für den Schreibzugriff gesetzt (`Write`) und ein Testspeicher, TestIn-FIFO oder TestRef-FIFO, adressiert. Im *Parameter*-Feld ist hierbei der Wert *Packet-Size* enthalten, der die Anzahl der darauffolgenden Datenwörter bestimmt (Abbildung 4.8). Somit können die Daten mit voller Bandbreite, ohne den Overhead weiterer Steuerdaten, übertragen werden, bis der angegebene Wert mittels eines Zählers erreicht ist. Das Instruktionswort kann hier als Header eines Datenblocks betrachtet werden. Jeder zu übertragene Datenblock muss durch solch einen Header angeführt werden.

Die Anfrage von Testausgaben erfolgt über den *Test-Control*-Code für den Lesezugriff (`Read`). Die Übertragung verläuft äquivalent zur Schreibanweisung. Durch *Packet-Size* ist die Anzahl der zu lesenden Datenwörter angegeben. Soll der gesamte Speicher der Testausgaben ausgelesen werden, gibt es die Möglichkeit, diesen Wert auf null zu setzten. Somit ist die Obergrenze für die Anzahl der zu lesenden Datenwörter unterdrückt

Datenfeld	**Beschreibung**		
Test-Control (TC)	Instruktion zur Steuerung des *Universal Scan-Test Interface*		
	Befehls-code	**Mnemonik**	**Beschreibung**
	0xF	StartCom	Start der Kommunikation mit dem USIF-Controller
	0x7	StopCom	Beendigung der Kommunikation
	0x1	ConfigWrite	Schreibzugriff auf Konfigurationsregister
	0x2	ConfigRead	Lesezugriff auf Konfigurationsregister
	0x3	*reserved*	
	0x4	Write	Schreibzugriff auf die Datenspeicher
	0x5	Read	Lesezugriff auf die Datenspeicher
	0x6	*reserved*	
	0x8	StartSC	Start des Scan-Tests
	0x9	StopSC	Beendigung/Abbruch des Scan-Tests
	0xA	StartLoopback	Loop-Back-Modus für funktionalen Test der Kommunikationsschnittstelle
	0xB	StopLoopback	Beendigung des Loop-Back-Modus
	0xC	RunBIST	Initiierung des on-chip Startup-Tests
	0xD-0xE	*reserved*	
Component -Address	Komponente, in der die Konfiguration oder Schreib-/Lesezugriff auf ein Datenspeicher erfolgen soll		
	0x0	USIF-Controller	
	0x1	USB-Controller	
	0x2	FlexRay-Controller	
	0x3	CAN-Controller	
	0x4	*reserved*	
	0x5	JTAG-Buffer	
	0x6	*reserved*	
	0x7	optionaler Speicherblock	
	0x8-0xF	*reserved*	
Offset-Address	Controller-interner Adressraum/ Speicherblock (siehe individuelle *Memory Map* der spezifizierten Controller-Einheit im Anhang A.3)		
Parameter	Optionales Feld für den Wert des adressierten Konfigurationsregisters bzw. für die Kombination von Werten des adressierten Konfigurationsregistersatzes		
Packet-Size	Anzahl der zu übertragenden Datenwörter, die auf das Instruktionswort folgen		

Tabelle 4.3: USIF-Instruktionen

und der Lesezugriff ist erst beendet, wenn der adressierte FIFO-Speicher über keine weiteren Daten verfügt.

4.4.2 USIF-Controller

Mittels des definierten Instruktionssatzes wird der Testzugang und somit der Testablauf kontrolliert. Die Steuerung des Testzugangs kann durch das Zustandsdiagramm in Abbildung 4.9 nachvollzogen werden. Sobald einer der integrierten Kommunikations-Controller empfangene Daten meldet, wird der USIF-Controller über Multiplexer mit der jeweiligen Schnittstelle verbunden. Das bedeutet, in dieser Komponente ist nur eine allgemeine "virtuelle" Schnittstelle sichtbar, ohne Informationen des zugrundeliegenden Kommunikations-Controllers verwalten zu müssen.

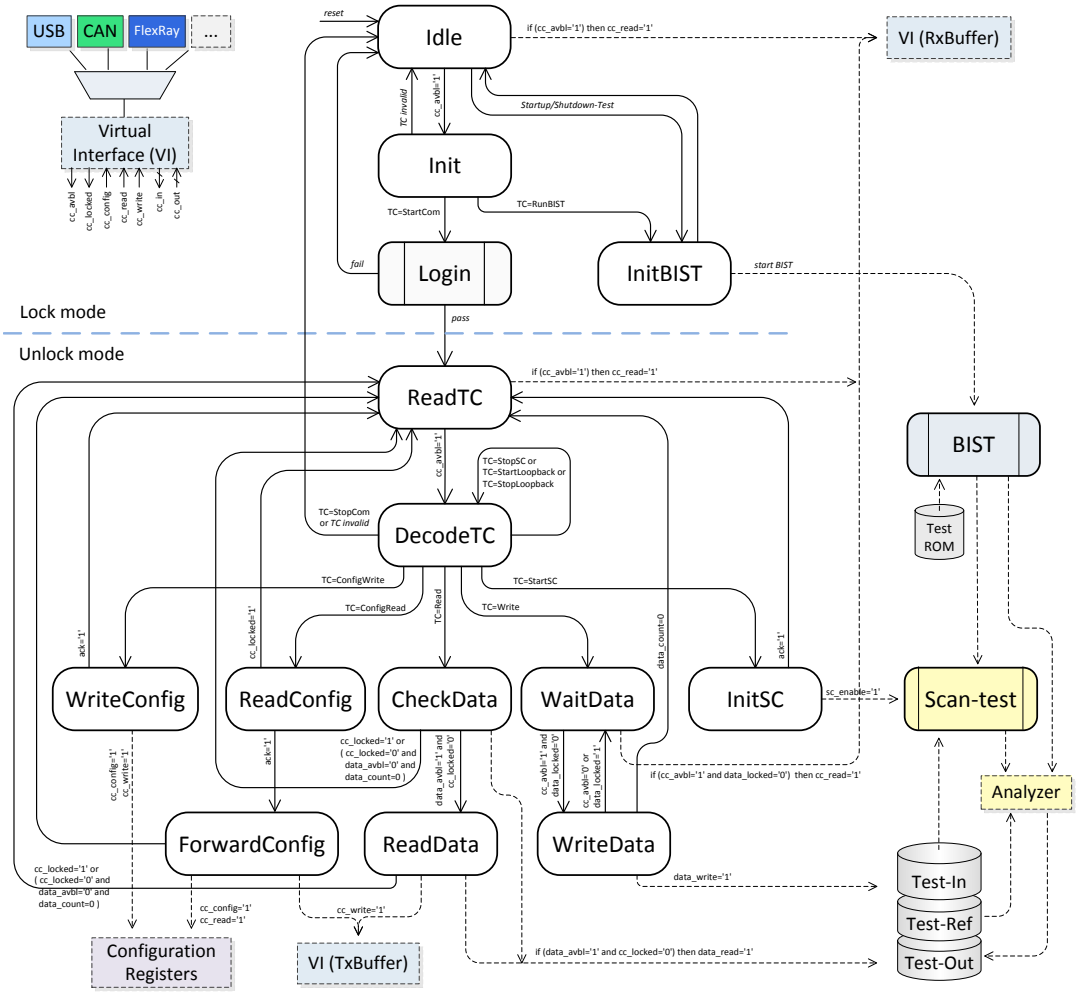


Abbildung 4.9: Erweitertes Zustandsdiagramm des USIF-Controllers

Signal[1]	**Beschreibung**
Kommunikations-Controller (CC): Die folgenden CC-Signale der "virtuellen" Schnittstelle werden auf die spezifischen Signale des eingestellten CCs gemappt.	
`cc_avbl <i>`	Signalisiert verfügbare Daten im Empfangspuffer *RxFIFO* des eingestellten CCs (entspricht dem invertierten *Empty*-Signal des *RxFIFO*).
`cc_locked <i>`	Signalisiert einen gesperrten Sendepuffer *TxFIFO* des eingestellten CCs. Der Sperrzustand wird signalisiert bevor das *TxFIFO* komplett voll ist (mindestens eine freie Speicheradresse). Somit kann auch noch ein bereits aus dem TestOut-FIFO gelesenes Datenwort geschrieben und so die Datenkonsistenz sichergestellt werden kann.
`cc_config <o>`	Signal zur Konfiguration des CCs. Wird in den Zuständen WriteConfig und ReadConfig zusätzlich zum *Write*- bzw. *Read*-Signal gesetzt.
`cc_read <o>`	Signal für den Lesezugriff auf den Empfangspuffer *RxFIFO* des eingestellten CCs, falls `cc_config='0'`. Ansonsten wird der Wert dem adressierten Konfigurationsregister gelesen.
`cc_write <o>`	Signal für den Schreibzugriff auf den Sendepuffer *TxFIFO* des eingestellten CCs, falls `cc_config='0'`. Ansonsten wird der Wert aus dem Parameter-Feld des USIF-Instruktionswortes ins adressierte Konfigurationsregister geschrieben.
`cc_in[] <i>`	Dateneingang vom Ausgangsport des eingestellten CCs. Enthält die Daten nach dem Lesezugriff auf den Empfangspuffer *RxFIFO* oder auf ein Konfigurationsregister.
`cc_out[] <o>`	Datenausgang zum Eingangsport des eingestellten CCs. Stellt die Daten für den Schreibzugriff auf den Sendepuffer *TxFIFO* oder auf ein Konfigurationsregister bereit.
Datenspeicher:	
Die folgenden Signale der "virtuellen" Speicherschnittstelle werden auf die spezifischen Signale des adressierten Datenspeichers gemappt.	
`data_avbl <i>`	Signalisiert verfügbare Daten im adressierten Datenspeicher (z.B. TestOut-FIFO).
`data_locked <i>`	Signalisiert, dass der adressierte Datenspeicher gesperrt ist.
`data_read <o>`	Signal für den Lesezugriff auf den adressierten Datenspeicher.
`data_write <o>`	Signal für den Schreibzugriff auf den adressierten Datenspeicher.
`data_in[] <i>`	Dateneingang vom adressierten Datenspeicher.
`data_out[] <o>`	Datenausgang zum adressierten Datenspeicher. Im USIF zu verteilende Daten, wie `data_out` oder `cc_out`, sind hier der Übersicht halber separat dargestellt, werden aber auf einen gemeinsamen Mainbus gebündelt.
Test-Controller:	
`sc_enable <o>`	Signal zum Starten des Test-Controllers.
`sc_stop <o>`	Signal zum Beenden des Scan-Tests.
`sc_status[] <i>`	Zustand des Test-Controllers (IDLE, ACTIVE, STOP, ERROR).

[1] Im Falle von Datenports zu anderen Komponenten, ist zusätzlich die Signalrichtung angegeben (Eingangsport <i>, Ausgangsport <o>)

Synchronizer:	
`ack <i>`	Bestätigungssignal der synchronisierten Kommunikation mit einer anderen Clock-Domain (CC oder Controller der Testarchitektur).
intern:	
`cc_source`	Register, das die für den Testzugang genutzte Schnittstelle speichert (siehe Komponenten in Tabelle 4.3) . Wird benötigt, um die Ports der konkreten Standardschnittstellen auf die internen "virtuellen" Ports (`cc_in`, `cc_out`, `cc_avbl`, `cc_read` usw.) umzuleiten.
`data_count`	Zählerwert zur Bestimmung eines komplettierten Datenpakets. Der Zähler wird vor einer Datenübertragung mit dem Wert aus dem Packet-Size-Feld des USIF-Instruktionswortes initialisiert und mit jedem weiteren empfangenen Datenwort dekrementiert. Ist der Wert 0, wurde somit das komplette Datenpaket empfangen.
`SCRun`	Zustandsregister für den Scan-Test-Modus. Aktiviert den Scan-Test sobald dieses auf '1' gesetzt wird. Wird Controller-intern zurückgesetzt, wenn der Scan-Test terminiert (z.B. bei erkanntem Fehler im Pass/Fail-Modus oder bei einem Error). Kann auch von außen durch den Befehl `TC=StopSC` zurückgesetzt werden, um den Scan-Test zu beenden bzw. abzubrechen.
`Loopback`	Zustandsregister für den Loopback-Modus. Ist der Modus aktiv, wird der Ausgang des TestIn-FIFOs direkt an den Eingang des TestOut-FIFOs angelegt. Somit lässt sich der Kommunikationspfad, von der Anwendung über den Bus und den CC bis zu den Testspeichern, funktional testen.

Tabelle 4.4: Signale des USIF-Controllers

In Abhängigkeit der Quelle der Eingangsdaten, die im *Source*-Register (`cc_source`) gespeichert wird, werden mittels Schalter die Ports der USIF-Komponente mit den Ports der jeweiligen Schnittstelle verbunden. Somit ist die Kommunikation unabhängig von einer spezifischen Schnittstelle gestaltet. Dieses Schaltnetz ist natürlich nur in der prototypischen Umsetzung notwendig, da hier optional eine aus mehreren integrierten Standardschnittstellen gewählt werden kann.

In Tabelle 4.4 sind die wichtigsten Signale des USIF-Controllers, die in der im Folgenden beschriebenen USIF-FSM benötigt werden, aufgelistet. Der Empfangspuffer des eingestellten Kommunikations-Controllers wird hier auch als *RxFIFO* und der Sendepuffer als *TxFIFO* bezeichnet.

Das erste Instruktionswort, gelesen im Zustand Init, muss im *Test-Control*-Feld den Befehl für den Start der Kommunikation (`StartCom`) oder für die Initiierung eines Startup-Tests (`StartSC`) enthalten. Im Falle eines Startup-Tests wird sofort die Prozedur des BISTs mit on-chip Steuerinformationen ausgeführt. Soll die Kommunikation gestartet werden, wird im Zustand Init zunächst der Wert im *Controller-Address*-Feld mit dem gespeicherten Wert im *Source*-Register (`cc_source`) abgeglichen, um einen korrekten Kommunikationsbeginn zu verifizieren. Außerdem werden im selben Zustand die Komponenten zur Zugangskontrolle initialisiert und die Testkomponenten wie Test-

Controller und Testspeicher zurückgesetzt. Daraufhin folgt die Zugangskontrolle, um die internen Strukturen vor unbefugtem Zugriff zu schützen. Ist der Zugang durch die Login-Prozedur zugelassen, können Schreib- und Lesezugriffe für die Steuerung eines umfangreichen Scan-Tests erfolgen.

Im Zustand ReadTC wird auf Daten im Empfangspuffer RxFIFO gewartet. Sobald durch das Signal `cc_avbl` verfügbare Daten gemeldet werden, erfolgt ein Lesezugriff mittels des Signals `cc_read`, um ein einzelnes Instruktionswort zu lesen. Der Zustand DecodeTC wertet die Instruktion aus dem *Test-Control*-Feld des empfangenen Instruktionswortes aus, um in den entsprechenden Folgezustand zu wechseln. Ist die Instruktion `TC=StopCom` oder liegt ein ungültiger Wert vor, wird der Testzugang beendet beziehungsweise abgebrochen. Bei den Instruktionen `StopSC`, `StartLoopback` und `StopLoopback` findet kein Zustandswechsel statt, es wird lediglich das entsprechende Flag-Register aktualisiert. Mit `TC=StopSC` wird das *SCRun*-Flag, das ein aktiven Scan-Test indiziert, zurückgesetzt und über das Signal `sc_stop` der Test-Controller terminiert. Mittels `TC=StartLoopback` und `TC=StopLoopback` wird der Loopback-Modus aktiviert beziehungsweise deaktiviert (*Loopback*-Flag).

Die Konfiguration eines adressierten Registers geschieht im Zustand WriteConfig. Hierzu wird das Schreibsignal zur Konfiguration auf logisch 1 gesetzt und die Handshake-Prozedur für den synchronisierten Datenaustausch gestartet. Der Zustand wird erst wieder verlassen und somit das Schreibsignal zurückgesetzt, sobald der Synchronizer den erfolgreichen Datenaustausch meldet (`ack='1'`).

Im Zustand ReadConfig wird geprüft, ob der Sendepuffer TxFIFO beschrieben werden kann. Ist dieser voll, wird dies über das Signal `cc_locked` registriert und unverzüglich zurück in den Zustand ReadTC gegangen. Ansonsten wird das adressierte Konfigurationsregister gelesen. Nach bestätigtem Zugriff (`ack='1'`) wird der Zustand ForwardConfig betreten, um den Wert an den Sendepuffer TxFIFO weiterzuleiten (`cc_write='1'`).

Durch den Zustand InitSC wird der Scan-Test gestartet. Hierzu werden sämtliche Testkomponenten initialisiert und der Test-Controller durch das Signal `sc_enable` aktiviert. Es wird vom Zustand InitSC wieder in den Zustand ReadTC übergegangen, sobald durch den Synchronizer die erfolgreiche Signalübermittlung und somit der Start des Test-Controllers bestätigt wird (`ack='1'`). Solange keine Daten in das TestIn-FIFO geschrieben wurden, befindet sich der Test-Controller im Wartezustand. Auf diese nötigen Testdatenpakte wird nun im Zustand ReadTC gewartet.

Wird durch das Instruktionswort ein Schreibzugriff auf einen Datenspeicher angekündigt, wird zunächst der Zähler `data_count` mit dem Wert aus dem *Packet-Size*-Feld initialisiert und in den Zustand WaitData gewechselt. Hier wird geprüft, ob sich die zu schreibenden Daten bereits im RxFIFO befinden und der adressierte Testspeicher nicht gesperrt ist. Sobald RxFIFO verfügbare Daten (`cc_avbl='1'`) und der Zielspeicher freien Speicherplatz (`data_locked='0'`) meldet, wird das erste Datenwort gelesen und im Zustand WriteData in den Testspeicher geschrieben. Es wird in diesem Zustand mit dem Lesen des RxFIFOs (`cc_read`) und dem Schreiben des Testspeichers (`data_write`) kontinuierlich fortgefahren und mit jedem Datenwort der Zählerwert `data_count` dekrementiert. Ist zwischenzeitlich entweder RxFIFO leer oder der Testspeicher gesperrt,

muss wieder in den Wartezustand WaitData gewechselt werden. Die Übertragung eines Datenpaketes ist beendet, wenn der Zähler `data_count` den Wert null erreicht.

Der Lesezugriff auf den Testausgabespeicher TestOut-FIFO geschieht durch den Zustand ReadData. Der Zähler `data_count` wird auch hier mit dem im Instruktionswort übertragenen *Packet-Size*-Wert initialisiert, um die Anzahl zu lesender Datenwörter zu bestimmen. Im Zustand CheckData wird vor dem ersten Lesezugriff geprüft, ob der Zielspeicher TxFIFO gesperrt ist (`cc_locked='1'`). In diesem Fall wird direkt wieder in den Zustand ReadTC gewechselt. Ebenso abgebrochen wird, wenn bei gleichzeitig leerem TestOut-FIFO der Zähler `data_count` mit null initialisiert ist. In den Zustand ReadData wird nur gewechselt, wenn TxFIFO nicht gesperrt (`cc_locked='0'`) und TestOut-FIFO verfügbare Daten enthält (`data_avbl='1'`). Hier werden nun kontinuierlich die Datenwörter aus dem TestOut-FIFO ins TxFIFO geschrieben und der Zähler `data_count` dekrementiert (nur falls nicht bereits null). Sollte während der Übertragung des angeforderten Datenpaketes ein gesperrtes TxFIFO signalisiert werden, so bricht der Lesezugriff ab und das Datenpaket gilt als komplettiert. Ansonsten ist der Lesezugriff erst beendet, wenn der Zähler ein vollständiges Datenpaket signalisiert (`data_count=0`). Für den Fall eines mit dem *Packet-Size*-Wert null initialisierten Zählers ist das Datenpaket erst mit komplett ausgelesenem TestOut-FIFO vervollständigt. Somit können auch alle verfügbaren Daten angefragt werden, ohne den Füllstand des TestOut-FIFOs kennen und eine genaue Paketgröße angeben zu müssen.

4.4.3 Zugangskontrolle

Für die prototypische Anwendung wurde hier eine einfache Authentifizierung für den autorisierten Zugang gewählt. Bei dieser simplen Alles-oder-Nichts-Lösung, bekannt als *All-or-Nothing Access Control*, wird der volle Zugriff nach Eingabe eines korrekten Passwortes gewährt. Dieses Verfahren erhebt natürlich keinen Anspruch auf hinreichenden Zugriffsschutz. Es findet weder eine regelmäßig aktualisierende Authentifizierung noch eine komplexe Verschlüsselung der Testdaten statt. Es ist hier nur exemplarisch eine Zugangsprozedur umgesetzt.

Die Zugangskontrolle erfolgt mittels eines AES-Cores (*Advanced Encryption Standard*) entsprechend der Abbildung 4.10. AES ist ein Verschlüsselungsverfahren, das einen Klartext mittels eines Schlüssels in einen kryptischen Text überführt [AES01, MVL07, XCL09, Lan11]. AES ist auch Teil des standardisierten und im Automobilbereich etablierten Sicherheitsmoduls SHE [SHE09]. Die Idee ist es AES zu nutzen, um den Zugang zu gewähren, wenn ein kryptisches Codewort decodiert mit einem Referenzwort übereinstimmt. Hierzu wird das Codewort c_{plain} und das durch den Schlüssel k codierte Referenzwort $c_{\text{crypt}} = \text{enc}(k, c_{\text{plain}})$ auf dem Chip abgelegt. Durch diese beiden Werte kann nicht auf den Schlüssel k zurückgeschlossen werden [AES01].

Soll eine Kommunikation begonnen werden, muss zunächst das verschlüsselt übertragene Passwort k_{crypt} eingelesen werden. Dieses wird durch den AES-Dekryptor mittels ei-

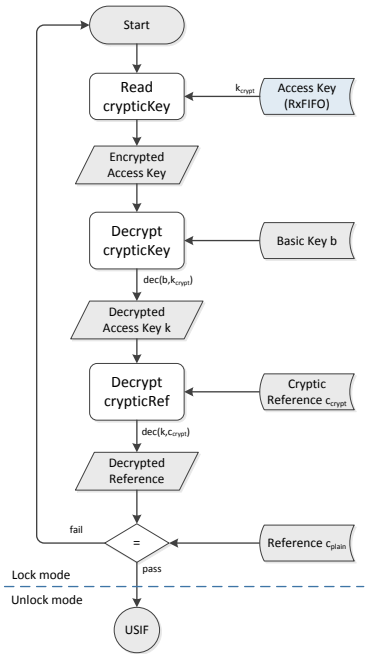


Abbildung 4.10: Kontrollfluss der Zugangsprozedur

nes Basisschlüssels b, der on-chip gespeichert ist, decodiert. Das entschlüsselte Passwort $k = \mathrm{dec}(b, k_{\mathrm{crypt}})$ wird nun als Schlüssel verwendet, um c_{crypt} durch den AES-Dekryptor zu decodieren. Ist nun c_{plain} gleich $\mathrm{dec}(k, c_{\mathrm{crypt}})$ war das Passwort korrekt und somit die Authentifizierung erfolgreich.

Für die umgesetzte Zugangsprozedur wird, sowohl für den Entschlüsselungs- als auch für den Authentifizierungsschritt, nur ein AES-Dekryptor benötigt.

4.4.4 Statusinformationen

Der Controller muss für den externen Zugriff Statusinformationen bereitstellen, um die Kommunikation zu synchronisieren und so korrekten Datentransfer sicherzustellen. In Tabelle 4.5 sind die für die Anwendung der umgesetzten Testschnittstelle nötigen Statusinformationen beschrieben. Diese Signale sind im Statusregister des USIF-Controllers zusammengefasst, welches regelmäßig über den Testzugang übermittelt wird. Die spezifische Umsetzung der Statusübermittlung über die USB- sowie die FlexRay-Verbindung ist zum jeweiligen Kommunikations-Controller unter Abschnitt 4.1 erläutert.

Statusbit	Beschreibung
`Unlocked`	Der Lock-Modus zeigt an, ob der Testzugang für den Zugriff von außen entsperrt ist.
`SCrun`	Zeigt den aktiven Scan-Test an.
`TestIn_locked`	TestIn-FIFO für weitere Daten gesperrt. Wird gesetzt wenn das FIFO mindestens halb voll ist.
`TestRef_locked`	TestRef-FIFO für weitere Daten gesperrt. Wird gesetzt wenn das FIFO mindestens halb voll ist.
`TestOut_avbl`	Zeigt verfügbare Daten im TestOut-FIFO an.
`TestOut_packet`	Zeigt an, dass mindestens ein Datenpaket (in konfigurierbarer Größe) im TestOut-FIFO verfügbar ist.
`TCsync`	Synchronisationsbit, dass der Applikation eine verarbeitete Anfrage signalisiert. Hierfür wird dieses Bit nach jeder erfolgreichen Abarbeitung einer Test-Control-Instruktion umgeschaltet. Das heißt, ist dieses Statusbit ungleich des zuvor durch die Applikation gespeicherten Zustandes, wird ein erfolgreicher Zugriff erkannt.
`TCavbl`	Zeigt an, dass Test-Control-Daten in den Eingangspuffern verfügbar sind. Dient der Applikation zur Synchronisation, indem die nächste Anfrage erst übermittelt wird, sobald keine weiteren zu verarbeitenden Instruktionen und Testdaten on-chip vorliegen.
`SCstatus[]`	Status des Test-Controllers (IDLE, ACTIVE, STOP, ERROR).
`SCfault`	Signalisiert entdeckten Fehler. Hierfür müssen zusätzlich Referenzsignaturen ins TestRef-FIFO übertragen werden.
`Buserror`	Signalisiert fehlerhafte Datenübertragung durch Kommunikations-Controller.

Tabelle 4.5: USIF-Statusinformationen

4.5 Ansteuerung der Teststrukturen

Für die USIF-Umsetzung wurde die vorgestellte SCT-Architektur als prototypische Testumgebung verwendet. Das USIF-Konzept soll sich aber nicht auf den SCT beschränken, sondern als allgemeiner Testzugang zu bestehenden BIST-Architekturen dienen. Daher wurde ebenfalls der als etablierter Industriestandard bekannte *Embedded Deterministic Test* (EDT) betrachtet und eine angepasste Testanbindung entworfen.

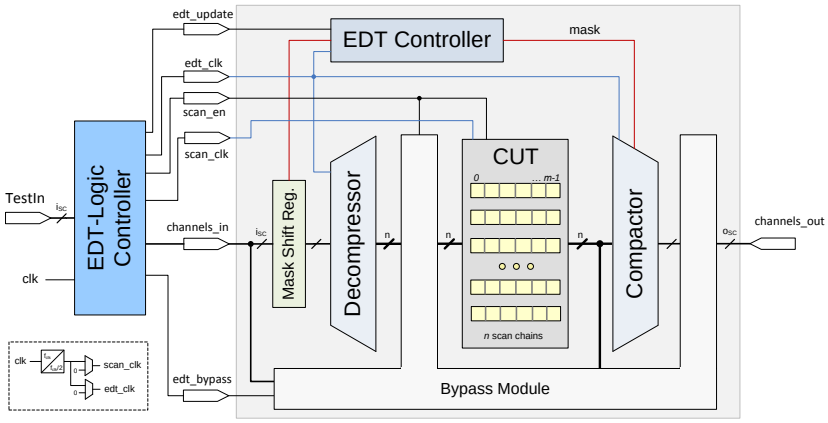


Abbildung 4.11: EDT-Logik

4.5.1 Scan-Controller

Der vorgestellte Scan-Controller erfordert keine separate Ansteuerungslogik. Durch den zugehörigen ATPG-Prozess, erläutert in Abschnitt 5.1, wird ein SCT-Programm ermittelt, das die deterministischen Muster on-chip erzeugt. Dieses Programm besteht aus einer Abfolge von Eingabewörtern für den Scan-Controller und wird dementsprechend als eine Folge von {0,1}-Strings in einer Textdatei abgespeichert. Bei einem über den Testzugang durchzuführenden Scan-Test kann diese Programmdatei ausgelesenen und der Inhalt direkt als Bitstrom interpretiert werden. Daraufhin wird dieser übertragen und im TestIn-FIFO abgespeichert. Aus dem TestIn-FIFO entnommen, werden die Testeingaben direkt an den Eingangsport des Scan-Controllers angelegt, um den Scan-Test durchführen zu lassen.

4.5.2 Embedded Deterministic Test

Als Alternative zum Scan-Controller soll der EDT von Mentor Graphics über den seriellen Testzugang ansteuerbar sein. Aufbau und Funktionsweise des EDT können in [RTK02, RTK04] nachvollzogen werden. An dieser Stelle wird nur kurz auf die Steuersignale und die dadurch erzeugten Testphasen eingegangen. Wichtig zu erwähnen ist zudem, dass im EDT eine kombinatorische Kompaktierung umgesetzt wird. Das heißt, mit jeder Testeingabe, die an die EDT-Eingangskanäle angelegt wird, ist an den Ausgangskanälen eine Testantwort zu erwarten. Somit hat der USIF-Analyzer in jedem Scan-Takt eine Testantwort zu verarbeiten.

Die EDT-Logik wird im Produktionstest durch ein ATE gesteuert. Für das USIF-Konzept muss diese Funktionalität von einer Komponente on-chip übernommen werden. Hierzu wurde der *EDT-Logik-Controller* (ELC) entworfen und implementiert. Dieser ist in erste Linie dafür zuständig, die Phasen des Scan-Tests (*Load/Unload, Shift, Capture*), wie sie durch ein ATE eingestellt würden, nachzubilden. Dabei werden die Kontrollein-

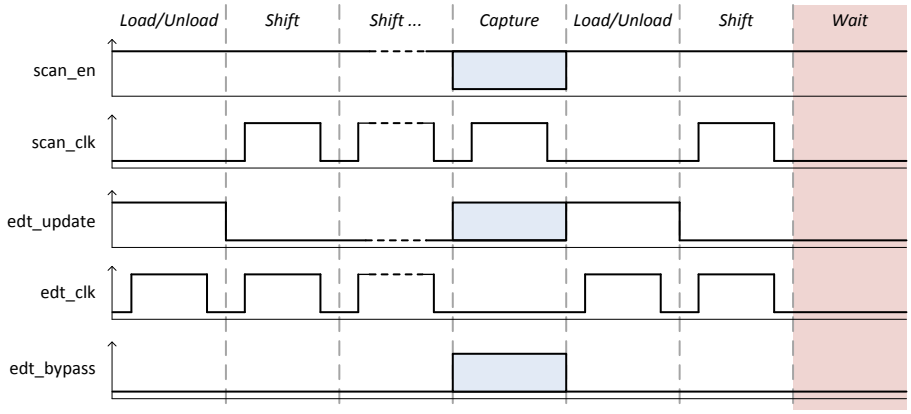


Abbildung 4.12: EDT-Phasen

gänge des EDT-Blocks (`scan_clk`, `scan_en`, `edt_clk`, `edt_update`, `edt_bypass`) durch den ELC entsprechend der Phasen gesetzt und die Testmuster an die Eingangskanäle `channels_in` angelegt, wie in Abbildung 4.11 verdeutlicht. Das Signal `edt_bypass` wird genutzt, um einen Bypass-Modus umsetzen zu können, in dem unkomprimierte Testdaten direkt in die Scan-Ketten geladen und Testantworten unkompaktiert ausgelesen werden. Da dieser Modus hier nicht explizit betrachtet werden soll, wird das Signal `edt_bypass` im Folgenden vernachlässigt.

Die Testphasen, entsprechend der Abbildung 4.12, werden für einen EDT, der eine Scan-Struktur mit einer Scan-Tiefe m_{Chain} betreibt, folgendermaßen erzeugt:

- *Load/Unload*-Phase:
  - EDT-Signale `scan_en` und `edt_update` werden für einen Takt `edt_clk` auf logisch 1 gesetzt (`scan_clk` unterdrückt).
- *Shift*-Phase:
  - EDT-Signal `scan_en` wird auf logisch 1 und `edt_update` auf logisch 0 gesetzt,
  - für m_{Chain} Takte werden `scan_clk` und `edt_clk` durchgeschaltet und
  - die Testeingabe 0 bis $m_{\text{Chain}} - 1$ des aktuellen Patterns (aus dem TestIn-FIFO) werden an die EDT-Kanäle `channels_in` angelegt.
- *Capture*-Phase:
  - EDT-Signale `scan_en` und `edt_update` werden auf logisch 0 gesetzt und
  - für einen Takt `scan_clk` angelegt (`edt_clk` unterdrückt).

Nach der Capture-Phase folgt wieder für einen Takt die *Load/Unload*-Phase. In den nächsten m_{Chain} Takten der Shift-Phase werden die Ergebnisdaten des vorangegangenen Patterns aus den EDT-Ausgangskanälen `channels_out` gelesen und gegebenenfalls nach

ELC-Control	Beschreibung
SCANCLK	EDT-Signal aus der Pattern-Datei; aktiviert die Clock scan_clk.
SCANENA	EDT-Signal aus der Pattern-Datei; wird an EDT-Eingang scan_en weitergeleitet.
EDTCLK	EDT-Signal aus der Pattern-Datei; aktiviert die Clock edt_clk.
UPDATE	EDT-Signal aus der Pattern-Datei; wird an EDT-Eingang edt_update weitergeleitet.
BYPASS	EDT-Signal aus der Pattern-Datei; wird an EDT-Eingang edt_bypass weitergeleitet.
SHIFT	Zusätzliches Kontrollbit, das eine folgende Shift-Phase indiziert, um den Shift-Counter zurückzusetzen und in den Zustand Shift zu wechseln.
STOP	Zusätzliches Kontrollbit, das benötigt wird um den Selbsttest terminieren zu lassen.

Tabelle 4.6: ELC-Steuersignale

Filterung und Analyse im TestOut-FIFO abgespeichert. Im Falle einer on-chip-Analyse werden die Testantworten mit den zugehörigen Referenzwerten verglichen.

Der EDT ist in jedem Scan-Takt komplett konfigurierbar, es muss also nicht eine statische Abfolge der Testphasen *Load/Unload*, *Shift* und *Capture* durchlaufen werden. Um einen Test mittels der aus der EDT-Pattern-Datei extrahierten Testinformationen zu gewährleisten, müssen die zugehörigen EDT-Informationen gelesen und zusammen mit den Testeingaben übermittelt werden. Hierzu werden zusätzlich zu den Testeingaben ELC-Instruktionswörter übertragen. Solch ein Instruktionswort setzt sich aus den in Tabelle 4.6 aufgelisteten ELC-Steuersignalen zusammensetzen.

Um mittels dieser Steuersignale die EDT-Phasen rechtzeitig einzustellen, muss der ELC mit doppeltem Scan-Takt, im Folgenden als ELC-Clock bezeichnet, betrieben werden. Die ELC-Clock muss on-chip verfügbar sein. Die Scan-Clock lässt sich aus dieser ableiten und entsprechend an scan_clk und/oder edt_clk anlegen. Es werden zuerst die Steuerinformationen aus dem TestIn-FIFO mit ELC-Takt gelesen und anhand derer die darin enthaltenen Eingangswerte der EDT-Kontrollports (scan_en, edt_update, edt_bypass) gesetzt oder die Scan-Clock an die EDT-Clockeingänge (scan_clk, edt_clk) durchgeschaltet. Anschließend werden die Testdaten aus dem TestIn-FIFO mit Scan-Clock direkt an die EDT-Kanäle channels_in angelegt. Somit sind die Kontrolleingänge rechtzeitig zum Scan-Takt eingestellt, um je nach Testphase die EDT-Logik zu initialisieren, die Testdaten in die EDT-Kanäle zu schieben oder einen funktionalen Takt auszuführen. Die EDT-Logik wird also komplett durch die Daten aus einer standardisierten Pattern-Datei, wie STIL, CTL, TDL oder weitere, gesteuert.

Um den Testablauf über die ELC-Steuersignale zu realisieren, wurde der in Abbildung 4.13 dargestellte Zustandsautomat umgesetzt. Sobald der Controller aktiviert ist (enable=1), wird in den Zustand LoadControl gewechselt. In diesem Zustand wird stets ein ELC-Instruktionswort, gelesen aus dem TestIn-FIFO, erwartet. Sind die ELC-

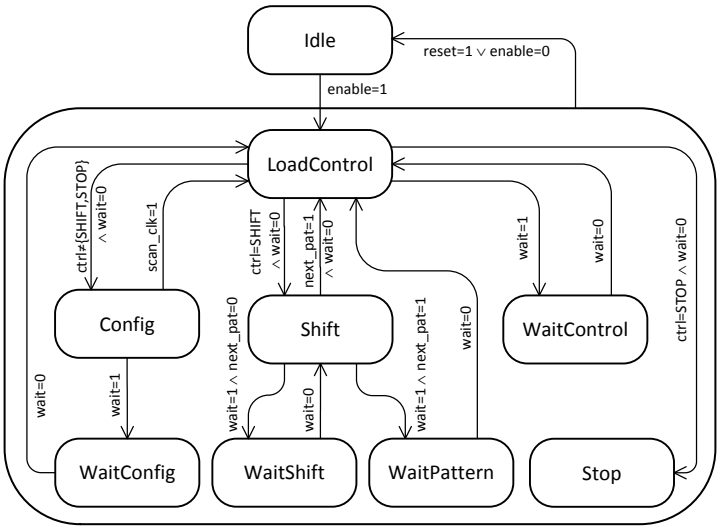


Abbildung 4.13: ELC-Zustandsdiagramm

Steuersignale `SHIFT=0` und `STOP=0`, wird der Zustand Config betreten und die im Instruktionswort enthaltenen Signale `SCANENA`, `UPDATE` und `BYPASS` an die entsprechenden EDT-Eingänge angelegt. Ist `UPDATE=1` wird zusätzlich ein Scan-Takt über den Eingang `edt_clk` durchgeschaltet und somit die *Load/Unload*-Phase erzeugt. Ist `UPDATE=0` handelt es sich um eine Capture-Phase und es wird die Clock `scan_clk` für einen Takt angelegt. Vom Zustand Config wird nach einen Scan-Takt sofort wieder in den Zustand LoadControl zurückgewechselt. Enthält ein gelesenes Instruktionswort das aktive ELC-Steuersignale `SHIFT=1`, wird der Zustand Shift betreten. Hier wird die Shift-Phase initiiert und die nächsten m_{Chain} Scan-Takte die Testeingaben aus dem TestIn-FIFO an den EDT-Eingangsport `channels_in` angelegt. Sind keine Testeingaben im TestIn-FIFO verfügbar, wird der Test angehalten (`wait=1`). Falls der Analysemodus *LogFaults* oder *PassFail* ist, muss der Test auch bei einem leeren TestRef-FIFO oder bei einem vollen TestOut-FIFO angehalten werden. Der ELC wechselt in den entsprechenden Wartezustand und nimmt die Clock-Signale `scan_clk` und `edt_clk` zurück (siehe Abbildung 4.12). Sobald das Wartesignal zurückgesetzt wird, kann mit dem Test fortgefahren werden. Ist in einem Instruktionswort das Stoppsignal `STOP=1` gesetzt, wird der Zustand Stop betreten, alle EDT-Eingangssignale zurückgesetzt und so der Scan-Test beendet.

4.6 Eingebauter Selbsttest

Der BIST-Modus ist Teil des Tests während des Hochfahrens oder Herunterfahrens eines Steuergerätes. Dieser entspricht dem *PassFail*-Modus, nur dass hier die Testkomponenten von den Testdatenspeichern TestIn-, TestRef-, TestOut- und TS-FIFO entkoppelt sind. Die Testeingabedaten und die Referenzdaten werden hierbei aus den on-chip Testspeichern *TestIn-ROM* und *TestRef-ROM* geladen.

4.6.1 BIST-Controller

Um die Testarchitektur, sei es SCT oder EDT, on-chip zu betreiben, wird eine Steuerung benötigt, die die Anbindung der on-chip Testinformationen an die Eingabeports der Testarchitektur umsetzt. Die Ablaufsteuerung des Selbsttests kann durch einen Prozessor, der dabei nicht Teil der CUT ist, vorgenommen werden. Diese Möglichkeit bietet etwa ein separater Testprozessor, wie der an der BTU Cottbus entworfene *T5016tp* [GPV02, FRG07, KKV09].

Für die Realisierung eines Startup-Tests wurde ein kompakter BIST-Controller in die USIF-Architektur integriert, da für diese spezifische Anwendung keine umfangreichere Prozessorfunktionalität vonnöten ist. Sobald der BIST-Controller gestartet wird, also hier nach dem Anschalten oder dem Reset des zu testenden Gerätes, werden Testinformationen aus dem TestIn-ROM gelesen und dem Test-Controller übergeben. Die Steuerung des Tests geschieht durch das im ROM abgelegte Testprogramm für den Test-Controller SC oder ELC. Neben der Testarchitektur ist auch die Analysekomponente an den BIST-Controller angeschlossen, um die Auswertung und gegebenenfalls eine Vorfilterung durchzuführen. Die Referenzdaten hierfür werden dem TestRef-ROM entnommen.

Die ROM-Komponenten können in der USIF-Architektur generisch der Größe des Startup-Testsatzes angepasst werden. Für die Simulation oder Synthese wird eine ROM-Komponente mit initialen Werten aus einer spezifizierten Datei im MIF-Format (*Memory Initialisation File*) geladen. Der Speicher TestIn-ROM enthält die Testeingabedaten. Zu jedem Eingangsdatum ist die Anzahl der Takte angegeben, die das Datum am Testeingang anzuliegen hat. Somit muss bei aufeinanderfolgenden gleichen Werten nur ein Wert mit zugehöriger Wiederholungsrate im ROM abgelegt werden. Bei einem Eingangsport mit einer Datenbreite von 16 Bit ergibt sich mit dem zusätzlichen Byte für die Wiederholungsrate eine Datenbreite des TestIn-ROMs von 24 Bit. Der BIST-Controller lädt für ein gelesenes Datum einen Zähler mit der Wiederholungsrate und dekrementiert diesen mit jedem folgenden Scan-Takt. Sobald der Zähler den Wert null erreicht, also bereits nach einem Takt oder nach spezifizierter Anzahl an Wiederholungstakten, wird das nächste Datum dem TestIn-ROM entnommen. Aus dem TestRef-ROM werden entsprechend der Konfiguration des Analysefilters die Referenzwerte gelesen, um die Testantworten aus der Testarchitektur auszuwerten. Die Datenbreite des TestRef-ROMs entspricht der Datenbreite des Testausgangsports der Testarchitektur.

Algorithmus 4.1 Beispiel eines TestIn-ROMs (MIF-Format) für SCT-Eingabedaten

```
1  depth = 64;           -- Anzahl der Adressen
2  width = 24;           -- Anzahl der Bits pro Datenwort
3  address_radix = hex; -- Basis der Adressen
4  data_radix = hex;     -- Basis der Datenwörter

6  content begin
7    00 : 00F200;        -- Starte Scan-Test (Start-Instr.+Init)
8    01 : 000105;        -- Lade Konfigurationsregister
9    02 : 008416;        -- Lade LFSR-Feedback
10   03 : 00BF5F;        -- Lade LFSR-Seed
11   04 : 64E000;        -- 1. Pattern: Sequenzlänge+Instr. 'Phaseshift'
12   05 : 00F000;        -- Starte neues Pattern (Start-Instr.+Init)
13   06 : 00D83A;        -- Lade LFSR-Seed (während Capture-Phase)
14   ...
15   x-3: 646000;        -- Letztes Pattern: Sequenzlänge+Instr. 'NOP'
16   x-2: 00E000;        -- Stoppe Patterngen./ Schiebe Signatur aus MISR
17   x-1: 00F000;        -- Stoppe Scan-Test (Start/Stop-Befehl)
18   [x..3F] : 000000;
19 end;
```

BIST-Datensatz für den Scan-Controller

In den folgenden Auszügen aus den Test-ROMs ist ein Steuerprogramm des Scan-Controllers (Algorithmus 4.1) und die erwarteten Testantworten (Algorithmus 4.2) zu sehen. Mit den ersten beiden Eingabewörtern des TestIn-ROMs wird der Scan-Controller gestartet und konfiguriert. Danach wird die Rückkopplung des LFSRs konfiguriert und ein Startwert geladen. Die Testmustergenerierung erfolgt in diesem Beispiel durch ein freilaufendes LFSR, ohne deterministische Nachbearbeitung der erzeugten Vektoren, indem lediglich eine Instruktion für die komplette Shift-Phase am Eingangsport des Scan-Controllers angelegt wird. Die Instruktion *Phaseshift* wird eingestellt, um bei der Erweiterung des Vektors aus der LFSR-Sequenz auf die Datenbreite des SFRs einen Pseudozufallsvektor zu erhalten (siehe Abschnitt 3.2.2).

Es folgen die Eingaben für das nächste Testmuster. Für jede Testmustergenerierung kann optional ein neuer LFSR-Startwert und auch ein neues LFSR-Feedback geladen werden. Um eine für den begrenzten Testmustersatz relativ hohe Fehlerüberdeckung zu erzielen, muss sichergestellt werden, dass jedes Testmuster eine entsprechend hohe Anzahl an neuen, nicht bereits durch vorangegangene Testmuster erfassten Fehlern überdeckt. Dies kann zum einen über eine geschickte Wahl der Feedback-Seed-Konfigurationen geschehen. Zum anderen ist aber auch eine Musteroptimierung durch die Erzeugung spezifischer Testvektoren möglich (ALU- und Bitflip- Operationen), um so gezielt bestimmte RP-resistente Fehlerpunkte einzustellen. Ferner besteht auch die Möglichkeit, einen Satz deterministischer Testvektoren unkomprimiert im ROM vorzuhalten, um diese für RP-resistente Fehler direkt in die Scan-Ketten einzugeben. Werden Pseudozufallsmuster durch solche deterministischen Muster für essentielle Fehlerpunkte, um zum Beispiel sicherheitskritische Funktionen zu überdecken, ergänzt, lässt sich ein aussagekräftiger Selbsttest durchführen. Sowohl der zusätzliche Befehlssatz für die

Algorithmus 4.2 Beispiel eines TestRef-ROMs (MIF-Format) für erwartete SCT-Testantworten

```
1  content begin
2    00 : FFCC;          -- kompaktierte MISR-Signaturen
3    01 : 006A;
4    02 : 3F29;
5    03 : B1AD;
6    ...
7  end;
```

deterministische Nachbearbeitung der LFSR-Sequenzen, als auch der Satz an deterministischen Testvektoren erfordert entsprechenden Speichermehraufwand.

Die Referenzwerte werden durch die Analysekomponente aus dem TestRef-ROM gelesen und mit den Testantworten aus dem MISR des Scan-Controllers verglichen. Auch im Startup-BIST ist eine Vorfilterung der zu analysierenden Testantworten durch Angabe einer Analyserate a_{Rate} möglich, um den Speicherplatz für die Referenzdaten einzuschränken. Dieser Ansatz wird durch das Prinzip der sequentiellen Kompaktierung des MISRs begünstigt, da sich erkannte Fehler in den Signaturen fortpflanzen. Anstatt also eine Ausgabe nach jedem Scan-Takt auszuwerten, lässt sich durch das MISR jeweils eine definierte Anzahl an Testantworten in einer Signatur akkumulieren. Da es sich um einen Pass/Fail-Test handelt, kann die Analyserate a_{Rate} hoch angesetzt werden. Im Grunde genügt eine einzige Auswertung am Ende des Testlaufs, da mit hoher Wahrscheinlichkeit erkannte Fehler nicht maskiert werden (siehe Abschnitt 2.1.9.1).

BIST-Datensatz für den EDT-Logik-Controller

Im Algorithmus 4.3 ist ein einfaches Beispiel eines Testprogramms für den ELC auszugsweise im MIF-Format angegeben. Der Testablauf ist dem Eingabeschema des EDT angepasst. Es ist hier jeweils ein ELC-Instruktionswort mit den EDT-Steuersignalen angegeben, dem ein oder mehrere Testeingabedaten für die Testkanäle (`channels_in`) folgen. Auch hier ist es ermöglicht, ein Datenwort durch Angabe einer Wiederholungsrate am EDT-Eingang zu halten.

Im Gegensatz zum Scan-Controller mit dem MISR geschieht in der EDT-Logik eine kombinatorische Kompaktierung der Ausgaben aus den Scan-Ketten. Das bedeutet, dass alle Testantworten aus den Ausgangskanälen (`channels_out`) auszuwerten sind. Die Fehlerüberdeckung sollte hier nicht durch die ungünstige Angabe einer Analyserate $a_{\text{Rate}} > 1$ im Analysefilter reduziert werden.

Algorithmus 4.3 Beispiel eines TestIn-ROMs (MIF-Format) für ELC-Eingabedaten

```
1  depth = 64;           -- Anzahl der Adressen
2  width = 24;           -- Anzahl der Bits pro Datenwort
3  address_radix = hex; -- Basis der Adressen
4  data_radix = hex;     -- Basis der Datenwörter

6  content begin
7    00 : 00000E;        -- Instruktion: Update
8    01 : 000000;        -- Channels_in: für Update-Takt
9    02 : 000027;        -- Instruktion: Shift
10   03 : 040000;        -- Channels_in: Config (4 Takte)
11   07 : 642C63;        -- Channels_in: Pattern (100 Takte 0x2C63)
12   08 : 0B0000;        -- Channels_in: Mask (11 Takte)
13   09 : 000002;        -- Instruktion: Capture
14   0A : 000000;        -- Channels_in: für Capture-Takt
15   0B : 00000E;        -- Instruktion: Update
16   0C : 000000;        -- Channels_in: für Update-Takt
17   0D : 000027;        -- Instruktion: Shift
18   0E : 040000;        -- Channels_in: Config (4 Takte)
19   0F : 64A192;        -- Channels_in: Pattern (100 Takte 0xA192)
20   10 : 0B0000;        -- Channels_in: Mask (11 Takte)
21     ...
22   x-1 : 000040;       -- Stoppe ELC
23   [x..3F] : 000000;
24 end;
```

4.7 Applikation auf Anwenderseite

Die Steuerung des USIF-Controllers und somit des diagnostischen Tests soll von einem Anwendersystem aus, das die Kommunikation mit dem peripheren USIF-Gerät beherrscht, möglich sein. Hierzu wurde das Anwendungsprogramm *Universal Scan-Test Interface Applikation* inklusive einer Benutzeroberfläche in Java implementiert. Dieses Programm, das auf einem handelsüblichen PC läuft, stellt die Schnittstelle der Testumgebung zum Anwender dar. Für die Verbindung zum zu testenden Gerät wird die USB-Schnittstelle genutzt.

Die Benutzeroberfläche der USIF-Applikation stellt Eingabemasken zur Konfiguration der Testumgebung und zur Eingabe der Testdaten bereit. Die angegebenen Daten können eingelesen, entsprechend umgewandelt und über die zum USIF-Gerät hergestellte Verbindung übertragen werden. Um die USB-Geräte, die an dem Host des Rechners angeschlossen sind, zu identifizieren, ist in der Applikation eine Java USB API mit zugehörigem Treiber eingebunden. Die Wahl fiel hier auf die USB-Bibliothek *libusb/libusb-win32*, ein Open-Source-Projekt lizenziert unter *GNU Lesser General Public License version 2.1*.

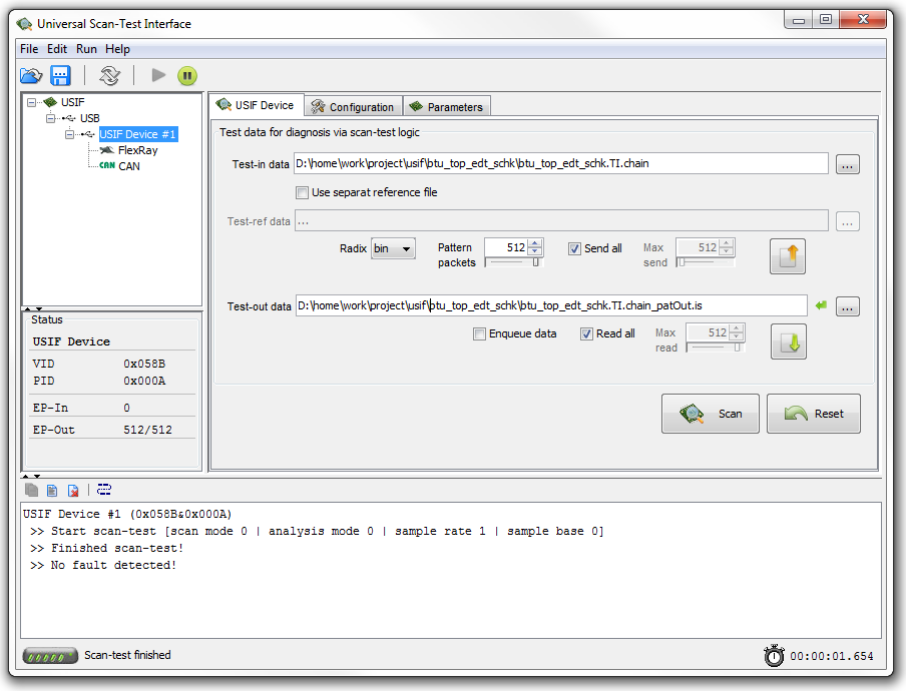


Abbildung 4.14: GUI der USIF-Applikation

Alle am USB-Host angeschlossenen USIF-Geräte werden identifiziert und konfiguriert. Hierzu werden zunächst die USB-Geräte mittels der Geräte-Identifikatoren VID und PID auf die spezifischen USIF-Geräte gefiltert. Daraufhin wird die Verbindung zum USIF-Gerät über die USB-Konfiguration eingerichtet. Für jeden Schreib- und Lesezugriff wird der entsprechende Kommunikationskanal (*Downstream*/ *Upstream Pipe*) zu den USB-Endpunkten, die für den Bulk-Transfer ausgelegt sind, geöffnet. Die zu sendenden Testdaten werden in den Endpunkt *EPOut* des Geräts geschrieben. Daten für den Lesezugriff liegen im Endpunkt *EPIn* des Geräts. Um nicht ständig Daten aus einem leeren Endpunkt *EPIn* anzufragen und so NAK-Antworten zu produzieren (welche zu *Exceptions* in der Libusb-API führen), werden zusätzliche aktuelle Geräteinformationen auf Anwenderseite benötigt. Hierzu verfügt der USB-Controller über einen zusätzlichen Interrupt-Endpunkt (siehe Abschnitt 4.1.1), um stets aktuelle Statusinformationen bereitzustellen. Ein Thread der Anwendung liest nun regelmäßig, mittels *Polling* im Intervall von 1 ms, die Statusinformationen aus, die unter anderem den Zustand des Tests und die Anzahl der im Endpunkt *EPIn* verfügbaren Daten enthalten. Detaillierte Beschreibungen der Statusinformationen sind den Tabellen 4.1 und 4.5 zu entnehmen.

Alle Funktionen der Applikation, wie das Einlesen und Parsen einer Pattern-Datei, das Konfigurieren der Testumgebung, das Senden eines Datenpakets oder das Empfangen eines Testausgabepakets, sind separat ausführbar. Um einen kompletten Testlauf zu automatisieren, wurde auch eine entsprechende Scan-Prozedur implementiert, die sämtliche Schritte vom Einlesen der Testdaten bis hin zur Aufbereitung der Testergeb-

nisse zusammenfasst. Hierzu müssen zunächst die Authentifizierung und die Angabe der einzulesenden Testinformationen erfolgen.

Wird ein USIF-Gerät in der Baumstruktur angewählt (siehe Abbildung 4.14), erscheint die Passworteingabe, um sich für den Zugriff zu authentifizieren. Das Passwort wird verschlüsselt (AES128) über die USB-Schnittstelle gesendet. Ist das Passwort korrekt, wird dies vom USIF bestätigt und die Applikation öffnet die zugehörige Kommunikationsseite (*USIF Device*, Abbildung 4.14). Hier kann die Pattern-Datei angegeben werden. Enthält diese keine Informationen zu den erwarteten Testantworten, kann für eine Testauswertung zusätzlich eine Datei mit den Referenzwerten zu den Testausgaben spezifiziert werden. Zu diesen Eingaben ist unter anderem der Radix zur korrekten Interpretation der Daten (binär, hexadezimal) und die maximale Größe der Pakete, in die der Testsatz für die Übertragung aufgeteilt wird, einzutragen. Die Paketgröße darf die Kapazität des TestIn- beziehungsweise TestRef-FIFOs nicht übersteigen, was aber durch die Applikation sichergestellt wird. Des Weiteren ist eine Ausgabedatei, in der die empfangenen Testantworten abgespeichert werden sollen, anzugeben.

Um den Scan-Test zu konfigurieren, wurde eine weitere Seite in der Benutzeroberfläche erstellt (*Configuration*, Abbildung 4.15). Über eine Eingabemaske können die nötigen Parameter spezifiziert werden. Diese sind in Tabelle 4.7 beschrieben. Darüber hinaus ist es möglich, die Scan-Test-Parameter und weitere USIF-Register (siehe *USIF-Memory Map* im Anhang A.3) über ein zusätzliches Editorfenster zu konfigurieren. In diesem Editor sind die zu übermittelnden Daten im USIF-Datenformat binär oder hexadezimal anzugeben.

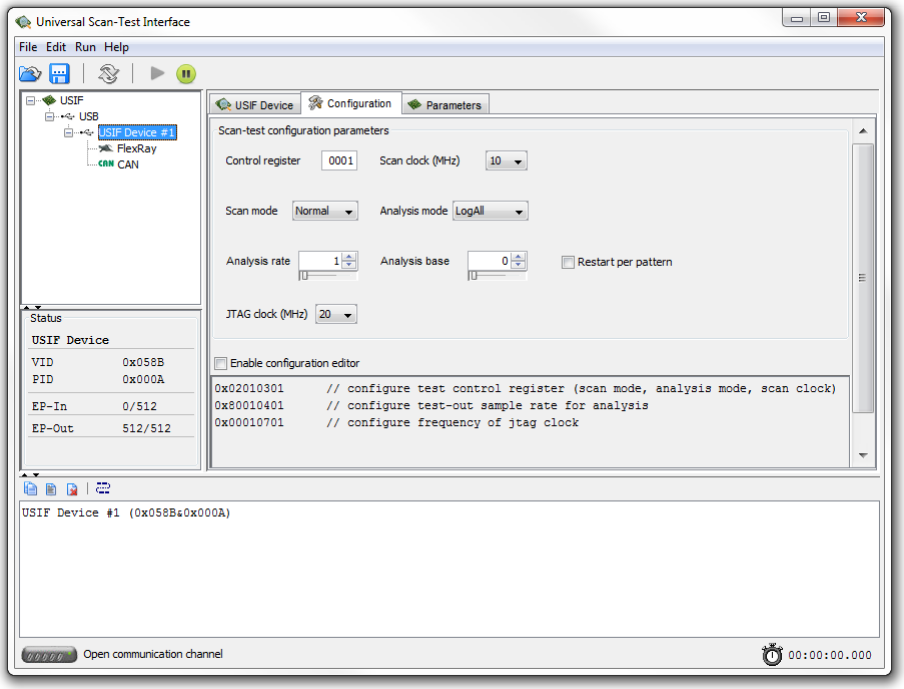


Abbildung 4.15: Konfigurationsparameter der USIF-Applikation

Scan-Test-Parameter	Beschreibung
Control Register	Das 4-Bit Control-Register soll Steuersignale für das USIF-Controller bereitstellen. Das erste Bit (LSB) erzwingt ein Reset der Testkomponenten (inkl. Test-FIFOs) vor der Konfiguration. Die weiteren Bits sind bisher nicht genutzt.
Scan Mode	Angabe des Scan-Modus: *Normal* (*Decompress*) oder *DirectScan* (*Bypass*). Der Scan-Modus *Decompress* bedeutet, dass die Testarchitektur mit den komprimierten Testdaten angesteuert wird. Mittels des Modus *DirectScan* können die unkomprimierten Testmuster in die Scan-Ketten geschoben werden.
Analysis Mode	Angabe des Analysemodus: *LogAll*, *LogFaults* oder *PassFail*. Der Modus *LogAll* ermöglicht die Aufzeichnung aller Testausgaben (in Abhängigkeit vom Rate-Basis-Tupel). Im *LogFaults*-Modus werden nur die Testantworten gespeichert, die ungleich ihren Referenzwerten sind, also durch die Fehler erkannt wurden. Durch den *PassFail*-Modus wird der Test abgebrochen, sobald ein Fehler erkannt wird.
Scan Clock	Angabe des Scan-Taktes. Die möglichen Werte sind 10, 20 oder 50 MHz.
Analysis Rate	Die Analyserate a_{Rate} gibt an, dass nur jede a_{Rate}-te Testantwort auszuwerten und ggf. abzuspeichern ist ($1 \leq a_{\text{Rate}} < 256$).
Analysis Base	Die Analysebasis a_{Base} (wobei $a_{\text{Base}} < a_{\text{Rate}}$) gibt an, ab welcher Testantwort mit der Auswertung begonnen wird (falls $a_{\text{Base}} \geq a_{\text{Rate}}$, wird $a'_{\text{Base}} = a_{\text{Base}} \mod a_{\text{Rate}}$ gespeichert).
Restart per Pattern	Der Parameter gibt an, ob die Rate über ein Pattern hinaus weitergezählt (0) oder jeweils mit der (a_{Base})-ten Testantwort der entsprechende Zähler zurückgesetzt wird (1).
JTAG Clock	Angabe des JTAG-Taktes. Die möglichen Werte sind 10, 20, 30 oder 50 MHz.

Tabelle 4.7: Konfigurationsparameter der USIF-Applikation

Die für einen zu testenden IC nötigen Metadaten, Parameter der Testarchitektur und gegebenenfalls weitere Strukturinformationen können in einer Konfigurationsdatei abgespeichert und für die Ausführung eines diagnostischen Tests geladen werden.

4.7.1 Extraktion der Testmuster

Das Testprogramm für den Scan-Controller wird durch einen zugehörigen ATPG-Prozess erstellt und als Bitstrom aufbereitet in einer Pattern-Datei abgespeichert. Somit

kann die Applikation die erstellte Pattern-Datei einlesen, direkt als Bitstrom interpretieren und an das USIF übertragen.

Die durch kommerzielle ATPG-Programme erzeugten Testmuster werden in einem standardisierten Format einer *Pattern Description Language* (PDL) abgespeichert. Gebräuchliche Pattern-Formate sind unter anderem *Standard Test Interface Language* (STIL1450, STIL2005), *Core Test Language* (CTL), *Waveform Generation Language* (WGL), *Texas Instruments Test Description Language* (TDL 91) oder *Fujitsu Test Data Description Language* (FTDL). Diese werden üblicherweise durch ATEs interpretiert, um die BIST-Architektur im Fertigungstest anzusteuern.

Die Applikation kann auch für eine automatisierte Pattern-Extraktion aus einer solchen Dateischnittstelle genutzt werden. Somit kann ein durch das USIF angesteuerter EDT mittels vorliegender EDT-Pattern betrieben werden. Aus der Pattern-Datei müssen die nötigen Informationen extrahiert und umgewandelt werden, um die Testdaten als Bitstrom auf dem seriellen Bus zu übertragen. Die USIF-Applikation unterstützt hierzu die Pattern-Formate STIL, CTL und TDL91.

Es wurde eine Funktion zur Aufbereitung der Testmuster aus Dateien im STIL1450-, STIL2005- und CTL-Format implementiert. Da diese Formate einen äquivalenten Aufbau besitzen, kann hierfür eine gemeinsame Funktion, die die Pattern-Dateien scannt und die nötigen Testdaten extrahiert, genutzt werden. Aus der textuellen Testablaufbeschreibung wird der Pattern-Bitstrom für die Testkanäle (`channels_in`) und die Referenzausgaben (`channels_out`) gelesen. Die Kontrolldaten werden durch die Funktion je nach Scan-Phase gesetzt und vor jedem Testpattern als ELC-Instruktion übertragen. Die Makro- und Prozeduraufrufe in diesen Pattern-Dateien werden nicht behandelt.

Um sämtliche Testinformationen der Pattern-Datei direkt, ohne Konfiguration über Makros und Prozeduren, zu entnehmen, wurde ein weiteres Format hinzugezogen. Hier erwies sich das TDL-Format als gut lesbar. Es handelt sich um ein statisches Format, in dem sämtliche Eingaben, inklusive der Kontrollinformationen, direkt nacheinander folgend definiert sind. Es wird zuerst die Initialisierung der Testschnittstelle gelesen, um dann die Testdaten für die EDT-Kontrollports (`scan_en`, `edt_update`, `edt_bypass`) und Testkanäle (`channels_in`) zu interpretieren und als Bitstrom zu speichern.

4.7.2 Scan-Prozedur

Mittels der Scan-Prozedur ist der Testablauf, die paketweise Übertragung der Testeingaben und -ausgaben über die USB-Schnittstelle und die Testauswertung, automatisiert. Zunächst wird der Testsatz aus der eingelesenen Datei je nach Format extrahiert und in einem Bytearray für die Testeingaben und gegebenenfalls in einem separaten Array für die Referenzwerte abgespeichert. Als Erstes werden dem USIF die spezifizierten Konfigurationsdaten übermittelt und daraufhin bereits der Test-Controller aktiviert. Dieser startet aber erst, sobald die nötigen Testdaten verfügbar sind. In Algorithmus 4.4 ist ein Auszug der Scan-Prozedur im Pseudocode dargestellt. Zusätzlich kann die Prozedur auch anhand des Kontrollflussdiagramms im Anhang A.6 nachvollzogen werden.

Mittels der Funktion *send*(*int* s_1, *int* s_2, *byte*[] s_3, *int* s_4) werden Instruktionen oder Daten an das USIF-Gerät übertragen, wobei der Parameter s_1 die USIF-Instruktion (siehe Tabelle 4.3) und der Parameter s_2 die adressierte Komponente (siehe Anhang A.3) spezifiziert. Ist hier der dritte optionale Parameter s_3 angegeben, ist das entsprechende Datenpaket zu übertragen. Der Parameter s_4 gibt dabei die Größe (in Byte) des zu sendenden Paketes an. Im Falle einer einzelnen Instruktion ($s_4 = 4$) wird dieser Parameter nicht explizit aufgeführt.

Die durch eine Lese-Instruktion (`Read`, `ReadConfig`) angeforderten Daten werden durch den Funktionsaufruf *receive*(*byte*[] r_1, *int* r_2) empfangen und im Bytearray r_1 abgespeichert. Der Parameter r_2 gibt die Größe (in Byte) des zu empfangenden Paketes an. Falls Referenzdaten aus der Testmusterdatei oder einer separaten Datei vorhanden sind und diese nicht zur on-chip Analyse verwendet werden, besteht die Möglichkeit, eine Auswertung und Aufbereitung der Testergebnisse mittels der Funktion *doAnalysis*(*byte*[] a_1, *byte*[] a_2, *byte*[] a_3) durchzuführen. Hierzu werden die Referenzdaten a_1, die empfangenen Testausgaben a_2 und, falls vorhanden, die zugehörigen Zeitstempel a_3 dieser Auswertungsfunktion übergeben. Die Zeitstempel werden für den *LogFaults*-Modus benötigt, um die Referenzwerte den Testantworten zuzuordnen. Für den Fall, dass im *LogAll*-Modus durch eine Analyserate $a_{\text{Rate}} > 1$ nicht sämtliche Ausgaben abgespeichert werden, sind keine Zeitstempel vonnöten, da die Rate a_{Rate} der Applikation bekannt ist und so auch die entsprechenden Referenzwerte für die Analyse herausgefiltert werden können. Steht der Auswertungsfunktion ein Fehlerwörterbuch zur Verfügung, kann eine verfälschte Signatur auch auf eine Klasse von Fehlerpunkten zurückgeführt, also potentielle Defekte diagnostiziert werden. Die Funktion *saveFile*() fügt die durch die vorangestellte Funktion *doAnalysis*() aufbereiteten Testergebnisse in eine spezifizierte Zieldatei ein. Der dadurch realisierte Testlauf wird im Folgenden beschrieben.

Aus dem Array der Testeingabedaten `arrayIn` wird je nach spezifizierter maximaler Paketgröße ein Teildatensatz entnommen und an das TestIn-FIFO übertragen. Dem Datenpaket wird hierzu das nötige USIF-Instruktionswort als Header vorangestellt. Vor dem Versenden wird die Paketgröße des Datensatzes gegebenenfalls auf den Umfang des verfügbaren freien Speicherplatzes im USB-Empfangspuffer verringert. Das Datenpaket wird schließlich durch den USB-Host wiederum zerlegt, in USB-Pakete verpackt und versendet.

Im Falle der Analysemodi *LogFaults* oder *PassFail* muss zudem ein Paket aus dem Array der Referenzdaten `arrayRef` in das TestRef-FIFO geschrieben werden. Daraufhin startet der Test-Controller den Scan-Test, also dekomprimiert die Testdaten zu Testmustern, schiebt diese in die Scan-Ketten und kompaktiert die Testantworten zu Signaturen. Diese Testausgaben werden je nach Analysemodus und Analyserate entweder direkt oder nach Vorfilterung in das TestOut-FIFO geschrieben. Der Test befindet sich im Wartezustand, wenn das TestIn-FIFO leer oder, falls der Analysemodus nicht *LogAll*, das TestRef-FIFO leer oder das TestOut-FIFO voll ist.

Die Test- und FIFO-Zustände (siehe Tabelle 4.5) werden über eine Statusbotschaft mittels Polling des USB-Statusendpunktes regelmäßig abgefragt. Wird im *PassFail*-Modus

ein gefundener Fehler signalisiert, ist der Scan-Test beendet und die Scan-Prozedur terminiert. Optional kann die Testantwort, durch die der Fehler erkannt wurde, mit zugehörigem Zeitstempel ausgelesen werden. Ansonsten liest die Applikation das TestOut-FIFO komplett aus, sobald verfügbare Testantworten gemeldet werden. Hierzu wird dem USIF-Controller die entsprechende Leseanweisung, mit dem TestOut-FIFO als Quelladresse, übermittelt. Im *LogFaults*-Modus werden daraufhin ebenfalls die zugehörigen Zeitstempel aus dem TS-FIFO gelesen.

Die nächsten Testdatenpakete werden gesendet, sobald Speicherplatz im TestIn- beziehungsweise TestRef-FIFO frei ist. Dies könnte theoretisch direkt ohne weitere Verzögerung erfolgen, da die Geschwindigkeit der Datenverarbeitung durch die Applikation und die Übertragung über die USB-Verbindung im Verhältnis zur Verarbeitung der Testdatenpakete auf dem IC sehr langsam erfolgt. Da aber auf die Statusmeldungen reagiert wird und diese im festen Intervall abgefragt werden, können Wartezeiten entstehen.

Nach Übermittlung des gesamten Testsatzes ist der Scan-Test abgeschlossen. Es werden die restlichen Testantworten ausgelesen und der Testantwortsatz mit zugehörigen Zeitstempeln in die spezifizierte Ausgabedatei geschrieben. Im Falle des *LogAll*-Modus werden die Zeitstempel durch die Applikation eingefügt und zudem, falls Referenzdaten vorhanden sind, die fehlerhaften Testantworten markiert.

Algorithmus 4.4 Scan-Prozedur

```
1  - Sei Integer s die spezifizierte Paketgröße in Byte.
2  - Sei Integer d die Datenbreite der Datenspeicher (TestIn-/TestRef-/
       TestOut-FIFO) in Byte.
3  - Sei arrayIn ein Byte-Array, das die Testeingabedaten aus der
       eingelesenen Pattern-Datei enthält.
4  - Sei arrayRef ein Byte-Array, das ggf. die Referenzdaten (aus der
       eingelesenen Pattern-Datei) enthält.
5  - Sei arrayOut ein Byte-Array der Größe s (soll jeweils ein empfangenes
       Paket des TestOut-FIFOs aufnehmen).
6  - Sei arrayTS ein Byte-Array der Größe s (soll jeweils ein empfangenes
       Paket des TS-FIFOs aufnehmen).

8  // Starte Scan-Test
9  send(StartSC, USIF)

11 while arrayIn not empty
12   // Lese ein Paket aus dem TestOut-FIFO aus
13   if TestOut_packet = true
14     send(Read, TestOut)
15     receive(arrayOUT, s)
16     if AnalysisMode != LogAll
17       send(Read, TS)
18       receive(arrayTS, s)
19     endif
20     doAnalysis(arrayRef, arrayOut, arrayTS)
21     saveFile()
22   endif
```

```
23    // Schreibe Referenzdaten in das TestRef-FIFO
24    if AnalysisMode != LogAll & TestRef_locked = false
25       & arrayRef not empty
26      - Sei packetRef ein Byte-Array der Größe s.
27      - Entnehme arrayRef s Byte und schreibe diese in packetRef.
28      send(Write, TestRef, s, packetRef)
29    endif

31    // Schreibe Testeingabedaten in das TestIn-FIFO
32    if TestIn_locked = false
33      packetIn = removePacket(arrayIn)
34      - Sei packetIn ein Byte-Array der Größe s.
35      - Entnehme arrayIn s Byte und schreibe diese in packetIn.
36      send(Write TestIn, s, packetIn)
37    endif

39    if AnalysisMode = PassFail & SCfault = true
40      break while
41    endif
42  endwhile

44  // Lese restliche Daten des TestOut-FIFOs aus
45  while TestOut_avbl = true
46    if TestOut_packet = true
47      send(Read, TestOut)
48      receive(arrayOut, s)
49    else
50      send(ReadConfig, TestOutUsedw)
51      - Sei wRest ein Bytearray (soll den Wert des Registers TestOutUsedw
            aufnehmen)
52      receive(wRest, 4)
53      - Sei Integer r = (Integer)wRest * d
54      send(Read, TestOut)
55      receive(arrayOut, r)
56    endif

58    if AnalysisMode != LogAll
59      send(Read, TS)
60      receive(arrayTS, s)
61    endif

63    doAnalysis(arrayRef, arrayOut, arrayTS)
64    saveFile()
65  endwhile

67  // Beende Scan-Test
68  send(StopSC, USIF)
```

4.8 Zusammenfassung

In diesem Kapitel wurde die Umsetzung von der Testarchitektur bis hin zu der Anwenderschnittstelle präsentiert. Zunächst wurden die erstellten Kommunikations-Controller, die den Zugang über die FlexRay- beziehungsweise USB-Schnittstelle ermöglichen, erläutert. Der Austausch der Testdaten geschieht hier über paketweise Übermittlung an die Testspeicher. Diese Speicher wurden als separate FIFO-Einheiten dargestellt. Das TestIn- und das TestRef-FIFO nehmen empfangene Datenpakete auf. Mittels der Testinformationen aus dem TestIn-FIFO wird die Testarchitektur betrieben. Die optionalen Referenzdaten aus dem TestRef-FIFO dienen der schaltungsinternen Auswertung. Die Testausgaben werden nach optionaler Filterung und Auswertung in dem TestOut-FIFO abgelegt. Den Testausgaben sind zudem optionale Zeitstempel zugeordnet.

Mit dem USIF-Controller wurde die zentrale Steuerungseinheit des Testzugangs erläutert. Durch diesen wird der komplette Zugriff innerhalb des Testmodus gesteuert. Um neben dem Test mittels des Scan-Controllers (SCT) auch den industriell weit verbreiteten *Embedded Deterministic Test* zu unterstützen, wurde ein zusätzlicher EDT-Controller entworfen. Somit kann diese prototypische Konzeptlösung sowohl für den SCT also auch für den EDT ausgelegt werden.

Zudem wird ein Selbsttest auf Basis der integrierten Teststruktur ermöglicht. Hierzu wurde ein kompakter BIST-Controller umgesetzt, der die Ansteuerung der Testarchitektur ausschließlich mit den im IC gespeicherten Testinformationen realisiert. Es kann ein effizienter Selbsttest mittels on-chip generierter Pseudozufallsmuster, ergänzt um zusätzliche deterministische Muster für RP-resistente Fehler, durchgeführt werden.

Abschließend wurde die implementierte Applikation, die die Anwenderschnittstelle zur Verfügung stellt, präsentiert. Durch diese wird die Verbindung zum peripheren Gerät über die USB-Schnittstelle hergestellt. Es wird ein komplett automatisierter Testablauf vom Einlesen umfangreicher Testdaten aus spezifischen Testmusterdateien bis hin zur Auswertung der empfangenen Ergebnisdaten unterstützt.

Kapitel 5

5 Ergebnisse und Auswertung

In diesem Kapitel findet eine Auswertung des entwickelten Testkonzepts statt. Hierzu werden die Ergebnisse, die durch die prototypische Umsetzung ermittelt wurden, betrachtet. Zuvor wird der genutzte ATPG-Prozess vorgestellt, der die Modifizierung der Schaltung entsprechend dem DFT-Konzept, die Erzeugung der Testmuster und deren Komprimierung umfasst.

5.1 Erzeugung der Testdaten

Zur Erzeugung der Testdaten im ATPG-Prozess, der dem SCT angepasst ist, wurde der Fast-VHDL-Simulator (FVHDL)[1] genutzt. Dieses Programm wurde entwickelt, um Fehlersimulation und Testmustererzeugung einer strukturbeschriebenen VHDL- oder Verliog-Schaltung umzusetzen. FVHDL basiert auf Perl-Skripten, die unter anderem auf Programme wie FastScan, MILEF, ATALANTA oder HOPE zurückgreifen.

Das FVHDL-Programm bietet folgende Funktionalitäten:

- Fehlersimulation für Einzelhaftfehler und Transitionsfehler,
- Testmuster-Relaxation für Einzelhaftfehler und Transitionsfehler,
- Optimierung eines Testmustersatzes,
- Erzeugung eines Scan-Controller-Programms aus einem Testmustersatz,
- Funktionen zur Optimierung des Low-Power Scan-Test, wie
  - Berechnung der Schaltungsaktivität mittels des Zero-Delay-, Unit-Delay- oder Library-Delay-Modells
  - Optimierung der Schaltungsaktivität anhand des Zero-Delay- oder Preferred-Fill-Modells

[1] Der FVHDL-Simulator ist eine proprietäre Lösung, die in einem vorangegangenen Projekt der Forschungsgruppe entstanden ist.

5.1.1 ATPG-Toolchain

Die folgende ATPG-Toolchain wird zur Erzeugung komprimierter Testmuster für den Test von Schaltungen mittels des Scan-Controller-Konzepts genutzt. Der beschriebene Ablauf und die dafür verwendeten Skripte sind innerhalb eines vorangegangenen Projekts der Forschungsgruppe entstanden. Nähere Erläuterungen zu den hier erwähnten Skripten finden sich im Anhang A.7.

5.1.1.1 Technologie-Mapping

Die zu modifizierende Schaltung wird für die weiteren Schritte des ATPG-Prozesses als strukturelle Verilog-Beschreibungen mit der zugrundeliegenden Technologiebibliothek *Si2 NanGate FreePDK45 Generic Open Cell Library* erwartet. Daher hat zunächst ein Technologie-Mapping in das benötigte Format zu erfolgen.

Für die Untersuchungen wurden ISCAS89-, ITC99-Schaltungen und auch VLIW-Prozessoren als VHDL-Beschreibungen herangezogen. Diese werden in strukturelle Verilog-Schaltungen ausschließlich bestehend aus Gattern der angegebenen Technologiebibliothek konvertiert.

5.1.1.2 Integration der Scan-Strukturen

In diesem Schritt wird die nötige Teststruktur in die Schaltung eingebracht. Hierzu werden die internen Speicherelementen um Multiplexer und Steuersignale ergänzt, um die nötigen Scan-Flipflops zu erzeugen und diese zu Scan-Ketten verknüpfen zu können. Es handelt sich hierbei um ein *Muxed-D*-Design. Um die gewünschte Scan-Struktur zu erhalten, werden aus den Scan-Flipflops n_{Chain} Ketten möglichst gleicher Länge gebildet. Die Scan-Tiefe m_{Chain} wird dabei durch die Scan-Kette(n) mit der höchsten Anzahl an Flipflops bestimmt. Zusätzlich wird eine Version mit nur einer Scan-Kette, in der sämtliche Flipflops nacheinander verbunden sind, erstellt. Diese Erweiterung der Netzliste erfolgt mittels eines Skripts unter Angabe der zugrundeliegenden Technologiebibliothek, der Kettenanzahl und weiteren optionalen Parametern, die die Ausgabe interner Signale betreffen.

5.1.1.3 Testmustererzeugung

Testmuster für die Schaltung werden mittels Mentor Graphics FastScan erstellt. Hierbei wird aufgrund einer effizienteren Weiterverarbeitung die mit einer Scan-Kette erzeugte Schaltung genutzt. Außerdem wird für die anschließenden Arbeitsschritte der von FastScan ausgegebene Testsatz im ASCII-Format in das MILEF-Format konvertiert. MILEF ist ein offenes frei verfügbares ATPG-Tool, das Fehlersimulation durchführen und kompakte Testmuster generieren kann [GV92]. Es bietet auch die Möglichkeit,

direkt einen (umfangreicheren) Testmustersatz mit zusätzlichen Freiheitsgraden zu erzeugen, um eine weitergehende Komprimierung zu erlauben. Aber aufgrund einiger Einschränkungen[2] wurde es in dieser Toolchain durch FastScan ersetzt.

Nach Abschluss dieses Schrittes liegen das erzeugte FastScan-Testmuster, das MILEF-Testmuster und die zugehörigen Log-Dateien, unter anderem mit den Angaben zur Teststruktur und Fehlerüberdeckung, vor.

5.1.1.4 Relaxation

Relaxation bedeutet aus einem stark spezifizierten Testmustersatz, mit wenigen bis keinen Freiheitsgraden, einen schwach-spezifizierten Testmustersatz mit einer hohen Anzahl an *Don't Care Bits* (DCB) zu erstellen. Diese DCB-Erzeugung ist für die Komprimierung der Testdaten notwendig und wird durch das zuvor erwähnte Programm FVHDL unter Angabe der Netzliste, der Testmusterdatei, des Fehlermodells und weiterer Parameter (siehe Anhang A.7) realisiert.

Es wird zunächst zu jedem Muster eine vollständige Fehlersimulation durchgeführt. Nachdem dies für alle Muster geschehen ist, werden zu jedem Muster essentielle Fehler ermittelt. Überdeckt ein Muster keine essentiellen Fehler, wird es aus dem Testmustersatz entfernt. Danach beginnt die Relaxation. Anschließend wird durch Merging (statische Komprimierung) versucht, den Testmustersatz zu optimieren. Das Resultat ist ein Testmustersatz mit hohem DCB-Anteil, wobei die Fehlerüberdeckung erhalten bleibt. Ein Testsatz mit einem hohen Wert an Freiheitsgraden bietet eine bessere Ausgangslage für die Programmerzeugung des Scan-Controllers. Um eine signifikante Komprimierung der Testdaten zu ermöglichen, sollte ein Testmustersatz mit einem DCB-Anteil von über 80% erreicht werden [KV08, KV10].

In Tabelle 5.1 wird der DCB-Anteil für einige Schaltungen, die in den folgenden Abschnitten näher untersucht werden, dargestellt. Im Anhang B.1 ist die ausführliche Tabelle zu dieser Betrachtung zu finden.

Die Testschaltungen sind in der ersten Spalte aufgeführt. Spalte 2 gibt die Anzahl der durch FastScan ermittelten Testmuster und Spalte 3 die nach dem Relaxationsschritt erhaltene Testmusteranzahl an. Der DCB-Anteil ist der letzten Spalte zu entnehmen. Für jede Schaltung wurden zwei Testsätze, die sich nur in einem Prozessschritt unterscheiden, erstellt. In einer Version wurde der durch FastScan erstellte Testmustersatz durch einen anschließenden Optimierungsprozess, der ebenfalls Teil des FastScan-ATPGs ist, reduziert. In der anderen Version wurde diese Optimierung durch FastScan unterbunden. Es sollte dadurch geprüft werden, ob die Relaxation mit anschließender Optimierung durch den größeren Suchraum eines umfangreicherer Testsatz begünstigt wird oder ein bereits optimierter Testsatz bessere Ergebnisse liefert.

[2] MILEF beherrscht nur rein kombinatorische Schaltungen und nur das Stuck-at-Fehlermodell. Zudem kommt es bei der Testmustererzeugung für größere Schaltungen zu Speicherverwaltungsproblemen.

Testschaltung	Anzahl Testmuster		DCB-Anteil %
	FastScan	Relaxation u. Optimierung	
s13207	309	262	94,23
s13207 voroptimiert	284	259	94,12
s15850	201	155	87,66
s15850 voroptimiert	179	151	87,62
s38417	182	165	85,88
s38417 voroptimiert	182	164	85,78
b17	592	562	92,50
b17 voroptimiert	575	564	92,54
b18	532	530	89,51
b18 voroptimiert	532	530	89,51
b19	539	538	89,54
b19 voroptimiert	538	538	89,54
b22	521	472	78,15
b22 voroptimiert	500	470	78,01
VLIW16S4	470	455	92,29
VLIW16S4 voroptimiert	455	455	92,29
VLIW32S4	717	716	97,84
VLIW32S4 voroptimiert	717	716	97,84

Tabelle 5.1: DCB-Anteil in Testmustersätzen mit und ohne Voroptimierung

Es ist zu erkennen, dass sich eine Voroptimierung, wenn überhaupt, nur marginal auf den DCB-Anteil im resultierenden Testsatz auswirkt. Ebenso werden durch die Optimierung nach der Relaxation annähernd die gleichen Werte für den Testsatzumfang, unabhängig von dem zusätzlichen Optimierungsschritt durch FastScan, erreicht. Im Falle der Schaltung b22 wird der Testmustersatz sogar etwas reduziert, dies aber auf Kosten des DCB-Anteils. Gerade bei den komplexeren Schaltungen wie b18, b19, VLIW16S4 oder VLIW32S4 hat eine Voroptimierung keinen Einfluss auf den Umfang und den DCB-Anteil. Das bedeutet auch, dass die Optimierung des Relaxationsalgorithmus der Optimierung durch FastScan gleichwertig ist.

Es zeigt sich, dass die für die Testmusterkomprimierung nötige Anzahl an DCBs schon für kleinere Schaltungen, wie beispielsweise an s13207 oder s15850 zu sehen, realisierbar ist. Je umfangreicher die Schaltungen, desto größer ist die Wahrscheinlichkeit, einen hohen DCB-Anteil mit Werten um die 90% zu erzielen.

5.1.1.5 Programmerzeugung für den Scan-Controller

Der abschließende Schritt des ATPG-Prozesses ist die Testsatzkomprimierung. Hierzu wird ein Programm für den Scan-Controller durch das FVHDL-Programm erstellt. Das bedeutet, die Testmuster werden hier auf LFSR-Sequenzen und Scan-Controller-

Operationen abgebildet. Vor der Erzeugung eines Testmusters werden ein LFSR-Seed und gegebenenfalls ein Feedback angegeben. Für jeden Testvektor eines Testmusters wird ein ALU-Befehl (siehe Tabelle 3.1) und eine Reihe von Bitflip-Adressen spezifiziert, um diesen auf Basis der eingestellten LFSR-Sequenz on-chip nachbilden zu lassen.

Für die Programmermittlung wird die für den Scan-Test erweiterte Schaltung mit zuvor spezifizierter Anzahl an Scan-Ketten verwendet. Diese Schaltung wird auch in die USIF-Architektur als CUT integriert und mit dem Scan-Controller verknüpft. Durch die kompakten Steuerinformationen, die über den Testzugang übertragen werden, generiert der Scan-Controller die für die Schaltung ermittelten Testmuster.

In Tabelle 5.2 sind bekannte ISCAS89- und ITC99-Benchmark-Schaltungen zu sehen, für die der ATPG-Prozess bis hin zum Scan-Controller-Programm durchlaufen wurde. Um auch größere Designs zu betrachten, wurde der am Lehrstuhl entworfene VLIW-Prozessor[3] (*Very Long Instruktion Word*) in verschieden Auslegungen herangezogen [Sch06]. Dabei handelt es sich um eine superskalare Architektur mit statischer Programmablaufplanung. Dem Bezeichner der Schaltung ist hier die Anzahl der parallelen Datenpfade, die sogenannten Slots, und die Datenbreite eines Datenpfades zu entnehmen. Der VLIW16S4 beispielsweise wurde als 16-Bit-Architektur mit 4 Slots synthetisiert. Eine ausführliche Tabelle mit weiteren Schaltungen ist im Anhang B.2 angegeben.

Zu jeder Schaltung der Spalte 1 wurde ein Scan-Design mit unterschiedlicher Anzahl an Scan-Ketten erstellt. In der zweiten Spalte ist die Anzahl der zu Scan-Ketten verknüpften Flipflops n_{SFF} und zusätzlich die Anzahl aller enthaltenen Gatter angegeben. In den Spalten 3 und 4 ist die Scan-Struktur dargestellt. Da die Scan-Flipflops zu möglichst gleichlangen Ketten aufgeteilt werden, ergibt sich die Scan-Tiefe m_{Chain} aus der spezifizierten Anzahl an Scan-Ketten n_{Chain}. Die Spalte 5 enthält die Anzahl der durch Relaxation und Optimierung erhaltenen Testmuster. Die durch diese Testmuster erreichbare Fehlerüberdeckung ist in der darauffolgenden Spalte zu finden. In Spalte 7 wird der im Testsatz enthaltene Anteil an DCBs wiedergegeben. Der Datenumfang des unkomprimierten und komprimierten Testsätze ist in den Spalten 8 und 9 zu sehen. Der unkomprimierte Testsatz ergibt sich hier aus der Testmusterzahl (Spalte 5) und der Scan-Struktur, umfasst also $n_{\mathrm{Chain}} * m_{\mathrm{Chain}} * n_{\mathrm{Pat}}$ Bit. Als komprimierter Testsatz ist das ermittelte Scan-Controller-Programm zu verstehen.

Die letzte Spalte stellt schließlich den Komprimierungsfaktor dar. Dieser bezieht sich hier nur auf die Reduzierung durch die Abbildung des bereits optimierten Testmustersatzes auf ein Scan-Controller-Programm. Der genutzte ATPG-Prozess von FastScan nutzt die Freiheitsgrade, die durch die DCBs gegeben sind, um Testmuster zusammenzuführen und somit den Testsatz um äquivalente Testvektoren zu reduzieren. Ebenso wird nach der Relaxation verfahren, um eine weitere Reduzierung zu erreichen. Das heißt, in diesem Prozessschritt der Programmerzeugung wird schon von einem möglichst minimalen Testmustersatz ausgegangen. Es findet nach der Ermittlung des Test-

[3] Die generische Hardware-Beschreibung ermöglicht es, den VLIW-Prozessor in diversen Konfigurationen zu synthetisieren, also die Größe der Speicher und Verarbeitungseinheiten und Anzahl der Slots zu variieren.

Testschaltung		SFFs / Gatter	Scan-Ketten n_{Chain}	Scan-Tiefe m_{Chain}	Testmuster n_{Pat}	Fehlerüberdeckung (stuck-at) %	DCB-Anteil %	Unkomprimierter Testsatzes [Byte]	Komprimiertes SC-Progr. [Byte]	Komprimierungs-faktor
ISCAS89	s13207	790 / 7.951	96	9	262	94,10	94,23	28.296	10.960	2,36
			128	7				29.344	10.272	2,52
			256	4				33.536	9.184	2,82
	s15850	684 / 9.772	96	8	155	94,30	87,66	14.880	8.814	1,69
			128	6				14.880	8.266	1,80
			256	3				14.880	7.464	1,99
	s38417	1.742 / 22.179	180	10	165	95,23	85,88	37.125	25.218	1,47
			192	10				39.600	25.096	1,58
			256	7				36.960	24.346	1,52
ITC99	b15s	485 / 8.338	64	9	452	95,53	91,75	32.544	15.860	2,05
			128	5				36.160	14.222	2,54
			256	3				43.392	14.118	3,07
	b17	1.452 / 22.645	128	12	562	95,39	92,50	107.904	39.668	2,72
			256	6				107.904	38.260	2,82
			512	3				107.904	40.696	2,65
	b18	2.797 / 42.015	128	22	530	95,38	89,51	186.560	92.732	2,01
			256	11				186.560	90.558	2,06
			512	6				203.520	100.168	2,03
	b19	5.571 / 82.109	128	44	538	95,32	89,54	378.752	185.962	2,04
			256	22				378.752	180.996	2,09
			512	11				378.752	197.910	1,91
			1.024	6				413.184	220.678	1,87
VLIW16S4		2.904 / 34.973	128	23	455	96,21	92,28	167.440	64.616	2,60
			256	12				174.720	61.842	2,83
			320	10				182.000	68.803	2,65
			512	6				174.720	65.007	2,69
VLIW32S4		3.303 / 81.862	128	26	716	97,84	90,36	297.856	135.506	2,20
			256	13				297.856	130.248	2,29
			512	7				320.768	139.386	2,30
			1.024	4				366.592	149.870	2,45
VLIW32S8		4.419 / 172.408	128	35	790	98,56	86,06	442.400	282.866	1,56
			256	18				455.040	267.900	1,70
			512	9				455.040	286.342	1,59
			1.024	5				505.600	310.603	1,63

Tabelle 5.2: Komprimierte Testsätze für verschiedene Schaltungen und Scan-Strukturen

mustersatzes und nach der Relaxation also bereits eine Komprimierung statt, die nicht in diesen hier dargestellten Komprimierungsfaktor eingeht.

Des Weiteren handelt es sich bei dem komprimierten Testsatz um ein Programm, das voll deterministische Muster erzeugt. Dies ist zu berücksichtigen, um die nur geringe Komprimierung um den Faktor 2 nachzuvollziehen. Eine höhere Komprimierung wird verhindert durch die Vielzahl an zusätzlichen Bitflip-Operationen pro Testvektor, die eine bitgenaue Anpassung an deterministische Vorgaben realisieren. Dafür wird die höchstmögliche Fehlerüberdeckung, die hier durch FastScan ermittelt wurde, mit relativ geringem Datenaufwand erreicht. Eine weitaus höhere Komprimierung kann durch Pseudozufallsmuster erzielt werden, da auf deterministische Modifizierungen verzichtet wird. Es ist aber für eine annähernd hohe Fehlerüberdeckung auch eine Vielzahl an Testmustern und somit eine längere Testzeit nötig.

Für einen strukurorientierten Test mit hoher diagnostischer Auflösung sollte die höchstmögliche Fehlerüberdeckung angestrebt werden. Daher werden im Folgenden auch nur Testsätze beziehungsweise Testprogramme für den deterministischen Test betrachtet.

In Abbildung 5.1 ist exemplarisch an VLIW-Schaltungen die Anzahl der Scan-Ketten dem komprimierten Testsatz gegenübergestellt. Es ist zu sehen, dass in allen Fällen ein minimaler Datenumfang für den komprimierten Testsatz bei Versionen mit 256 Scan-Ketten erreicht wird. Der Scan-Controller ist hier mit einem Eingangsport von 16 Bit ausgelegt, um das LFSR in möglichst wenigen Takten mit dem Feedback und dem Seed laden oder zwei Adressen parallel anlegen zu können.

Da ein Testvektor eines Testmusters viele DCBs enthält, kann dieser aus einem kompakten LFSR auf die Anzahl der Scan-Ketten erweitert werden, so dass nur wenige Modifizierungen notwendig sind. Ein Testmuster kann bei einer höheren Anzahl an Scan-Ketten auf weniger Testvektoren höherer Datenbreite aufgeteilt werden. Da sich im Scan-Controller-Programm die Datenbreite des Testvektors nur indirekt als Anzahl zusätzliche Bitflip-Adressen bemerkbar macht, ist somit eine höhere Komprimierung möglich. Die Anzahl der nötigen Bitflips eines Testmusters sollte die Anzahl der Bitflips des entsprechenden Testmusters mit kürzeren Testvektoren nicht übersteigen. Dies ist in den aufgelisteten Schaltungen auch nicht der Fall.

Das der Datenumfang in den hier betrachteten komprimierten Testdaten bei Schaltungsversionen mit mehr als 256 Scan-Ketten dennoch wieder steigt, liegt vor allem daran, dass die erforderliche Datenbreite für die Bitflip-Adressen entsprechend zunimmt. Um einen Testvektor für n_{Chain} Scan-Ketten zu adressieren, sind $\lceil \text{ld}(n_{\text{Chain}}) \rceil$ Bit vonnöten. Da jeweils zwei Adressen für die parallel auszuführenden Bitflip-Operationen geladen werden, erfordert ein Testvektor für bis zu 512 Scan-Ketten ein Eingabewort von 18 Bit. Für die Schaltungsversionen mit 1024 Scan-Ketten müssen 20 Bit für das Eingabewort vorgesehen werden. Somit nimmt der Datenumfang der komprimierten Testdaten entsprechend zu.

Ferner kann sich eine ungünstige Wahl der Ketten-Anzahl negativ auf den Testsatzumfang auswirken. Dies ist hier am Beispiel des VLIW16S4 mit 320 Scan-Ketten dargestellt (Abbildung 5.1). Nur wenige dieser Ketten haben eine Länge von 10 Scan-Flipflops. Für

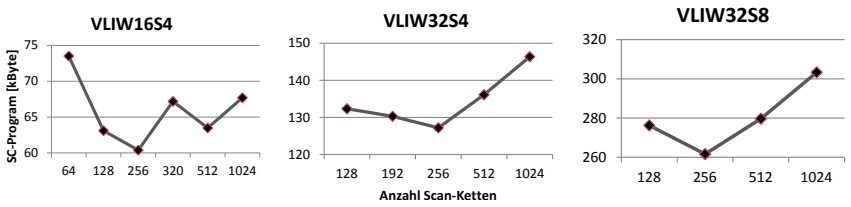


Abbildung 5.1: Testsatz in Abhängigkeit zur Scan-Kettenanzahl

die Mehrzahl der Ketten mit einer Länge von 9 Scan-Flipflops müssen die Testmuster aufgefüllt werden. Die Scan-Struktur ist mit einem Testmuster komplett initialisiert, nachdem es in 10 Scan-Takten hineingeschoben wurde. Dabei wurden die Füllbits mit dem letzten Scan-Takt wieder aus den 9er-Ketten herausgeschoben, haben also im anschließenden Capture-Takt keinen Einfluss auf die zu testende Logik. Bei einer Vielzahl dieser Füllbits und einer hohen Anzahl an Testmustern kann der Mehraufwand mehrere Kilobyte betragen. Zudem trägt auch in diesem Beispiel die Anpassung der Datenbreite des Scan-Controller-Eingangsports an die Bitflip-Adressen zum hohen Umfang des Testsatzes bei. Für 320 Scan-Ketten wird ein 18-Bit Eingabewort benötigt, um zwei Adressen parallel zu laden.

In Abbildung 5.2 ist der Komprimierungsfaktor für diese VLIW-Schaltungen dargestellt. Hier ist zu sehen, dass die Komprimierung hauptsächlich vom Anteil unbestimmter Bits in den Testmustern abhängt. So kann für die Schaltung VLIW16S4 mit einem DCB-Anteil von 92,28% eine nahezu 3-fache Komprimierung erreicht, für VLIW32S8 mit 86,08% DCBs der Testsatz aber nur um etwa 40% reduziert werden.

Für die Schaltungen VLIW16S4, VLIW32S8 und auch den meisten Schaltungen der Tabelle 5.2, wird die höchste Komprimierung in der Version mit 256 Scan-Ketten erreicht. Die Schaltung VLIW32S4 hat ihre maximale Komprimierung bei 1024 Scan-Ketten. Dies liegt aber daran, dass das schlechte Verhältnis von Kettenanzahl zu Scan-Tiefe für einen hohen unkomprimierten Testsatz sorgt. Es ist jeweils ein komplettes Testmuster in die Scan-Ketten über die volle Scan-Tiefe zu schieben, selbst wenn nur ein paar wenige Ketten dieser Länge entsprechen. Im Allgemeinen ist bei Scan-Strukturen mit vielen Ketten und geringer Tiefe die Wahrscheinlichkeit eines Mehraufwands durch Füllbits höher. Dies zeigt sich in der Größe des unkomprimierten Testsatzes in Tabelle 5.2. Je umfangreicher dieser Testsatz gegenüber dem minimalen Umfang g_{Min} ist, mit $g_{\text{Min}} = {}^{n_{\text{Pat}} * n_{\text{SFF}}}/_{8}$ Byte, desto mehr solcher Füllbits sind enthalten.

5.2 Auswertung von Testzeiten

Die benötigte Zeit eines Testlaufs hängt vom Umfang des Testsatzes, der mit der Komplexität der zugrundeliegenden Schaltung und der zu erreichenden Fehlerabdeckung

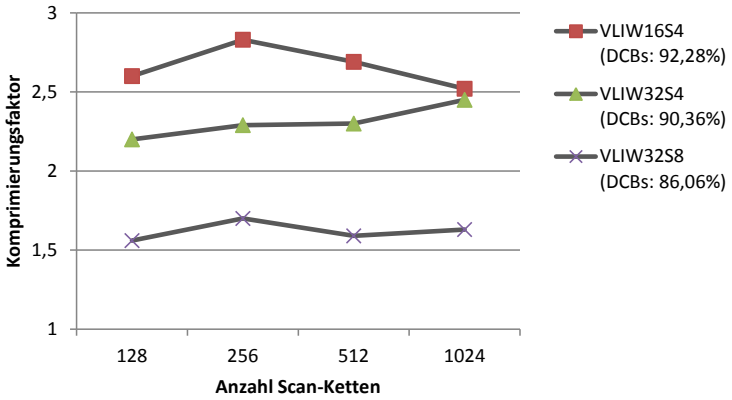


Abbildung 5.2: Komprimierungsfaktor in Abhängigkeit zur Scan-Kettenanzahl

korreliert, ab. Ein Testsatz umfasst eine Vielzahl an Testmustern, um je nach Fehlerdichte hinreichend viele Schaltungszustände einzustellen. Das Testmuster, die Matrix über alle Scan-Flipflops, ergibt sich aus der Anzahl an Scan-Ketten n_{Chain} mal der Scan-Tiefe m_{Chain}. Für einen scan-basierten Test kann das Datenvolumen des unkomprimierten Testsatzes näherungsweise, bei in etwa ausgeglichener Länge der Scan-Ketten, als Anzahl der Scan-Zellen mal Anzahl der Testmuster n_{Pat} angenommen werden. Die Testzeit wird hauptsächlich durch das Hineinschieben der Testmuster bestimmt. Werden die Scan-Ketten parallel geladen, ist die Testzeit also von der Anzahl der Testvektoren des Testsatzes und von der Scan-Frequenz f_{SC} abhängig:

$$t_{\text{Test}} \approx \frac{m_{\text{Chain}} * n_{\text{Pat}}}{f_{\text{SC}}}.$$

Als Beispiel für eine zu testende Schaltung sei hier eine Scan-Architektur mit 1000 Scan-Ketten, einer Scan-Tiefe von 100 und einem Scan-Takt von 20 MHz herangezogen. Diese Schaltung erfordert ein Testmuster von 12,21 KByte. Sei zunächst der Fall ohne Komprimierungsverfahren, also das direkte Füllen der parallelen Scan-Ketten, betrachtet. Dann ergibt sich für beispielsweise 2000 Testmuster eine Testzeit von 10 ms mit einem Gesamtdatenvolumen des reinen Testsatzes von 23,84 MByte. Soll über eine serielle Schnittstelle der reine Testsatz übertragen und dieser direkt mit Scan-Takt in die Scan-Ketten eingeschoben werden (Bypass um Dekompressor), stößt die Umsetzung schnell an die Grenzen der realisierbaren Datenübertragung. Das Füllen von 1000 parallelen Scan-Ketten bei einem 20 MHz-Takt würde eine Datenrate von 20 GBit/s erfordern. Um unnötig hohe Testlaufzeiten und hohe Datenpufferung zu vermeiden, ist eine geeignete Komprimierung erforderlich. Es werden daher lediglich die Steuerinformationen an den integrierten Test-Controller (SC, ELC) übertragen und so die nötigen Testdaten generiert. Neben der Komprimierung des gesamten Testsatzes ist für die seriellen Schnittstellen die Komprimierung des Eingabedatums pro Zeitschritt entscheidend, um die erforderliche Datenrate zu senken.

Der Scan-Controller liest in jedem Takt ein Datum des komprimierten Testsatzes ein. Als komprimierter Testsatz ist der komplette Datensatz an Steuerinformationen für den Scan-Controller zu verstehen. Das heißt, pro zu erzeugenden Testvektor ist mindestens ein Takt des Scan-Controllers erforderlich, im Falle von zu verarbeiteten Bitflip- und/oder Bitfix-Adressen entsprechend mehr. Ausgehend von einer Konfiguration des Scan-Controllers mit einer Taktfrequenz f_{SC} von 20 MHz und einem Eingangsport der Datenbreite i_{SC} von 16 Bit muss alle 50 ns ein neues Datenwort zu je 16 Bit anliegen. Das entspricht einer Datenrate von 320 MBit/s. Bei einem Eingang von 20 Bit, was für zwei parallele Adressen zur Bitmodifikation eines bis zu 1024 Bit breiten Testvektors erforderlich ist, steigt die Datenrate auf 400 MBit/s. Diese ist durch ein USB HighSpeed-Schnittselle, mit einer Übertragungsrate der reinen Payload-Daten von bis zu 425,984 MBit/s, zu bewerkstelligen [USB00]. In solch einem Fall kann die Testzeit für einen Testsatz g_{Test} (in Bit) näherungsweise angegeben werden als:

$$t_{\mathrm{Test}} \approx \frac{g_{\mathrm{Test}}}{i_{\mathrm{SC}} * f_{\mathrm{SC}}}.$$

Bei einem Scan-Test im Feld stellen die bezüglich Geschwindigkeit etwas schwächeren seriellen Schnittstellen, wie USB-FullSpeed, FlexRay oder CAN, den Engpass dar, sodass die Testzeit durch die erreichbare Datenrate bestimmt wird. In diesem Fall wird der Test zyklisch jeweils mit den bereits verfügbaren Testeingaben, parallel zur langsameren Übertragung des Testsatzes, durchgeführt. Das heißt, die Testzeit hängt von der effektiven Datenrate r_{Eff}, mit der die reinen Nutzdaten übertragen werden können, ab:

$$t_{\mathrm{Test}} \approx \frac{g_{\mathrm{Test}}}{r_{\mathrm{Eff}}}.$$

Da der Testzugang prototypisch auf FPGA-Basis umgesetzt wurde, steht die USB-Schnittstelle nur in der FullSpeed-Verion zur Verfügung. Das heißt, für die Payload-Daten ist die Bandbreite auf 9,728 MBit/s begrenzt [USB00]. Tatsächlich werden im Mittel etwas über 8 MBit/s mit der implementierten USB-Schnittstelle erreicht. Die Testdaten werden paketweise übertragen und jeweils im Empfangspuffer zwischengespeichert. Ein Datenpaket kann vom Test-Controller mit Scan-Takt verarbeitet werden, um Testmuster zu erzeugen und diese in die Scan-Ketten hineinzuschieben. Sind keine Daten verfügbar, befindet sich der Test-Controller im Wartezustand. Die Kommunikation über Pakete erfordert regelmäßige Statusabfragen, um beispielsweise verfügbaren Speicherplatz in den Puffern sicherzustellen. Dieser Kommunikationsaufwand wirkt sich zusätzlich auf die Testzeit aus. Bei einer USB HighSpeed-Verbindung muss aber ebenso solch ein Verfahren angewandt werden, da nicht sichergestellt werden kann, dass stets mindestens ein neues Datenwort synchron zum Takt des Test-Controllers anliegt. Das liegt daran, dass für die Datenübertragung im Bulk-Transfer keine Bandbreite reserviert ist.

Für die FlexRay-Schnittstelle hängt die Datenrate von der individuellen Konfiguration des FlexRay-Netzes ab. Angenommen der Kommunikationszyklus beträgt 5 ms und einem Netzwerkknoten ist lediglich ein statischer Slot zugeordnet, so steht dieser Slot 200-mal pro Sekunde zur Verfügung. Bei einem Nutzdatenumfang eines Frames von 64 Byte kann eine effektive Datenrate von 100 KBit/s pro Knoten erreicht werden. Können mehrere statische Slots oder zusätzlich dynamische Slots genutzt werden, vervielfacht sich die effektive Datenrate entsprechend. Bei einer Netzkonfiguration mit kürzeren

Zykluszeiten, größeren statischen Slots und/oder mehreren Slots pro Knoten kann eine weitaus höhere Datenrate erzielt werden. Da der Testzugang über ein bestehendes FlexRay-Netz erfolgen soll, ist die maximale Datenrate und somit die Testlaufzeit durch dieses bestimmt.

5.2.1 Versuchsaufbau

Um Messungen der Testzeiten für verschiedene Schaltungen und Testkonfigurationen vorzunehmen, wurde jeweils die zu testende Schaltung in entsprechender Auslegung der Scan-Struktur und die USIF-Architektur, inklusive der Testarchitektur und der Kommunikations-Controller, synthetisiert und auf dem FPGA des Entwicklungsboards *DE2-70* implementiert (siehe Anhang C). Es wurde zusätzliche eine Aufsteckplatine entworfen und erstellt, die die Entwicklungsumgebung um die nötigen Transceiver und Steckverbindungen für die umzusetzenden Schnittstellen erweitert. Der Testablauf wird komplett durch die Applikation auf dem PC gesteuert. Die Kommunikation zwischen PC und USIF findet dabei über die USB-Verbindung statt.

Den anschließenden Betrachtungen lag der folgende Versuchsaufbau zugrunde:

- Entwicklungsboard DE2-70
- Altera Cyclone II FPGA (EP2C70F896C6N)
- Aufsteckplatine für Schnittstellen
  - USB Transceiver Texas Instruments TUSB1106,
    USB 2.0-Buchse Typ Mini-B
  - FlexRay Transceiver NXP Semiconductors TJA1080A , D-Sub DE9-Buchse
  - Oszillator Crystek Crystals CPRO33, 80 MHz (für FlexRay-Controller)
- Applikation betrieben auf
  - PC Intel Core i7-2600 CPU 3,40 GHz (Arbeitsspeicher 8 GB),
  - OS Windows 7 (64 Bit),
- Testzugang: USB FullSpeed, Bulk-Transfer.

5.2.2 Auswertung der Analysemodi

In Tabelle 5.3 wird der Datenumfang und die Testzeiten der verschiedenen Analysemodi gegenübergestellt. Hierzu wird ein VLIW-Prozessor, konfiguriert als 32-Bit-Architektur mit 8 Slots, in einer Version mit 256 Scan-Ketten betrachtet.

Der Test wurde hier folgendermaßen konfiguriert:

- Fehlermodell: *Stuck-at*,
- Scan-Modus: *Decompress*,

- Scan-Clock: $f_{\mathrm{SC}} = 20$ MHz,
- Scan-Ketten: $n_{\mathrm{Chain}} = 256$,
- Scan-Tiefe: $m_{\mathrm{Chain}} = 18$,
- Analyse-Rate/-Basis: $(a_{\mathrm{Rate}}, a_{\mathrm{Base}}) = (18, 17)$,
- Kompaktierung: *Mixed* (MISR + XOR-Tree),
- Paketgröße: $g_{\mathrm{Packet}} = 512$ Byte,
- Testmusteranzahl: 790 (FC 98,56%),
- Fehler in Schaltung injiziert (*Stuck-at-0*), der insgesamt 790 verfälschte Testantworten[4] zu Folge hat.

In den Spalten zwei bis fünf der Tabelle 5.3 sind die jeweiligen zu sendenden beziehungsweise zu empfangenden Daten in Byte zu sehen. Pro Paket fallen zusätzlich Daten für ein Instruktionswort, als Header in zu sendenden Testdaten oder als Anfrage für zu empfangene Testantworten, an. Bei einer Datenbreite der USIF-Architektur von 32 Bit umfasst eine Instruktion also 4 Byte. Der Mehraufwand für die USIF-Instruktionen ist abhängig von der Anzahl der Pakete, die für den Satz an Testdaten aufgebracht werden müssen. Somit ergibt sich für einen Testdatenumfang von g_{Test} Byte ein Datenumfang für Instruktionswörter von $g_{\mathrm{Instr}} = 4 \lceil g_{\mathrm{Test}} / g_{\mathrm{Packet}} \rceil$ Byte. Dieser Wert ist hier zu jedem Datensatz zusätzlich in Klammern angegeben.

Analyse-modus	**TestIn [Byte]**	**TestRef [Byte]**	**TestOut [Byte]**	**Time-stamps [Byte]**	**Gesamt-daten [Byte]**	**Testzeit [s]**
LogAll	267.900 (2.096)	-	1.580 (16)	-	271.592	2,724
LogFaults	267.900 (2.096)	1.580 (16)	1.580 (16)	3160 (28)	276.376	2,838
PassFail	512 (4)	512 (4)	4 (4)	8 (4)	1.052	0,128

Tabelle 5.3: Testaufwand in Abhängigkeit des Analysemodus am Beispiel des VLIW32S8

Die vorletzte Spalte gibt den Gesamtumfang der Daten wieder, die während des Testlaufs zu übertragen sind. Es sind also die Werte aus den Spalten zwei bis fünf, inklusive

[4] Da die Testantworten auf Signaturen eines MISRs beruhen und dieses während des Testlaufs nicht zurückgesetzt wird, bleibt der Fehler in sämtliche Folgesignaturen erhalten.

der in Klammern aufgelisteten Instruktionsdaten aufaddiert. Die mittlere Testzeit, die sich aus dem Median über zehn gemessene Zeiten ergibt, ist schließlich in der letzten Spalte angegeben. Im Anhang B.3 sind weitere diesbezüglich untersuchte Schaltungen erfasst.

Im Modus *LogAll* sind die kompletten Datensätze der Testeingaben (261,62 KByte) und Testantworten (1,54 KByte) zu übertragen. Dies ergibt inklusive der USIF-Instruktionswörter etwa 265,23 KByte an Gesamtdaten. In die Gesamtzeit fließt hier außerdem die Aufbereitung der Testergebnisse durch die Applikation mit ein.

Im Modus *LogFaults* ist neben dem kompletten Datensatz der Testeingaben auch der der Referenzwerte (1,54 KByte) zu übermitteln. Der Testantwortsatz umfasst in diesem Beispiel alle 790 Testausgaben (entsprechend der eingestellten Analyserate), da bereits ein Fehler durch die Signatur nach dem ersten Testmuster erkannt wird und sich somit in allen nachfolgenden Signaturen fortsetzt. Der Gesamtdatensatz ist durch die zusätzlich zu übertragenen Referenzdaten und Zeitstempel etwas größer als der des *LogAll*-Modus. Die verhältnismäßig um einiges längere Testzeit ist vor allem auf die zusätzlichen Statusabfragen und Zugriffszyklen durch die Applikation zurückzuführen. Der *LogFaults*-Modus zahlt sich nur bei wenigen zu erwartenden verfälschten Signaturen aus, so dass die Testausgaben durch die on-chip Filterung auf nur wenige Testantworten und zugehöriger Zeitstempel beschränkt bleibt. Dies ist beim EDT mit einer rein kombinatorischen Kompaktierung der Fall (siehe Anhang B.3).

Der *PassFail*-Modus erfordert ebenfalls Referenzen zur Auswertung der Testantworten. Hierzu wird das erste Paket an Referenzdaten übertragen (512 Byte). Danach folgt das erste Paket der Testeingaben. Da in diesem Beispiel aber bereits nach dem ersten Testmuster ein Fehler erkannt wird, stoppt der Scan-Test, und weitere Eingabedaten sind nicht vonnöten. Die Signatur, durch die der Fehler erkannt wurde, wird in das TestOut-FIFO und der zugehörige Zeitstempel in das TS-FIFO geschrieben.

5.2.3 Auswertung von Testeigenschaften

Für die Untersuchung von Testzeiten wurden wieder die ISCAS89-, ITC99-Schaltungen und die VLIW-Prozessoren in den verschiedenen Auslegungen herangezogen. Hierzu wurden die ermittelten komprimierten Testsätze, wie in Abschnitt 5.1.1.5 beschrieben, verwendet. Dementsprechend wurde hierfür der Scan-Controller als Testarchitektur und die zu testenden Schaltung in die USIF-Architektur eingebunden.

Der Ermittlung der Testzeit lag folgende Konfiguration zugrunde:

- Fehlermodell: *Stuck-at*,
- Scan-Modus *Decompress*,
- Scan-Clock $f_{\mathrm{SC}} = 20$ MHz,
- Scan-Controller Eingangsport $i_{\mathrm{SC}} = 16$ Bit,

- Scan-Controller Ausgangsport $o_{\mathrm{SC}} = 16$ Bit,
- Analysemodus *LogAll*,
- Analyse-Rate/-Basis $(a_{\mathrm{Rate}}, a_{\mathrm{Base}}) = (1, 0)$ und $(a_{\mathrm{Rate}}, a_{\mathrm{Base}}) = (m_{\mathrm{Chain}}, m_{\mathrm{Chain}} - 1)$,
- Kompaktierung: *Mixed* (MISR + XOR-Tree),
- Paketgröße $g_{\mathrm{Packet}} = 512$ Byte,

Die Konzeptlösung wurde auch für den Einsatz des EDTs als Testarchitektur synthetisiert und verifiziert. Hierzu wurde von Infineon Technologies im Rahmen des BMBF-Verbundprojektes DIANA ein EDT-Demonstrator auf Basis des 32-bit Mikrocontrollers *Infineon TriCore 1797* zur Verfügung gestellt. Bei dieser EDT-Logikbeschreibung handelt es sich um ein Skelett-Design, bei dem die durch den EDT zu testende Schaltung nur aus den Scan-Ketten besteht. Für diesen Demonstrator wurde ein Chain-Test im Umfang von 140 Testmustern (stuck-at) durchgeführt. Um auch einen etwas größeren Test zu simulieren, wurde der vorhandene Testsatz des Chain-Tests, durch zehnfache Wiederholung der Testmuster, erweitert. Somit umfasst dieser Pseudotestsatz 1400 Testmuster.

Weitere Parameter des EDT-Demonstrators sind:

- Anzahl Scan-Ketten $n_{\mathrm{Chain}} := 900$,
- Scan-Tiefe $m_{\mathrm{Chain}} := 103$,
- EDT-Eingangskanäle $i_{\mathrm{SC}} := 16$,
- EDT-Ausgangskanäle $o_{\mathrm{SC}} := 16$.

In der Tabelle 5.4 ist eine Auswahl der untersuchten Schaltungen zu sehen. Eine umfangreiche Auflistung ist in Anhang B.4 zu finden. Die Werte zur Scan-Struktur, zur Testmusteranzahl, zur Fehlerüberdeckung und zum Testsatzumfang sind der Tabelle 5.2 entnommen. In der Spalte 7 ist die nötige Testzeit für einen Selbsttest, dem der komprimierte Testsatz für einen deterministischen Test on-chip zur Verfügung steht (Spalte 6), dargestellt. In den weiteren Spalten sind die mittels des beschriebenen Versuchsaufbaus eruierten Testzeiten zu sehen.

Für den Selbsttest kann in jedem Scan-Takt ein Datum aus dem komprimierten Testsatz entnommen und dieses an den Eingangsport des Scan-Controllers angelegt werden. Über den kompletten Testlauf verarbeitet der Scan-Controller in jedem Takt ein Eingabedatum, sowohl in der Scan- also auch in der Capture-Phase. In der Capture-Phase findet bereits die Initialisierung der LFSR-Sequenz für das darauffolgende Testmuster statt. Somit ist die Testzeit t_{BIST} nur von der Anzahl der Eingaben und der Taktfrequenz des Scan-Controllers f_{SC} abhängig. Für ein Testsatzumfang von g_{Test} Byte und einem Eingangsport der Datenbreite i_{SC} gilt demnach:

$$t_{\mathrm{BIST}} = \frac{8 g_{\mathrm{Test}}}{i_{\mathrm{SC}}} * \frac{1}{f_{\mathrm{SC}}}$$
$$\Rightarrow t_{\mathrm{BIST}} = \frac{n_{\mathrm{Input}}}{f_{\mathrm{SC}}},$$

wobei $n_{\mathrm{Input}} = {}^{8 g_{\mathrm{Test}}}/_{i_{\mathrm{SC}}}$ die Anzahl der Eingangsdatenworte des Testsatzes ist.

Die Werte für die Testzeit eines über die Standardschnittstellen durchgeführten Scan-Tests wurden durch die Applikation, über die der kompletten Testlauf organisiert wird, ermittelt. Ein Testlauf umfasst hier das Einlesen der Testdaten aus einer spezifizierten Pattern-Datei, die in Algorithmus 4.4 beschriebene Scan-Funktion und das Aufbereiten und Speichern der Testergebnisse. Da die gemessenen Zeiten mehrerer Testläufe desselben Versuchsaufbaus etwas schwankten und teilweise auch relativ hohe Werte annahmen, wurde ein Mittelwert gewählt, der robust gegen Extremwerte ist. Hierzu wurde der Median aus zehn Testläufen ermittelt.

Die Testzeit t_{Test} lässt sich folgendermaßen ausdrücken:

$$t_{\mathrm{Test}} \approx \frac{g_{\mathrm{Test}} + g_{\mathrm{Res}}}{r_{\mathrm{Eff}}} + t_{\mathrm{Init}} + t_{\mathrm{Req}} n_{\mathrm{Packet}} + t_{\mathrm{Res}}$$

Sämtliche Testdaten, deren Umfang sich sowohl aus dem der Testeingaben g_{Test} als auch aus dem der Testantworten g_{Res} ergibt, werden mit der effektiven Datenrate r_{Eff} übertragen. Hinzu kommen die Zeiten, die für die Initialisierung des Tests t_{Init} und für die Auswertung der Testantworten t_{Res} benötigt werden. Außerdem fällt eine bestimmte Zeit für die Statusanfragen t_{Req} pro Datenpaket an. Wobei sich die Anzahl der Pakete n_{Packet} aus dem Umfang des Testsatzes g_{Test}, dem Umfang des Testantwortsatzes g_{Res} und der spezifizierten Paketgröße g_{Packet} ergibt, also

$$n_{\mathrm{Packet}} = \left\lceil \frac{g_{\mathrm{Test}}}{g_{\mathrm{Packet}}} \right\rceil + \left\lceil \frac{g_{\mathrm{Res}}}{g_{\mathrm{Packet}}} \right\rceil.$$

Die Initialisierungsphase der Applikation besteht neben der Konfiguration des USIF-Gerätes hauptsächlich aus der Extraktion der Testdaten aus der einzulesenden Pattern-Datei. Die hierfür aufzuwendende Zeit ist also vor allem von dem Datenumfang des Testsatzes abhängig, fällt aber gegenüber der Sendezeit eines umfangreichen Testsatzes relativ gering aus. Da der Analysemodus *LogAll* gewählt und kein Referenzdatensatz spezifiziert wird, fällt auch keine umfangreiche Auswertung der Testergebnisse durch die Applikation an. Somit kann sowohl die Initialisierungszeit als auch die Analysezeit hier vernachlässigt werden. Die Gesamttestzeit wird also auf die Übertragung der Testdaten und der dazu nötigen Kommunikation (Status- u. Request-Botschaften) beschränkt:

$$t_{\mathrm{Test}} \approx \frac{g_{\mathrm{Test}} + g_{\mathrm{Res}}}{r_{\mathrm{Eff}}} + t_{\mathrm{Req}} n_{\mathrm{Packet}}.$$

Sei in Tabelle 5.4 zunächst der Zugang über USB betrachtet, mit den Testzeiten aus Spalte 8 und 9. Es ist zu erkennen, dass die Testzeit lediglich von der Größe des Testsatzes und der gewählten Analyserate a_{Rate} abhängt. Der eingestellte Scan-Takt und die Scan-Tiefe, die die Zeit für die Shift-Phase des Scan-Tests bestimmt, wirkt sich nicht auf die Gesamtzeit aus, da zum einen der Test parallel zur Übertragung weiterer Testdaten geschieht. Zum anderen ist die on-chip Verarbeitungsgeschwindigkeit um ein Vielfaches höher als die Übertragungsgeschwindigkeit, sodass die Applikation nicht auf das Hineinschieben der Testmuster in die Teststrukturen warten muss. Da aber vor der Übertragung der Datenpakete sichergestellt werden soll, dass genügend freier

Testschaltung		Scan-Ketten n_{Chain}	Scan-Tiefe m_{Chain}	Testmuster n_{Pat}	Fehlerüberdeckung (stuck-at) %	Testsatz [KByte]	Zeitaufwand Selbsttest [ms]	Testzeit [s]			
								USB		FlexRay[5]	
								$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$	$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$
ISCAS89	s13207	128	7	262	94,10	10,03	0,256	0,138	0,104	0,587	0,475
		256	4			8,97	0,230	0,119	0,089	0,507	0,396
	s15850	64	11	155	94,30	9,14	0,234	0,126	0,093	-	-
		128	6			8,07	0,207	0,102	0,084	-	-
	s35932	128	16	36	87,86	10,65	0,273	0,144	0,127	-	-
		256	8			10,02	0,256	0,131	0,119	-	-
ITC99	b18	256	11	530	95,38	88,44	2,264	1,049	0,906	4,329	3,592
		512	6			97,82	2,504	1,087	1,029	-	-
	b19	256	22	538	95,32	176,75	4,525	1,994	1,774	9,599	8,543
		512	11			193,27	4,948	2,171	2,059	8,715	8,152
VLIW16S4		128	23	454	96,21	63,10	1,615	0,785	0,561	-	-
		256	12			60,39	1,546	0,665	0,541	3,142	2,509
		512	6			63,48	1,625	0,673	0,620	-	-
VLIW32S4		256	13	716	97,84	127,20	3,256	1,467	1,300	5,992	5,313
		512	7			136,12	3,485	1,518	1,435	-	-
		1024	4			146,36	3,747	1,631	1,545	-	-
VLIW32S8		256	18	790	98,56	261,62	6,698	3,050	2,724	12,159	11,067
		512	9			279,63	7,159	3,098	2,949	13,327	12,863
		1024	5			303,32	7,765	3,251	3,164	-	-
EDT-Demon-strator		900	103	140	~100	33,64	0,861	0,661	-	2,734	-
				1400	-	336,33	8,610	6,524	-	28,223	-

Tabelle 5.4: Testaufwand, Fehlerüberdeckung und Testzeit

Speicherplatz im Empfangspuffer des USB-Controllers beziehungsweise auszulesender Speicher im TestOut-FIFO vorhanden ist, erfolgt eine regelmäßige Abfrage des USB-Statusendpunktes. Das bedeutet, es entstehen Wartezyklen pro Datenpaket, die sich negativ auf die Testzeit auswirken.

Da die Übertragung paketweise geschieht und für jedes Paket ein nicht unerheblicher administrativer Aufwand durch die Applikation hinzukommt, ist es hier günstig, die Testzeit über die Zeit pro Paket auszudrücken. Die Paketanzahl n_{Packet} ergibt sich, wie zuvor erwähnt, aus dem Umfang des zu sendenden Testsatzes g_{Test} und des zu empfangenen Testantwortsatzes g_{Res}. Der Umfang der Testantwortdaten ist abhängig von dem gewählten Analysemodus und der Analyserate a_{Rate}. Es sei hier nur der Analy-

[5] Hier wurde nur eine Auswahl an Schaltungsexemplaren hinsichtlich Testlaufzeit untersucht. Von denen wurden die bezüglich des Testdatenumfangs günstigen Versionen umgesetzt.

semodus *LogAll* betrachtet. Die kompaktierten Testantworten haben eine Datenbreite entsprechend des Ausgangsports der Testarchitektur, der hier $k = {}^{o_{SC}}/_{8}$ Byte beträgt. Die Anzahl der Testantworten kann über die Anzahl der Testmuster n_{Pat} und Testantworten pro Testmuster ${}^{m_{\text{Chain}}}/_{a_{\text{Rate}}}$ ausgedrückt werden. Der Datensatzumfang für die Testantworten ist demnach:

$$g_{\text{Res}} = \frac{k * m_{\text{Chain}} * n_{\text{Pat}}}{a_{\text{Rate}}}.$$

Daraus ergibt sich eine Paketanzahl n_{Packet} von:

$$n_{\text{Packet}} = \left\lceil \frac{g_{\text{Test}}}{g_{\text{Packet}}} \right\rceil + \left\lceil \frac{k * m_{\text{Chain}} * n_{\text{Pat}}}{g_{\text{Packet}} * a_{\text{Rate}}} \right\rceil.$$

Bei einer Analyserate $a_{\text{Rate}} = 1$ werden im *LogAll*-Modus $\lceil {}^{k * m_{\text{Chain}} * n_{\text{Pat}}}/_{g_{\text{Packet}}} \rceil$ Pakete für die Testausgaben benötigt. Im Falle einer Analyserate $a_{\text{Rate}} = m_{\text{Chain}}$, also nur eine Testausgabe pro Testmuster, ist die Anzahl der Testausgabepakete nur $\lceil {}^{k * n_{\text{Pat}}}/_{g_{\text{Packet}}} \rceil$. Ist ausschließlich am Ende des Testlaufs eine MISR-Signatur der Datenbreite d_{MISR} auszulesen, reduzieren sich die Testausgaben auf $\lceil {}^{d_{\text{MISR}}}/_{8} \rceil$ Byte und können in einem einzigen Paket übermitteln werden (falls nicht $d_{\text{MISR}} > 8 g_{\text{Packet}}$). Dies würde sich in einer Analyserate $a_{\text{Rate}} \geq {}^{k * m_{\text{Chain}} * n_{\text{Pat}}}/_{g_{\text{Packet}}}$ ausdrücken lassen.

Die Gesamtzeit des Testlaufs ergibt sich aus der Zeit, die für ein Paket benötigt wird, und der Anzahl aller zu sendenden und zu empfangenden Pakete:

$$t_{\text{Test}} = t_{\text{Packet}} * n_{\text{Packet}},$$

$$t_{\text{Test}} = t_{\text{Packet}} * \left(\left\lceil \frac{g_{\text{Test}}}{g_{\text{Packet}}} \right\rceil + \left\lceil \frac{k * m_{\text{Chain}} * n_{\text{Pat}}}{g_{\text{Packet}} * a_{\text{Rate}}} \right\rceil \right).$$

Da über die Applikation die Gesamtzeit eines Testlaufs ermittelt wurde, kann auf die Zeit pro Paket geschlossen werden. Es folgt somit:

$$t_{\text{Packet}} = \frac{t_{\text{Test}}}{\left\lceil \frac{g_{\text{Test}}}{g_{\text{Packet}}} \right\rceil + \left\lceil \frac{k * m_{\text{Chain}} * n_{\text{Pat}}}{g_{\text{Packet}} * a_{\text{Rate}}} \right\rceil}.$$

Die Zeiten für die Initialisierung und die Auswertung des Scan-Tests fließen hier in die Paketzeit mit ein. Diese Darstellung der Paketzeit korreliert mit den über die Applikation gemessenen Werten (siehe Tabelle 5.4). Für die unterschiedlichen Testschaltungen ergeben sich bei gleicher Analyserate a_{Rate} in etwa gleiche Zeiten für ein Datenpaket.

Die Paketzeit beträgt hier im Mittel um die 5 ms. Daran ist zu sehen, dass ein erheblicher Zeitaufwand durch die Applikation verursacht wird. Über den Bus wird annähernd die volle Datenrate erreicht. Für die USB-Verbindung wurde eine Datenrate für die Nutzdaten von etwa 8 MBit/s ermittelt. Das heißt, ein Paket von 512 Byte kann in 0,512 ms übertragen werden. Die fast zehnfache Zeit für ein Paket ist bedingt durch die Statusabfragen, der Aufbereitung und Speicherung der Testantworten und Ausgaben an die Benutzerschnittstelle.

Die Applikation wurde nicht auf optimale Laufzeit, sondern in erster Linie auf garantierten fehlerfreien Datenaustausch ausgelegt. So wird beispielsweise vor und nach jedem Zugriff, innerhalb der Sende- und Empfangsfunktion (siehe Algorithmus 4.4), auf eine aktualisierte Statusabfrage, die durch den USB-Host mittels Polling im Intervall von 1

ms erfolgt, gewartet. Es wird vorher sichergestellt, dass ein Zugriff erfolgen darf und danach geprüft, ob der Zugriff erfolgreich war.

Versuchsaufbau für den Testzugang über FlexRay

Um den Testzugang über eine FlexRay-Verbindung zu ermöglichen, wurde die USIF-Architektur eines weiteren Entwicklungsboards als Gateway ausgelegt. Die Architektur wurde dahingehend konfiguriert, die Weiterleitung der Testdaten in beide Transferrichtungen umzusetzen. Hierzu werden jeweils die Daten im Empfangspuffer des einen Kommunikations-Controllers in den Sendepuffer des anderen geschrieben. Zudem werden die Statusinformationen aus den empfangenen FlexRay-Frames ausgelesen und an den Statusendpunkt des USB-Controllers weitergereicht. Mittels dieses Gateways wird also die Verbindung zwischen der USB-Schnittstelle des PCs und der FlexRay-Schnittstelle des zu testenden Gerätes herstellt.

Die Konfiguration des FlexRay-Netzes kann dem Anhang A.5 entnommen werden. Um einen ausgelasteten Bus zu simulieren, wurden jedem Knoten nur zwei statische Slots zugewiesen. In einem statischen Slot stehen hier 42 Byte für Nutzdaten zur Verfügung, von denen zwei Byte für die zu übermittelnden Statusinformationen reserviert sind.

Bei einem Kommunikationszyklus des FlexRay-Netzes von 5 ms und zwei statischen Slots für 80 Byte Nutzdaten ergibt sich eine Datenrate von 128 KBit/s pro FlexRay-Knoten. Da im Versuchsaufbau beide FlexRay-Knoten identisch ausgelegt wurden, ist auch die Datenrate in beiden Richtungen dieselbe. Ein Knoten empfängt jeweils die beiden Frames des anderen. Für den Testdatenaustausch wird somit, beide Transferrichtungen betrachtend, eine Datenrate für reine Nutzdaten von 256 KBit/s erreicht.

In den letzten beiden Spalten der Tabelle 5.4 sind Testzeiten für ausgewählte Schaltungen dargestellt. Die relativ hohe Testzeit liegt hier vor allem an der begrenzten effektiven Datenrate des spezifizierten FlexRay-Netzes, ist aber auch bedingt durch das Kommunikationsverfahren zwischen Applikation und den USB-Endpunkten.

Um beispielsweise Daten aus dem TestOut-FIFO auszulesen, wird zuerst der hierfür notwendige Steuerbefehl über das Gateway und das FlexRay-Netz an das Zielgerät gesendet. Nun kann nicht direkt mit dem Auslesen des USB-Endpunktes des Gateways begonnen werden, da dieser erst einmal die angeforderten Testantworten über das FlexRay-Netz empfangen muss. Es ist hier eine weitere Anfrage von der Applikation nötig, um verfügbare Daten im Endpunkt *EPIn* des USB-Controllers zu registrieren. Hier muss also ein entsprechender Mehraufwand der Testzeit in Kauf genommen werden.

Im Falle einer direkten FlexRay-Verbindung zum Zielgerät begünstigt das TDMA-Verfahren eine regelmäßige Statusmeldung beziehungsweise die Übertragung der Testantwortdaten über die zugeordneten Slots des zu testenden Gerätes. Da die Datenpakete in Slots mit garantierter Latenz übermittelt werden, ist hier kein Polling wie beim USB notwendig. Die nötigen Statusinformationen werden in Frames eines oder mehrerer Slots eingefügt. Dadurch, dass hier zusätzliche Statusanfragen entfallen, kann der admi-

nistrative Aufwand erheblich reduziert werden. Die Testzeit ist demnach hauptsächlich von der effektiven Datenrate der FlexRay-Verbindung abhängig.

5.2.4 Vergleich der USIF-Paketgröße

Die Größe der zu übermittelnden Datenpakete hat einen entscheidenden Einfluss auf die Gesamttestzeit. Dies ist hier exemplarisch an den Testläufen des VLIW16S4 in Abbildung 5.3 und VLIW32S8 in Abbildung 5.4 dargestellt. Hier wurden die Versionen mit 256 Scan-Ketten gewählt. Zudem lag eine Analyse-Rate/-Basis von $(a_{\text{Rate}}, a_{\text{Base}}) = (m_{\text{Chain}}, m_{\text{Chain}} - 1)$ zugrunde. In den Untersuchungen des Abschnitts zuvor war die Paketgröße konstant auf 512 Byte gesetzt. Hier wurden mit demselben Versuchsaufbau über die USB-Verbindung weitere Testläufe für die Paketgrößen 32, 64, 128 und 256 Byte durchgeführt. Die Paketgröße ist hierbei inklusive des Testdaten-Headers für die USIF-Instruktion angegeben.

In den Abbildungen ist zu erkennen, dass eine halbierte Paketgröße in annähernd der doppelten Testzeit resultiert. Diese Beobachtung spiegelt sich auch in der im vorangegangenen Abschnitt dargestellten Gleichung der Paketzeit t_{Packet} wieder.

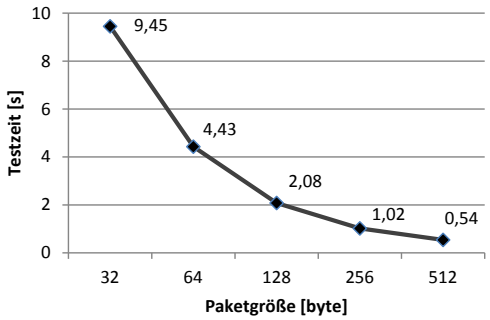


Abbildung 5.3: Testzeiten in Abhängigkeit zur Paketgröße am Beispiel des VLIW16S4

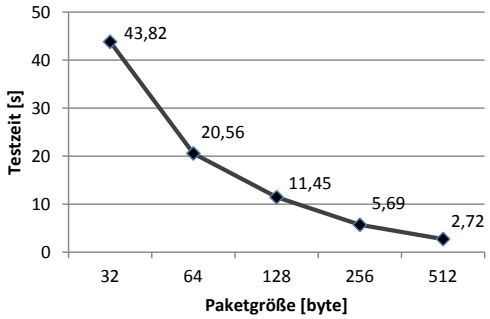


Abbildung 5.4: Testzeiten in Abhängigkeit zur Paketgröße am Beispiel des VLIW32S8

Auch im FlexRay-Netz wird sich die Wahl der Paketgröße negativ auf die Testzeit auswirken, wenn für kleine Datenpakete regelmäßig Sendezeit in den zugeordneten Slots ungenutzt verstreicht. Das heißt, es sollte ein Mindestumfang für Datenpakete spezifiziert werden, der der verfügbaren effektiven Datenrate angepasst ist.

Die Paketzeit ist abhängig von der Auslegung des für die Testdaten verfügbaren Speicherplatzes. Es muss hier also ein Kompromiss zwischen zu spendierenden Speicher on-chip und der Anforderung an zu erreichende Testzeiten gefunden werden. In der industriellen Anwendung bedeutet jede auch nur geringe Hardware-Einsparung in Steuergeräten aufgrund hoher Stückzahlen eine wesentliche Kostenreduzierung. Zudem bedingen die Anwendungsfälle, wie die Diagnose nach einer Fehlermeldung oder die Fehleranalyse durch den Halbleiterhersteller, keine wesentlichen Beschränkungen der Testzeit. Daher wird eine, wenn auch erhebliche, Reduzierung der Testzeit nicht unbedingt einen hohen Speichermehraufwand rechtfertigen können.

5.3 Hardware-Aufwand

Um eine Aussage über den Hardware-Aufwand des entwickelten Designs zu machen, wurden durch Synthese, basierend auf der *Si2 NanGate FreePDK45 Generic Open Cell Library*, die Logikzellen und die dadurch eingenommene Chipfläche der einzelnen Komponenten ermittelt. Die Auslegung der Komponenten entspricht dabei der in den Abschnitten zuvor beschriebenen. In Tabelle 5.5 sind die resultierenden Chipflächen und der relative Aufwand bezüglich einer Beispielschaltung dargestellt. Hierzu wurde der VLIW-Prozessor in einer 32-Bit-Architektur mit 8 Slots als zu testende Schaltung herangezogen.

Komponenten	**Chipfläche in µm²**	**Relativer Aufwand zur Beispiel-CUT**
USIF-Architektur	3.957	2,08 %
USIF-Controller	2.269	1,19 %
Config-Register	658	0,35 %
BIST-Controller	209	0,11 %
Analyzer	791	0,42 %
Scan-Controller[6]	6.487	3,41 %
VLIW32S8	189.993	

Tabelle 5.5: Hardware-Aufwand der USIF-Architektur

[6] Der Scan-Controller wurde hier für 256 Scan-Ketten ausgelegt. Dies betrifft den Eingangsport (für zwei 8Bit-Adressen) und interne Komponenten wie SFR, ALU und MISR.

Die internen Speicherblöcke, wie die Testdatenspeicher des USIF-Controllers, die Empfangs- und Sendepuffer der Kommunikations-Controller und der ROM des Selbsttests, sind aber nicht in die Betrachtung einbezogen worden. Diese Speicher sind in der implementierten Architektur generisch ausgelegt, um die Größe je nach Anwendungsfall anpassen zu können. Die fehlenden Datenspeicher der Testeingaben, Referenzdaten und Testantworten, die den Großteil der USIF-Komponenten ausmachen, führen somit zu dem sehr geringen Anteil.

Es ist zu erkennen, dass das Design für den Testzugang zur internen Produktionstestlogik gegenüber einem komplexen IC sehr gering ausfällt. Da im Steuergerät bereits vorhandene Schnittstellen und Sicherheitsmodule genutzt werden sollen, wird also nur verhältnismäßig wenig Zusatzlogik für die Verknüpfung der Anwendungsschnittstelle mit der Testarchitektur benötigt.

6 Kapitel 6
Zusammenfassung

In dieser Arbeit wurde aufgezeigt, dass eine feingranulare Fehlerdiagnose von hochintegrierten Schaltungen auch in der Postproduktionsphase durchgeführt werden kann. Hierzu wird durch das vorgestellte Konzept die schaltungsinterne Produktionstestlogik über serielle Standardschnittstellen anwendbar gemacht.

Für Steuergeräte im Automobilbereich bedeutet dies, dass im Falle einer Fehlfunktion eine vorhandene Anwendungsschnittstelle als Testzugang für einen detaillierten diagnostischen Test des eingebetteten Systems dient, somit also keine aufwendige Demontage für nähere Untersuchungen notwendig ist. Kann mittels einer Diagnose ein in der Fahrzeugelektronik gemeldeter Fehler auf einen fehlerhaften IC eines Steuergerätes zurückgeführt werden, kann die betroffene Komponente gezielt ausgetauscht werden. Somit können wiederholte Werkstattaufenthalte und teuer Reparatur- und Servicekosten vermieden werden.

Neben der gewonnenen Diagnosefähigkeit im Feld kann der Testzugang bereits den Produktionstest beim Steuergerätehersteller unterstützen. Es wird hier ein erweiterter Produktionstest bereitgestellt, der über den Leiterplattentest hinaus eine nachweisbare hohe Prüfschärfe bietet. So kann ein IC nach dem Aufbringen auf die Leiterplatte oder nach dem Einbetten in das Steuergerät, mehr als durch den üblichen Boundary Scan möglich, strukturorientiert mit hoher Fehlerüberdeckung getestet werden. Die Fehlerüberdeckung, mit der ein eingebetteter IC als funktionstüchtig nachgewiesen werden soll, kann als aussagekräftiger Parameter für Sicherheitskriterien, gerade hinsichtlich des neuen Standards für funktionale Sicherheit von Automobilen *ISO26262*, dienen.

Eine weitere Testumgebung ist die feingranulare Fehleranalyse von Rückläufern. Das heißt, ein fehlerhaftes Steuergerät kann entweder beim Steuergerätehersteller oder, falls der IC als fehlerverursachende Komponente ausgemacht wurde, beim Halbleiterhersteller detailliert untersucht werden. Hierbei ist es möglich, den IC, ohne diesen von der Steuergeräteplatine lösen zu müssen, mittels umfangreicher Testdaten, die über die Anwendungsschnittstelle des Steuergerätes übertragen werden, zu analysieren und so den beziehungsweise die Fehler in der Schaltungsstruktur zu diagnostizieren. Der Halbleiterhersteller kann mittels der daraus gewonnenen Informationen entsprechende

Anpassungen im Fertigungsprozess oder auch im Fertigungstest vornehmen, die eine Verbesserung der Chipqualität versprechen.

Neben dem Testzugang wurde auch ein integrierter Selbsttest umgesetzt. Solch ein Test, der beispielsweise während der Hoch- oder Nachlaufphase eines Fahrzeugs durchgeführt wird, kann strukturelle Fehler des ICs bereits vor einer möglichen Auswirkung auf die Funktionalität des Systems identifizieren und somit die Systemzuverlässigkeit erhöhen. Hierzu wurde zusätzlich ein kompakter Controller implementiert, der mittels Testinformationen aus einem schaltungsinternen ROM die Produktionstestlogik betreibt. Das heißt, es wird mit nur geringem Hardware-Aufwand und geringem Speicheraufwand, der für Rekonfigurationen und Reinitialisierungen des Mustergenerators oder gegebenenfalls für deterministische Muster beansprucht wird, ein effektiver Selbsttest mit relativ hoher Fehlerüberdeckung realisiert.

Das Konzept wurde durch mangelnde Analysefähigkeit der Automotive-ICs, die sich in ihrem Funktionsumfeld befinden, motiviert. Gerade in der Automobilindustrie wirken sich die nach einer Störung in der Elektronik nicht feststellbaren Fehler hinsichtlich des Sicherheitsanspruchs, aber auch des Kostenfaktors, besonders stark aus. Die Lösung des Testzugangs ist aber nicht nur in diesem Umfeld, sondern grundsätzlich auf alle eingebetteten Systeme anwendbar. Daher wurde der Testzugang auch nicht auf eine spezifische Schnittstelle eines Automotiv-Bussystems ausgelegt, sondern allgemein für eine dem eingebetteten System zur Verfügung stehenden Anwendungsschnittstelle beschrieben. In der Umsetzung wurden dies exemplarisch anhand FlexRay und USB demonstriert.

Der Testzugang setzt dabei die maximale über die genutzte Standardschnittstelle realisierbare Datenrate um. Die im Verhältnis zur theoretischen Testzeit, die die verfügbare Datenrate erwarten lässt, etwas längere Testdauer, die in den Ergebnissen im vorangegangenen Kapitel präsentiert wurde, hängt von der implementierten Applikation ab. Diese setzt den Testablauf mittels eines garantierten fehlerfreien Austausches von Datenpaketen um. Das in der Applikation angewandte Kommunikationsverfahren, das auf ein zeitintensives Polling von Statusinformationen beruht, bietet daher noch Optimierungspotential.

Ein wichtiger Punkt, der hier nochmals Erwähnung finden muss, ist der Schutz vor unbefugtem Zugriff. Der Zugang über standardisierte Schnittstellen zu internen Strukturen des ICs, erfordert adäquate Maßnahmen gegen unautorisierte Kommunikation und missbräuchliche Anwendungen. Es gibt bewährte industriell genutzte Hardware-Lösungen, wie beispielsweise der von deutschen Automobilherstellern angewandte SHE-Standard, und darauf basierende oft herstellerspezifische Verfahren, die Authentifizierung und Verschlüsselung für Systemzugriffe gewährleisten.

Bei einer angemessene Lösung für sicheren Zugriff schafft der Einsatz der präsentierten Testanbindung, die gegenüber eines komplexen ICs eine Zusatzlogik mit relativ geringem Hardware-Aufwand darstellt, einen autorisierten Zugang zur Produktionstestlogik und bietet somit erheblichen Nutzen bezüglich Systemzuverlässigkeit und Diagnosefähigkeit.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACK	Acknowledgement
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BILBO	Build-In Logic Block Observation
BISR	Built-In-Self-Repair
BIST	Built-In Self-Test
BSS	Byte Start Sequence (FlexRay)
CAD	Computer Aided Design
CAN	Controller Area Network
CC	Communication Controller
CDC	Clock Domain Crossing
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access/ Collision Avoidance
CSMA/CR	Carrier Sense Multiple Access/ Collision Resolution
CTL	Core Test Language
CUT	Circuit Under Test
D^2B	Domestic Digital Bus
DCB	Don't Care Bit
DFT	Design For Testabilty
DTS	Dynamic Trailing Sequence (FlexRay)
DUT	Device Under Test
ECU	Electrical Control Unit
EDA	Electronic Design Automation
EDT	Embedded Deterministic Test
ELC	EDT-Logik-Controller
FAN	Fan-Out Oriented Algorithm
FDD	Fault Detection and Diagnosis
FES	Frame End Sequence (FlexRay)
FIFO	Datenspeicher nach dem First-In-First-Out-Prinzip
FMEA	Fault Mode and Effects Analysis
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSS	Frame Start Sequence (FlexRay)

FTDMA ..... Flexible Time Division Multiple Access
FVHDL ..... Fast-VHDL-Simulator
GALS ..... Globally Asynchronous Locally Synchronous
GUI ..... Graphical User Interface
HID ..... Human Interface Device (USB)
HTTF ..... Hard-To-Test Faults
IC ..... Integrated Circuit
IDDQ ..... Quiescent Supply Current
IEEE ..... Institute of Electrical and Electronics Engineers
IJTAG ..... Internal JTAG, IEEE P1687
ISCAS ..... IEEE International Symposium on Circuits and Systems
ISO ..... International Organization for Standardization
ITC ..... International Test Conference
ITRS ..... International Technology Roadmap for Semiconductors
JTAG ..... Joint Test Action Group, IEEE1149.1
KFF ..... Kein Fehler feststellbar
LBIST ..... Logic Built-In Self-Test
LFSR ..... Linear Feedback Shift Register
LIN ..... Local Interconnect Network
LOC ..... Launch-On-Capture Scan
LOS ..... Launch-On-Shift Scan
LSB ..... Least Significant Bit
LSSD ..... Level-Sensitive Scan Design
MIF ..... Memory Initialisation File
MILEF ..... Mixed Level Automatic Test Pattern Generation
MISR ..... Multiple Input Signature Register
MOST ..... Media Oriented Systems Transport
MP-LFSR ..... Multiple-Polynomial LFSR
MSB ..... Most Significant Bit
MT ..... Minimum Transition
MTTF ..... Mean Time To Failure
NAK ..... Negative Acknowledgement
NIT ..... Network Idle Time (FlexRay)
OEM ..... Original Equipment Manufacturer
ORA ..... Output Response Analyzer
OSI ..... Open Systems Interconnection
PCB ..... Printed Circuit Board
PID ..... Product Indentificator (USB)
PLL ..... Phase-Locked Loop
PODEM ..... Path-Oriented Decision Making
PRPG ..... Pseudo-Random Pattern Generator
RAM ..... Read Access Memory
ROM ..... Read-Only Memory
RTL ..... Register-Transfer Level
RTOS ..... Real-Time Operating System

Rx ..... Receiver, allg. bez. Empfangskomponente
SBST ..... Software-Based Self-Test
SC ..... Scan-Controller
SCT ..... Scan-Controller-based Test
SFF ..... Scan-Flipflop
SFR ..... Scan Feed Register
SHE ..... Secure Hardware Extension
SOPC ..... System on a Programmable Chip
SRSG ..... Shift Register Sequence Generator
STIL ..... Standard Test Interface Language
STUMPS ..... Self-Test Using MISR and Parallel Shift Register Sequence Generator
SW ..... Symbol Window (FlexRay)
TAP ..... Test Access Port (JTAG)
TCK ..... Test Clock (JTAG)
TDI ..... Test Data Input (JTAG)
TDL ..... Texas Instruments Test Description Language
TDMA ..... Time Division Multiple Access
TDO ..... Test Data Output (JTAG)
TMS ..... Test Mode Select (JTAG)
TP ..... Testprozessor
TPG ..... Test Pattern Generator
TRST ..... Test Reset (JTAG)
TSS ..... Transmission Start Sequence (FlexRay)
TTCAN ..... Time-Triggered CAN
Tx ..... Transmitter, allg. bez. Senderkomponente
USB ..... Universal Serial Bus
USIF ..... Universal Scan-Test Interface
VHDL ..... Very High speed integrated circuits Description Language
VID ..... Vendor Indentificator (USB)
VLIW ..... Very Long Instruction Word

Literaturverzeichnis

[ACE14] U. Abelein, A. Cook, P. Engelke, M. Glaß, F. Reimann, L. Rodríguez Gómez, T. Russ, J. Teich, D. Ull & H.-J. Wunderlich. Non-Intrusive Integration of Advanced Diagnosis Features in Automotive E/E-Architectures. In *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, 2014.

[AES01] Advanced Encryption Standard (AES) Federal Information Processing Standards Publication 197. National Institute of Standards and Technology, 2001.

[ALH12] U. Abelein, H. Lochner, D. Hahn & S. Straube. Complexity, Quality and Robustness - The Challenges of Tomorrow's Automotive Electronics. In *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition*, pp. 870–871, 2012.

[Axe05] J. Axelson. *USB Complete: Everything You Need to Develop Custom USB Peripherals*. Complete Guides Series. Lakeview Research, 2005.

[Axe07] J. Axelson. *USB 2.0 - Handbuch für Entwickler*, Vol. 3. Hüthig Jehle Rehm, 2007.

[AZT05] S. Sattler. Applikationsspezifische Testmethodik für hochkomplexe Systeme der Kommunikations- und Kraftfahrzeugtechnik: BMBF-Verbundprojekt AZTEKE; Abschlussbericht. Technical report, ATMEL Germany GmbH, Infineon Technologies AG, Philips Semiconductors GmbH, 2005.

[BA00] M. L. Bushnell & V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits*. Frontiers in Electronic Testing. Springer Science New York, 2000.

[BBD02] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, A. Ferko, B. Keller, D. Scott, B. Könemann & T. Onodera. Extending OPMISR beyond 10x Scan Test Efficiency. *IEEE Design & Test of Computers*, Vol. 19, pp. 65–72, 2002.

[BCA75] N. Benowitz, D. F. Calhoun, G. E. Alderson, J. E. Bauer & C.T. Joeckel. An Advanced Fault Isolation System for Digital Logic. *IEEE Transactions on Computers*, Vol. 24, pp. 489–497, 1975.

[BM82] P. H. Bardell & W. H. McAnney. Self-Testing of Multiple Logic Modules. In *Proc. IEEE International Test Conference*, pp. 200–204, 1982.

[CAN91] CAN Specification 2.0. Robert Bosch GmbH, 1991.

[CFK04] V. Chickermane, B. Foutz & B. Keller. Channel Masking Synthesis for Efficient On-Chip Test Compression. In *Proc. IEEE International Test Conference*, pp. 452–461, 2004.

[CHI12] A. Cook, S. Hellebrand, M. E. Imhof, A. Mumtaz & H.-J. Wunderlich. Built-In Self-Diagnosis Targeting Arbitrary Defects with Partial Pseudo-Exhaustive Test. In *Proc. Latin American Test Workshop*, pp. 1–4, 2012.

[Cum08] C. E. Cummings. Clock Domain Crossing (CDC) Design and Verification Techniques using SystemVerilog. *Sunburst Design, SNUG Boston*, Vol. 1.0, 2008.

[DIA13] H. Obermeir. Durchgängige Diagnosefähigkeit in Halbleiterbauelementen und übergeordneten Systemen zur Analyse von permanenten und sporadischen Elektronikausfällen im Gesamtsystem Automobil: BMBF-Verbundprojekt DIANA, Schlussbericht. Technical report, Audi AG, Conti Temic Microelectronic GmbH, Infineon Technologies AG, Zentrum Mikroelektronik Dresden AG, Ingolstadt, 2013.

[Eng02] H. Engels. *Can-Bus*, Vol. 2. Franzis Verlag, 2002.

[Fis03] H. Fischer. Delay-Fault-Test. In *Proc. Test Kompendium*, pp. 160–162, 2003.

[FPS05] FlexRay Protocol Specification 2.1 Rev.A. FlexRay Consortium, 2005. Registered copy for cgleichn@informatik.tu-cottbus.de.

[FRG07] R. Frost, D. Rudolph, C. Galke, R. Kothe & H.T. Vierhaus. A Configurable Modular Test Processor and Scan Controller Architecture. In *Proc. IEEE International On-Line Testing Symposium*, pp. 277–284, 2007.

[GGV04] U. Gätzschmann, C. Galke & H. T. Vierhaus. Ein flexibles Verfahren zur Testdaten-Kompaktierung und -Dekompaktierung für den Scan-Test. In *Proc. GI/GMM/ITG Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen*, 2004.

[GGV06] C. Galke, U. Gätzschmann & H. T. Vierhaus. Scan-Based SoC Test using Space/ Time Pattern Compaction Schemes. In *Proc. IEEE Euromicro Symposium on Digital Systems Design*, pp. 433–438, 2006.

[GPV02] C. Galke, M. Pflanz & H. T. Vierhaus. A Test Processor Concept for Systems-On-a-Chip. In *Proc. IEEE International Conference on Computer Design*, pp. 210–212, 2002.

[Gre09] M. Greinwald. *USB und Firewire- Funktionalitäten und Anwendungen.* GRIN Verlag GmbH, 2009.

[GV92] U. Glaser & H. T. Vierhaus. MILEF: An Efficient Approach to Mixed Level Automatic Test Pattern Generation. In *Proc. Design Automation Conference*, pp. 318–321, 1992.

[GVE12a] C. Gleichner, H. T. Vierhaus & P. Engelke. Anbindung scanbasierter Tests an Standard-Schnittstellen: Möglichkeiten und Grenzen. In *Proc. GI/GMM/ITG Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen*, pp. 17–22, 2012.

[GVE12b] C. Gleichner, H. T. Vierhaus & P. Engelke. Scan Based Tests via Standard Interfaces. In *Proc. IEEE Euromicro Conference on Digital System Design*, pp. 844–851, 2012.

[HM84] S. Z. Hassan & E. J. McCluskey. Increased Fault Coverage through Multiple Signatures. In *Proc. Fault-Tolerant Computing Symposium*, pp. 354–359, 1984.

[HTR92] S. Hellebrand, S. Tarnick, J. Rajski & B. Courtois. Generation of Vector Patterns through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers. pp. 120–129, 1992.

[IFX14] Infineon Technologies AG, Monika Sonntag. The DIANA Research Project Explores End-to-End Diagnostic Capabilities for Automotive Electronics Systems. July 2010. http://www.infineon.com/cms/en/corporate/press/news/releases/2010/INFXX201007-058.html, [Stand 16.04.2014].

[IJT13] P1687/D1.62 - IEEE Draft Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device. IEEE Association, 2013.

[Ise06] R. Isermann. *Fault-Diagnosis Systems - An Introduction from Fault Detection to Fault Tolerance*. Springer, 2006.

[ISO10a] ISO 10681-1:2010 Road Vehicles - Communication on FlexRay - Part 1: General Information and Use Case Definition. International Organization for Standardization, 2010.

[ISO10b] ISO 10681-2:2010 Road Vehicles - Communication on FlexRay - Part 2: Communication Layer Services. International Organization for Standardization, 2010.

[ISO11] ISO 26262: Road Vehicles - Functional Safety. International Organization for Standardization, 2011.

[ITR11] International Technology Roadmap for Semiconductors. Technical report, European Semiconductor Industry Association (ESIA), Japan Electronics, Information Technology Industries Association (JEITA), Korean Semiconductor Industry Association (KSIA), Taiwan Semiconductor Industry Association (TSIA), and United States Semiconductor Industry Association (SIA), 2011.

[JAR11] A. Jutman, I. Aleksejev & J. Raik. *Sequential Test Set Compaction in LFSR Reseeding. In Design and Test Technology for Dependable Systems-On-Chip*, Kapitel 22, pp. 476–493. Hershey - New York: Information Science Reference IGI Global, 2011.

[JTG01] IEEE 1149.1-2001 Standard Test Access Port and Boundary-Scan Architecture. IEEE Association, 2001.

[KA87] M. Khare & A. Albicki. Cellular Automata used for Test pattern Generation. In *Proc. IEEE International Conference on Computer Design*, pp. 56–59, 1987.

[KGV04] C. Kretzschmar, C. Galke & H. T. Vierhaus. A Hierarchical Self-Test Scheme for SoCs. In *Proc. IEEE International On-Line Testing Symposium*, pp. 37–42, 2004.

[KGV05] R. Kothe, C. Galke & H. T. Vierhaus. A Multi-Purpose Concept for SoC Self-Test Including Diagnostic Features. In *Proc. IEEE International On-Line Testing Symposium*, pp. 241–246, 2005.

[KK07] I. Koren & C. M. Krishna. *Fault-tolerant Systems.* Elsevier / Morgan Kaufmann Publishers, 2007.

[KKV09] T. Koal, R. Kothe & H. T. Vierhaus. Der erste Mikroprozessor der BTU Cottbus - Die Entwicklung des Testprozessors. *Forum der Forschung*, Vol. 22, pp. 127–132, 2009.

[Kön91] B. Könemann. LFSR-Coded Test Patterns for Scan Designs. In *Proc. IEEE European Test Conference*, pp. 237–242, 1991.

[KV08] R. Kothe & H. T. Vierhaus. A Scan Controller Concept for Low-Power Scan Tests. *Journal of Low-Power Electronics, American Science Publishers*, Vol. 4, pp. 1–9, 2008.

[KV10] R. Kothe & H. T. Vierhaus. Test Data and Power Reductions for Transition Delay Tests for Massive-Parallel Scan Structures. In *Proc. IEEE Euromicro Symposium on Digital Systems Design*, pp. 283–290, 2010.

[KW98] G. Kiefer & H.-J. Wunderlich. Deterministic BIST with Multiple Scan Chains. In *Proc. IEEE International Test Conference*, pp. 1057–1064, 1998.

[Lan11] L. Lan. The AES Encryption and Decryption Realization Based on FPGA. In *Proc. IEEE International Conference on Computational Intelligence and Security*, pp. 603–607, 2011.

[Law11] W. Lawrenz. *CAN Controller Area Network: Grundlagen und Praxis.* Hüthig Verlag, 2011.

[LLH08] H. Liu, H. Li, Y. Hu & X. Li. A Scan-Based Delay Test Method for Reduction of Overtesting. In *Proc. IEEE International Workshop on Electronic Design, Test and Applications*, pp. 521–526, 2008.

[LZ12] E. Larsson & F. G. Zadegan. Accessing Embedded DFT Instruments with IEEE P1687. In *Proc. IEEE Asian Test Symposium*, pp. 71–76, 2012.

[McC86] E. J. McCluskey. *Logic Design Principles: With Emphasis on Testable Semi-Conductor Circuits*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[MHB00] P. Maxwell, I. Hartanto & L. Bentz. Comparing Functional and Structural Tests. In *Proc. IEEE International Test Conference*, pp. 400–407, 2000.

[MIW11] A. Mumtaz, M. E. Imhof & H.-J. Wunderlich. P-PET: Partial Pseudo-Exhaustive Test for High Defect Coverage. In *Proc. IEEE International Test Conference*, pp. 1–8, 2011.

[MLM04] S. Mitra, S. S. Lumetta & M. Mitzenmacher. X-Tolerant Signature Analysis. In *Proc. IEEE International Test Conference*, pp. 432–441, 2004.

[MVL07] C. Mucci, L. Vanzolini, A. Lodi, A. Deledda, R. Guerrieri, F. Campi & M. Toma. Implementation of AES/Rijndael on a Dynamically Reconfigurable Architecture. *IEEE Design, Automation & Test in Europe Conference & Exhibition*, pp. 355–360, 2007.

[Nol09] M. Nolte. *Entwicklung eines CAN-Bus-Adapters für spezielle Anforderungen zur Fahrzeuganbindung*. Europäischer Hochschulverlag, 2009.

[NPR03] M. Naruse, I. Pomeranz, S. M. Reddy & S. Kundu. On-Chip Compression of Output Responses with Unknown Values using LFSR Reseeding. In *Proc. IEEE International Test Conference*, pp. 1060–1068, 2003.

[Okl08] V. G. Oklobdzija. *Digital Design and Fabrication*, Vol. 2. CRC Press, Taylor & Francis Group, 2008.

[Par03] K. P. Parker. *The Boundary-Scan Handbook*, Vol. 3. Springer US, 2003.

[PF06] I. Polian & H. Fujiwara. Functional Constraints vs. Test Compression in Scan-Based Delay Testing. In *Proc. IEEE Design, Automation & Test in Europe Conference & Exhibition*, pp. 218–223, 2006.

[PR07] D. Paret & R. Riesco. *Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire...* Wiley, 2007.

[Pra96] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Rau07] M. Rausch. *FlexRay - Grundlagen, Funktionsweise, Anwendung*, Vol. 1. Carl Hanser Fachbuchverlag, 2007.

[REP05] J. Rearick, B. Eklow, K. Posse, A. Crouch & B. Bennetts. IJTAG (Internal JTAG): A Step Toward a DFT Standard. In *Proc. IEEE International Test Conference*, pp. 8–15, 2005.

[RMC06] K. Roy, T. M. Mak & K.-T. Cheng. Test Consideration for Nanometer-Scale CMOS Circuits. *IEEE Design & Test of Computers*, Vol. 23, pp. 128–136, 2006.

[RTK02] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide & J. Qian. Embedded Deterministic Test for Low Cost Manufacturing Test. In *Proc. IEEE International Test Conference*, pp. 301–310, 2002.

[RTK04] J. Rajski, J. Tyszer, M. Kassab & N. Mukherjee. Embedded Deterministic Test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, Vol. 23, pp. 776–792, 2004.

[RTW05] J. Rajski, J. Tyszer, C. Wang & S. M. Reddy. Finite Memory Test Response Compactors for Embedded Test Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, Vol. 24, pp. 622–634, 2005.

[Sch06] Mario Schölzel. *Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren*. Dissertation, BTU Cottbus, 2006.

[SHE09] Secure Hardware Extension - Functional Specification V1.1, Rev. 439. HIS - Herstellerinitiative Software, 2009.

[SP04] M. A. Shah & J. H. Patel. Enhancement of the Illinois Scan Architecture for Use with Multiple Scan Inputs. In *Proc. Annual Symposium on VLSI*, pp. 167–172, 2004.

[Ste78] J. H. Stewart. Application of Scan/Set for Error Detection and Diagnostics. In *Proc. IEEE Semiconductor Test Conference*, pp. 152–158, 1978.

[Sto11] N. Stollon. *On-Chip Instrumentation - Design and Debug for Systems on Chip*. Springer, 2011.

[TM95] N. A. Touba & E. McCluskey. Transformed Pseudo-Random Patterns for BIST. In *Proc. VLSI Test Symposium*, pp. 410–416, 1995.

[TR09] S. C. Talbot & S. Ren. Comparision of FieldBus Systems CAN, TTCAN, FlexRay and LIN in Passenger Vehicles. In *Proc. International Conference on Distributed Computing Systems*, pp. 26–31, 2009.

[URV11] R. Ubar, J. Raik & H. T. Vierhaus. *Design and Test Technology for Dependable Systems-on-Chip*. Information Science Reference, IGI Global, 2011.

[USB00] USB Specification 2.0 Rev. 2.0. USB Implementers Forum, Inc., 2000.

[USB08] USB Specification 3.0 Rev. 1.0. USB Implementers Forum, Inc., 2008.

[VCI14] Vector Informatik GmbH. FlexRay Protocol Reference Chart. 2008. http://vector.com/portal/medien/solutions_for/flexray/schematic_graphics/chart_flexray.png, [Stand 12.05.2014].

[VDI14] Verein Deutscher Ingenieure. Komplexe Bordnetze entwickeln, VDI-Konferenz Automobile Bordnetzentwicklung, VDI Wissensforum. April 2012. http://www.vdi-wissensforum.de/de/nc/presse/details/article/komplexe-bordnetze-entwickeln/, [Stand 16.04.2014].

[VSP14] VehicleServicePros.com, D. Kolman. DIANA is Looking into Vehicle Electronic Controls. July 2010. http://www.vehicleservicepros.com/blog/10342815/diana-is-looking-into-vehicle-electronic-controls, [Stand 16.04.2014].

[Wan06] L.-T. Wang, C.-W. Wu & X. Wen. *VLSI Test Principles and Architectures: Design for Testability.* Morgan Kaufmann Publishers, Elsevier Inc., San Francisco, CA, USA, 2006.

[WK96] H.-J. Wunderlich & G. Kiefer. Bit-flipping BIST. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp. 337–343, 1996.

[Wol83] S. Wolfram. Statistical Mechanics of Cellular Automata. In *Reviews of Modern Physics*, Vol. 55, pp. 601–644, 1983.

[WWP03a] P. Wohl, J. A. Waicukauski, S. Patel & M. B. Amin. Efficient Compression and Application of Deterministic Patterns in a Logic BIST Architecture. In *Proc. Design Automation Conference*, pp. 566–569, 2003.

[WWP03b] P. Wohl, J. A. Waicukauski, S. Patel & M. B. Amin. X-Tolerant Compression and Application of Scan-ATPG Patterns in a BIST Architecture. In *Proc. IEEE International Test Conference*, pp. 727–736, 2003.

[WWP04] P. Wohl, J. A. Waicukauski & S. Patel. Scalable Selector Architecture for X-tolerant Deterministic BIST. In *Proc. Design Automation Conference*, pp. 934–939, 2004.

[XCL09] S. Xiao, Y. Chen & P. Luo. The Optimized Design of Rijndael Algorithm Based on SOPC. In *Proc. IEEE International Conference on Information & Multimedia Technology*, pp. 384–387, 2009.

[XS06] G. Xu & A. D. Singh. Low Cost Launch-on-Shift Delay Test with Slow Scan Enable. In *Proc. IEEE European Test Symposium*, pp. 9–14, 2006.

[XS07] G. Xu & A. D. Singh. Scan Cell Design for Launch-On-Shift Delay Tests with Slow Scan Enable. *Proc. IET Computers & Digital Techniques*, Vol. 1, pp. 213–219, 2007.

[ZS07] W. Zimmermann & R. Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards.* ATZ-MTZ Fachbuch. Vieweg, 2007.

Anhang

A Anhang A
Implementierungsdetails

In diesem Anhang sind Darstellungen und Parameter bezüglich der in Kapitel 4 beschriebenen prototypischen Umsetzung zusammengefasst.

A.1 Parameter und Bezeichner

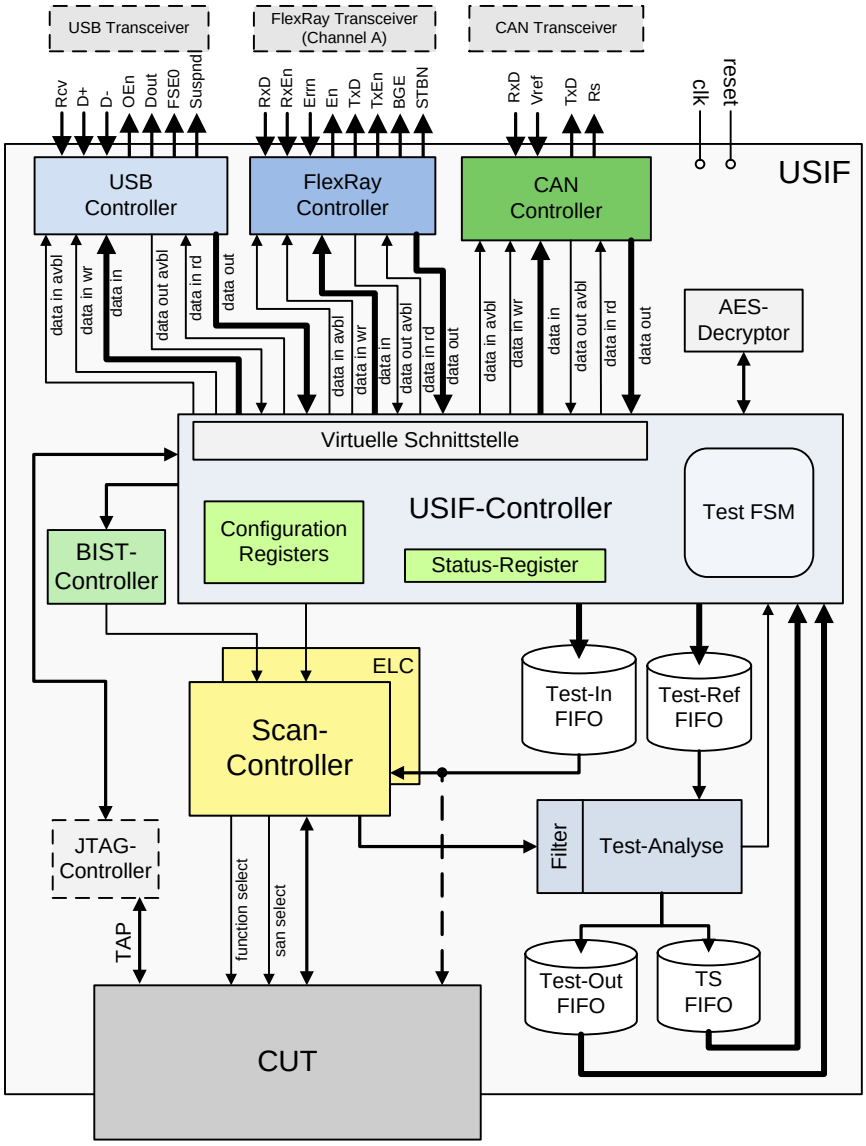
In der folgenden Tabelle sind Parametern, die an mehreren Stellen in dieser Arbeit wiederholt auftreten, aufgelistet.

Scan-Kettenanzahl	n_{Chain}	Anzahl der in die Schaltung integrierten Scan-Ketten
Scan-Tiefe	m_{Chain}	Scan-Tiefe; entspricht der Anzahl an Testvektoren pro Testmuster und wird durch die längste Scan-Kette bestimmt
Scan-Clock-Frequenz	f_{SC}	Frequenz der Scan-Clock clk_{SC}
Datenbreite des Testeingangs	$i_{\text{SC}} = \max(\left\lceil \frac{d_{\text{LFSR}}}{2} \right\rceil, 2\lceil \text{ld}(d_{\text{SFR}}) \rceil)$	Datenbreite des Eingangsports der Testarchitektur (SCT, EDT); Der Eingang muss mindestens so breit sein, um das LFSR in maximal 2 Takten laden oder das komplette SFR für Bitmodifizierungen adressieren zu können
Datenbreite des Testausgangs	o_{SC}	Datenbreite des Eingangsports der Testarchitektur (SCT, EDT); Am Ende eines Testlaufs ist über diesen Port das komplette MISR in $\left\lceil \frac{d_{\text{MISR}}}{o_{\text{SC}}} \right\rceil$ Takten auslesbar
LFSR-Datenbreite	d_{LFSR}	Datenbreite des LFSRs
SFR-Datenbreite	$d_{\text{SFR}} \geq n_{\text{Chain}}$	Datenbreite des SFRs

MISR-Datenbreite	$d_{\text{MISR}} = n_{\text{Chain}}$	Datenbreite des MISRs; Im Falle einer modularisierten Scan-Controller-Architektur, werden die MISR-Komponenten $\{1, ..., k\}$ als ein MISR der Datenbreite $\sum_{i=1}^{k} d_{\text{MISR}i}$ betrachtet
Datenbreite des Testvektors	$d_{\text{Testvector}} = n_{\text{Chain}}$	Der Testvektor ist das Eingabemuster für die parallelen Scan-Ketten
USIF-Datenbreite	d_{USIF}	Datenbreite des Datenpfads der USIF-Architektur
Datenbreite des Eingangsports des TestIn-FIFOs	$i_{\text{TestIn}} = d_{\text{USIF}}$	Das TestIn-FIFO wird über den USIF-Datenpfad beschrieben
Datenbreite des Ausgangsports des TestIn-FIFOs	$o_{\text{TestIn}} = i_{\text{SC}}$	Das TestIn-FIFO wird von der Testarchitektur (SC, EDT) gelesen
Datenbreite des Eingangsports des TestRef-FIFOs	$i_{\text{TestRef}} = d_{\text{USIF}}$	Das TestRef-FIFO wird über den USIF-Datenpfad beschrieben
Datenbreite des Ausgangsports des TestRef-FIFOs	$o_{\text{TestRef}} = o_{\text{SC}}$	Das TestRef-FIFO wird von der Analyzer gelesen
Datenbreite des Eingangsports des TestOut-FIFOs	$i_{\text{TestOut}} = o_{\text{SC}}$	Das TestOut-FIFO wird von der Testarchitektur bzw. vom Analyzer beschrieben
Datenbreite des Ausgangsports des TestOut-FIFOs	$o_{\text{TestOut}} = d_{\text{USIF}}$	Das TestOut-FIFO wird über den USIF-Datenpfad gelesen
Größe des Testsatzes	g_{Test}	Umfang des Testsatzes bzw. Scan-Controller-Programms
Paketgröße	g_{Packet}	Datenumfang eines Testdatenpaketes
Testmusteranzahl	n_{Pat}	Anzahl der einen Testlauf umfassenden Testmuster
Paketanzahl	n_{Pat}	Anzahl der Pakete eine Testsatzes
Analyserate	a_{Rate}	Sample-Rate mit der der Analyzer die Testausgaben der Testarchitektur entnimmt
Analysebasis	a_{Base}	Die Basis gibt den Startwert an, ab dem mit dem Sampling der Testausgaben begonnen wird

A.2 USIF-Architektur

Der Aufbau der USIF-Architektur ist hier in einem vereinfachten Blockschaltbild wiedergegeben. Die Darstellung der Ports und Signale der Standardschnittstellen fällt hier etwas ausführlicher aus, um die Anbindung an die spezifischen Kommuniktions-Controller und an die zugehörigen Transceiver hervorzuheben.



A.3 USIF-Memory Map

Adr.	Parameter	Beschreibung	Typ	Werte-bereich
0x00	DEVICEID	Identifikationsnummer des USIF-Gerätes (*read only*)	Integer	$[1,(2^{16}-1)]$
	USIF	Findet ein Zugriff auf die Adresse 0 statt, bei einer Instruktion ungleich ReadConfig, bedeutet dies eine Konfiguration des USIF-Controllers. Die Instruktion ist in dem Fall StartCom, StopCom, StartSC, StopSC, StartLoopback, StopLoopback oder RunBIST, um den entsprechenden Modus zu initiieren bzw. zu beenden.		
0x01	STATUS	Status-Register (*Read Only*):	Bitarray	
	[0]	USIFLock - Status USIF-Zugangskotrolle		
	[1]	SCRun - Scan-Test aktiv		
	[2]	TestInLock - Speicher der Testeingaben voll		
	[3]	TestRefLock - Speicher der Referenzdaten voll		
	[4]	TestOutAvbl - Speicher der Testantworten nicht leer		
	[5]	TestOutPacket - komplettes Paket verfügbar		
	[6]	TCSync - Synchronisationsbit (Anfrage verarbeitet)		
	[7]	TCAvbl - weitere Anfragen/Daten zu verarbeiten		
	[8..9]	SCStatus - Status des Scan-Controllers		
	[10]	SCFault - Fehlerstatus beim Pass/Fail-Test		
	[11]	BusError - Kommunikations-Controller meldet Error		
	[12..15]	*Reserved*		
0x02	ERRORS	Error-Register (*Read Only*)	Bitarray	
	[0]	SCError - Scan-Controller Error		
	[1]	FLEXError - FlexRay-Controller Error		
	[2]	CANError - CAN-Controller Error		
	[3..15]	*Reserved*		
0x03	Control-Register			
	CtrlReg	Control-Bits	Bitarray	
	[0]	USIFReset - Zurücksetzen des USIF-Controllers und der Test-FIFOs (*Active High*)		
	[1..3]	*Reserved*		
	ScanMode	Scan-Modus (*Default* 0)	Integer	$[0,1]$
	[4..7]	0 - *Decompress* (SCT/ EDT)		
		1 - *DirectScan* (*Bypass*)		
	Analysis-Mode	Analysis-Modus (*Default* 0)	Integer	$[0,3]$
	[8..11]	0 - *LogAll*		
		2 - *LogFaults* (Referenzdaten notwendig!)		
		3 - *PassFail* (Referenzdaten notwendig!)		
	ScanClock	Scan-Clock (*Default* 0)	Integer	$\{0;1;2\}$
	[12..13]	0 - 50 MHz		
		1 - 10 MHz		
		2 - 20 MHz		

Adr.	Parameter	Beschreibung	Typ	Werte-bereich
0x04	Analyzer-Rate/Base			
	Rate [0..7]	Sample-Rate a_{Rate} mit der die Testantworten ausgelesen werden. Wird genutzt für die zeitliche Kompression bei Verwendung des MISR. Ist der Wert bei der Konfiguration 0, wird dieser durch den Controller auf den Minimalwert 1 gesetzt (*Default*: 1)	Integer	$[1,(2^8-1)]$
	Base [8..14]	Startwert a_{Base} der Speicherung der Testantworten. Ist der Wert 0, wird mit der ersten Testausgabe begonnen und, entsprechend der Rate a_{Rate}, jede weitere (a_{Rate})-te Testausgabe ausgewertet/gespeichert (*Default*: 0)	Integer	$[1,(2^7-1)]$
	RestartPerPattern [15]	Dieser Wert gibt an, ob der zugehörige Sample-Counter für jedes Testmuster zurückgesetzt wird. D.h., es wird die Auswertung/Speicherung bei jedem neuen Testmuster wieder mit der (a_{Base})-ten Testausgabe begonnen (*Default*: 0)	Boolean	$[0,1]$
0x05-0x06 *reserved*				
0x07	TestOutPacket	Paketgröße (Anzahl der Adresszeilen), die genutzt wird um über die Statusinformation anzuzeigen, dass mindestens ein komplettes Paket im TestOut-FIFO abrufbar ist (*Default*: 512 Byte). Sei a hier die Adressweite des TestOut-FIFOs.	Integer	$[0,(2^a-1)]$,
0x08	TestOutUsedw	Aktueller Füllstand des Test-Out-FIFOs (*Read Only*)	Integer	$[0,(2^a-1)]$
0x09	JTAGClock	JTAG-Clock (*Default* 0) 0 - 50 MHz 1 - 10 MHz 2 - 20 MHz 3 - 40 MHz	Integer	$\{0;1;2;3\}$
0x0A	UsedAsGateway	Das USIF wird als Gateway zwischen USB und einer weiteren Schnittstelle konfiguriert; Das Mapping auf den entsprechenden Kommunikations-Controller (FlexRay, CAN) wird in der USIF-Architektur vor der Synthese spezifiziert	Boolean	$[0,1]$
Testdatenspeicher:				
0x10	TestIn	FIFO der Testeingaben		
0x11	TestRef	FIFO der Referenzdaten		
0x12	TestOut	FIFO der Testantworten (*Read Only*)		
0x13	TS	FIFO der Timestamps zu den Testantworten, falls AnalysisMode LogFaults oder PassFail (*Read Only*)		
0x14	JTAGIn	FIFO der JTAG-Eingaben TDI und TMS		
0x15	JTAGOut	FIFO der JTAG-Antworten TDO (*Read Only*)		

A.4 USB-Deskriptoren zu 4.1.1

Die für den implementierten USB-Controller gewählte Konfiguration ist in den folgenden Deskriptoren aufgelistet.

Device Descriptor

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x12
bDescriptorType	DEVICE Descriptor	0x01
bcdUSB	USB Spezifikation Versionsnummer im BCD-Format (*Binary Coded Decimal*)	0x0200
bDeviceClass	*Class Code* gibt die Geräteklasse an (siehe *USB Implementers Forum, Inc.*), 0x00 Geräteklasse im Interface definiert, 0xFF Hersteller-spezifische Geräteklasse	0x00
bDeviceSubClass	*SubClass Code* (siehe *USB Implementers Forum, Inc.*)	0x00
bDeviceProtocol	*Protocol Code* (siehe *USB Implementers Forum, Inc.*)	0x00
bMaxPacketSize0	Max. Paketgröße des Endpunkt 0 (8, 16, 32 oder 64)	0x40
idVendor	Vendor ID - muss vom *USB Implementers Forum, Inc.* erworben werden	0x0000
idProduct	Product ID - wird durch den Hersteller festgelegt	0x000A
bcdDevice	Geräte-Versionsnummer im BCD-Format	0x0010
iManufacturer	Index des String Deskriptors für die Herstellerbeschreibung (kein String Deskriptor falls 0)	0x01
iProduct	Index des String Deskriptors für die Produktbeschreibung	0x02
iSerialNumber	Index des String Deskriptors für die Seriennummer des Geräts	0x01
bNumConfigurations	Anzahl der Konfigurationen	0x01

Configuration Descriptor

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x09
bDescriptorType	CONFIGURATION Descriptor	0x02
wTotalLength	Gesamtanzahl (in Byte) dieses und aller folgenden Deskriptoren	0x0027
bNumInterfaces	Anzahl der Interfaces dieser Konfiguration	0x01
bConfigurationValue	Identifikator dieser Konfiguration	0x01
iConfiguration	Index des String Deskriptors dieser Konfiguration (0 falls kein String Deskriptor)	0x00
bmAttributes	D7: fest auf 1 D6: Self-powered D5: Remote Wakeup D4...D0: fest auf 0	0xC0
bMaxPower	Maximalstrom, der vom Gerät in dieser Konfiguration gezogen wird (in Einheiten von 2mA)	0x10

Interface Descriptor

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x09
bDescriptorType	INTERFACE Descriptor	0x04
bInterfaceNumber	Identifikator dieses Interfaces	0x00
bAlternateSetting	Identifikator der alternativen Einstellung dieses Interfaces	0x00
bNumEndpoints	Anzahl an Endpunkten dieses Interfaces (ausgenommen des Control-Endpunktes 0)	0x03
bInterfaceClass	Klassen-Code zugewiesen durch USB-IF (00h ist reserviert, FFh bedeutet Anbieter-definiert)	0xFF
bInterfaceSubClass	SubClass-Code, zugewiesen durch USB-IF	0xFF
bInterfaceProtocol	Protokol-Code, zugewiesen durch USB-IF	0xFF
iInterface	Index des String Deskriptors dieses Interfaces (0 falls kein String Deskriptor)	0x00

Endpoint Descriptor für IN-Endpunkt im Bulk-Transfer (Sendpuffer)

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x07
bDescriptorType	ENDPOINT Descriptor	0x05
bEndpointAddress	Die Endpunktadresse innerhalb des USB-Gerätes, Bit 7: Kommunikationsrichtung (0 = OUT, 1 = IN), Bit 4-6: 0, Bit 0-3: Endpunkt-Nummer	0x82
bmAttributes	Bit 0-1: Transfertyp (0x0 = Control, 0x1 = Isochronous, 0x2 = Bulk, 0x3 = Interrupt), Bit 2-3: Synchronisationstyp für isochr. Endpunkte (0x0 = No Synchronisation, 0x1 = Asynchronous, 0x2 = Adaptive, 0x3 = Synchronous), Bit 4-5: Nutzungsart (0x0 = Data, 0x1 = Feedback, 0x2 = Implicit Feedback Data), Bit 6-7: 0	0x02
wMaxPacketSize	Maximale Größe der Pakete, die der Endpunkt senden bzw. empfangen kann (Im FullSpeed-Modus sind nur 8, 16, 32 oder 64 Byte gültige Werte)	0x0040
bInterval	Polling-Intervall, in dem der Endpunkt für den Datentransfer abgefragt wird; Angabe in Frames (1ms) im LowSpeed/FullSpeed- oder in Microframes (125us) im HighSpeed-Modus	0x00

Endpoint Descriptor für OUT-Endpunkt im Bulk-Transfer (Empfangspuffer)

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x07
bDescriptorType	ENDPOINT Descriptor	0x05
bEndpointAddress	Die Endpunktadresse innerhalb des USB-Gerätes, Bit 7: Kommunikationsrichtung (0 = OUT, 1 = IN), Bit 4-6: 0, Bit 0-3: Endpunkt-Nummer	0x03
bmAttributes	Bit 0-1: Transfertyp (0x0 = Control, 0x1 = Isochronous, 0x2 = Bulk, 0x3 = Interrupt)	0x02
wMaxPacketSize	Maximale Größe der Pakete, die der Endpunkt senden bzw. empfangen kann (Im FullSpeed-Modus sind nur 8, 16, 32 oder 64 Byte gültige Werte)	0x0040
bInterval	Polling-Intervall, in dem der Endpunkt für den Datentransfer abgefragt wird; Angabe in Frames (1ms) im LowSpeed/FullSpeed- oder in Microframes (125us) im HighSpeed-Modus	0x00

Endpoint Descriptor für Status-Endpunkt

Das Statusregister (4 Byte) wird als Interrupt-Endpunkt umgesetzt, um die Statusinformationen in jedem Frame (1ms) zu versenden.

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x07
bDescriptorType	ENDPOINT Descriptor	0x05
bEndpointAddress	Die Endpunktadresse innerhalb des USB-Gerätes, Bit 7: Kommunikationsrichtung (0 = OUT, 1 = IN), Bit 4-6: 0, Bit 0-3: Endpunkt-Nummer	0x81
bmAttributes	Bit 0-1: Transfertyp (0x0 = Control, 0x1 = Isochronous, 0x2 = Bulk, 0x3 = Interrupt)	0x03
wMaxPacketSize	Maximale Größe der Pakete, die der Endpunkt senden bzw. empfangen kann	0x0004
bInterval	Polling-Intervall, in dem der Endpunkt für den Datentransfer abgefragt wird; Angabe in Frames (1ms) im LowSpeed/FullSpeed- oder in Microframes (125us) im HighSpeed-Modus	0x01

String Descriptor 0

Der String Deskriptor 0 wird vor den anderen Deskriptoren gelesen und gibt die Sprache bzw. Sprachen der darin enthaltenen Beschreibungen an.

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x04
bDescriptorType	STRING Descriptor	0x03
wLANGID[0]	LANGID Code (0x0409 Englisch, 0x0407 Deutsch)	0x0409

String Descriptor für die Herstellerbeschreibung

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x18
bDescriptorType	STRING Descriptor	0x03
bString	Beschreibung in UNICODE-Kodierung	0x007300750062 00740074006F 004300200055 00540042 ('BTU Cottbus')

String Descriptor für die Produktbeschreibung

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x18
bDescriptorType	STRING Descriptor	0x03
bString	Beschreibung in UNICODE-Kodierung	0x006500630069 007600650044 002000460049 00530055 ('USIF Device')

String Descriptor für die Seriennummer

Parameter	Beschreibung	Wert [hex]
bLength	Größe des Deskriptors in Byte	0x18
bDescriptorType	STRING Descriptor	0x03
bString	Beschreibung in UNICODE-Kodierung	0x003100300030 003000300030 003000460049 00530055 ('USIF0000001')

A.5 FlexRay-Konfiguration zu 4.1.2

Der folgende Parametersatz wurde der FlexRay-Beispielkonfiguration des Programms *Vector Informatik CANalyzer 8.0.35* entnommen. Einige Knoten-spezifische Parameter, wie *pKeySlotID, pKeySlotUsedForStartup, pKeySlotUsedForSync, pChannels* u. a., wurde individuell angepasst. Der entworfene FlexRay-Controller wurde in dieser Protokollkonfiguration durch das CANalyzer-Programm und den zugehörigen Busanalyzer *VN7600 FlexRay Interface* von Vector Informatik verifiziert. Der FlexRay-Controller wird nach dem Power-up des FPGAs mit diesen Parametern als Defaultwerte initialisiert. Über die implementierte USB-Schnittstelle können die Parameter auch im Nachhinein modifiziert werden. Hierzu wird das USIF-Instruktionswort, mit dem Befehl `TC=WriteConfig`, der entsprechenden Registeradresse und dem gewünschten Parameterwert, genutzt. Die grau unterlegten Parameter sind im implementierten Controller konstante Werte und somit nach der Synthese nicht konfigurierbar.

Protokoll-relevante globale Parameter

Parameter	Einheit	Wert
Bautrate	MHz	10
gChannels	A=1, B=2, A&B=3	1
gClusterDriftDamping	µT	2
gColdstartAttempts	#	8
gdActionpointOffset	MT	2
gdBit	µs	0,1
gdCASRxLowMax	gdBit	99
gdCycle	µs	5000
gdDynamicSlotIdlePhase	Minislots	1
gdMacrotick	µs	1,375
gdMaxInitializationError	µs	2,648
gdMiniSlot	MT	5
gdMiniSlotActionpointOffset	MT	2
gdNIT	MT	6
gdSampleClockPeriod	µs	0,0125
gdStaticSlot	MT	43
gdSymbolWindow	MT	0
gdTSSTransmitter	gdBit	15
gdWakeupSymbolRxIdle	gdBit	59
gdWakeupSymbolRxLow	gdBit	55
gdWakeupSymbolRxWindow	gdBit	301
gdWakeupSymbolTxIdle	gdBit	180
gdWakeupSymbolTxLow	gdBit	60
gListenNoise	#	2
gMacroPerCycle	MT	3636

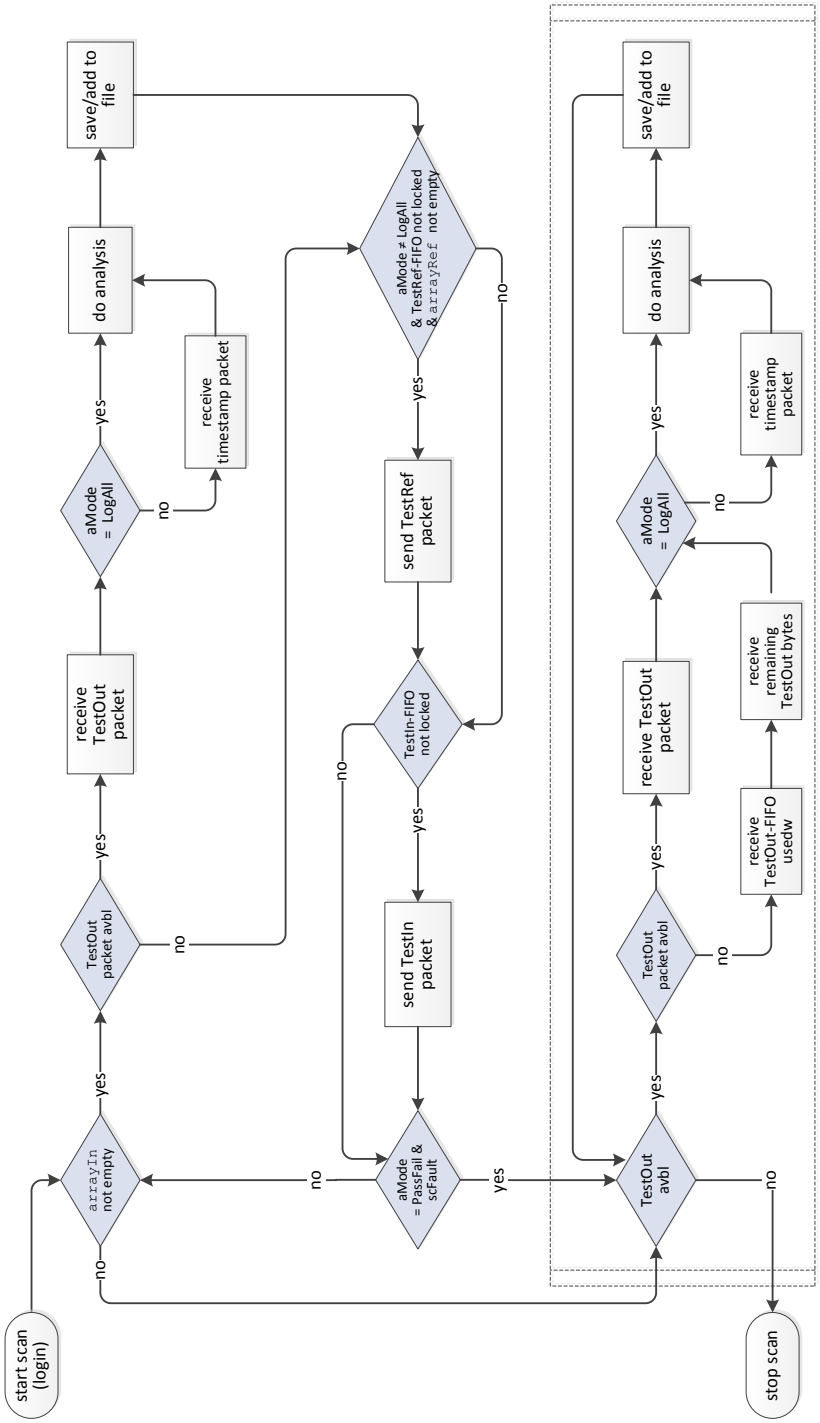
Parameter	Einheit	Wert
gMaxWithoutClockCorrectionFatal	Zykluspaar	2
gMaxWithoutClockCorrectionPassive	Zykluspaar	2
gNetworkManagementVectorLength	Byte	0
gNumberOfMinislots	#	210
gNumberOfStaticSlots	#	60
gOffsetCorrectionMax	µs	3,154
gOffsetCorrectionStart	MT	3632
gPayloadLengthStatic	Word	21
gSyncNodeMax	#	15

Protokoll-relevante lokale Parameter

Parameter	Einheit	Wert
pAllowColdstart	Boolean	1
pAllowHaltDueToClock	Boolean	1
pAllowPassiveToActive	Boolean	1
pChannels	A=1, B=2, A&B=3	1
pClusterDriftDamping	µT	2
pdAcceptedStartupRange	µT	213
pDecodingCorrection	µT	72
pDelayCompensation[A]	µT	10
pDelayCompensation[B]	µT	10
pdListenTimeout	µT	401202
pdMaxCycleLengthDeviation	µT	300
pdMaxDrift	µT	601
pdMicrotick	µs	0.025
pExternOffsetCorrection	µT	0
pExternRateCorrection	µT	0
pKeySlotUsedForStartup	Boolean	1
pKeySlotUsedForSync	Boolean	1
pLatestTx	Minislots	202
pMacroInitialOffset[A/B]	MT	4
pMicroInitialOffset[A/B]	µT	28
pMicroPerCycle	µT	200000
pMicroPerMacroNom	µT	55
pOffsetCorrectionOut	µT	127
pRateCorrectionOut	µT	601
pPayloadLengthDynMax	Word	16
pSamplesPerMicrotick	#	2
pSingleSlotEnabled	Boolean	0
pWakeupChannel	A=0, B=1	0
pWakeupPattern	#	2

A.6 Applikation

Kontrollfluss der Scan-Prozedur



A.7 Skripte

Für den ATPG-Prozess (Kapitel 5) wurden folgende Skripte verwendet.

Integration der Scan-Ketten

Um die Teststrukturen in eine Schaltung zu integrieren, werden mittels eines Skriptes die internen Flipflops um Multiplexer und Steuersignale ergänzt, um diese zu Scan-Ketten zu verknüpfen. Dies geschieht mittels des Skripts *scan4verilog.pl* unter Angabe der Kettenanzahl, der Technologiebibliothek und weiteren optionalen Parametern für die Scan-Flipflops.

Die zu modifizierende Schaltung muss für dieses Skript als strukturelle Verilog-Beschreibungen mit der zugrundeliegenden Technologiebibliothek *Si2 NanGate FreePDK45 Generic Open Cell Library* vorliegen. Daher hat zuvor ein Technologie-Mapping in dieses Format zu erfolgen.

Skriptaufruf: `scan4verilog.pl [-scan <value>] [-noqn] [-allrst] [-onlyff2scan] [-clk <value>] [-rst <value>] [-ckckn] [-powersim <value>] [-1500] [-nangate45] <verilog-file>`

Testmustererzeugung

Um Testmuster für die Schaltung erzeugen zu lassen wird Mentor Graphics FastScan genutzt. Das von FastScan ausgegebene Testmuster im ASCII-Format wird für weitere Schritte des ATPG-Prozesses mittels des Skriptes *fastscan2milef.pl* in das MILEF-Format (Mixed Level Automatic Test Pattern Generation) konvertiert. Die Testmusterzeugung und Konvertierung sind im Skript *patternGen.pl* zusammengefasst. Es wird schließlich das erzeugte FastScan-Testmuster, das MILEF-Testmuster und die zugehörigen Log-Dateien, die unter anderem Angaben zur Schaltung, Teststruktur, Test- und Fehlerüberdeckung enthalten, ausgegeben.

Skriptaufruf: `patternGen.pl <verilog-file>`

- Aufruf und Konfiguration von FastScan unter Angabe der Schaltung und der Technologiebibliothek
- Aufruf des Konvertierungsskriptes `fastscan2milef.pl`

Relaxation

Die Relaxation geschieht durch das an der BTU Cottbus entwickelte Programm FVHDL. In der folgenden Tabelle sind die für die Relaxation relevanten Parameter aufgelistet.

Skriptaufruf: `FVHDL.exe -verilog <verilog-file> -pat <pattern-file> -faultmodel <value> [-Nangate45nm] [-Orelax] [-OrelaxProc <value>] [-OrelaxDiv <value>] [-Omerge] [-OmergeRounds <value>]`

Parameter	Beschreibung
verilog <file>	Angabe der Netzliste im Verilog-Format.
vhdl <file>	Angabe der Netzliste im VHDL-Format.
pat <file>	Angabe der Pattern-Datei im MILEF-Format (jede Zeile enthält ein komplettes Testmuster als (0,1)-String).
faultmodel <value>	Das zugrundeliegende Fehlermodell (0 = *stuck-at*, 1 = *transition*).
Nangate45nm	Zur Interpretation der Netzliste wird 45nm-Bibliothek *Si2 NanGate FreePDK45 Generic Open Cell Library* genutzt. Ansonsten wird die Lattice-Bibliothek (VHDL) und eine 180nm-Bibiothek (Verilog) unterstützt.
Orelax	Relaxation durchführen.
OrelaxProc <value>	Versuch Testsatz mit höchstens <value> Prozent Care-Bits zu erhalten.
OrelaxDiv <value>	Anzahl Zwischenschritte pro Muster (falls *OrelaxProc* > 0), höhere Zahlen liefern typischerweise bessere Testsätze, bei höherem Zeitaufwand.
Omerge	Statische Kompaktierung durchführen. Dabei wird *OrelaxProc* als Care-Bit-Schranke verwendet.
OmergeRounds <value>	Anzahl der Merge-Runden. Nach jeder Runde wird hinsichtlich DCB optimiert.

Programmerzeugung für den Scan-Controller

Das Scan-Controller-Programm, also der komprimierte Testdatensatz, wird ebenfalls durch das FVHDL-Tool ermittelt.

Skriptaufruf: `FVHDL.exe -verilog <verilog-file> -pat <pattern-file> [-Nangate45nm] -scancontroller -storeSCInstructionStream -storeRAWpattern [-Opower]`

Parameter	Beschreibung
scancontroller	Scan-Controller-Programm erzeugen. Die Kompressionstechnik stellt dabei die Scan-Controller-Architektur dar. Hierzu muss die Anzahl an Scan-Ketten 2^n mit $n \geq 4$ sein. Ansonsten wird die *Alternierende Lauflängen-Codierung* verwendet.
pat <file>	Angabe der Pattern-Datei im MILEF-Format nach Relaxation (jede Zeile enthält ein komplettes Testmuster als (0,1,x)-String).
storeSCInstructionStream	Ausgabe der Scan-Controllers-Programms als (0,1)-Befehlsstrom.
storeRAWpattern	Ausgabe der unkomprimierten Testmuster-Muster (Rohdaten). Diese Muster sind also jeweils der komplette Inhalt der Scan-Ketten nach Abschluss der Shift-Phase.
SCLFSRcount <value>	Anzahl der bei der Optimierung parallel ablaufenden Scan-Controller-Emulationen.
LFSRfeedback <value>	Definition eines statischen Polynoms.
Opower	Power-Optimierung vornehmen. Das heißt, bei Wahl der Muster zugunsten der effizienteren hinsichtlich Stromverbrauch (geringere Anzahl an Transitionen) entscheiden.

B Anhang B
Analysedaten

In diesem Anhang befinden sich die in Kapitel 5 ausgewerteten Mess- und Berechnungsdaten in ausführlicher Form.

B.1 Testmuster zu 5.1.1.4

Testschaltung	Anzahl Testmuster		DCB-Anteil %
	FastScan	Relaxation u. Optimierung	
s9234	206	167	75,36
s9234 voroptimiert	194	169	75,74
s13207	309	262	94,23
s13207 voroptimiert	284	259	94,12
s15850	201	155	87,66
s15850 voroptimiert	179	151	87,62
s35932	76	36	77,91
s35932 voroptimiert	56	35	77,55
s38417	182	165	85,88
s38417 voroptimiert	182	164	85,78
b15	531	452	91,75
b15 voroptimiert	457	454	91,81
b17	592	562	92,50
b17 voroptimiert	575	564	92,54
b18	532	530	89,51
b18 voroptimiert	532	530	89,51
b19	539	538	89,54
b19 voroptimiert	538	538	89,54

Testschaltung	Anzahl Testmuster		DCB-Anteil %
	FastScan	Relaxation u. Optimierung	
b20	524	422	75,38
b20s voroptimiert	479	421	75,35
b21	478	410	76,20
b21 voroptimiert	448	409	76,24
b22	521	472	78,15
b22 voroptimiert	500	470	78,01
VLIW16S4	470	455	92,29
VLIW16S4 voroptimiert	455	455	92,29
VLIW32S4	717	716	97,84
VLIW32S4 voroptimiert	717	716	97,84
VLIW32S8	790	790	86,06
VLIW32S8 voroptimiert	790	790	86,06

B.2 Komprimierte Testmuster zu 5.1.1.5

Testschaltung		SFFs / Gatter	Scan-Ketten n_{Chain}	Scan-Tiefe m_{Chain}	Testmuster n_{Pat}		Fehlerüberdeckung (stuck-at) %	DCB-Anteil %	Größe d. Testsatzes		
					FastScan	Relaxation			Unkomprimiert [Byte]	Komprimiert [Byte]	Komprimierungsfaktor
ISCAS89	s5378	228 / 2779	32	8	160	89	94,59	69,69	5120	4330	1,18
			48	5					4800	4172	1,15
			64	4					5120	4134	1,24
	s9234	250 / 5597	32	8	206	167	92,27	75,36	6592	8642	0,76
			48	6					7416	8324	0,89
			64	4					6592	7860	0,84
	s13207	790 / 7951	32	25	309	262	94,10	94,23	26200	17868	1,45
			64	13					27248	12522	2,07
			96	9					28296	10960	2,36
			128	7					29344	10272	2,52
			256	4					33536	9184	2,82
			512	2					33536	8820	2,93
	s15850	684 / 9772	64	11	201	155	94,30	87,66	13640	9360	1,46
			96	8					14880	8814	1,69
			128	6					14880	8266	1,80
			256	3					14880	7464	1,99
			512	2					19840	8161	2,43
	s35932	2048 / 16065	64	32	76	36	87,86	77,91	9216	11828	0,78
			96	22					9504	11422	0,83
			128	16					9216	10910	0,84
			256	8					9216	10258	0,90
			512	4					9216	11129	0,83
	s38417	1742 / 22179	96	19	182	165	95,23	85,88	37620	26670	1,41
			100	18					37125	26568	1,40
			180	10					37125	25218	1,47
			192	10					39600	25096	1,58
			256	7					36960	24346	1,52
			512	4					42240	26121	1,61
ITC99	b14	277 / 4444	64	5	465	412	95,10	80,40	18600	13616	1,37
			128	3					22320	13280	1,68
			256	2					29760	13240	2,25
	b15s	485 / 8338	64	9	531	452	95,53	91,75	32544	15860	2,05
			128	5					36160	14222	2,54
			256	3					43392	14118	3,07
			512	2					57856	15948	3,63

Testschaltung		SFFs / Gatter	Scan-Ketten n_{Chain}	Scan-Tiefe m_{Chain}	Testmuster n_{Pat}		Fehlerüberdeckg. (stuck-at) %	DCB-Anteil %	Größe d. Testsatzes		
					FastScan	Relaxation			Unkomprimiert [Byte]	Komprimiert [Byte]	Komprimierungsfaktor
ITC99	b17	1452 / 22645	64	24	592	562	95,39	92,50	107904	46792	2,31
			128	12					107904	39668	2,72
			256	6					107904	38260	2,82
			512	3					107904	40696	2,65
			1024	2					143872	46625	3,09
	b18	2797 / 42015	128	22	532	530	95,38	89,51	186560	92732	2,01
			256	11					186560	90558	2,06
			512	6					203520	100168	2,03
			1024	3					203520	114315	1,78
	b19	5571 / 82109	128	44	539	538	95,32	89,54	378752	185962	2,04
			256	22					378752	180996	2,09
			512	11					378752	197910	1,91
			1024	6					413184	220678	1,87
	b20	522 / 8875	64	9	524	422	95,79	75,38	37728	27596	1,37
			128	5					41920	26896	1,56
			256	3					50304	26198	1,92
			512	2					67072	29291	2,29
	b21s	522 / 9259	64	9	478	410	95,87	76,20	34416	26940	1,28
			128	5					38240	26108	1,46
			256	3					45888	25404	1,81
			512	2					61184	28353	2,16
	b22	767 / 14282	64	12	521	472	95,96	78,15	45312	38916	1,16
			128	6					45312	37670	1,20
			256	3					45312	36202	1,25
			512	2					60416	40262	1,50
VLIW16S4		2904 / 34973	64	46	470	455	96,21	92,28	167440	75282	2,22
			128	23					167440	64616	2,60
			256	12					174720	61842	2,83
			320	10					182000	68803	2,65
			512	6					174720	65007	2,69
			1024	3					174720	69323	2,52
VLIW32S4		3303 / 81862	128	26	717	716	97,84	90,36	297856	135506	2,20
			256	13					297856	130248	2,29
			512	7					320768	139386	2,30
			1024	4					366592	149870	2,45
VLIW32S8		4419 / 172408	128	35	790	790	98,56	86,06	442400	282866	1,56
			256	18					455040	267900	1,70
			512	9					455040	286342	1,59
			1024	5					505600	310603	1,63

B.3 Testzeiten zu 5.2.2

VLIW16S4:

- Fehlermodell: *Stuck-at*,
- Scan-Modus: *Decompress*,
- Scan-Clock: $f_{\mathrm{SC}} = 20$ MHz,
- Scan-Ketten: $n_{\mathrm{Chain}} = 256$,
- Scan-Tiefe: $m_{\mathrm{Chain}} = 12$,
- Analyse-Rate/-Basis: $(a_{\mathrm{Rate}}, a_{\mathrm{Base}}) = (12, 11)$,
- Kompaktierung: *Mixed*,
- Paketgröße: 512 Byte,
- Testmusteranzahl: 454 (FC 96,21%),
- Fehler in Schaltung injiziert (*Stuck-at-0*), der insgesamt 454 verfälschte Testantworten zu Folge hat.

Analyse-modus	**TestIn [byte]**	**TestRef [Byte]**	**TestOut [Byte]**	**TS [Byte]**	**Gesamt [Byte]**	**Testzeit [s]**
LogAll	61.842 (484)	-	908 (8)	-	63.242	0,541
LogFaults	61.842 (484)	908 (8)	908 (8)	1.816 (16)	65.990	0,864
PassFail	512 (4)	512 (4)	4 (4)	8 (4)	1.052	0,118

EDT-Demonstrator:

- Fehlermodell: *Stuck-at*,
- Scan-Modus: *Decompress*,
- Scan-Clock: $f_{\mathrm{SC}} = 20$ MHz,
- Scan-Ketten: $n_{\mathrm{Chain}} = 900$,
- Scan-Tiefe: $m_{\mathrm{Chain}} = 103$,
- Analyse-Rate/-Basis: $(a_{\mathrm{Rate}}, a_{\mathrm{Base}}) = (1, 0)$,
- Paketgröße: 512 Byte,
- Testmusteranzahl: 140 (für den Chain-Test),
- Fehler in Schaltung injiziert (*Stuck-at-0*), der insgesamt 208 verfälschte Testantworten zu Folge hat.

Analyse-modus	**TestIn [Byte]**	**TestRef [Byte]**	**TestOut [Byte]**	**TS [Byte]**	**Gesamt [Byte]**	**Testzeit [s]**
LogAll	34.444 (276)	-	32.804 (260)	-	67.784	0,661
LogFaults	34.444 (276)	32.804 (260)	416 (4)	832 (8)	69.044	0,848
PassFail	$<$ 512 (4)	512 (4)	4 (4)	8 (4)	$<$ 1.052	0,012

B.4 Testeigenschaften zu 5.2.3

In der folgenden Tabelle sind mehrere ISCAS89- und ITS99-Schaltungen sowie diverse Auslegungen des VLIW-Prozessors zusammengefasst. Für diese wurde mittels des vorgestellten ATPG-Prozesses je ein Testmustersatz und ein komprimierter Testsatz erzeugt. In der Tabelle sind pro Schaltung mehrere Auslegungen der Scan-Strukturen dargestellt. Zu jeder Version sind hier die Anzahl der Testmuster, die erreichte Fehlerüberdeckung und die Größe des komprimierten Testsatzes angegeben. Es wurde zudem eine on-chip Testzeit berechnet, um die benötigte Zeit für einen Selbsttest, der den kompletten komprimierten Testsatz on-chip zur Verfügung hätte, aufzuzeigen. Dieser Berechnung lag dabei eine Scan-Taktfrequenz von 20 MHz zugrunde. Die Testzeiten für den Testzugang über USB oder FlexRay wurden für einige ausgewählte Schaltungsversionen durch den in Abschnitt 5.2.3 vorgestellten Testaufbau ermittelt. Es wurde nicht jede Schaltungsversion hinsichtlich Testlaufzeit untersucht, da die Messungen nach erforderlicher aufwendiger Synthese keinen Erkenntnisgewinn erwarten lassen.

Testschaltung		**Scan-Ketten** n_{Chain}	**Scan-Tiefe** m_{Chain}	**Testmuster** n_{Pat}	**Fehlerüberdeckung (stuck-at) %**	**Testsatz [KByte]**	**Zeitaufwand Selbsttest [ms]**	**Testzeit [s]**			
								USB		**FlexRay**	
								$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$	$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$
ISCAS89	s13207	64	13	262	94,10	12,23	0,313	-	-	-	-
		96	9			10,70	0,274	-	-	-	-
		128	7			10,03	0,256	0,138	0,104	0,587	0,475
		256	4			8,97	0,230	0,119	0,089	0,507	0,396
	s15850	64	11	155	94,30	9,14	0,234	0,126	0,093	-	-
		96	8			8,61	0,220	-	-	-	-
		128	6			8,07	0,207	0,102	0,084	-	-
		256	3			7,29	0,187	-	-	-	-
	s35932	64	32	36	87,86	11,55	0,296	-	-	-	-
		96	22			11,15	0,286	-	-	-	-
		128	16			10,65	0,273	0,144	0,127	-	-
		256	8			10,02	0,256	0,131	0,119	-	-
	s38417	100	18	165	95,23	25,95	0,664	-	-	-	-
		192	10			24,66	0,627	-	-	-	-
		256	7			23,77	0,609	0,259	0,228	1,216	1,125
ITC99	b17	128	12	562	95,39	38,74	0,992	-	-	-	-
		256	6			37,36	0,957	0,454	0,398	-	-
		512	3			39,74	1,017	-	-	-	-
	b18	128	22	530	95,38	90,56	2,318	-	-	-	-
		256	11			88,44	2,264	1,049	0,906	4,329	3,592
		512	6			97,82	2,504	1,087	1,029	-	-

Testschaltung		**Scan-Ketten** n_{Chain}	**Scan-Tiefe** m_{Chain}	**Testmuster** n_{Pat}	**Fehlerüberdeckung (stuck-at) %**	**Testsatz [KByte]**	**Zeitaufwand Selbsttest [ms]**	**Testzeit [s]**			
								USB		**FlexRay**	
								$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$	$a_{\text{Rate}} = 1$	$a_{\text{Rate}} = m_{\text{Chain}}$
ITC99	b19	256	22	538	95,32	176,75	4,525	1,994	1,774	9,599	8,543
		512	11			193,27	4,948	2,171	2,059	8,715	8,152
		1024	6			215,51	5,517	-	-	-	-
	b21	128	5	410	95,87	25,50	0,653	-	-	-	-
		256	3			24,81	0,635	-	-	-	-
	b22	128	6	472	95,96	36,79	0,942	-	-	-	-
		256	3			35,35	0,905	-	-	-	-
VLIW16S4		128	23	454	96,21	63,10	1,615	0,785	0,561	-	-
		256	12			60,39	1,546	0,665	0,541	3,142	2,509
		512	6			63,48	1,625	0,673	0,620	-	-
VLIW32S4		128	26	716	97,84	132,33	3,388	-	-	-	-
		256	13			127,20	3,256	1,467	1,300	5,992	5,313
		512	7			136,12	3,485	1,518	1,435	-	-
		1024	4			146,36	3,747	1,631	1,545	-	-
VLIW32S8		128	35	790	98,56	276,24	7,072	-	-	-	-
		256	18			261,62	6,698	3,050	2,724	12,159	11,067
		512	9			279,63	7,159	3,098	2,949	13,327	12,863
		1024	5			303,32	7,765	3,251	3,164	-	-
EDT-Demonstrator		900	103	140	~100	33,64	0,861	0,661	-	2,734	-
				1400	-	336,33	8,610	6,524	-	28,223	-

C Anhang C
Software & Hardware

Für die Umsetzung des Konzeptes wurden die folgenden kommerziellen Hard- und Softwaretools verwendet:

Simulation und Validierung

- Mentor Graphics® ModelSim® SE-64 10.1aRevision 2012.02

Synthese und Fitting des VHDL-Designs auf FPGA

- Altera® Quartus II Version 11.1sp2 (Build 173)
- Cadence® Encounter RTL Compiler

Tracing und Validierung des Designs auf dem FPGA

- Altera® Quartus SignalTab II Logic Analyzer

Tracing und Validierung der Kommunikationsschnittstellen

- USB Protocol Analyzer USBlyzer 2.0 Build 20
- Vector Informatik® CANalyzer 8.0.35
- Vector Informatik® FlexRay™ Interface Busanalyzer VN7600
  - CAN, 3 Kanäle
  - FlexRay, 2 Kanäle (A/B)
- Hameg 100 MHz Analog/Digital Oszilloskop HM1007

FPGA zum Validieren des Designs in Hardware

- Altera® Cyclone™ II EP2C70F896C6N, Entwicklungsboard DE2-70
- Aufsteckplatine für zusätzliche Schnittstellen
  - Texas Instruments TUSB1106 Universal Serial Bus (USB) Transceiver, USB 2.0-Buchse Typ Standard-B (oder USB 2.0-Buchse Typ Mini-B 5-polig)
  - 2x NXP Semiconductors TJA1080A FlexRay Transceiver, 2x D-Sub DE9-Buchse
  - 2x IFX1050G High Speed CAN-Transceiver, 2x Stiftleiste 2-polig
  - Oszillator Crystek Crystals CPRO33-80.000, 80 MHz

vormals auch:

- Altera® Cyclone™ II EP2C70F896C6N, Entwicklungsboard DE2-70 mit Aufsteckplatine
  - Philips ISP1106DH (TSSOP16) USB Transceiver, USB 2.0-Buchse Typ Standard-B
  - 2x Austriamicrosystems AS8221 FlexRay Transceiver, 2x Stiftleiste 2-polig
- Altera® Apex™ EP20K300EQC240-2X, Entwicklungsboard Digilab 20Kx240
  - Philips PDIUSBP11A USB Transceiver, USB 2.0-Buchse Typ Standard-B

Testmustergenerierung

- Mentor Graphics® FastScan™ v8.2009 1.10
- Si2 NanGate FreePDK45 Generic Open Cell Library

Implementierung der Applikation

- Java™ Standard Edition Development Kit (JDK) 1.7.0.15 (64-bit)
- Java™ Standard Edition Runtime Environment (JRE) 7.0.15 (64-bit)
- Eclipse® Source Development Kit (SDK) Version 3.7.1 (Build M20110909-1335)
- USB Bibliothek Libusb- Version 1.2.6.0, gemäß den Bedingungen der GNU Lesser General Public License (LGPL)
  - Java libusb/libusb-win32 Wrapper