UNIVERSITÄT LEIPZIG

Fakultät für Mathematik und Informatik

Institut für Informatik

Hardware-Debugging durch die Kombination von Emulation und Simulation

Diplomarbeit

Aufgabenstellung:

Prof. Dr. U. Kebschull

vorgelegt von:

Jens Frauenschläger

Leipzig, November 2002

Vorwort

Im Rahmen des Schaltkreisentwurfes treten immer auch Fehler auf. Alle bisher bekannten Methoden zur Fehlererkennung besitzen trotz positiver Eigenschaften auch zahlreiche Einschränkungen und Nachteile.
In der vorliegenden Arbeit wird der von Prof. Kebschull patentierte Ansatz [Pat01a] für das Hardware-Debugging untersucht, implementiert und getestet. Auf die dazu notwendige neuartige Verbindung zwischen Emulation und Simulation geht diese Arbeit im Besonderen ein. Die erforderlichen Modifikationen an einer Schaltung sowie die dafür entwickelten Softwarekomponenten werden im Anschluss erklärt.
Abschließend stellt diese Diplomarbeit den Funktionsumfang des entwickelten Programms NIHD© (***N**ew **I**ntegrated **H**ardware **D**ebugger*) anhand verschiedener praktischer Beispiele dar und vergleicht deren Messergebnisse miteinander, um Rückschlüsse auf des Gesamtverhalten zu erlauben.

Folgenden Personen möchte ich an dieser Stelle explizit danken:

- Herrn Prof. Dr. U. Kebschull für die Aufgabenstellung und die Unterstützung der Arbeit,
- Herrn Dipl.-Inf. C. Nitsch für seine aktive Hilfe bei die technischen Umsetzung der Zielarchitektur,
- sowie allen, die mich bei dieser sehr umfangreichen Arbeit unterstützt haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

„*...Zugriffsfehler im Modul xyz...*“ - solche oder ähnliche Fehlermeldungen sehen Benutzer von Computern meist dann, wenn sie versuchen mit einem Programm Operationen auszuführen, die fehlerhaft programmiert wurden.

Die Ursachen für derartige Fehler sind dabei recht vielfältig. Oftmals sind sie bereits in der benutzten Programmiersprache zu suchen, dazu kommen zusätzlich Fehler in den Compilern, Beschränkungen in dem Betriebssystem sowie Seiteneffekte anderer Programme, wie z.B. falsches Speichermanagement.

Angeführt wird die Liste möglicher Fehlerquellen allerdings vom Menschen selbst. Wobei auch hier noch zu unterscheiden ist, ob der Mensch einen Fehler aus Unwissenheit gemacht hat, oder ob er dessen Relevanz einfach unterschätzt hat. In beiden Fällen wird es immer schwer sein in einem Endprodukt die genaue Ursache eines Fehlverhaltens zu lokalisieren. Ähnlich wie es zu Problemen bei der Erstellung von Software kommen kann, treten natürlich auch während der Entwicklung einer Schaltung grundlegende Schwierigkeiten auf.

Im folgenden Kapitel kommt es zu grundsätzlichen Überlegungen wie man derartigen Tendenzen wirksam begegnen kann. Dabei bietet die vorliegende Arbeit eine Möglichkeit zukünftig Fehler beim Schaltkreisentwurf effektiver zu finden und zu beseitigen.

1.1 Warum Hardware-Debugging?

Die Gründe für ein Fehlverhalten einer Schaltung sind heutzutage vielfältig. Einerseits ist dabei die große Anzahl der verschiedenen Zielarchitekturen zu nennen. Jeder der zahlreichen Schaltkreisfamilien liegen andere, zumeist recht unterschiedliche Technologien zu Grunde die dazu führen können, dass eine Schaltung teilweise anders als es sich der Entwickler vorgestellt hat umgesetzt wird. Andererseits spielen hierbei auch die Programme für die Erzeugung einer Schaltungsbeschreibung eine wichtige Rolle. Oftmals unterscheiden sich die Ergebnisse der verschiedenen Entwicklungsprogramme trotz gleicher Zielarchitektur erheblich. Der Vorteil von Software ist, dass sie, einmal entwickelt, immer in dieser „Form“ existieren wird. Später erkannte Fehlfunktionen kann man durch das Überschreiben der fehlerhaften Programmteile nachträglich beheben, ohne damit das komplette System zu zerstören.

Bei Hardware hingegen ist es so, dass nach einer Prototypenphase oft dazu übergegangen wird, die Schaltungen in großer Stückzahl dauerhaft auf einer bestimmten Zielarchitektur unterzubringen. Danach sind keinerlei Veränderungen mehr an den Schaltkreisen möglich. Sowohl in Software, als auch in Hardware werden die Hersteller immer vor die Frage gestellt, ab wann ihr Produkt als „fehlerfrei“ gilt. Natürlich ist auch die Betrachtung der finanziellen Aspekte dabei nicht zu vernachlässigen.

Selbst nach noch so intensiven Testphasen kann man ein Fehlverhalten nie vollkommen ausschließen. Mitunter treten auch Fehler auf, die nicht reproduzierbar sind. Nach [Chi02] sinkt jedoch die Anzahl der Fehler, je länger, genauer und häufiger man testet. Speziell für Hardware ist es daher notwendig so viele Probleme wie möglich bereits im Prototypenstadium zu beheben, bevor die Massenproduktion überhaupt beginnt.

Moderne Systeme bestehen jedoch nicht mehr nur aus einem einzigen Chip, sondern stellen vielmehr eine Verbindung von zahlreichen Bausteinen dar. Viele dieser Chips interagieren untereinander mit Hilfe verschiedener Steuerprogramme. Eine Möglichkeit zur Fehlerminimierung kann nun z.B. darin bestehen, den Anteil von Hardware an einem derartigen Gesamtsystem zu verringern und stattdessen weniger zeitkritischere Hardwaresegmente in die Software zu verlagern. Heutige Software-Systeme erlauben bereits Geschwindigkeiten die mit den Performance-Eigenschaften von Hardware vergleichbar sind.

Im Fehlerfall müsste man nun nicht mehr große Teile der statisch programmierten Hardware austauschen, sondern modifiziert die entsprechenden Segmente in der Software.

Gegen diese Idee als einzige Lösung zur Behebung von Fehlverhalten in Hardware sprechen jedoch einige Argumente.
Zwar kann man natürlich Teile der Hardware auch in Software auslagern, dennoch bleiben genügend mögliche Fehlerquellen in der Hardware bestehen. Des weiteren wird eine Bearbeitung in Software zwar eventuelle Hardwarefehler beseitigen, jedoch können dafür neue Fehler in der Software auftreten. Zusätzlich dazu stellt die Interaktion zwischen Hard- und Software einen erheblich höheren Entwicklungsaufwand dar. Alles in allem kann dies nicht der Ausweg sein!
Ein sinnvoller Ausweg, der sich bietet, ist das sog. *Debugging*. Hinter diesem Begriff, der ursprünglich aus der Softwareentwicklung stammt, steht die Möglichkeit u.a. durch die Überwachung von Signalen und Variablen den Programmverlauf schrittweise zu verfolgen und Fehlverhalten zu entdecken bzw. zu beseitigen. Dieser Grundgedanke kann recht einfach auch auf den Entwicklungsprozess bei Hardware angewendet werden.
Um eine große Anzahl von Zielarchitekturen abzudecken, ist es notwendig ein passendes Ausgangsformat zu wählen. Aus diesem Grund bietet sich eine standardisierte HDL (engl. ***H**ardware **D**escription **L**anguage*) an. Ein großer Vorteil für die vorliegende Arbeit war dabei, dass die verwendete Hardwarebeschreibungssprache VHDL weltweit bereits seit vielen Jahren vereinheitlicht wird. Fast alle Anbieter von Zielarchitekturen bieten zahlreiche Schnittstellen zu dieser Sprache. Innerhalb dieses Ausgangsformates sollte man in Analogie zum Debugging bei Software die Möglichkeit haben, aus einer Reihe von Signalen, Variablen und Registern auszuwählen. Diese können beliebig überwacht werden.
Zusätzlich zu diesen Gemeinsamkeiten müssen natürlich auch einige Unterschiede in der Realisierung einer Schaltung beachtet werden. Hierzu zählen u.a. Fragen nach dem Umfang der dafür notwendigen Zusatzlogik, nach der Reaktionszeit der Schaltung oder nach der Anzahl der zu überwachenden Zeiteinheiten.

1.2 Motivation

In modernen Hardwareanwendungen findet man eine sehr hohe Integrationsdichte der Schaltelemente vor. Um innerhalb eines derartig komplexen Systems fehlerhafte Einheiten aufzuspüren, benötigt man einiges an Technik. Heutzutage benutzt man dazu z.B. einen sog. LA (engl. ***L**ogic **A**nalyzer*), Oszilloskope, oder man greift auf Simulatoren zurück.

Ein Simulator ermöglicht es dem Programmierer eine Schaltung ähnlich wie Software zu überwachen. Dies geschieht allerdings komplett außerhalb des IC's (engl. ***I**ntegrated **C**ircut's*). Die gesamte Schaltung inklusive der Logik wird innerhalb einer Simulationsumgebung auf einem Computer nachgebildet. Dieses Unterfangen setzt ein erhebliches Maß an Performance des Computers voraus. Der effektive Einsatz eines Simulators bietet sich deshalb nur für kleinere Schaltungen, oder Teile eines großen Gesamtsystems an.
Die andere Alternative dazu ist ein Logic Analyzer. Mit Hilfe eines solchen Gerätes hat der Entwickler die Möglichkeit die Werte einer Schaltung direkt an den Pins des ICs zu messen, während die Hardware aktiv ist. Allerdings stößt man auch bei diesem Ansatz schnell auf dessen Grenzen. Die Werte von Variablen, die in der Definition der Schaltung benutzt wurden, sind nicht ohne erheblichen Mehraufwand an den Pins auslesbar. Es fehlt hierbei jeder Bezug zwischen ihnen und den an den Pins anliegenden Signalwerten. Aus diesen Erkenntnissen ergibt sich die zwingende Notwendigkeit der Entwicklung eines umfassenderen Lösungsansatzes.

Für diese Arbeit bildeten vorerst wiederbeschreibbare Zielarchitekturen, wie z.B. FPGAs (engl. ***F**ield **P**rogramable **G**ate **A**rray's*) die Grundlage. Diese Beschränkung gilt aber nur der Einfachheit halber. Zusätzlich dazu ist eine Verschmelzung der jeweiligen Vorteile der beiden o.g. Ansätze wünschenswert. Dies bedeutet im Einzelnen, dass die aktive Schaltung sich auf einem beliebigen FPGA befindet, während die Auswertung der gewünschten Signale mit Hilfe eines Simulators am Computer erfolgt. Die dafür notwendigen technischen Grundlagen zur Umsetzung dieser Diplomarbeit standen in der Abteilung Technische Informatik der Universität Leipzig zur Verfügung.
Das zur Realisierung dieses Ziels eine Modifikation der Ausgangsschaltung unumgänglich ist, steht außer Frage. Um den Hardwareentwicklern allerdings die Arbeit zu erleichtern, sollten diese Veränderungen automatisch durchgeführt werden. Für einen Bezug der an den Pins anliegenden Signalwerte zu den im Quellcode verwendeten Variablen bedarf es mehrerer neuer Bibliothekskomponenten. Die Schnittstelle für die Kommunikation zwischen dem FPGA und dem Computer wird auf der Hardware in einer Hardwarebeschreibungssprache und seitens des Computers mit Hilfe der Programmiersprachen *C* bzw. *Java* realisiert.

1.3 Aufgabenstellung

Während des Prozesses der Implementierung von Komponenten in digitalen Systemen treten immer wieder die gleichen Schwierigkeiten auf. Auf der einen Seite kann man auf Simulatoren zurückgreifen, die den Vorteil haben, dass man das System sehr intensiv überwachen kann. Der Nachteil hierbei aber ist, dass die Simulationsgeschwindigkeit mit steigendem Ausmaß der Kontrolle abnimmt.

Alternativ kann man auf Hardware zurückgreifen, auf der die zu untersuchende Schaltung emuliert wird. Hierbei sind natürlich wesentlich höhere Taktraten als mit der Simulation möglich. Allerdings fehlt dann die Möglichkeit, die Werte von internen Signalen auszuwerten. In der vorliegenden Arbeit wurden diese beiden Ansätze vereint. Dabei werden die Schaltungen derart modifiziert, dass sie bis zur Feststellung eines Fehlverhaltens auf einer vorhandenen Hardware emuliert werden.

Ein Fehler liegt dann vor, wenn eine logische Bedingung erfüllt wird. Diese logischen Bedingungen werden mit Hilfe von sog. BP (engl. ***B****reak****p****oint*) Komponenten überwacht. Sobald eine Fehlerbedingung erreicht ist, werden die aktuellen Zustände der Komponenten der Schaltung in der Emulationshardware gespeichert.

Anschließend werden diese gespeicherten Werte an den angeschlossenen Simulator übergeben. Im Simulator läuft zu diesem Zeitpunkt das gleiche Schaltungsdesign, das auf der Hardware emuliert wird. Mit Hilfe des Simulators ist der Entwickler nun in der Lage, die internen Signale der Schaltung zu betrachten. Anhand der an der emulierten Schaltung vorgenommenen Modifikationen wird nicht nur der aktuelle Wert zum Zeitpunkt des Breakpoints gespeichert, sondern auch eine festgelegte Anzahl von Werten davor und danach.

Somit hat man im Simulator die Möglichkeit das Verhalten der Schaltung einige Zeittakte vor und nach dem BP zu observieren. Die Speicherung der Schaltungszustände über mehrere Takte hinweg erfordert neben den neuen BP-Komponenten noch weitere.

So ist es z.B. notwendig angepasste Schieberegister zu implementieren um für jedes zu überwachende Signal die erforderliche Menge externer Signalwerte zu erhalten. Gleichzeitig muss man eine geeignete Steuerlogik zur Verknüpfung der BP-Komponenten mit den Puffern entwickeln.

Damit der Benutzer einen strukturierten Überblick über den Aufbau der Datei mit der Hardwarebeschreibung bekommt, wurde als Ausgangspunkt ein Parser für das Dateiformat entwickelt. Dessen Einsatzgebiet sollte vorerst nur auf einem speziellen Zwischenformat,

das während der Synthese erzeugt wird, liegen. Obwohl die Erweiterung der Schaltungslogik um die genannten Komponenten hauptsächlich automatisch erfolgt, muss der Benutzer dennoch einige Grundangaben machen. Hierzu sollen u.a. die Liste der überwachbaren internen Signalnamen zählen, die Auswahl eventuell bereits vorhandener Steuersignale sowie das Durchlaufen der einzelnen Zeittakte.
Um das zu erreichen wird dem Nutzer eine einfache Schnittstelle geboten, die grafisch realisiert wird. Als Grundlage für die Schaltungsbeschreibung wird eine HDL genutzt. Zur Synthese selbiger werden Programme der Firma *Xilinx* eingesetzt.

1.4 Was kann man damit erreichen?

Mit Hilfe der vorliegenden Arbeit hat der Entwickler die Möglichkeit Fehler in einer bestehenden Schaltung wesentlich leichter zu lokalisieren. Durch den Einsatz dieser patentierten Technologie muss der Hardware Designer nicht mehr, wie heutzutage üblich, getrennt zwischen den beiden Varianten (Emulator, Simulator) zum Hardware-Debugging wählen.
Vielmehr kann er durch die mit dieser Arbeit gelungenen Verknüpfung von Simulation und Emulation viel effizienter Fehler aufspüren. Das Auswerten von internen Werten, die bei der Fehleranalyse eine große Rolle spielen, stellt keine Schwierigkeit mehr dar!
Es bedarf keiner kostenintensiven Technik, wie z.B. Logic Analyzer. Aufgrund der Verwendung von Java für das Verarbeiten und Verändern der Hardwarebeschreibung ist dieses Verfahren unabhängig von der Plattform auf der sich der Simulator befindet. Der hier vorgestellte Ansatz lässt sich beliebig auf andere Zielarchitekturen als das FPGA umsetzen. Eine Erweiterung auf moderne Kommunikationsschnittstellen wie z.B. JTAG (engl. ***J**oint **T**est **A**ction **G**roup*) ist in den kommenden Versionen vorgesehen.

1.5 Was wurde im Rahmen der Diplomarbeit erreicht?

Die vorliegende Arbeit zeigt, dass es durchaus Sinn macht, zwei prinzipell getrennte Prozesse, wie Emulation und Simulation, zu kombinieren. Mit Hilfe weniger Veränderungen in der Hardwarebeschreibung hat der Benutzer Einfluss darauf, welche internen Werte er überwachen möchte. Dieser Prozess geschieht weitgehend automatisch.

Da bereits leistungsfähige Simulatoren existieren, ist die in dieser Arbeit entwickelte Software daran angepasst. Der Parser sowie große Teile der entwickelten Software sind in Java implementiert. Einzig die Steuerung der Schaltung ist in der Sprache C umgesetzt.
Das hierfür benutzte FPGA der Firma *Xilinx* befindet sich auf einem für solche Anwendungen vorgesehenen Prototypen-Board. Die Steuerbefehle des Computers werden mit Hilfe eines anderen Boards an das FPGA herangeführt. Analog dazu ist auch das Auslesen der Signalwerte implementiert. Dem Nutzer steht für die wenigen notwendigen Eingaben eine grafische Oberfläche zur Verfügung. Die ausgelesenen Werte werden in ein passendes Format für den Simulator *ModelSim* umgewandelt und können anschließend dort ausgewertet werden. Die Belegungen der Signale und Flipflops sowohl vor dem Auftreten eines Breakpoints, als auch danach sind problemlos auszuwerten.

1.6 Aufbau dieser Arbeit

Die vorliegende Arbeit ist folgendermaßen aufgebaut: Nachdem im ersten Kapitel die Notwendigkeit des Hardware-Debuggings dargelegt wird, werden diese Ausführungen durch die derzeitigen praktischen Ansätze untermauert und die dabei benutzte Begriffswelt abgegrenzt.
Das dritte Kapitel beschreibt ausführlich die für das Verständnis dieser Arbeit wichtigen Grundlagen. Der eigentliche Grundgedanke der benutzten Methode wird im vierten Kapitel dargelegt. Das fünfte Kapitel beschäftigt sich mit den technischen Einzelheiten, insbesondere im Rahmen der entwickelten Software. Zum Abschluss werden Tests durchgeführt und deren Ergebnisse zusammengefasst, um danach einen Ausblick auf die weiteren Entwicklungen zu geben.

Kapitel 2

Abgrenzung

Wenn man eine Idee hat und diese umsetzen möchte, muss man sich zuerst einen Überblick verschaffen, inwieweit vergleichbare Ideen bereits erforscht oder umgesetzt wurden. Auf den ersten Blick stellt man häufig fest, dass vorhandene Ideen sich stark ähneln. Erst auf den zweiten Blick erkennt man meistens gravierende Fehler oder Nachteile der bereits existierenden Implementierungen.

Dieses Kapitel soll daher zuerst einen Überblick über bestehende Ansätze gewähren. Bei den erwähnten Ansätzen handelt es sich sowohl um bereits kommerziell vertriebene Produkte, als auch um theoretische Grundlagen, die derzeit noch erforscht werden.

Im zweiten Teil des Kapitels wird geklärt, worin sich die Begriffe Simulation, Emulation und Debugging unterscheiden. Bezüglich der Simulation wird auf die möglichen Konzepte näher eingegangen. Zusätzlich bedarf es zum besseren Verständnis dieser Arbeit einer Betrachtung der verschiedenen Abstraktionsebenen.

Da es sich bei der verwendeten Hardware um eine Emulationsplattform für das sog. *Rapid Prototyping* handelt, wird tiefer auf die Vor- und Nachteile der Emulation sowie die Gründe für das Prototyping eingegangen. Abschließend untersucht ein Teilkapitel die generellen Eigenschaften des Debuggings.

2.1 Begriffsbildung

2.1.1 Simulation

Insbesondere bei der Fehlersuche in Hardwaresystemen spielt die Simulation eine entscheidende Rolle. Generell handelt es sich bei der Simulation, unabhängig von der Abstraktionsebene, um die Darstellung einer Schaltungsbeschreibung in Software. Dabei werden sämtliche Funktionen der Schaltung komplett nachgebildet. Drei verschiedene Abstraktionsebenen lassen sich nach [Ulm98] bei der Simulation unterscheiden. Die simpelste Darstellungsform ist die Simulation auf Gatterebene, wobei dieser Beschreibung eine Netzliste bestehend aus einfachen Gattern zugrunde liegt. Das zeitliche Verhalten ist dabei nur anhand der einzelnen Gatterlaufzeiten geschätzt. Andererseits kann diese Netzliste auch auf einem konkreten Layout basieren. Dann sind dieser Form auch die echten Gatterlaufzeiten der Zielarchitektur zugrunde gelegt worden. Die nächst höhere Abstraktionsstufe ist die Simulation auf RTL (engl. ***R**egister **T**ransfer **L**evel*). In dieser Stufe der Simulation wird die Richtigkeit der Schaltungsbeschreibung noch vor der Logiksynthese überprüft. Der implizite und explizite Bezug auf bestimmte Taktzyklen ist die Grundlage von RTL-Beschreibungen.

Die höchste Stufe der Simulation ist die sog. High-Level-Simulation. Sie arbeitet auf der Grundlage einer völlig abstrakten Verhaltensbeschreibung und dient deren Überprüfung. Die Angaben über die Realisierung des beschriebenen Verhaltens fehlen gänzlich. Wie genau läuft nun aber eine Simulation ab?

Heutigen Schaltungssimulatoren liegen laut [Ulm98] entsprechende weltweite Standards der IEEE zugrunde. Darin wird eine Simulation, insbesondere auf der RTL-Ebene, in 2 Stufen eingeteilt. Eine Simulation beginnt immer mit einer Initialisierung. Dabei werden alle Signale und Variablen mit den vordefinierten Anfangswerten belegt. Die Prozesse werden komplett ausgeführt, jedoch die berechneten Werte noch nicht aktualisiert. Daran schliest sich die eigentliche Simulationsphase an. Jetzt werden die in Prozessen berechneten Signalwerte aktualisiert und Prozesse erneut angestoßen, sobald das dafür erforderliche Ereignis auftritt und ein Weiterlaufen der Zeit veranlasst.

Grundsätzlich lassen sich nach [Ulm98] und [Sim02] zwei verschiedene Simulationskonzepte aus diesem IEEE-Standard ableiten. Einerseits der Ereignis-gesteuerte (engl. *Event-driven*), andererseits der Zyklus-basierende (engl. *Cycle-based*) Ansatz. Ein Ereignis-gesteuerter Simulator reagiert auf alle Ereignisse während eines Taktes und verarbeitet diese unter Verwendung eventueller Verzögerungszeiten (engl.

Delays) sofort. Der Vorteil dieses Systems ist sein Anwendungsgebiet, da es auf auf allen drei Abstraktionsebenen angewandt werden.
Bei einem Zyklus-basierendem Simulator werden alle Angaben über das zeitliche Verhalten einer Schaltung ignoriert und die Stabilität der Signale zwischen den Taktflanken vorrausgesetzt. Bei einer steigenden Taktflanke findet die Berechnung der neuen Signalwerte statt. Dieser Ansatz eignet sich nur für die High-Level oder RTL-Simulation, jedoch nicht für die Simulation auf Gatterebene.

2.1.2 Emulation

Unter Emulation versteht man das Ausführen einer Schaltung in einer anderen Hardwareumgebung, einer sog. *Emulationsplattform*, zum Zwecke umfangreicher Tests der Funktionen. Emulation bzw. das Testen von Prototypen (engl. *Prototyping*) bei einem FPGA wird benutzt, um eine Überprüfung der darauf befindlichen Schaltung in harter Echtzeit bzw. Fast-Echtzeit durchzuführen. Falls die Zielarchitektur bereits ein gewisses Minimum an Geschwindigkeit bietet, so kann man entsprechend die Schaltung auch direkt auf der zukünftigen Umgebung überprüfen. Der Zeitaufwand für das Erstellen einer Testumgebung, die zusätzlich fehleranfällig ist, wird dadurch erheblich reduziert.
Der größte Nachteil solcher Ansätze ist der hohe Zeitaufwand für die begleitenden Aufgaben, insbesondere bei größeren Designs. Nach jeder Entdeckung und Behebung eines Fehlers ist der Entwickler gezwungen, den kompletten Prozess zur Erstellung einer hardwarefähigen Schaltungsbeschreibung zu durchlaufen. Der Vorteil industriell eingesetzter Emulationsplattformen ist laut [Dat01] ihre, im Vergleich zu Simulatoren, einfache Handhabung und die recht gute Unterstützung bei der Fehlersuche. Nachteilig an Emulationssystemen sind der hohe Preis und die mitunter zu geringen Taktfrequenzen bei Echtzeitanwendungen.
Durch die ständige Weiterentwicklung der FPGA-Technologie erreichten die Emulationsplattformen einen ausgesprochen hohen Grad an Komplexität und Performance. Dadurch wurde ihre Stellung speziell bei Anwendungen der Telekommunikation gestärkt. Insbesondere werden sie dort bei der Entwicklung von Prototypen und zur Erhöhung der Geschwindigkeit bei der Fehlersuche eingesetzt. Auf FPGAs basierende Prototypen haben jedoch den Nachteil, dass das Hardware-Debugging nicht ausreichend unterstützt wird. [Dat01]

Industriell wird die Emulation hauptsächlich bei der Entwicklung komplexer ASICs (engl. ***A**pplication **S**pecific **I**ntegrated **C**ircuts*) oder Prozessoren eingesetzt. Durch diese Methode können die eigentlichen Bausteine sowie die dafür notwendige Systemsoftware relativ schnell und kostengünstig getestet bzw. angepasst werden. Obwohl dieses Argument bedingt auch auf die Simulation zutrifft, so ist die Effizienz aufgrund des Geschwindigkeitsvorteils wesentlich größer. Wegen der hohen Entwicklungskosten für Emulationsplattformen lohnt sich deren Verwendung jedoch nur im großem Maßstab.

2.1.3 Debugging

Unter Debugging versteht man einen Prozess bestehend aus der Fehlerüberprüfung, -erkennung und -behebung. Dieser Prozess wird bei Software schon seit längerem angewendet. Trotz der längeren Erfahrung in dieser Branche kommt es recht häufig vor, das Fehler in der Herstellungsphase nicht entdeckt wurden. Dann bedeutet das aber nicht die Überarbeitung der gesamten Software, sondern nur den „raschen“ Austausch der fehlerhaften Softwarekomponente.

In Hardware ist dieses Vorgehen nicht ohne weiteres anwendbar. Hier steht zur Fehlersuche eine ganze Reihe von Methoden zur Verfügung. Die Auswahl an möglichen Methoden hierfür reicht von einer einfachen Simulation über die Emulation bis hin zu kombinierten Weiterentwicklungen daraus. Wie aus Abschnitt 2.1 zu erkennen ist, existieren zwei grundsätzliche Arten des Debugging. Primär nutzt man direkte Methoden, mit denen die kritischen Signale unmittelbar am Baustein überwachbar sind. Sekundäre Ansätze basieren auf einer indirekten Auswertung nicht unmittelbar sichtbarer Signale. Meist bedarf es erst einer Art „Markierung“ der Signale, damit sie überhaupt auffindbar sind.

Selbst wenn nach zeitintensiven Tests kein Fehlverhalten mehr festgestellt wird, bedeutet dies noch lange nicht, dass eine Schaltung fehlerfrei ist. Insbesondere an den Schnittstellen kann es passieren, dass z.B. ein bereits getesteter Softwaretreiber nicht mehr korrekt mit der geringfügig geänderten Hardware zusammenarbeitet. Durch das Anpassen des Treibers entstehen aber auch wieder neue Fehlerquellen, womit sich ein Ringschluss der Fehlerquelle beim Debugging ergibt. Oftmals kann es passieren, dass die Fehlerursache nicht genau lokalisiert werden kann. Dann ist es notwendig, zumindestens die Quelle soweit wie möglich einzugrenzen. Am Ende dieses komplexen Prozesses wurde die Hardware und die dafür notwendige Software vielfach modifiziert.

Zielstellung für das Debugging wird es zukünftig sein, mit Hilfe neuer Methoden Fehler in Hardware unabhängig von der Zieltechnologie in Echtzeit zu finden.

2.2 Stand der Technik

Für die Fehlersuche in einen Hardware-System bietet die Praxis derzeitig eine Reihe von Möglichkeiten, die jedoch in ihrer Funktionalität immer auch eine Reihe von Nachteile besitzen. Wenn dabei von Hardware-Debugging gesprochen wird, so ist häufig die Verwendung eines LAs oder Oszilloskops gemeint. Allein die Anschaffung solcher Geräte übersteigt oftmals die finanziellen Mittel, die für die Entwicklung von Prototypen vorgegeben worden sind. In einen komplexen System kommunizieren die verschiedenen ICs mittels fest verdrahteter Kupferbahnen. Da diese ICs fest auf die Platine gelötet sind, hat man primär keinen Angriffspunkt für die hochsensiblen Messspitzen eines Logic Analyzers. Um dennoch Zugriff auf diese Leitungsbahnen zu erhalten, entwickelten einige Hersteller von Platinen sog. *Test-Pins* (vgl. [Ter02]) auf ihren Produkten. Entlang des Verlaufs vorher festgelegter Leitungen werden diese *Test-Pins* als Pfostenstecker in die Leiterplatte eingearbeitet, an denen man nun die Möglichkeit hat entsprechende Messungen mit einen LA vorzunehmen, ohne in den Ablauf einer Schaltung einzugreifen. Da man mit Hilfe derartiger Pins jedoch nur externe, also zwischen den Bausteinen einer Schaltung befindliche Signale messen kann, eignet sich das Verfahren nicht zum effektiven Hardware-Debugging.

Viel wichtiger als Auswertung externer Signale eines digitalen Systems sind meist die internen Signale innerhalb der einzelnen Bausteine, deren Signalwerte mit Hilfe eines Logic Analyzers nicht direkt auswertbar sind. Um dennoch die Werte auslesen zu können, bedienen sich die Entwickler eines implementierungstechnischen „Tricks“ bei der Beschreibung ihrer Schaltung durch eine beliebige HDL. Wenn einige Pins eines Designs ungenutzt bleiben, so verbindet man die diese, zusätzlich zu ihrer usprünglichen Funktion, mit den zu untersuchenden internen Signalen. Dadurch werden einige interne Signale aus dem IC herausgeführt und können dann mittels Test-Pins und LA überwacht werden. So einfach wie dieser Ansatz theoretisch erscheint, ist er in der Praxis jedoch nicht anwendbar. Die Anzahl der externen Pins eines Chips ist begrenzt, da viele der zahlreichen Pins für die eigentliche Schaltung zwingend notwendig sind, oder sie fest definierte Steuerleitungen des Schaltkreises repräsentieren. Die wenigen unbenutzten Pins genügen

oft nicht mehr um eine ausreichende Anzahl von internen Signalen gleichzeitig zu verfolgen. Um diesem Problem zu begegnen und die Anzahl der simultan kontrollierbaren internen Signale zu erhöhen, kann der Entwickler diese internen Signale durch einen Multiplexer zusammenfassen, oder alternativ das Design in seinen Bestandteilen einzeln überwachen. Spätestens an diesem Punkt hat jedoch die Verwendung eines Logic Analyzers in Verbindung mit Test-Pins ihre technischen Grenzen erreicht. Grundsätzlich ist diese Vorgehensweise zwar wirksam, aber technisch sehr aufwendig, denn für jedes neue interne Signal das überwacht werden soll, muss die entsprechende Hardwarebeschreibung angepasst werden. Danach muss bei jeder Änderung der Prozess zur Erzeugung der Schaltkreisbeschreibung (*Design Flow*) neu durchlaufen und dessen Ergebnis auf den Baustein geladen werden. Bei großen Designs ist diese Vorgehensweise sehr aufwendig und beansprucht sehr viel Zeit. Daneben stellt die Benutzung eines Logic Analyzers bzw. Oszilloskops eine eher passive Methode des Hardware-Debuggings darstellt, was dazu führt das nur aktuelle Zustände von wenigen internen Signalen überwachbar sind.Oft treten Fehler aber auch zeitlich verzögert auf, oder das Fehlverhalten einer Schaltung ist nicht reproduzierbar. Technisch wird dann zwar erkannt, das sich eine Signalleitung im Chip fehlerhaft verhält, jedoch nicht, welche anderen Signalwerte dazu führten. Ein dafür erforderlicher illegaler Zustand der Schaltung ist somit nicht extern des Bausteins setzbar.

Die finanziell derzeitig günstigere Alternative zu den kostenintensiven Geräten (LA, Oszilloskop) ist ein Simulator, bei dem sich der Entwickler die komplette Schaltung automatisch in Software nachbilden lässt und dadurch auch Zugriff auf die Werte sämtlicher interner Signale hat. Hierzu zählen nicht nur die aktuellen Werte in einem bestimmten Zeittakt, daneben hat der Benutzer auch die Möglichkeit illegale Zustände manuell zu setzen und schrittweise verschiedene Eingangsbelegungen zu testen. Ein Simulator erfüllt zumindestens technisch betrachtet die grundlegenden Anforderungen an die Fehleranalyse bei einem beliebigen Design. Diese Vorgehensweise ist jedoch nicht immer praktikabel, da es Einschränkungen gibt die zu beachten sind. Sowohl heutige Computersysteme, als auch die zahlreichen verschiedenen Zielarchitekturen unterliegen einem permanenten Fortschritt in Bezug auf ihre Geschwindigkeit. Dennoch existieren nach wie vor viele Anwendungen in denen das Design mit einer geringeren Taktfrequenz als der auf der Hardware vorkommenden, simuliert wird. Im Gegensatz dazu, tritt selten auch der umgekehrte Fall auf, das die Hardware eine Funktion langsamer ausführt als der Simulator in Software. Gerade bei sehr umfangreichen Schaltungen können diese

Unterschiede in den Ausführungsgeschwindigkeiten zu schwerwiegenden Problemen bei der Fehlersuche führen. Zusätzlich dazu treten trotz intensiver Qualitätskontrollen seitens der Hersteller natürlich auch rein physikalische Fehler in der Zielarchitektur auf, deren Ermittlung besonders schwierig ist, da ein Fehlverhalten nicht immer statisch auftreten muss. Ein weiterer Kritikpunkt bei der Simulation ist das Fehlen jeglichen Echtzeitverhaltens. In Hardware auftretende Fehler möchte man bereits während der Ausführung des Design erkennen und wenn möglich beheben, das Laden der Schaltung in einen Simulator impliziert aber das Setzen der zum Fehlerzeitpunkt vorliegenden Signalwerte, diese fehlen jedoch. Deshalb ist man gezwungen, den in der Schaltung implementierten Algorithmus solange im Simulator schrittweise abzuarbeiten, bis der Fehler eventuell wieder auftritt und die passende Belegung der externen und internen Signale gefunden wird. Dabei treten Fehler auch durchaus erst bei der Interaktion zwischen mehreren Bausteinen, die mit einem Simulator nicht nachgebildet werden können, auf. Zur Behebung dieses Problems, erweitert man die Funktionsweise des Simulators durch die die zusätzliche Verwendung eines Logic Analyzers.

Für die erste Variante dieser Methode wird davon ausgegangen, dass die Simulationssoftware die externen Signale der Schaltung während des laufenden Betriebes über den Logic Analyzer ausliest und die Schaltung damit simuliert. Dem gegenübergestellt existiert eine zweite Variante, in der zuerst alle Signalwerte durch den Logic Analyzer gespeichert werden, um anschließend als Stimuli für die in Simulator nachgebildete Schaltung zu dienen. Durch die Verbindung der Schaltung mit einem Computer benötigt man zur Übertragung der Daten eine Schnittstelle mit einer hohen Datentransferrate. Die heutzutage benutzten Kommunikationsschnittstellen *RS232* und *Seriell*, können laut [Date00] die zur Übertragung erforderlichen Geschwindigkeiten nicht umsetzen. Erst durch die Implementierung moderner Schnittstellen wie z.B. *Gigabit-Ethernet* in Hardware, bekommen die beiden Methoden eine praktische Relevanz.

Als Reaktion auf dieses Dilemma schlossen sich einige Hersteller von Hardware-Produkten zusammen und definierten eine neue einheitliche Schnittstelle auf ihren Schaltkreisen. Wenig später wurde dieses Verfahren durch den weltweit einheitlichen Standard der *IEEE* normiert und bekam den Namen *JTAG*. Um die JTAG-Funktionalität nutzen zu können, müssen seitens der Hardware sog. *Boundary Scan-Zellen* implementiert werden mit denen die Funktionsweise der Ein- und Ausgaberegister eines Bausteins erweitert wird. Sämtliche dieser Boundary-Scan-Zellen können seriell in einem sog. *Scan Path* zusammengefasst werden, um sie anschließend über maximal 5 Pins eines ICs

auszulesen (vgl. [Bou96]). Die Steuerung dieses Verhaltens wird mittels eines endlichen Automaten umgesetzt der mit 16 Zuständen arbeitet. Der Vorteil des Verfahrens liegt in der Vielzahl der vorhandenen Funktionen zum Beschreiben und Auslesen eines Bausteins. Nachteilig ist allerdings die serielle Übertragung der Daten die bei umfangreichen Schaltungsbeschreibungen einen unverhältnismäßig hohen Zeitaufwand erfordert. Deshalb kann bei der Verwendung von JTAG im Allgemeinen, sowie bei der Kopplung mit anderen Verfahren zum Debugging im Speziellen, nicht von einer Fehleranalyse unter realen Echtzeitbedingungen gesprochen werden. Zusätzlich bietet die Schnittstelle ebenfalls keinen direkten Zugriff auf interne Signale eines Chips.

Einige Hardware-Hersteller begannen deshalb, in ihren Schaltkreisen neben JTAG alternative Schnittstellen zu schaffen. Die Firma *Xilinx* bietet z.B. für viele ihrer Zielarchitekturen ein sog. *Readback*-Verhalten an (vgl. [Xil138]). Dabei handelt es sich um ein spezielles Ausleseverfahren, bei dem der komplette Inhalt eines angehaltenen ICs ausgelesen wird. Im Anschluss daran berechnet ein Programm aus diesen Daten die Belegung aller Signale einer Schaltung zu einem einzigen Zeitpunkt. Die Berechnung aller internen Werte ist nur mit Hilfe komplexer Gleichungen und der Kenntnis der Platzierung innerhalb des Chips möglich. Die Übertragung der Daten erfolgt wiederum seriell, was einen hohen Zeitaufwand verursacht, der aufgrund der Datenmenge noch größer ist als bei der Übertragung durch JTAG. Darüber hinaus kann man in komplexen Systemen nicht davon ausgehen, dass nur Schaltkreise einer Firma Anwendung finden.

ChipScope ist eines weiteren Produkte des Unternehmens *Xilinx.* Es handelt sich dabei um eine Art Logic Analyzer, der mit in das Design integriert wird (vgl. [Xil02]). Dazu bindet die *Xilinx*-Software einen sog. ILA (engl. *Integrated Logic Analyzer*) in die zu synthetisierende Schaltungsbeschreibung ein. Anschließend kann man durch diese Zusatzkomponente in Verbindung mit einem speziellen Programm der zuvor markierten internen Signale auslesen. Nachteilig an diesem Ansatz ist, dass man vor Beginn des Debuggings festlegen muss, welche Signale man später überwachen möchte. Aufgrund der Benutzung nicht verwendeter Pins des ICs, kann die maximale Anzahl der durch den *ILA* kontrollierbaren Signale variieren. Positiv zu bewerten, ist die Fähigkeit die Signale während der Laufzeit auszulesen.

Die Firma *Altera* bietet ein vergleichbares Produkt mit dem Namen *SignalTAP* (vgl. [Alt99]). Seitens der Hardware wird hierbei in den Schaltkreisen der *APEX*-Serie allerdings die notwendige Zusatzlogik fest verdrahtet vordefiniert. Die Werte der zu überwachenden internen Signale einer Schaltung werden in einem eigens dafür angelegten

RAM (engl. ***R**andom **A**ccess **M**emory*) abgelegt. Die gespeicherten Signalwerte im RAM kann man nun während der Laufzeit über der JTAG-Schnittstelle ausgelesen und in einem Simulator weiterverarbeiten. Auch dieser Ansatz bietet eine Reihe von Kritikpunkten. Zum Einen handelt es sich auch hierbei wieder um ein Verfahren das nicht unabhängig von Hersteller ist, zum anderen hängt die Anzahl der überwachbaren internen Signale direkt von der Größe des verfügbaren Speichers ab. Zusätzlich dazu ist es erforderlich, dass jedes System neben den primär von der eigentlichen Schaltung benötigten Ressourcen noch weitere RAM-Bausteine benötigt.

Im Jahr 1998 wurde eine weitere Variante zum Hardware-Debugging vorgestellt. In verteilten Echtzeitsystemen wird dabei ein sog. *Hardware Monitor* verwendet, der dem Benutzer erlaubt Signale in Echtzeit zu überwachen und die Kommunikation in einen solchem System zu kontrollieren. (vgl. [RAW98]).

Ähnlich wie der in der vorliegenden Arbeit benutzte Ansatz, verwendet auch [Koch98] zusätzliche Komponenten in der Schaltungsbeschreibung die auf ein zuvor festgelegtes Signalverhalten reagieren. Diese Modifikationen laufen dabei automatisch ab, der Nutzer muss nur angeben, welche internen Signale er überwachen möchte. Diese Signale werden anschließend mit *Breakpoints* versehen und warten auf die Erfüllung der für sie relevanten logischen Bedingung. Mit Hilfe dieser Vorgehensweise ist es möglich Hardware-Debugging während der Laufzeit des Systems durchzuführen. Als negativ an dieser Stelle gilt allerdings, das diese Methode eine Schaltung nur passiv überwacht und damit keine Zustände explizit setzen kann. Die Taktfrequenz des modifizierten Designs verringert sich in der Regel durch die Vergrößerung der eingefügten kombinatorischen Logik sowie durch die längeren Datenpfade. Da Emulation mit der Simulation dabei verbunden werden, ist ein Weg beschrieben mit dem man eine Verbindung zwischen der Verhaltensbeschreibung mit Hilfe einer HDL und einem während der Synthese erzeugten Zwischenformat auf Register-Transfer-Ebene herstellt.

Ein weiteres System für das Hardware-Debugging während der Laufzeit unter Verwendung von Zusatzkomponenten, stammt vom *Fraunhofer Institut für Integrierte Schaltungen* aus Deutschland (vgl. [Date00]). Ziel dieses Vorgehens ist es, alle internen und externen Signale beginnend mit dem Initialisieren der Schaltung zu speichern. Sobald eine Fehlerbedingung an einem der Breakpoints erfüllt wird, werden alle gespeicherten Daten an den Simulator übertragen, der die auf Gatterebene umgewandelte Schaltungsbeschreibung mit diesen Signalwerten initialisiert. Somit kann man die Zustände der Schaltung beliebig weit in Vergangenheit zurückverfolgen, ohne beim vom

Einschalten an den Simulator nutzen zu müssen. Die Zusatzkomponenten für das Hardware-Debugging werden in die Verhaltensbeschreibung einer Schaltung integriert. Man kann drei unterschiedliche Arbeitsweise des Debuggers nutzen, dazu gehört das Speichern (*trace mode*) sowie das Setzen (*update mode*) und die Fehlererkennung (*breakpoint mode*). Der Vorteil des *update modes* ist u.a. die Fähigkeit schwer zu erreichende Zustände manuell zu setzen. Der Nachteil dieser vorgestellten Idee liegt in der Speicherung sämtlicher Signalwerte, was einen erheblichen Speicheraufwand erfordert. Bei der Übertragung an den Simulator wird außerdem die Ausführung des Design gestoppt.

Aus dem Unternehmen *Bridges2Silicon* stammt ein bereits patentierter Ansatz eines neuartigen Hardware-Debuggers (vgl. [Bri02]). Auch hierbei wird die Schaltungsbeschreibung durch neue Komponenten erweitert, wobei das Kernstück dieses Vorhabens ein sog. IICE (engl. ***I**ntelligent **I**n **C**ircut **E**mulator*) darstellt. Dieser ermöglicht in Zusammenarbeit mit weiteren Komponenten die Speicherung der internen Signalwerte im RAM, um diese im Fehlerfall auf einen Computer zu übertragen, während die Schaltung normal weiterläuft. Auf dem Computer befindet sich kein Simulator, sondern das eigene Programm der Firma, wobei es gelungen ist einen Bezug zwischen der Verhaltensbeschreibung und den ausgelesenen Signalwerten zu finden. Der Benutzer hat jedoch die Einschränkung, dass er sich nur die Signale anzeigen lassen kann, die vor dem Debugging ausgewählt hat, alle anderen sind nicht sichtbar. Zur Übertragung wird die JTAG-Schnittstelle genutzt.

Kapitel 3

Grundlagen

In diesem Kapitel werden zahlreiche Grundlagen erklärt, die in diese Arbeit eingeflossen sind. Für das weitere Verständnis sind diese Ausführungen unumgänglich, insbesondere wird die Hardwarebeschreibungssprache VHDL zu Beginn kurz umrissen. Von besonderer Bedeutung sind bei diesen Betrachtungen sind die unterschiedlichen Abstraktionsebenen auf denen eine Schaltungsbeschreibung beruht. Daneben gilt es, die Vor- und Nachteile der Sprache aufzuzeigen und darzulegen weshalb gerade diese Sprache benutzt wurde, obwohl es zahlreiche andere Möglichkeiten gibt.

Aufgrund der bei dieser Arbeit zum Einsatz gekommenen Hardware beschäftigt sich ein Teil des Kapitels mit dem Aufbau und der Funktionsweise der einzelnen Komponenten sowie der Struktur des verwendeten FPGAs. Neben der Hardware wird auch die benutzte Software näher erläutert. Dazu wird einerseits die angewendete Design Software der Firma *Xilinx* beschrieben, zum anderen der verwendete Simulator *ModelSim*. Der damit im Zusammenhang stehende Entwurfsablauf, der in diesen Produkten seine Umsetzung findet, wird ebenfalls untersucht.

3.1 VHDL

Die amerikanische Regierung gab 1980 ein Projekt mit den Namen VHSIC (engl. ***V**ery **H**igh **S**peed **I**ntegrated **C**ircut*) in Auftrag. Durch dieses Projekt sollte es insbesondere im Verteidigungssektor möglich werden, den Prozess zur Erstellung elektronischer Schaltungen zu vereinfachen und damit die Entwicklung fortschrittlicherer ICs voranzutreiben (vgl. [Cha97]). Darüber hinaus sollte es möglich werden, Schaltungen besser als bisher zu dokumentieren und zu vereinheitlichen. Im Rahmen dieses Projektes entstand fünf Jahre später die Hardwarebeschreibungssprache VHDL (***VHSIC H**ardware **D**escription **L**anguage*), deren Syntax an die der Programmiersprache *ADA* angelehnt wurde. 1987 wurde der erste VHDL-Standard, *IEEE 1076.1*, von der IEEE herausgegeben und 1993 durch *VHDL'93* erweitert (vgl. [Ash96]). Aufgrund dieser Sprache sind Hardwareentwickler nicht mehr gezwungen einen Schaltplan schematisch aus Einfachstbausteinen zu erstellen, oder gar auf Transistor-Transistor-Logik (*TTL*) zurückzugreifen, sondern können viel höhere Abstraktionsebenen nutzen (vgl. [Cha97]). Da VHDL die Basis für das in der vorliegenden Arbeit entwickelte Programm bildet; ist es erforderlich näher auf die Gründe einzugehen, warum gerade dieses Format benutzt wird. Aufgrund der regelmäßigen internationalen Standardisierungsbestrebungen für die Sprache kann man auf einen Befehlssatz zurückgreifen, der seit vielen Jahren ständig verbessert wird. So kann man davon ausgehen, dass heute mit VHDL entwickelte Schaltungsbeschreibungen auch in einigen Jahren noch verarbeitet werden können.

Durch die Verbindung von Simulation und Emulation für das Debugging einer Schaltung, muss man die Besonderheiten des zu Grunde liegenden Logiksystems von VDHL beachten. Entsprechend des IEEE-Standards ist es für Simulationszwecke 9-wertig (*'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'*), für die technische Anwendung (Emulation) sind allerdings nur drei dieser Werte (*'0', '1', 'Z'*) nutzbar. Trotz dieser Einschränkung wird der anwendbare Sprachumfang kaum beeinflusst. Daneben unterscheidet sich VHDL in der Ausführungsreihenfolge von Befehlen. Während bei der Programmierung von Software alle Anweisung sequentiell verarbeitet werden, so lässt sich der Ablauf in VHDL vorher genau festlegen und somit auch eine parallele Ausführung erzwingen. Gerade die Modifikationen an einer Schaltung, die im Rahmen dieser Diplomarbeit erforderlich sind, bedürfen dabei einer genauen Abfolge, um das Verhalten des ursprünglichen Designs nicht zu verändern. Die Unterstützung des sog. *Top-Down* Entwurfs durch VHDL ist für das Ausgangsformat des hier vorgestellten Debuggers sehr wichtig. Nur dadurch ist es möglich

geworden den strukturellen Aufbau einer Schaltungsbeschreibung zu schnell zu analysieren und Veränderungen der Struktur effizient umzusetzen. Das zum Einsatz gekommene VHDL-Format dient neben der hier beschriebenen Verwendung auch als Grundlage für eine weitere Diplomarbeit zur Umsetzung einer patentierten (vgl. [Pat01b]) *Virtuellen Hardware-Maschine*. Dahingehend ist das menschenlesbare und selbsterklärende Dateiformat ein zusätzlicher Vorteil bei der Benutzung von VHDL. Der Code kann dabei problemlos dokumentiert und zwischen den verschiedenen Arbeitsgruppen ausgetauscht werden. Da die beiden Projekte unterschiedliche Zielarchitekturen nutzen, ist die Herstellerunabhängigkeit ebenso von Vorteil wie die Technologieunabhängigkeit. Die im Rahmen dieser Diplomarbeit zum Einsatz gekommenen Eigenschaften von VHDL lassen folgendermaßen zusammenfassen:

- ⇒ Vielseitigkeit,
- ⇒ Herstellerunabhängig,
- ⇒ Technologieunabhängig,
- ⇒ Lesbarkeit,
- ⇒ Strukturierung.

3.2 Ebenen und Sichten des Hardwareentwurfes

Bei einer HDL bildet nicht nur der eigentliche Algorithmus die Beschreibungsebene, wie es bei einer Programmiersprache der Fall ist, vielmehr existieren darüber hinaus noch weitere. Während in Software nur ein Verhalten beschrieben wird, beschreibt eine HDL Verhalten und Struktur der Schaltung. Da die Auswahl einer geeigneten Ebene für das Hardware-Debugging entscheidend ist und einen erheblichen Zeitaufwand erfordert, werden mit Hilfe der Abbildung 3-1 die einzelnen Abstufungen nach [Mod85] näher erläutert.

Bei den Ebenen zur Beschreibung unterscheidet man grundsätzlich fünf verschiedene Abstraktionsgrade deren Komplexität in der Reihenfolge: Systemebene, Algorithmische Ebene, Register-Transfer (*RT*) -Ebene, Logikebene, Schaltkreisebene zunimmt. Nachdem der Entwickler geklärt hat auf welchem dieser allgemeinen Niveaus er seine Schaltung beschreiben möchte, muss er sich entscheiden, welche Sicht dieser Entwurfsebene am besten gerecht wird. Dazu stehen beim Hardwareentwurf drei verschiedene Abstufungen zur Auswahl: die Verhaltenssicht, die Struktursicht und die Geometriesicht. In Verbindung

mit einer der erwähnten Entwurfsebenen kann man jeder dieser drei Gruppen eine eigene Art der Schaltungsbeschreibung zuordnen.

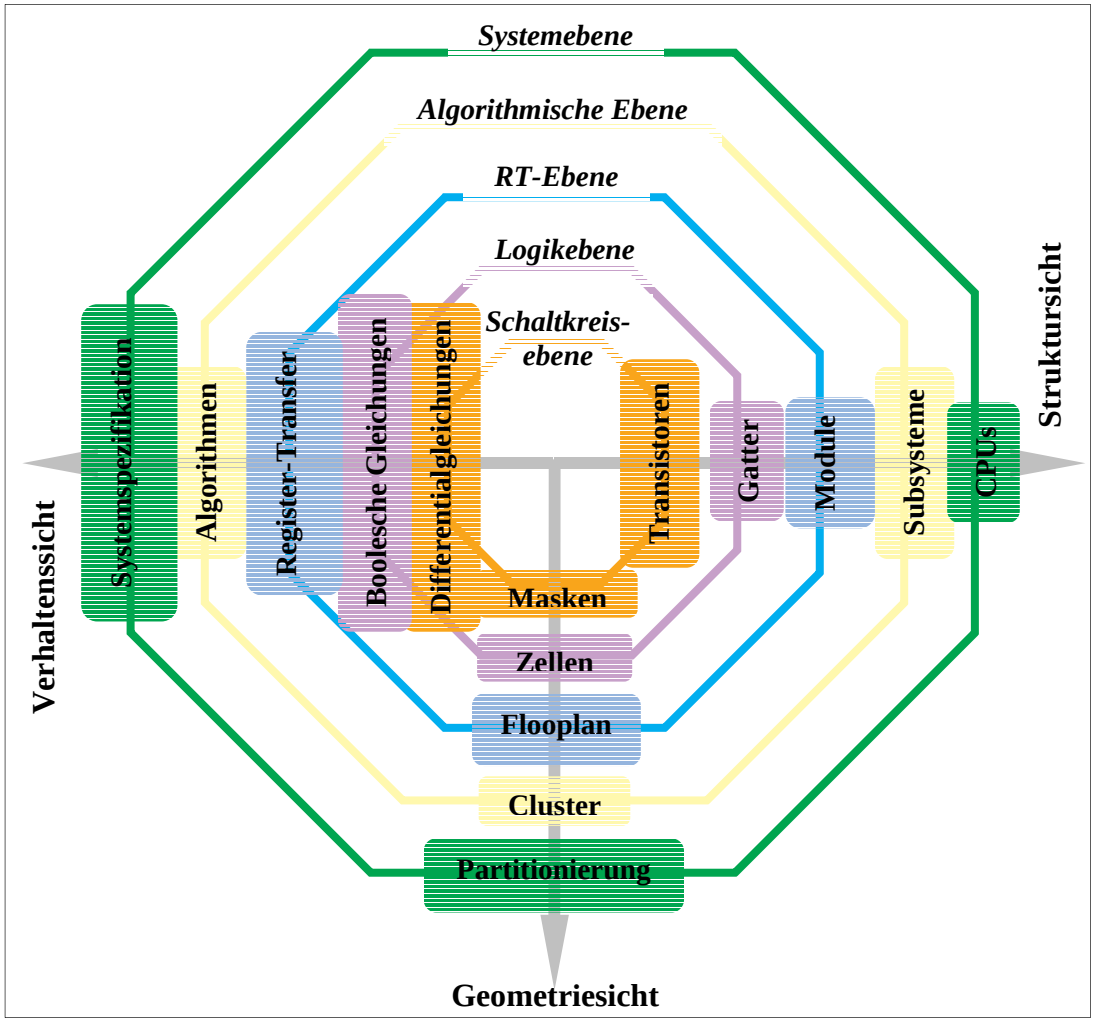

Abbildung 3-1: Entwurfsebenen und Sichten in Anlehnung an [Mod85]

Am Beispiel der RT-Ebene wird dieses Vorgehen dargelegt. Die Verhaltenssicht einer Schaltung wird auf RT-Ebene durch die notwendigen Operationen (Subtraktion, Multiplikation, etc.) sowie durch die Übertragungen von Daten zwischen den Registern beschrieben. Zusätzlich integriert man hierbei Rücksetz- und Taktsignale in die Darstellung was zur Folge hat, das man die Operationen in Abhängigkeit von Pegeln und Flanken dieser Steuersignale definiert. In der Struktursicht verknüpft man die Komponenten (Addierer, Komparator, etc.) mit Hilfe von Signalen. Die Geometriesicht stellt die Schaltung anhand von zu Blöcken oder Makrozellen zusammengefassten Einheiten eines Bausteins dar.

Heutige Systementwurfsabläufe beginnen meist mit der manuellen Verhaltensbeschreibung auf der Systemebene, in der man die zu entwickelnde Anwendung genau spezifizieren

muss. Anschließend verläuft der Entwurfsvorgang über die Algorithmen bis zur Darstellung des Register-Transfers. Auf dieser RT-Ebene wird die Schaltung durch die Synthese automatisch in eine strukturelle Sichtweise aus Leitungen und Komponentengruppen übertragen. Der Grad der Abstraktion wird nun bis hin zur Schaltkreisebene verringert und danach auf die geometrische Sicht aus Polygonen bzw. Masken eines Chips übersetzt. Im Anschluss daran liegt eine fertige Schaltungsbeschreibung für eine bestimmte Zielarchitektur vor.

Für das Hardware-Debugging im Sinne dieser Diplomarbeit spielen verschiedene Ebenen eine Rolle. Die eigentliche Schaltungsbeschreibung kann der Benutzer auf jeder Ebene durchführen, vorausgesetzt das dafür verwendete Programm übersetzt diesen Code anschließend auf die RT-Ebene in die Struktur- oder Verhaltenssicht. Das danach vorliegende Format dient als Grundlage sowohl für die durchzuführenden Erweiterungen, als auch für die Simulation.

3.3 Verwendete Hardware

3.3.1 Virtex FPGA

Der *XC800BG432* ist ein auf SRAM basierendes FPGA der Firma *Xilinx* und stammt aus deren Virtex Familie. Diese Familie erlaubt Taktfrequenzen bis zu 200 MHz sowie bis zu 1 Million Gatter. Die verwendete interne Struktur hat dabei den in Abbildung 3-2 dargestellten Aufbau. Durch die Virtex-Architektur werden einige Eigenschaften der

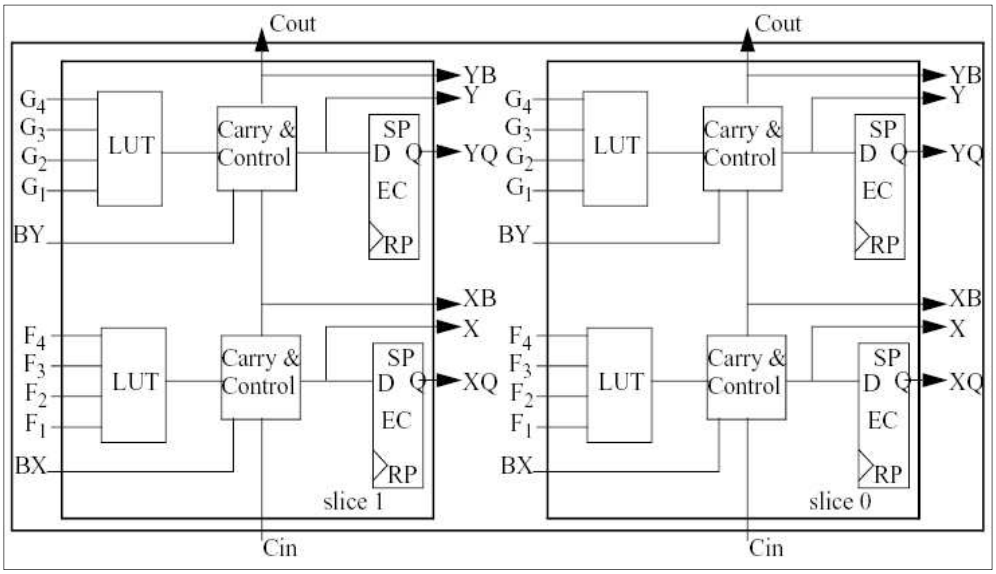

Abbildung 3-2: 2-Slice Virtex CLB

XC6000-Familie mit den Weiterentwicklungen aus der *XC4000*-Serie kombiniert. In diesem Zusammenhang entstanden CLBs (engl. ***C**onfigurable **L**ogic **B**locks*) die aus 2 identischen Bereichen (*slices*) bestehen. Diese Abschnitte werden als *Slice0* und *Slice1* bezeichnet und sind wiederum jeweils aus zwei LCs (engl. ***L**ogic **C**ells*) aufgebaut. Eine LC besteht aus einem Funktionsgenerator, dem sog. *Carry/Control-Block* und einem Speicherelement (vgl. [Car99]). Die LUTs (engl. ***L**ook **U**p **T**ables*) dienen hauptsächlich der Umsetzung von logischen Funktionen. Darüber hinaus ist es möglich diese LUTs als primitiven RAM zu nutzen, sog. Select-RAM (*SRAM*). Die Verbindungen zwischen den Flipflops und den LUTs werden durch den Carry/Control-Block realisiert, der daneben noch weitere untergeordnete Funktionen liefert. Nach [Car99] ist es dadurch z.B. möglich einen Volladdierer mit nur einer LC umzusetzen.

Neben den SRAM der durch die LUTs implementiert werden kann, bietet die Virtex-Architektur noch weitere Speichermöglichkeiten die statisch in den Aufbau des FPGAs integriert sind. Dabei sind um die CLBs feste Block-RAM-Zellen (*BRAM-Zellen*) angeordnet. Durch die GRM (engl. ***G**eneral **R**outing **M**atrix*) sind die einzelnen CLBs miteinander verbunden. An den Verbindungspunkten zwischen vertikalen und horizontalen Leitungen werden mit Hilfe von *Routing Switches* die Signale zu den CLBs implementiert. In der GRM ist dazu ein Array der jeweiligen Routing Switches hinterlegt. Die an den äußeren Schichten des FPGAs sitzenden IOBs (engl. *Input-Output-Blocks*) werden über eine ringförmige Struktur, den *Versa-Ring*, mit den CLBs verbunden (s. Abbildung 3-6).

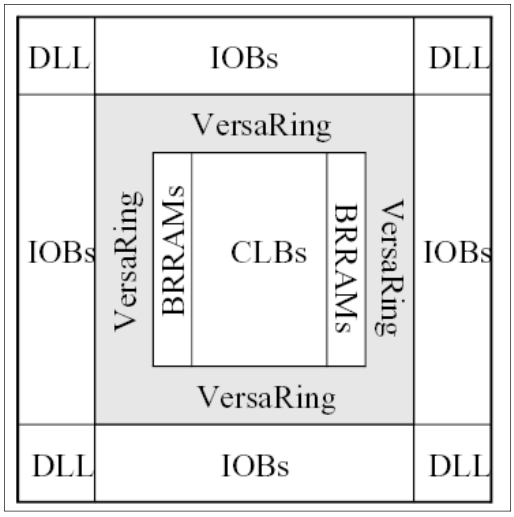

Abbildung 3-3: Virtex-Architektur

Da die IOBs eine festgelegte Lage haben, was zu bestimmten schaltungstechnischen Schwierigkeiten führen kann, entkoppelt dieser Versa-Ring die interne Vernetzung dieser

IOBs. Weil verschiedene Schnittstellen auch verschiedene Spannungen implizieren, werden die IOBs des FPGAs in acht Gruppen eingeteilt. Dadurch ist es möglich gleichzeitig verschiedene Versorgungsspannungen an die IOBs anzulegen. Einen Virtex FPGA kann man daher mit unterschiedlichsten elektrischen Schnittstellen wie USB (engl. ***U**niversal **S**erial **B**us*) und PCI (engl. ***P**eripheral **C**omponent **I**nterconnect*) verbinden. Weiterhin enthält die Virtex-Architektur an den Eckbereichen sog. DLL (engl. ***D**elayed **L**ocked **L**oop*) Komponenten, die zur Verhinderung von Signalverzögerungen an jedem Clock-Input-Puffer dienen. Derartige Laufzeitunterschiede entstehen meist dann, wenn Signalverschiebungen zwischen dem *Clock-Input-Pad* des Chips und den internen Clock-Input-Pins auftreten. Es wird dem FPGA durch die DLL ein synchrones Taktsignal gegeben (vgl. [Car99]). Vergleichbar mit der *XC6000*-Serie erlaubt auch der Virtex die partielle Rekonfiguration zur Laufzeit. Somit können während des laufenden Betriebes Teile der Konfiguration ausgetauscht werden. Bei der Festlegung dieser Konfigurationsmodi stehen zwei unterschiedliche zur Auswahl: Einerseits die serielle, andererseits die schnellere parallele Variante (vgl. [KHW99]).
Für die vorliegende Diplomarbeit bildet der *Xilinx* Virtex-FPGA einen wichtigen Bestandteil. Mit ihm als Zielarchitektur ist es möglich selbst große Schaltungsdesigns mit hohen Taktfrequenzen zu emulieren, was die praxisrelevante Bedeutung des zu Grunde liegenden Ansatzes noch zusätzlich untermauert. Obwohl das FPGA mit 432 Pins eine sehr komplexe Steuerung benötigt, relativiert sich dieser Aufwand spätestens durch die guten Steuerungsmöglichkeiten seitens des *Spyder-Virtex-X2*-Boards.

3.3.2 Spyder-Virtex-X2

Das *Spyder-Virtex-X2*-Board ist ein Teil des *Spyder*-Systems und dient in diesem Zusammenhang zur Emulation einer Schaltung auf dem integrierten Virtex-FPGA der Firma *Xilinx*. Kernstück des *Spyder-Virtex-X2*-Boards ist ein FPGA der Virtex Familie (*XCV300* bis *XCV800*), bei dieser Arbeit ein *XC800BG432*. Mit Hilfe eines Kontrollerbausteins für die PCI-Schnittstelle ist es möglich das *Spyder-Virtex-X2*-Board als Steckkarte in einem Computer zu benutzen. Über den verwendeten Chipsatz (*PCI90880*) der Firma *PLX* ist es außerdem möglich, zahlreiche Funktionen auf den Board über PCI zu steuern. Darunter fällt auch das Konfigurieren des FPGAs mittels PCI-Schnittstelle. Zusätzlich befinden sich auf dem Board zahlreiche RAM-Bausteine, entweder als SSRAM (engl. ***S**ynchronous **S**tatic **RAM***) mit einer Kapazität von *128k x 32*,

oder als SDRAM (engl. ***S**ynchronous **D**ynamic **RAM***) mit *4M x 32*. Darüber hinaus existieren sechs serielle EEPROMs (engl. ***E**lectrical **E**raseble **P**rogrammable **R**ead **O**nly **M**emory*) mit 1 Mbit (vgl. [Pro99]). Gesteuert werden die Konfigurationen des FPGA zusätzlich über einen sog. Bus-Arbiter, wofür ein CPLD (engl. ***C**onfigurable **P**rogrammable **L**ogic **D**evive*) der Serie *XC95144XL* zum Einsatz, der auch zur Konfiguration des FPGA über die parallele Schnittstelle sowie über JTAG dient. An den

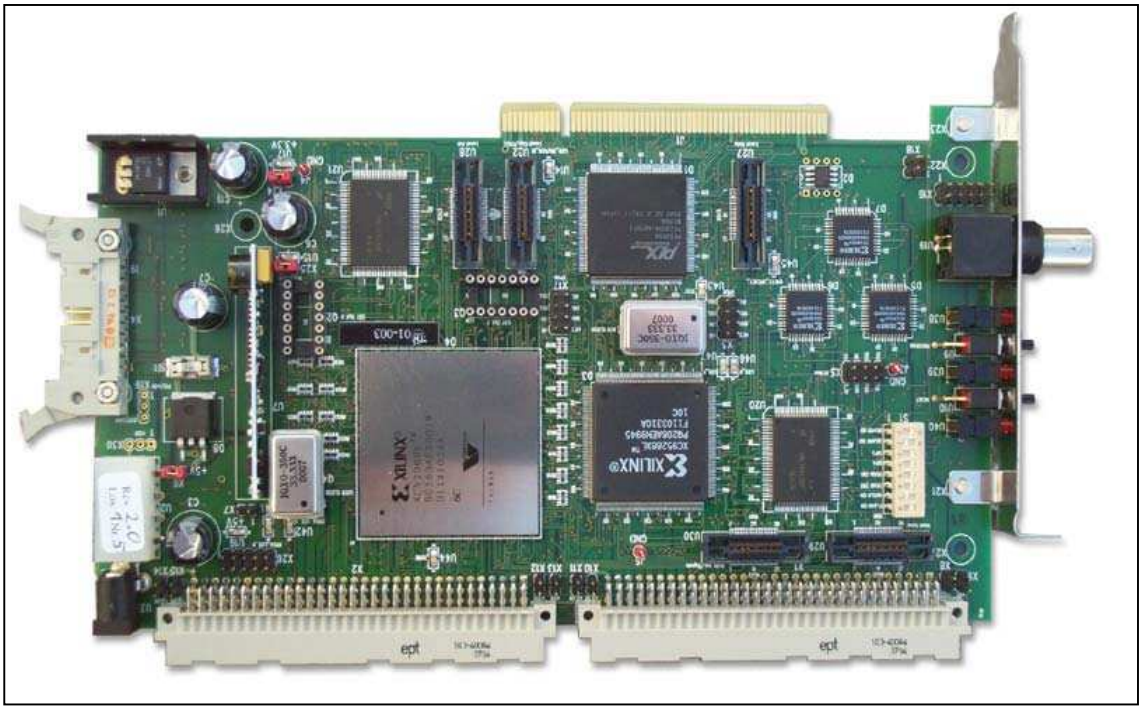

Abbildung 3-4: Spyder-Virtex-X2 aus [Pro99]

Verbindungen zwischen FPGA und *PCI9080* sowie zwischen dem FPGA und den Erweiterungssteckern zum Anschluss an die Mikrokontrollerkarte sind bereits standardisierte Stecker für einen Logic Analyzer vorgesehen. Speziell für das Aufspüren von Fehlern während der Emulation sind diese Stecker sehr nützlich (vgl. [KHW99]).

Die andere bereits in das Board integrierte Variante zum Hardware-Debugging ist die Verwendung der großen zur Verfügung stehenden RAM-Bausteine. Die intern ausgelesenen Werte können aus dem FPGA in den RAM geschrieben werden und anschließend über das *Spyder-Core-P2*-Board ausgelesen werden. Der „Umweg“ über den RAM ist bei dieser Methode nicht zwingend notwendig, da das *Spyder-Core-P2*-Board die Werte des FPGAs auch direkt auslesen kann.

3.3.3 Spyder-Core-P2/SH3

Das *Spyder-Core-P2/SH3* Board liefert im Prozess der Umsetzung eines eingebetteten Systems die Umgebung zur sofortigen Entwicklung der notwendigen Software, noch während die Hardware implementiert wird. Somit repräsentiert es das Software-Gegenstück zum *Spyder-Virtex-X2*-Board. Kernstück dieses Boards ist ein Prozessor mit den Eigenschaften eines DSPs (engl. ***D**igital **S**ignal **P**rocessors*). Es handelt sich dabei um einen *SH37709A* bzw. *SH37729* Baustein der Firma *Hitachi*, der mit 133 bzw. 66 MHz betrieben wird. Dieser Chip bietet die Möglichkeit über eine JTAG-Schnittstelle Daten zu lesen oder zu schreiben. Darüber hinaus wurde er mit zwei seriellen Schnittstellen

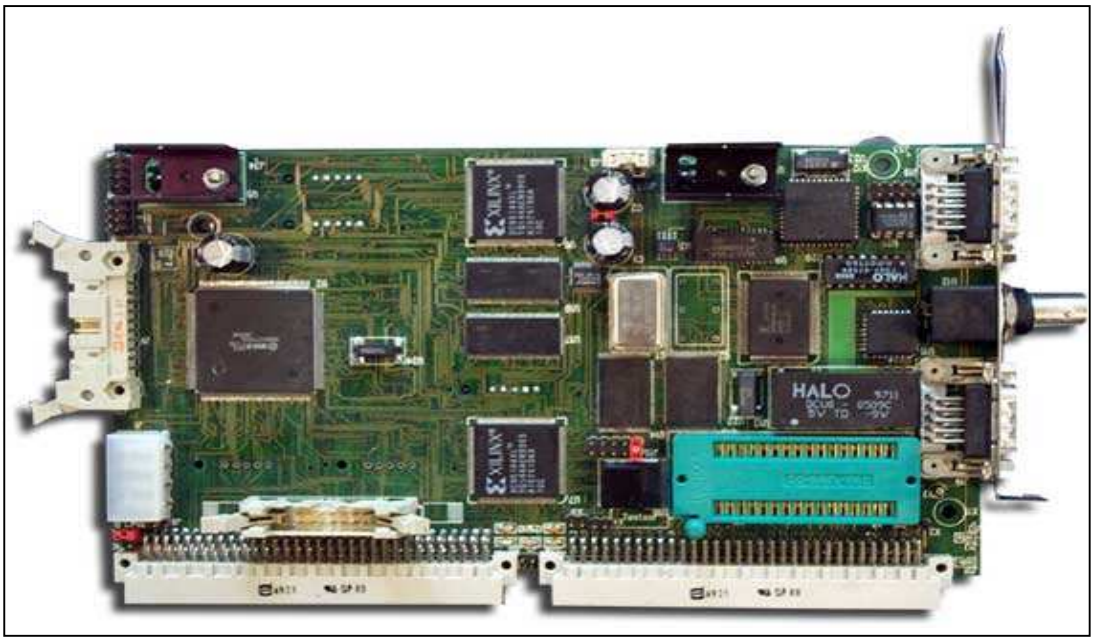

Abbildung 3-5: Spyder-Core-P2/SH3 aus [Pro99]

gekoppelt. Über einen gemeinsamen Bus existiert je eine Verbindung zu dem 16 MB großen SDRAM sowie zu einem CPLD-Puffer und zu den Erweiterungssteckern. Der CPLD-Puffer kann aus einem EPROM (engl. ***E**raseble **P**rogrammable **R**ead **O**nly **M**emory, 1M x 8*) lesen, den Flash-Speicher (*1M x 32*) lesen bzw. schreiben und über eine Schnittstelle mit dem in der Industrie häufig vorkommenden CAN-Bus kommunizieren. Eine Verbindung zu einer 10Base2 Ethernet-Schnittstelle ist ebenfalls implementiert (vgl. [Pro99]), wodurch die Kommunikation des Prozessors mit einem Computer realisiert wird. Auf dem Prozessor ist das Echtzeitbetriebssystem *VxWorks* aus dem Programmpaket *Tornado* der Firma *WindRiver* portiert und zusätzliche Protokolle wie TCP/IP (engl. ***T**ransmission **C**ontrol **P**rotocol/ **I**nternet **P**rotocol*) oder RS232 umgesetzt. Da auch eine

sog. *GNU-C* Umgebung zur Verfügung steht, können Anwendungen problemlos in *C* geschrieben und auf dem *SH3*-Prozessor ausgeführt werden.

3.3.4 Das Spyder-System

Bei dem *Spyder*-System handelt es sich um eine Emulationsplattform, die im Zusammenhang mit zahlreichen Industriepartnern am *FZI Karlsruhe* Ende der 90er Jahre entwickelt wurde (s. Abbildung 3-6). Das Spyder-System dient zur verbesserten

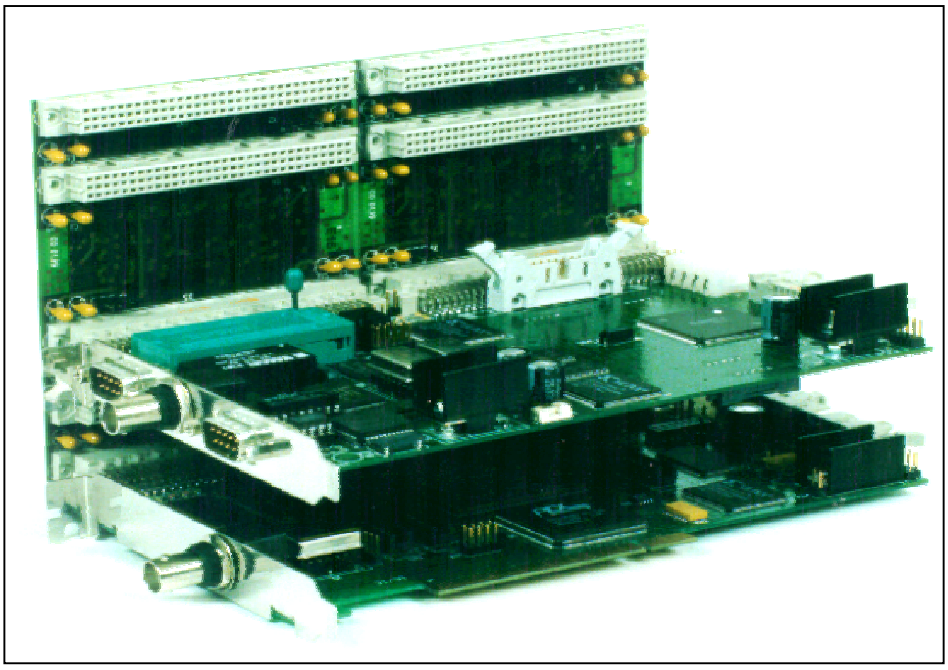

Abbildung 3-6: Spyder-System aus [Pro99]

Entwicklung von eingebetteten Systemen (engl. *Embedded Systems*). Solche eingebetteten Systeme bestehen nach [KHW99] aus wenig, dafür aber hochintegrierten Bausteinen. Die Teile eines derartigen Systems bestehen einerseits aus anwendungsspezifischer Software, die auf entsprechenden Chips implementiert wird, z.B. *SH3* oder *PowerPC*. Diese Bausteine bilden den eigentlichen Kern eines Mikrokontrollers. Andererseits lassen sich Teile eingebetteter Systeme auch in anwendungsspezifischer Hardware, z.B. FPGAs und ASICs umsetzen. Die beiden genannten Bereiche kommunizieren sowohl mit der Umwelt, alsauch untereinander über festgelegte Kommunikationsschnittstellen (vgl. [KHW99]).
Beim Entwurf von eingebetteten Systemen treten jedoch immer wieder Probleme auf. Die Mikrokontroller legen die Randbedingungen eines solchen Systems fest und bieten wenige Möglichkeiten zu variieren. Falls sich die Anforderungen an das eingebettete System nur geringfügig ändern, kann es durchaus passieren, dass die Mikrokontroller komplett ersetzt

werden müssen. Ein anderes Problem entsteht durch die zu verwendenden ASICs. Deren Verhalten ist sicherlich vom Hersteller getestet, jedoch können neue Fehlerquellen im Zusammenhang mit anderen Komponenten auftreten. Die Verwendung von ASICs bietet gerade in der heutigen Zeit einen erheblichen Zeit- und Kostenvorteil. Daher unterteilt man ein komplexes eingebettetes System in kleinere Untersysteme und ersetzt deren Teilfunktionen durch auf dem Markt bereits existierende ASICs. Diese Variante sollte noch Vorrang vor eigenen Entwicklungen haben. Insbesondere gilt dies bei dem derzeit immer kürzer werdenden Entwicklungsprozess und der hohen Komplexität der Anwendungen.

Bisher findet in der Wirtschaft ein getrennter Entwicklungsablauf statt. Nach der Spezifikation der Aufgabenstellung wird diese in Software- und Hardwareaufgaben segmentiert. Erst wenn die Hardware implementiert wird, geht man derzeit dazu über, die Software Architektur zu definieren und zu implementieren. Im Anschluss daran wird das gesamte System getestet und notfalls überarbeitet. In der Zeit zwischen der Aufgabenverteilung für Hard- und Software sowie der Beendigung der Hardwareimplementierung wird mit der Entwicklung der erforderlichen Software nicht begonnen - nach [Pro99] wird sie bisher in dieser Zeit komplett außen vor gelassen. Erst danach beginnt man die Software auf die Hardware anzupassen. Aufgrund der unvorhersehbaren Fehler die dadurch entstehen können sowie der zeitlichen Verluste im Entwicklungsprozess, bedarf dieses Modell einer Anpassung.

Bei einem weiteren Modell, dem *Hardware/Software Co-Design*, wird diese Lücke geschlossen, indem man die Hardware, die Software und die Schnittstellen gleichzeitig implementiert entwickelt. Bisher wird dieser Ansatz aber überwiegend nur in der Forschung eingesetzt. Die beim Spyder-Projekt zum Einsatz gekommene Methode erweitert das *Hardware/Software Co-Design* geringfügig, um die Entscheidungsfindung bei der Struktur eines eingebetteten Systems zu verbessern. Hierzu werden die Komponenten entsprechend ihres logischen und implementierungsspezifischen Verhaltens bewertet sowie die Effizienz der Partitionierung untersucht (vgl. [Pro99]). Das Spyder-System dient nun in jedem Iterationsschritt als Emulationsplattform zur Überprüfung dieser Rahmenbedingungen, wobei gleichzeitig Hardware und Software getestet werden.

Die Entscheidung für diese Zielarchitektur als Basis zur Umsetzung der Diplomarbeit wird durch mehrere Gründe gerechtfertigt. Das System ist an der *Universität Leipzig* vorhanden, es existieren umfangreiche Erfahrungen mit dieser Plattform und es bietet durch jahrelange praktische Tests einen hohen Grad an Fehlersicherheit. Daneben sind alle notwendigen

Software-Produkte für die Programmierung verfügbar bzw. die notwendigen Programme sind bereits entwickelt und müssen nur noch angepasst werden. Durch die Verbindung der beiden Bestandteile des Systems über die *back-plane* kann man die Schaltung auf dem FPGA ausführen, während man mit Hilfe des *Spyder-Core-P2*-Boards die Steuerung derer umsetzt. Dank der *Ethernet*-Schnittstelle auf diesem Board konfiguriert der SH3-Prozessor den FPGA des *Spyder-Virtex-X2*-Boards über das Internet mit den entsprechenden Schaltungsbeschreibungen. Diese Fähigkeit stellt einer der zahlreichen zukünftigen Erweiterungen des hier entwickelten Debuggers dar

3.4 Verwendete Software

3.4.1 Synthese-Software

Ursprünglich waren Schaltungsdesigner gezwungen für jeden Schritt des Entwurfsablaufes einzelne kommandozeilenbasierte Tools zu benutzen. Jedes dieser Programme benötigte eine Unzahl von Optionen zur korrekten Ausführung. In der Regel arbeiteten diese „kleineren“ Programme nicht zusammen, was Konvertierungen der Zwischenformate erforderte. Im Laufe der Zeit kaufte u.a. die Firma *Xilinx* eine ganze Reihe solcher Tools auf und entwickelte viele weitere hinzu. Um den Zeitaufwand bei der Designerstellung zu verringern, wurden diese vielen Werkzeuge unter einer großen gemeinsamen Oberfläche zusammengefasst, in der es nun möglich ist alle Schritte des Design Flows einzelnen und grafisch darzustellen, Fehler besser und schneller zu erkennen sowie Änderungen in den Vorgaben leichter anzupassen.

Es entstand u.a. die lizenzierungspflichtige Variante *Xilinx ISE*, die auch bei dieser Arbeit verwendet wurde. Als Design Compiler enthält das genannte Softwarepaket den *FPGA Express Compiler* der Firma *Synopsys*. Zusätzlich hat *Xilinx* auch einen eigenen Compiler (*XST*) entwickelt, der in einigen Konfigurationen nutzbar ist. Die Entwicklungswerkzeuge der Firma *Xilinx* sind ein wichtiger Bestandteil dieser Arbeit, zur Anwendung kam dabei speziell die Software *Xilinx ISE 4.1i*. Dieses grafische Tool bietet dem Benutzer eine ganze Reihe von zum Hardwareentwurf notwendigen Fähigkeiten. Diese reichen von der Schaltplaneingabe auf unterschiedlicher Basis, über die Festlegung der Rahmenbedingungen (z.B. feste Timingvorgaben), bis hin zur Erstellung einer lauffähigen Beschreibung für einen speziellen Baustein. Die Arten der unterstützten Zielarchitekturen sind vielfältig. So werden ältere Modelle wie z.B. die *XC9500* oder die *XC4000* Familie

ebenso wie moderne Architekturen wie der *Virtex2* unterstützt. Doch welche Vorteile bringt die Benutzung von *Xilinx ISE* im Rahmen dieser Arbeit? Aufgrund der jahrelangen Erfahrung mit den Produkten von *Xilinx* können die für die Diplomarbeit notwendigen Anpassungen an der Schaltungsbeschreibung schnell durchgeführt werden. Die hier vorgestellte Methode basiert auf einem bestimmten Zwischenformat des Entwurfsablaufes. Obwohl auch andere Hersteller die dafür erforderlichen Konvertierungen durchführen, so führt nur die *Xilinx*-Software diesen Schritt automatisch durch. Mit diesem Programmpaket werden Bausteine und Architekturen von zahlreichen Herstellern programmierbarer Logik unterstützt, wodurch das Hardware-Debugging zukünftig auch an weiteren Zielarchitekturen getestet werden kann. Die Qualität der Ergebnisse ist in der Industrie anerkannt. Speziell bei Virtex-Bausteinen wird die Inkrementelle Block-Level Synthese (*BLIS*) während des Entwurfes genutzt (vgl. [Bli00]). *Xilinx ISE* unterstützt natürlich auch die Standards anderer HDLs. Um eine Schaltungsbeschreibung einzugeben, stehen dem Benutzer viele Möglichkeiten zu Verfügung. Er kann die Beschreibung in Verilog, VHDL, ABEL, schematisch als Schaltplan, oder als Netzliste sowie in zahlreichen weiteren Formaten eingeben. Da viele dieser Formate automatisch in VHDL umgewandelt werden, ist das Hardware-Debugging somit auch bei anderen Ausgangsdateien möglich.

3.4.2 Software zur Simulation

ModelSim XE 5.5e ist der Name eines Simulators, der von der Firma *Model Technology Inc*. vertrieben wird und intern aus mehreren kleineren Simulationsmodulen besteht. Zum einen sei dabei *V-System* von *Modeltech Technology Inc*. genannt und zum anderen *QuickHDL* von *Mentor Graphics* sowie *QuickSim*. Für die Simulation auf Gatterebene ist das integrierte *QuickSim* Modul verantwortlich. Insgesamt handelt es sich nach [Ulm98] um ereignisbasierte Simulatoren, die *direct compiled* sind. Das bedeutet, dass zuerst ein plattformunabhängiger Pseudo-Code erstellt wird, der im Anschluss daran in die jeweilige Maschinensprache übersetzt wird.

Bei einer Simulation auf der RT-Ebene im Kommandozeilenmodus muss zuerst *vlib* aufgerufen werden, um das notwendige Arbeitsverzeichnis und die Bibliotheken zu verknüpfen. Mit dem Kommando *vcom* wird der Compiler aufgerufen, der auch gleichzeitig die Syntax überprüft. Dabei werden beide Standards, *VHDL'87* und *VHDL'93*, unterstützt. Nach dem Kompilieren startet man den eigentlichen Simulator mit dem Kommando *vsim*. Wie bei anderen Produkten dieser Art stehen dem Entwickler zahlreiche

Möglichkeiten zur Verfügung, Signale, Variablen, Prozesse, etc. zu überwachen. Ebenso kann er Signale einer Schaltung auf Fehlerbedingungen hin simulieren, Werte setzen und den Datenfluss kontrollieren. Für den weiteren Verlauf dieser Arbeit spielt speziell die Ausgabe der simulierten Signalwerte als *Waveform* eine wichtige Rolle. Dabei wird der zeitliche Verlauf aller zuvor festgelegten Signale bzw. Variablen einer Schaltungsbeschreibung simuliert und über ein Diagramm grafisch dargestellt. Daneben kann man sich diese Werte mit Hilfe des sog. *List views* auch textbasiert anzeigen lassen. In beiden Darstellungsoptionen kann der Benutzer den zeitliche Ablauf schrittweise interaktiv steuern, um die Auswertung detaillierter durchzuführen. *ModelSim* erlaubt es außerdem, ein Design völlig Skript-basiert zu simulieren.
Bei der Verwendung der hier erläuterten Arbeitsweise eines Hardware-Debuggers stellt die Benutzung von Skripten bzw. Kommandodateien einen wichtigen Aspekt dar. Die Signal werte die durch die Anpassung der Schaltung auf der Hardware gespeichert sind, werden auf einen Computer übertragen, automatisch in ein solches Skript umgewandelt und dienen dann dem Simulator als Stimuli.

3.5 Entwurfsablauf am Beispiel von *Xilinx ISE*

Wenn man heutzutage eine digitale Schaltung entwirft, kommt man um den Begriff des Entwurfsablauf (engl. *Design Flows*) nicht herum. Unter einem Design Flow versteht man im Allgemeinen den iterativen Prozess aller Tätigkeiten von der Eingabe der Schaltungsbeschreibung bis hin zum Vorliegen der „fehlerfreien“ Schaltung für einen bestimmten Chip. Die exakte Reihenfolge der einzelnen Unterschritte kann in Abhängigkeit von Zielarchitekturen und Software-Produkten variieren. In Analogie zu Abbildung 3-1 beginnt der Schaltungsentwurf meist auf der Systemebene mit der Spezifikation des zu entwickelnden Systems. Zusätzlich legt man die Schnittstellen fest und gibt die beschreibt die Schaltung überwiegend auf RT-Ebene. Dazu bietet die Software mehrere Möglichkeiten, so kann man z.B. eine HDL (VHDL, *Verilog*) nutzen, oder man verwendet einen endlichen Automaten, oder man definiert den Aufbau schematisch. Intern werden die meisten dieser Eingabeformat für die weitere Verarbeitung im Rahmen des Design Flows zuerst in VHDL und anschließend in EDIF-Daten umgewandelt. Nach der Schaltungseingabe mit *Xilinx ISE* ist die Synthese der nächste Schritt, bei der die Syntax und die Struktur des Designs überprüft werden. Danach wird die Implementierung

ausgeführt die sich in die vier Abschnitte *Übersetzen*, *Zuordnung*, *Platzierung* sowie *Verdrahtung* unterteilt. Während des *Übersetzens* bildet ein Programm die eingelesene Schaltungsstruktur auf Gatter der systeminternen Bibliotheken ab. Es kommt bei einigen anderen Produkten vor, dass dieser Schritt bereits in der Synthese erfolgt. Das in dieser Arbeit beschriebene Verfahren zum Hardware-Debugging benutzt als Ausgangsformat das beim *Übersetzen* automatisch generierte VHDL-Format des sog. *Post-Translate Simulation Models*. Dabei wird der Aufbau der Schaltung so angepasst, das sie nur noch aus Komponenten einer speziellen Simulationsbibliothek von *Xilinx* besteht. Für den nächsten Schritt, dem *Zuordnen*, wird das logische Design auf konkreten Komponenten des *Xilinx*-FPGAs definiert. Als Eingabedaten für diesen Prozess dient die logische Beschreibung der Schaltung, sowohl mittels eigener Komponenten, als auch auf Basis von *Xilinx*-Grundbausteinen sowie physikalisch vorkommenden Makrozellen des Bausteins.

Die *Platzierung* und *Verdrahtung* werden in der Regel gemeinsam ausgeführt. Das Programm bereitet die logische Beschreibung unter Beachtung einiger Rahmenbedingungen derartig auf, das nur noch einzelne CLBs bzw. Slices, etc. Verwendung finden. Am Ende des Entwurfsablaufes lädt man die generierte binäre *BIT*-Datei auf die Zielarchitektur, wo die Schaltung entsprechend der technischen Möglichkeiten verifiziert und dem Debugging unterworfen wird.

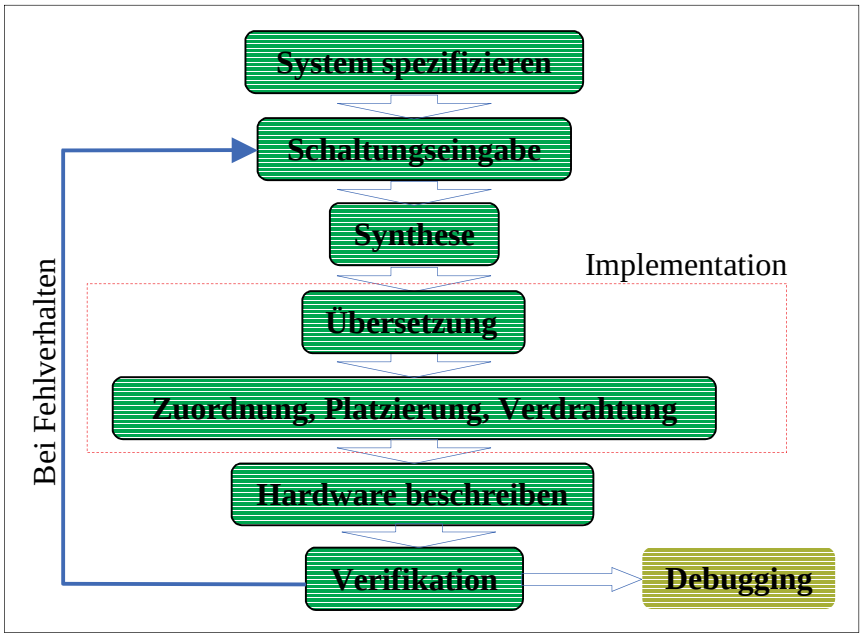

Abbildung 3-7: vereinfachtes Schema des FPGA-Entwurfsablaufes bei *Xilinx ISE*

Kapitel 4

Der Ansatz

Nachdem im vorherigen Kapitel ausführlich die für das weitere Verständnis erforderlichen Grundlagen beschrieben wurden, soll dieses Kapitel nun den im Rahmen dieser Diplomarbeit verwendeten Ansatz näher erklären. Im ersten Teil wird es darum gehen, die zu Grunde liegende Idee hinter der angewendeten Technik tiefer zu beleuchten.
Insbesondere werden die Vorteile des Vorgehens gegenüber anderen Methoden dargestellt. Der zweite Teil dieses Kapitels beschäftigt sich mit den technischen Einzelheiten des umgesetzten Patents für diese Methode des Hardware-Debuggings. Es geht speziell um die Anforderungen an das benutzte Dateiformat. Zusätzlich beschäftigt sich der Abschnitt mit den Vor- und Nachteilen der angewendeten Signalstruktur. Darunter fallen auch Fragen zur Realisierung der globalen Signale für die Schaltung und deren modifizierten Zusatzkomponenten.
Da während dieser Diplomarbeit das Programm NIHD© (engl. ***N**ew **I**ntegrated **H**ardware **D**ebugger*) zur Steuerung des Debugging-Vorgangs entwickelt wurde, werden die einzelnen Modifikationen die im Rahmen des Programms durchgeführt werden übersichtlich erklärt. Dazu zählt ebenfalls der allgemeine Programmablauf, besonders in den Phasen vor einen Fehlverhalten, dabei und das Verhalten nachdem ein Fehler auftrat.

4.1 Die Idee

Ziel des bei dieser Arbeit verwendeten Ansatzes ist es, eine neue Methode des Hardware-Debuggings auf der Grundlage bestehender Techniken zu entwickeln. Es handelt sich dabei um eine spezielle Verbindung von der Simulation und der Emulation. Durch den hier vorgestellten Ansatz zum Hardware-Debugging ist es möglich geworden, Fehlverhalten in Schaltungen noch effizienter als bisher zu beseitigen. Dieses bereits in [Pat01a] patentierte Vorgehen erlaubt sehr hohe Taktfrequenzen der Kernschaltung, da sich die erforderlichen Zusatzkomponenten automatisch an die jeweilige Taktrate anpassen. Der Benutzer bekommt eine völlig abstrakte Sicht auf die zu Grunde liegende Hardware und wird während des gesamten Vorgangs durch eine grafische Oberfläche unterstützt. Sowohl die Analyse des Quellcodes, als auch die notwendigen Modifikationen an der VHDL-Beschreibung, z.B. Breakpoints, erfolgen völlig automatisch. Anders als bei vergleichbaren Ansätzen (vgl. Abschnitt 2.2) wird kein zusätzlicher externer RAM für die Speicherung von Signalwerten benötigt. Gespeichert werden kann eine beliebige Anzahl von Werten der externen Eingangssignale sowie die Zustände aller Flipflops der Schaltung in Abhängigkeit der Größe des FPGAs. Die Veränderungen an der Kernschaltung sind minimal und beeinflussen deren Verhalten nicht. Deshalb ist auch der zusätzliche Schaltungsaufwand im Vergleich zur ursprünglichen Schaltung gering.

Absolut neu ist, dass nicht die internen Signale selbst gespeichert und dann in den Simulator eingesetzt werden, sondern vielmehr der Simulator (z.B. *ModelSim XE*) alle internen Signale aus den gespeicherten externen Eingangsbelegungen gekoppelt mit den Zuständen der Flipflops und der Schaltungsbeschreibung in VHDL rekonstruiert. Der Nutzer kann sich nun das Verhalten aller internen Signale vor, während und nach Auftreten einer Fehlerbedingung schrittweise im Simulator betrachten.

Die Abbildung 4-1 zeigt den grundsätzlichen Aufbau des Hardware-Debuggers. Die Eingabe der Schaltung durch den Nutzer erfolgt meist in VHDL, durch die internen Konvertierungen der *Xilinx*-Software sind aber auch andere möglich. Anschließend durchläuft diese Schaltungsbeschreibung den *Xilinx*-Entwurfsablauf über die *Synthese*, die *Übersetzung* bis hin zum Unterpunkt *Zuordnung*. An dieser Stelle exportiert man die Netzliste des vorliegenden Designs und wandelt sie mit Hilfe mehrerer Xilinx-Werkzeuge in eine nicht synthetisierbare *Xilinx*-Simulationsbeschreibung, das sog. *Post-Translation-Simulation Model*, um. Aufgrund des strukturellen Aufbaus der dadurch erzeugten VHDL-Datei, wird dieses Format im Weiteren explizit als *SVHDL* bezeichnet, SVHDL und

VHDL werden synonym gebraucht. Diese Datei geht zum Einen direkt in den Simulator *ModelSim XE* ein und zum Anderen bildet sie das Ausgangsformat für den in dieser Diplomarbeit entwickelten Hardware Debugger. NIHD© modifiziert die erhaltene SVHDL-Datei im Sinne der Arbeit zu einer erneut synthetisierbaren Beschreibung der Schaltung, die den kompletten FPGA-Entwurfsablauf erneut durchläuft. Die danach erstellte binäre *BIT*-Datei lädt man auf den FPGA des *Spyder-Virtex-X2*-Boards. Die im Fehlerfall ausgelesenen Signalwerte dienen dem *ModelSim XE* nun als Stimuli.

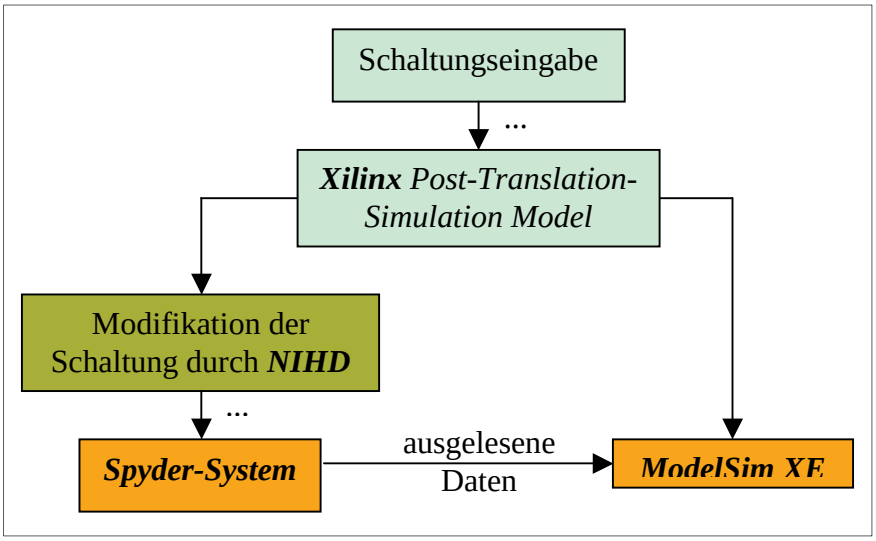

Abbildung 4-1: vereinfachtes Grundschema des Hardware-Debuggers

4.2 Das Prinzip

Wenn man beabsichtigt eine Schaltung entsprechend des in dieser Arbeit beschriebenen Ansatzes auf Fehler zu untersuchen, so exportiert nach der Schaltungseingabe die VHDL-Beschreibung mit Hilfe der *Xilinx ISE*-Software und speichert die entstandene Netzliste im XNF-Format (engl. ***X**ilinx **N**etlist **F**ormat*), bevor man sie mit dem *Xilinx*-Programm *XNF2NGD* in eine *NGO* (engl. ***N**ative **G**eneric **O**bjects*) -Datei umwandelt. Dabei handelt es sich um eine Binärdatei mit der logischen Beschreibung des Designs unter Verwendung spezieller *Xlinx*-Komponenten. Anhand der Abbildung 4.2 ist das beschriebene Vorgehen leicht nachvollziehbar. Das Programm *NGDBUILD* seinerseits transformiert diesen Aufbau in eine weiteres Zwischenformat, dass als NGD (engl. ***N**ative **G**eneric **D**atabase*) bezeichnet wird. *NGDBUILD* passt die Daten während des Vorgangs auf Komponenten an, die speziell für die Simulation entwickelt sind, wobei primär die *SIMPRIM*-Bibliothek (engl. ***Sim**ulation **Prim**itives*) zum Einsatz kommt. Mit Hilfe des *Xilinx*-Programmes

NGD2VHDL kann man aus dieser Netzliste eine neue strukturelle VHDL-Beschreibung, das bereits erwähnte SVHDL-Format, erzeugen. Durch die vorherigen Transformationen ist die dabei erzeugte Schaltungsbeschreibung nur noch rein strukturell und basiert auf Komponenten der *Xilinx*-Bibliothek *SIMPRIM*. Im FPGA-Entwurfsablauf von *Xilinx* wird diese SVHDL-Datei nach dem Übersetzungsschritt als *Post-Translation-Simulation Model* erzeugt, wobei das logische Verhalten dieses Designs identisch mit dem der ursprünglichen VHDL-Verhaltensbeschreibung ist.

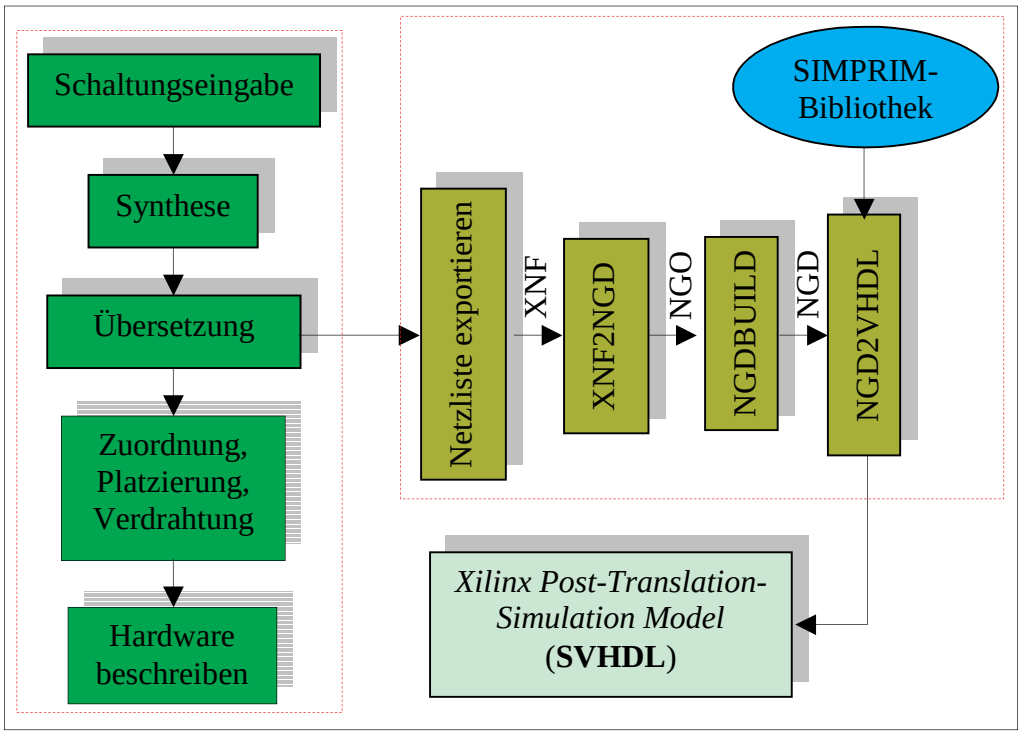

Abbildung 4-2: vereinfachtes Schema zur Erzeugung des *Post-Translation-Simulation Models* durch *Xilinx ISE*

Die *Xilinx*-Software benutzt dieses Dateiformat ausschließlich zu Simulationszwecken, weshalb sich in den Definitionen der einzelnen *SIMPRIM*-Komponenten neben dem logischen Verhalten auch alle spezifischen Zeitangaben, wie z.B. Signal Verzögerungen (engl. *signal delays*) befinden. Das von *NGD2VHDL* erzeugte SVHDL kann laut [NGD99] innerhalb des Design Flows auf noch zwei weiteren Stufen zur Simulation genutzt werden. Im Rahmen der *Xilinx ISE*-Software besteht die Möglichkeit auf Basis der erzeugten SVHDL-Datei auch eine Simulation nach der Zuordnung (engl. *Mapping*) sowie nach der Verdrahtung (engl. *Routing*) durchzuführen. Der durch die Konvertierungen entstandene SVHDL-Code hat daneben den Vorteil, dass aufgrund des rein strukturellen Aufbaus keine funktionalen Anweisungen der Verhaltensbeschreibung wie Schleifen, Bedingungen, etc.

existieren. Aus diesem Grunde ist es auch nicht erforderlich in NIHD© einen eigenen VHDL-Parser für den gesamten *VHDL'93*-Standard der IEEE zu entwickeln, vielmehr genügt es vorerst nur die benutzte Untermenge zu unterstützen.

Dieses automatisch generierte SVHDL bildet den Ausgangspunkt der Arbeit und muss nun zuerst von dem Parser des entwickelten NIHD© analysiert werden, um die Struktur zu erkennen. Durch die komplexen Datenstrukturen ist es dabei möglich geworden, den logischen Aufbau der Schaltung komplett in *Java* nachzubilden. Da jedes VHDL-Design aus nur einer einzigen Hauptkomponente, auch TLE (engl. *Top-Level Entitiy*), bestehen darf, aber mehrere Unterkomponenten (engl. *Sub-Level Entities*) besitzen kann, ist es zwingend notwendig, dass der Benutzer diese Top-Level Entitiy vorher festlegt. Unter erheblichen Einschränkungen kann man diesen Vorgang auch automatisch durchführen, jedoch steigt die Transparenz des Debugging-Vorgangs, wenn der Benutzer die Top-Level Entitiy aus der Liste aller erkannten Entities manuell selektiert.

Die zusätzlich einzufügenden Komponenten für das Debugging benötigen neben den obligatorischen Steuerleitungen auch ein eigenes primäres Taktsignal (engl. *Clock*) sowie ein Rücksetzsignal (engl. *Reset*). Dabei passt sich der NIHD© so an, das er bereits existierende Signale der ursprünglichen Kernschaltung mitbenutzen kann. Wenn z.B. in der Schaltung bereits ein globales Reset-Signal vorhanden ist, so wird dieses mit dem Reset-Pin der neuen Zusatzkomponenten verbunden. Der Vorteil davon ist, dass man einen zusätzlichen externen Pins am FPGA eingespart hat. Der Nachteil dagegen ist, dass dieses externe Reset-Signal dann nicht mehr durch den NIHD© überwachbar ist. Als globales Rücksetzsignal bietet sich das GSR-Signal (engl. *Global System Reset*) an, dass in den meisten Fällen bereits im von der *Xilinx*-Software erzeugten SVHDL-Code benutzt wird. Existiert in der Kernschaltung darüber hinaus bereits ein zentrales Taktsignal, so kann auch dieses analog zum Reset-Signal mitbenutzt werden. Sobald mindestens eines dieser Signale nicht von Xilinx verwendet wird, so fügt der NIHD© automatisch eigene Clock- bzw. Reset-Signale in die Schaltungsbeschreibung ein.

Problematischer ist der Fall, wenn z.B. mehrere Takt- bzw. Rücksetzsignale in der Schaltung vorkommen. Hierbei ist dann unklar, welches dieser vorhandenen Signale auch für die einzufügenden Komponenten verwendet werden soll. Weil das Taktsignal in den Zusatzkomponenten eine Art Abtastrate (engl. *sampling rate*) für andere Signale darstellt, bedeutet das aber auch, dass dieses Clock-Signal sich nicht selbst überwachen kann. Als Taktsignal wird in der Regel das Clock-Signal mit der höchsten Frequenz genutzt. In den kommenden Versionen des NIHDs© ist dieses Problem dadurch behoben, indem ein

separater Takt mit einer vielfachen Frequenz der höchsten in der Schaltung vorkommenden Frequenz benutzt wird.

Neben diesen Angaben muss der Nutzer ebenfalls spezifizieren, welche der ihm angezeigten internen Signale er überwachen möchte. Er wählt dazu all die internen Signale des Schaltungsdesigns aus, an denen er eine Fehlerquelle vermutet. Dort fügt der NIHD© selbstständig an jedes der markierten internen Signale eine Breakpoint-Komponente ein, die auf eine logische Fehlerbedingungen reagiert, wobei die beiden Zustände '*0*' oder '*1*' als Bedingung in Frage kommen.

Neben dem Einfügen der Breakpoints beinhaltet die in der vorliegenden Arbeit verwendete Methode zum Hardware-Debugging noch eine weitere Modifikationen an der Schaltungslogik. Dabei handelt es sich um die erforderlichen Schiebregister (engl. *shift-register*) zur Speicherung von Signalwerten. Zwei verschiedene Arten von Signalen müssen darin gespeichert werden. Beiden steht durch ein generisches Attribut im VHDL-Code eine große Anzahl an freien Speicherstellen in den zugehörigen Schieberegistern zu.

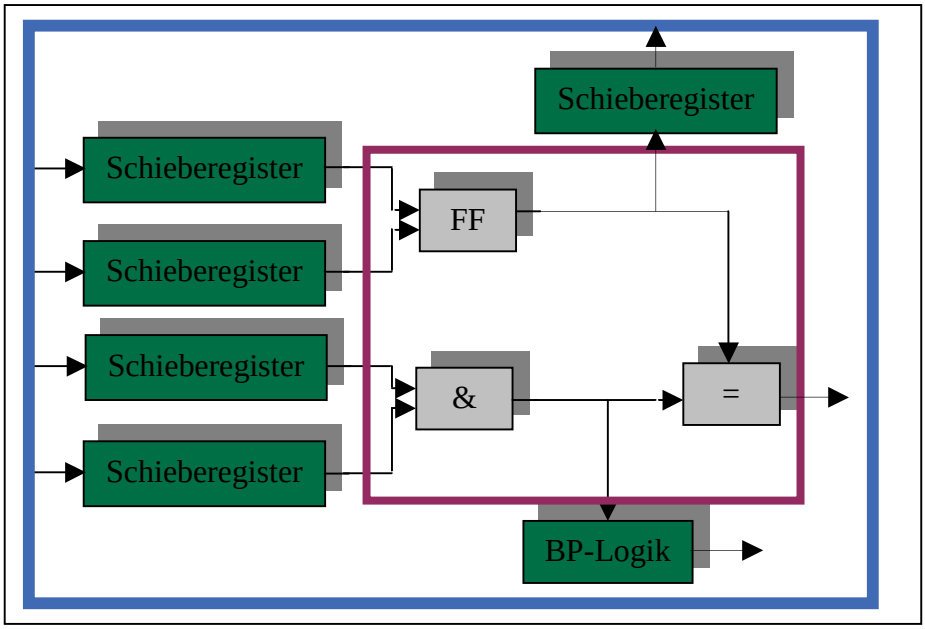

Abbildung 4-3 vereinfachte Übersicht über die durchzuführenden Erweiterungen durch *NIHD*

Die Abbildung 4.3 stellt schematisch den Umfang der Erweiterungen dar. Der violett umrandete Bereich ist die ursprüngliche Kernschaltung mit logischen Grundelementen und einem Flipflop. Die grünen Komponenten zeigen die zusätzlich notwendigen Komponenten. An allen externen Eingangsleitungen der Kernschaltung befindet sich ein Schieberegister mit drei unterschiedlichen Arbeitsmodi. Der Ausgang des Flipflops wird ebenfalls in ein solches Register geleitet. Um einen Breakpoint zu setzen wird das

entsprechende Signal von BP-Logik abgegriffen. Der blau umrandete Bereich ist der Aufbau der Schaltung nach den durchzuführenden Erweiterungen. Um später am Simulator die Werte der internen Signale berechnen zu können, sind einerseits die gespeicherten Signalwerte der primären Eingänge des Designs und andererseits die internen Zustände der Flipflops entscheidend (vgl. Abbildung 4-3). Bei den Flipflops stehen zwei Möglichkeiten zum Speichern ihrer internen Zustände zur Auswahl. Entweder man integriert die Puffer direkt in das Flipflop, oder man passt die Beschreibung der Eingabepuffer so an, dass diese auch als Ausgabepuffer fungieren, um sie anschließend dadurch an die Ausgangsleitung eines Flipflops zu legen. Da die letztere der beiden Möglichkeiten in Bezug auf den Aufwand günstiger ist, verwendet der NIHD© diesen Weg. Aufgrund der Tatsache, das die nach dem Einlesen im Speicher befindliche SVHDL-Beschreibung auf Komponenten der *SIMPRIM*-Bibliothek von *Xilinx* basiert, ist dieses Design nicht synthetisierbar. Daher ersetzt der NIHD© alle *Xilinx*-Komponenten durch das jeweilige synthetisierbare Äquivalent aus seiner umfangreichen Bibliothek.

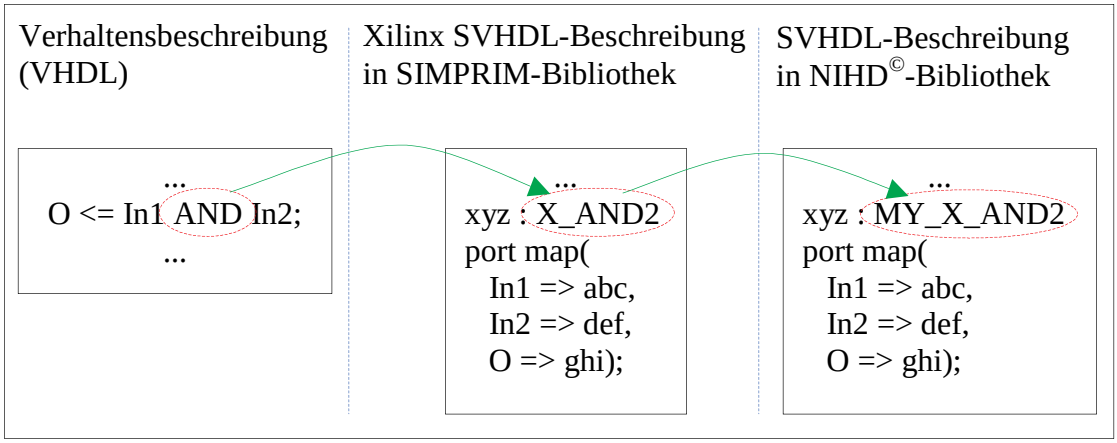

Abbildung 4-4: Umwandlung der Verhaltensbeschreibungen in die NIHD-Strukturen

Grundsätzlich hat jede auftretende Komponente darin ein synthetisierbares Gegenstück mit gleichem logischen Verhalten. Nach diesen Anpassungen der SVHDL-Beschreibung von *Xilinx* befindet sich in den Datenstrukturen von NIHD© eine vollständig synthetisierbare Schaltung mit gleichem Verhalten.

Im Rahmen der Weiterverarbeitung dieser NIHD©-Datenstrukturen werden zwei Dateien erzeugt. In der einen Datei wird die komplette Datenstruktur aus NIHD© in eine neue VHDL-Datei abgebildet. Dazu zählen auch alle Modifikationen und Optimierungen. Analog dazu erzeugt der NIHD© eine weitere neue VHDL-Datei, ebenfalls mit logisch äquivalenten Verhalten, in der jedoch sämtliche Anpassungen nicht integriert sind. Beiden Dateien ist gemein, dass sie nur noch auf Komponenten der hier entwickelten Bibliothek

basieren. Die zuletzt genannte Datei wird ausschließlich zu Simulationszwecken von *ModelSim XE* verwendet, während die andere Datei auf der dafür vorgesehenen Hardware umgesetzt wird. Für diese Umsetzung muss die generierte VHDL-Datei erneut den gesamten FPGA-Entwurfsablauf von *Xilinx* abarbeiten.

Der FPGA auf dem *Spyder-Virtex-X2*-Board wird nun aktiviert und über die Ethernet-Verbindung des *Spyder*-Systems mit der dazu erforderlichen *BIT*-Datei beschrieben. Sobald dieser Vorgang abgeschlossen ist, versetzt man diese Schaltung über das Rücksetzsignal in einen vordefinierten Zustand. Danach ist ihr allgemeiner Betriebsmodus erreicht und sie beginnt beliebig viele Stimuli seitens des *Spyder-Core-P2*-Boards zu verarbeiten. Während dieser gesamten Zeit arbeiten die Schieberegister transparent, d.h. sie speichern zwar den anliegenden Wert an ihrem Eingang, behindern ihn aber nicht. Sobald an einem der zahlreichen durch den NIHD© eingefügten Breakpoints eine Fehlerbedingung erfüllt wird, meldet dieser das Ereignis sofort an alle Schieberegister sowie über eine spezielle Steuerleitung an den Benutzer weiter. Die Register wechseln dadurch ihren internen Arbeitsmodus der besagt, dass nach einer festgelegten Anzahl von Takten keine weiteren Signalwerte mehr zu speichern sind, bzw. der zentrale Takt unterbrochen wird. Insbesondere für die Puffer an den externen Eingängen der Schaltung bedeutet das, dass auch keine weiteren Stimuli mehr in die Kernschaltung hineingelassen werden - somit wird dadurch die gesamte Schaltung „eingefroren“ (engl. *freeze*). Würde sowohl das Taktsignal, als auch die eigentliche Kernschaltung, wie bei einigen anderen Ansätzen, weiter aktiv sein, so würde man für den hier verwendeten Ansatz zusätzliche Signale benötigen. Jedes zusätzliche Signal impliziert jedoch einen weiteren Port, der belegt werden muss. Desweiteren wird NIHD© in der kommenden Version so erweitert, dass die Schaltung nach dem Auslesen der Werte exakt an dem Punkt weiterlaufen kann, an dem sie angehalten wird.

Dem Benutzer steht im *freeze*-Modus ein Steuersignal zur Verfügung, mit dem er alle Puffer in ihren nächsten Arbeitsmodus versetzen kann. Bei jeder steigenden Flanke des Taktsignals wird jeweils einer der gespeicherten Werte aller eingefügten Schieberegister in umgekehrter Reihenfolge ausgegeben. Um die Schaltungsanpassungen so gering wie möglich zu halten, ist es sinnvoll die Ausgabe parallel zu implementieren. Dieser Ansatz impliziert als Nachteil allerdings auch die Verwendung zusätzlicher externer Pins. In der nächsten Version des NIHDs© ist dieser Nachteil behoben, entweder durch eine serielle Übertragung, oder durch die Verwendung einer vorhandenen Schnittstelle wie z.B. JTAG.

Da eine Verbindung über Ethernet existiert, werden die ausgelesenen Daten darüber sofort an den PC gesendet. Beginnt der Benutzer nun das Taktsignal zu steuern, so wird bei jeder steigenden Flanke jeweils ein Wert aus allen Registern gleichzeitig durch das *Spyder-Core-P2*-Board ausgelesen und an den angeschlossenen Computer übertragen. Die Übertragung wird dabei über die vorliegende Ethernet-Schnittstelle realisiert. Auf dem PC läuft derweil der Simulator *ModelSim XE* der Firma *ModelTechnology*. Als Grundlage für die Simulation benutzt man die vom NIHD© ohne die zusätzlich eingefügten Komponenten generierte VHDL-Beschreibung. Die Stimuli für den Simulator stammen von den ausgelesenen externen Inputs sowie die gespeicherten internen Werten aller verwendeten Flipflops. Mit Hilfe dieser beiden Daten kann man den kompletten Zustand der Schaltung, inklusive aller internen Signale, zu jedem Zeitpunkt exakt nachbilden. Da nicht nur ein einziger Wert gespeichert wurde, kann der Entwickler den Zustand des gesamten Systems vor dem Fehlverhalten, während, und auch danach schrittweise analysieren um die Fehler einzeln zu beseitigen. Nachdem alle Fehler in diesem Bereich erkannt und beseitigt sind, kann der Entwickler mit dem NIHD© erneut Breakpoints an andere interne Leitungen setzen. Diese neue modifizierte Schaltungsbeschreibung durchläuft den FPGA-Entwurfsablauf und wird erneut auf den FPGA geladen, während die im Simulator befindliche Schaltungsbeschreibung unverändert weiterbenutzt wird und kein Neustart notwendig ist.

Kapitel 5

Die Implementierung

Das Kapitel 5 befasst sich mit den technischen Details für die Umsetzung der in dieser Arbeit vorgestellten Methode zum Hardware-Debugging. Dazu ist es notwendig näher auf die Bestandteile und Datenstrukturen des NIHDs© einzugehen. Einführend wird deshalb der Parser für die SVHDL-Dateien umfassend beschrieben, wobei die Weiterverarbeitung der eingelesenen Daten in den jeweiligen Datenstrukturen besonders wichtig ist.

Neben diesen Beschreibungen des Programms an sich, wird auch der Umfang der durchzuführenden Modifikationen dargestellt, insbesondere der Aufbau der erforderlichen Zusatzkomponenten sowie die Realisierung der NIHD©-Bibliothek auf der Basis der *SIMPRIM*-Bibliothek von *Xilinx* müssen tiefer betrachtet werden.

Nach dem Software-Abschnitt werden die betreffenden Details der verwendeten Hardwarekonfigurationen eingehender untersucht. Da es sich nicht um den originalen technischen Aufbau des Spyder-Systems handelt, werden die erforderlichen Änderungen bei der Zusammenstellung dieser Hardware zusammengefasst. Abschließend ist es erforderlich die grundsätzlichen Funktionsabläufe vom NIHD© im Zusammenhang mit den neu entwickelten Komponenten zu erläutern. Insbesondere wird detailliert die Initialisierung des *Spyder-Core-P2*-Boards, das Laden des FPGAs das Steuern selbigen und das Auslesen der Register beschrieben.

Diese Daten müssen nun in dem Simulator *ModelSim XE* verarbeitet werden. Deshalb beschäftigt sich ein eigener Abschnitt mit den dafür erforderlichen Konvertierungen der Daten, bishin zur Durchführung einer Auswertung im Simulator.

5.1 *New Integrated Hardware Debugger* (NIHD©)

Der NIHD© besteht aus zwei getrennten Bereichen die verschiedene Funktionen zu erfüllen haben. Der erste Teil ist der im Abschnitt 5.1.1 erläuterte Parser, der zweite umfasst alle Operationen der Klasse *VHDLFileCreator*. Um diese Funktionen im

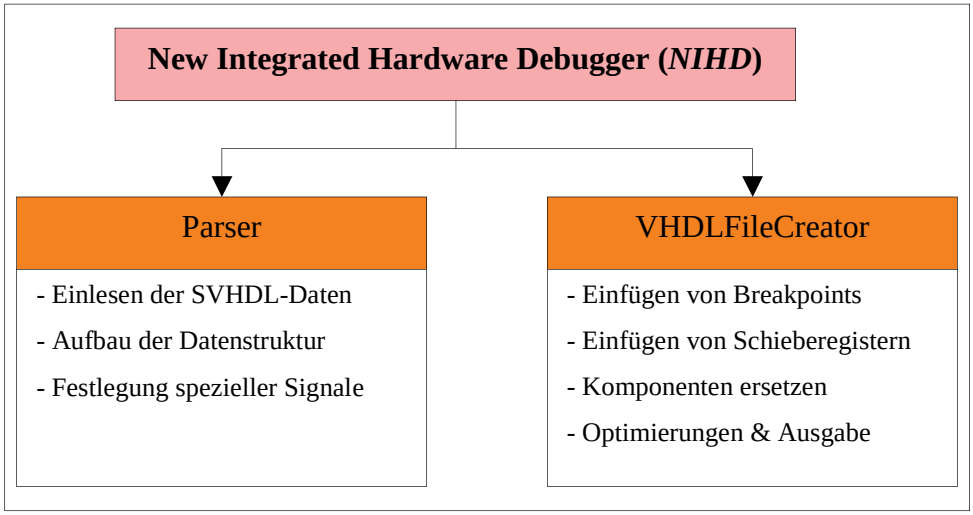

Abbildung 5-1: allgemeiner Aufbau und Aufgaben von *NIHD*

Einzelnen erklären zu können, ist es notwendig zuvor auf die entwickelten Datenstrukturen einzugehen. Zur Um die Logik einer Schaltung nachzubilden, bedarf es grob betrachtet nur Komponenten und Signalen. Daher wurden primär diese beiden Objekte in zwei abstrakten *Java*-Klassen umgesetzt. Da sie selber nicht instanziiert werden, übernehmen alle davon abgeleiteten Unterobjekte die in den abstrakten Klassen festgelegten Eigenschaften. So implementiert die Komponenten-Klasse (*Component*) nach Abbildung 5-2 einige Attribute die allen Unterklassen eigen sind. Dazu zählen u.a. eine eindeutige Kennung

```
public abstract class Component {
        protected int componentID                   =       -1;
        protected String instanceName               =       null;
        protected String vhdlComponentName          =       null;
        protected String javaComponentName          =       null;
        protected String genericName                =       null;
        protected String genericValueString         =       null;
        protected String vhdlEntityString           =       null;
        protected String vhdlDeclarationString      =       null;
        protected ConnectionsVector conVec          =       null;
        protected int numberOfConnections           =       0; …
```

Abbildung 5-2: Attribute der abstrakten *Component*-Klasse

(*componentID*), der Instanzname (*instanceName*), der Name der Komponente in VHDL (*vhdlComponentName*) sowie der Name der entsprechenden *Java*-Klasse

(*javaComponentName*). Darüber hinaus beinhaltet diese abstrakte Klasse Variablen für generische Attribute. Die für alle abgeleiteten Unterklassen typische Repräsentation als VHDL-Code wird jeweils statisch in zwei großen Strings gespeichert. Hierbei stellt *VhdlEntityString* die Definition einer Komponente dar, während *vhdlDeclarationString* die Deklaration beinhaltet. Somit ist bei einer Änderung des Verhaltens einer Komponente nur das Anpassen dieser einen Klasse notwendig.

Bei den Signalen ist die Implementierung ähnlich gestaltet (vgl. Abbildung 5-3). Eine eindeutige Kennung (*signalID*) erlaubt die genaue Zuordnung. Jedes Signal besitzt

```
public abstract class Signal {
        protected int       signaled              =       -1;
        protected String    signalName            =       null;
        private   String    initialValue          =       null;
        protected int       datatype              =       -1;
        protected int       vectorStart           =       -1;
        protected int       vectorEnd             =       -1;
        protected int       typeOfClass           =       -1;
        protected int       breakpointCondition   =       -1;
        private   int       clockFlag             =       0;
        private   int       resetFlag             =       0;
        protected boolean originalSignal          =       true;
…
```

Abbildung 5-3: Attribute der abstrakten *Signal*-Klasse

weiterhin einen eindeutigen Namen (*signalName*). Für Anpassungen der SVHDL-Beschreibung war es erforderlich Datentyp (*datatype*), Vektorgrössen (*vectorStart*, *vectorEnd*) und eine Reihe von Flags zur Markierung besonderer Signale zu verwenden. Die beiden möglichen Ausprägungen der Signal-Klasse stellen einerseits die internen Signale (*InternalSignal*), also alle Verbindungen zwischen den Ports von Komponenten und andererseits die externen Signale (*ExternalSignal*), alle Verbindungen zu Pins des

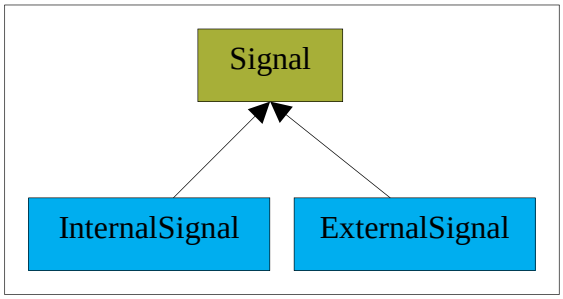

Abbildung 5-4: Hierarchie der *Signal*-Klasse

FPGAs, dar. Aufgrund der Komplexität vieler Schaltungsbeschreibungen sowie unter Beachtung des objektorientierten Paradigmas ist es nötig, Signale und Komponenten in der Datenstruktur komplett zu trennen. Um dennoch eine Verbindung beider Klassen zu

gewährleisten, wurde die Klasse *Connection* entwickelt. Jede Komponente beinhaltet deshalb eine Instanz des Vectors (*ConnectionsVector*) mit allen Verbindungen (*Connections*). Eine Instanz vom Typ *Connection* stellt die Verbindungslogik (engl. *glue logic*) zwischen dem Port genau einer Komponente und einem Signal her (vgl. Abbildung 5-5). Da eine Komponente in der Regel zahlreiche Ports hat, die zwar innerhalb der Komponente eindeutig sind, schaltungsweit jedoch nicht, wird auf die Zuweisung einer eindeutigen ID für jeden Port verzichtet. Daher bildet der Name eines Ports der Komponente (*componentSideName*) die eine Seite der *Connection* und die eindeutige *signalID* des mit diesem Port verbundenen Signals die andere. Die einzelnen Anbindungen der Komponente werden in dem erwähnten *ConnectionsVector* gespeichert. Er wird von der *Java*-Klasse im *JDK* bereits vorhandenen Klasse Vector abgeleitet und enthält bereits die notwendigen *Get*- bzw. *Set*-Methoden zum Auslesen und Schreiben von Daten.

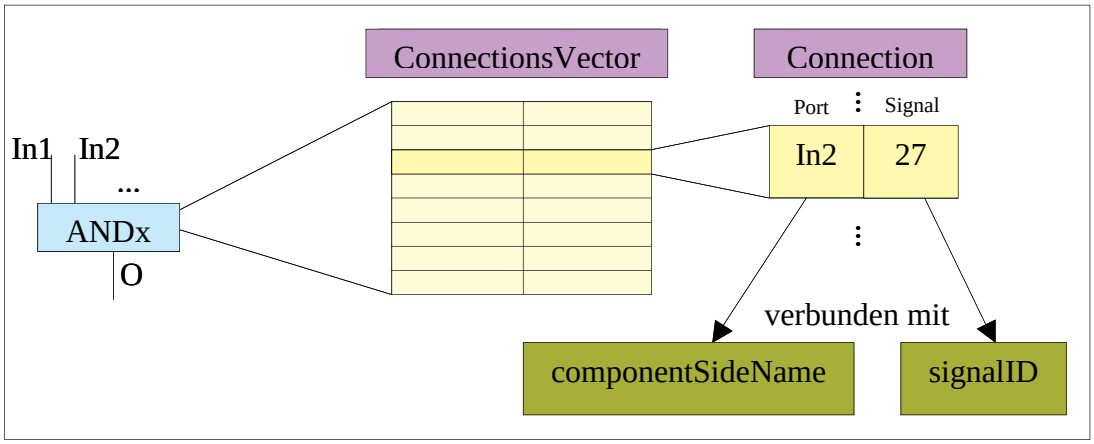

Abbildung 5-5: vereinfachte Darstellung der Einbindung eines n-fachen *AND*s

Neben den genannten Vektoren existieren aber noch weitere, wie z.B. der *ComponentVector*, in dem alle Objekte von Komponenten einer Schaltung gespeichert werden. Zusätzlich dazu gibt es noch einen Vektor der für die Speicherung aller Signalobjekte (*SignalVector)* verantwortlich ist. Insbesondere für die Komponenten ist in diesem Zusammenhang ein weiterer Vektor von Bedeutung. Mit Hilfe des sog. *AliasVectors* ist es möglich dynamisch während der Laufzeit von NIHD©-Instanzen neuer Komponenten einzufügen. Dazu ist es nicht erforderlich den gesamten Quellcode neu zu kompilieren. In einer Datei (*alias.lst*) stehen die Namen der VHDL-Komponenten mit dem Namen der zugehörigen Java-Komponente. NIHD© liest diese Datei ein und erzeugt zu jeder der Zuordnungen ein *AliasPair*-Objekt, dass im *AliasVector* gespeichert wird (vgl. Abbildung 5-6). Während der NIHD© die SVHDL-Datei einliest, kann später zur Laufzeit

mittels des Namens der *Xilinx*-Komponente über den *AliasVector* die entsprechende NIHD©-Komponente mit der Anweisung *„(Class.forName(className)).newInstance()“* dynamisch instanziiert.

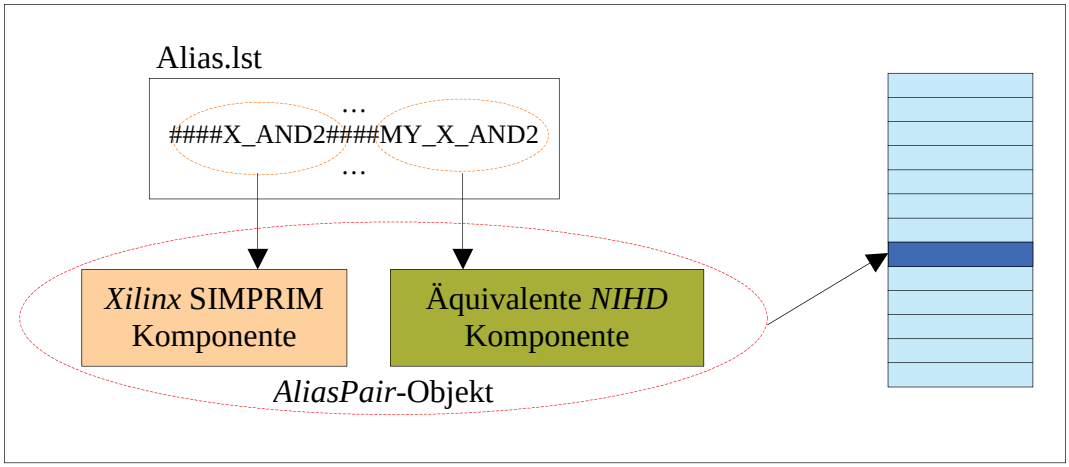

Abbildung 5-6: vereinfachter Aufbauvorgang der *Alias*-Daten

Auf Grundlage dieser hier erklärten Datenstrukturen lassen sich im Folgenden die Funktionsweisen der beiden Bestandteile des NIHDs© (Parser und *VHDLFileCreator*) effizienter darstellen.

5.1.1 Der Parser

Um an einer Schaltung Veränderungen vornehmen zu können, ist es erforderlich deren Struktur genau zu kennen. Da für NIHD© eine SVHDL-Beschreibung der Ausgangspunkt ist, muss dieses menschenlesbare Format zuerst mit Hilfe eines in *Java* geschriebenen Parsers genau analysiert werden. Einerseits durch die langjährigen Erfahrung mit dieser Sprache, andererseits wegen der guten Portierbarkeit auf andere Plattformen fiel die Entscheidung auf *Java*. Insbesondere da einige der Softwarepakete zum Schaltkreisentwurf nicht unter den Windows-Plattformen benutzt werden, sondern auch unter Plattformen wie z.B. *Linux* oder *Sun Solaris* bietet Java zum Bau eines Parsers die besten Möglichkeiten.
Ziel des entwickelten Parsers war es eine vorgegebene SVHDL-Datei zu analysieren, bevor deren Daten anschließend vom NIHD© weiterverarbeitet werden. Als Grundlage für den Parser dient eine bereits auf *Xilinx*-Komponenten umgewandelte Schaltungsbeschreibung (SVHDL). Es handelt sich dabei um das *Post-Translate-Simulation-Model* auf Grundlage der *SIMPRIM*-Bibliothek unter Verwendung der *VITAL* (engl. ***V**HDL **I**nitiative **T**owards **A**SIC **L**ibraries*) Bibliothek von Xilinx, in der die

zeitspezifischen Eigenschaften der Komponenten definiert sind. Das gesamte logische Verhalten der Schaltung wurde im Designschritt *Übersetzen* (engl. Translate) auf Komponenten dieser Bibliothek abgebildet. Das Design ist nun rein strukturell. Die dabei verwendeten Datenstrukturen werden im Abschnitt 5.1 ausführlich behandelt.

Für das Einlesen der Daten verwendet der Parser vom NIHD© die vordefinierte Java-Klasse *BufferedReader* in Verbindung mit der Klasse *FileReader* sowie die Methode *readLine()*, die ein zeilenweises Einlesen erlaubt (vgl. Abbildung 5-7). Zu Beginn des

```
…
BufferedReader vhdlBuffReader = new BufferedReader(new FileReader(vhdlFileName));
while (vhdlBuffReader.ready()) {
        vhdlLine = new String((vhdlBuffReader.readLine()).toUpperCase().trim());
…
```

Abbildung 5-7: Code-Ausschnitt für das Einlesen der SVHDL-Datei

Parsing benötigt der NIHD© den Namen der TLE (***T**op-**L**evel **E**ntity*). Entities können generell recht einfach im Code gefunden werden, da sie mit dem Schlüsselwort *Entitiy* beginnen. Dazu läuft das Programm einmal durch die komplette SVHDL-Datei. Danach bekommt der Benutzer eine Liste aller gefundenen Entities und wählt die gewünschte TLE aus.

Bei der folgenden zweiten Analyse der Datei werden die genauen Inhalte analysiert. Mit Hilfe einer großen *while*-Schleife wird die gesamte Datei zeilenweise eingelesen und verarbeitet. Der Beginn jeder Zeile ist verantwortlich dafür, von welcher der zahlreichen *if*-Verzweigungen diese Zeile verarbeitet werden soll. Jede Zeile wird vor der Behandlung in den entsprechenden *if*-Schleifen mit der vorhandenen Methode „*toUpperCase()*“ in Großbuchstaben umgewandelt, wobei die Methode *trim()* alle Leerzeichen davor und dahinter entfernt. Ist die Länge einer Zeile kleiner oder gleich Null, dann wird die nächste eingelesen. Im folgenden müssen die Kommentare behandelt werden. Alle Zeichen hinter den beiden Zeichen ‘--’ gelten laut VHDL-Standard als Kommentar. Sie treten meist in einer eigenen Zeile auf, können aber auch nach validen VHDL-Anweisungen stehen. Dies stellt einen Nachteil des zeilenweisen Einlesens durch die Klasse *BufferedReader* dar, da ein zeichenweises Einlesen unter dieser Vorraussetzung sicher vorteilhafter wäre. Um diese Zeilen dennoch weiter verarbeiten zu können, muss der Parser intern eine neue Zeichenkette bilden, die aus allen Zeichen bis zum Auftreten von ‘--’ besteht. Zu den Schlüsselworten mit niedriger Priorität, die aber ebenfalls behandelt werden müssen, gehören u.a. „*Library*“ und „*Use*“, durch die alle VHDL-Bibliotheken eingebunden sind.

Wenn der Parser auf eine Zeile trifft, die mit dem Schlüsselwort *Entity* beginnt, so sind weitere Unterscheidungen zu machen. Handelt es sich bei dem Namen dieser *Entity* nicht um den zuvor festgelegten Namen der TLE, so müssen die in den Block enthaltenen Anweisungen nicht weiter behandelt werden. Sobald jedoch die Namen der beiden *Entities* übereinstimmen, markiert der Parser in den Datenstrukturen mit Hilfe eines Attributes,

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
…
entity xor6 is
  port (
    out1 : out STD_LOGIC; in1 : in STD_LOGIC := 'X';
…
  ); end xor6;

architecture Structure of xor6 is
  component TOC  generic (InstancePath: STRING := "*"; WIDTH : Time := 0 ns);
    port (O : out STD_ULOGIC := '1');  end component;   …
  signal N_in1 : STD_LOGIC;
....
begin
  C0 : X_BUF  port map ( I => in1, O => N_in1 );   …
end Structure;
```

Abbildung 5-8: vereinfachte Beispiel-Datei

dass es sich dabei um die TLE handelt.

Da die Beschreibung einer Komponente immer aus zwei Abschnitten (*Entity und Architecture*) aufgebaut ist (vgl. Abbildung 5-8), so erfordert dies eine getrennte Verarbeitung. Der *Entity*-Abschnitt beginnt mit *entity* und wird durch das Wort „*end*“ geschlossen. In den dazwischenliegenden Zeilen befinden sich die Definitionen der primären externen Signale. Der Parser muss jetzt unterscheiden, wie diese Definitionen angeordnet sind. Der VHDL-Standard besagt jedoch auch, dass alle Zeilen zwischen dem Schlüsselwort „*port map(*" und der dazu schließenden Klammer, externe Signale darstellen. Somit stehen dem Benutzer zahlreichen Arten der Notation zur Verfügung. So könnte er beispielsweise sämtliche externen Signale in eine einzige Zeile schreiben, oder Leerzeilen und Kommentare in diese Definitionen einfügen. Alle diese Sonderformen muss bzw. kann der Parser vom NIHD© erkennen. Ein externes Signal selbst hat folgende Struktur: „*SignalName : direction datatype*“. Um den Parser nicht unnötig mit Konvertierungen zu belasten, verlagert der Parser die Analyse solcher Zeilen in die jeweiligen Datenstrukturen. Dort wird zuerst der Datentyp überprüft. Falls es sich um einen Vector (*STD_LOGIC_VECTOR oder BIT_VECTOR*) handelt, so wird dieses Signal sofort intern in einzelne Signale aufgespalten (engl. *flatten).* Für die weitere Verarbeitung

hat sich dieser Schritt als sinnvoll erwiesen, da jedes erzeugte Signalobjekt eindeutig sein muss. Ein VHDL-Vektor hingegen besteht aus mehreren Signalen. Eventuelle Initialwerte der externen Signale werden ignoriert, da sie bei der Synthese nicht erlaubt sind. Nachdem die externen Signale verarbeitet wurden, wartet der Parser auf das Schlüsselwort „*Architecture*“. Damit wird der Abschnitt eingeleitet, in dem die Verhaltensbeschreibung der TLE implementiert ist. Aufgrund des rein strukturellen Aufbaus stehen dort keinerlei funktionale Anweisungen. Es ist daher nur notwendig, den Parser nach internen Signalen und den Instanzen der deklarierten Komponenten suchen zu lassen. Beendet wird dieses Segment durch die Anweisung „*end*“. Die internen Signale beginnen mit dem Schlüsselwort „*signal*“, gefolgt von dem Namen des Signals und dessen Datentyp. Eine eingelesene Zeile wird ebenso wie ein externes Signal behandelt. Der einzige Unterschied bei internen Signalen im Vergleich zu externen ist das Nichtvorhandensein einer Richtungsangabe. Trifft der Parser nach einer Reihe interner Signale auf die Anweisung „*begin*“, so folgen danach die einzelnen Instanzen der Komponenten. Für jede auftretende Komponente in diesem SVHDL-Quellcode existiert in den Datenstrukturen von NIHD© eine äquivalente *Java*-Klasse, die alle grundlegenden verhaltenstechnischen Eigenschaften der Komponenten sowie deren vollständige Definition als VHDL-Code enthält (vgl.

```
…
protected String getVhdlDeclarationString() {
        this.vhdlDeclarationString = new String("component MY_X_AND2\n");
        this.vhdlDeclarationString = vhdlDeclarationString + "port\t(\n";
        this.vhdlDeclarationString = vhdlDeclarationString + "I0 : in std_logic;\n";
        …
        return(this.vhdlDeclarationString); }

protected String getVhdlEntityString() {
        this.vhdlEntityString = new String("library IEEE;\n");
        this.vhdlEntityString = vhdlEntityString + "use IEEE.std_logic_1164.all;\n\n";
        this.vhdlEntityString = vhdlEntityString + "entity MY_X_AND2 is\n";
        this.vhdlEntityString = vhdlEntityString + "port\t(\n";
        this.vhdlEntityString = vhdlEntityString + "I0 : in std_logic;\n";  …
        return(this.vhdlEntityString); }…
```

Abbildung 5-9: integrierte Deklaration und Definition einer Komponente im *NIHD*

Abbildung 5-9).

Eine Instanz besteht in der ersten Zeile grundsätzlich aus einem Instanznamen und dem Namen der ursprünglichen Komponente, auf die sie sich bezieht, d.h.“*name of instance : name of component*“ (vgl. Abbildung 5-9). Im Anschluss daran folgen nach „*port map(*„ die einzelnen Verbindungen von Signalen (extern und intern) mit den Pins der einzelnen Komponenten in Form von „*component_port_name => signal_name*“. Nachdem eine

komplette Instanz eingelesen ist, wird sie der Methode „*createComponent*“ als Zeichenkette übergeben, um anhand dessen die gesamte Instanz durch die Datenstrukturen vom NIHD© nachzubilden. Dabei wird die äquivalente Komponente im NIHD© als eigenes *Java*-Objekt generiert, gespeichert und alle Verbindungen eins-zu-eins nachgebildet. Darüber hinaus erkennt der Parser auch generische VHDL Attribute der Komponenten. Den Abschluss beim Parsen bildet das Schlüsselwort „*end nameOfArchitecture*“. Danach liegt im Speicher, durch entsprechende NIHD©-Datenstrukturen repräsentiert ein komplettes Abbild der eingegebenen SVHDL-Datei vor.

5.1.2 Die Klasse VHDLFileCreator

Nachdem die SVHDL-Datei unter Verwendung der erwähnten Datenstrukturen analysiert ist, beendet der Parser seine Tätigkeit, wodurch die generierten Datenstrukturen mit Hilfe der NIHD©-Klasse *VHDLFileCreator* weiterbehandelt oder optimiert werden. Diese Daten übergibt der Parser über einen Konstruktor an die Klasse *VHDLFileCreator*. Dazu zählt der *SignalVector*, der *ComponentVector* sowie ein temporärer Vektor *VHDLHeader* mit allgemeinen Informationen zur TLE und den zu benutzenden Bibliotheksanweisungen.
Um den späteren Bedarf an logischen Gattern zu verringern, sind einige Optimierungen an der Beschreibung des Schaltungsnetzes nötig. Diese optionalen Anpassungen sind insoweit erlaubt, sofern. In dieser Version vom NIHD© beschränken sich die Optimierungen auf die verwendeten *Xilinx*-Puffer *X_BUF* und *X_CKBUF* der *SIMPRIM*-Bibliothek (vgl. Abbildung 5-10).

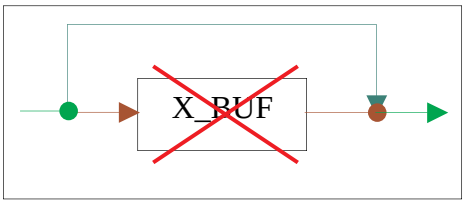

Abbildung 5-10: Optimierung eines *X_BUF*s

Mit Hilfe der entwickelten Methode *redirect()* im NIHD© ist es möglich, die Komponente, die in der Schaltung deaktiviert werden soll, als Parameter anzugeben, um sie danach automatisch aus dem *ComponentVector* löschen zu lassen. Der spezielle Algorithmus hinter diesem Vorgehen ist recht simpel. Zuerst sucht man über den Namen das Objekt im *ComponentVector*, danach liest man dessen Signalverbindungen aus, um diese nun so umzuleiten, das man die Komponente überbrückt. Da die so entstandene Schaltung redundante Verbindungen enthält, müssen diese im Anschluss daran entfernt werden.

Die Funktionen der Klasse *VHDLFileCreator* umfassen aber noch mehr, so bekommt der Benutzer neben der Gelegenheit die erwähnten Optimierungen durchführen zu lassen, auch die Möglichkeit aus einer Liste aller internen Signale einer Schaltung diejenigen auszuwählen, die überwacht werden sollen. Nachdem mindestens ein internes Signal ausgewählt wird, fragt das Programm die primären externen Reset- und Taktsignale ab, um anschließend die Schaltung mit den beschriebenen Zusatzkomponenten und ihren erforderlichen Verbindungen zu erweitern.
Insbesondere bei den Rücksetzsignalen muss jedoch noch eine Anpassung vorgenommen werden. Während der Erzeugung des SVHDLs durch die *Xilinx ISE*-Software wird eine *ROC*-Komponente eingefügt (vgl. Abschnitt 5.2.3), diese liefert ein Reset-Signal (*GSR*) für alle getakteten Komponenten, speziell die Flipflops, der Beschreibung. Da auch die neuen Zusatzkomponenten des NIHDs© ein Rücksetzsignal benötigen, vermischt man die beiden Steuersignale.

5.2 NIHD©-Komponenten

5.2.1 Breakpoints

Mit Hilfe einer Breakpoint-Komponente (*BP*) ist es möglich sämtliche Signalleitungen in einer Schaltung auf ein zuvor festgelegtes logisches Verhalten hin zu überwachen. Die benutzerdefinierte Bedingung (engl. *condition*), auf die der Breakpoint reagieren soll, wird bei VHDL mit Hilfe eines generischen Attributes definiert. Als Datentyp, mit dem dieses Attribut implementiert wird, verwendet der NIHD© den Typ „*natural*“. In VHDL wird dies folgendermaßen umgesetzt: „*generic (condition: natural range 0 to 1)*“. Als Schnittstelle nach außen benötigt eine solche Komponente insgesamt drei Signale (vgl. Abbildung 5-11). Dabei liefert das Input-Signal den Wert des zu überwachenden Signals und die high-

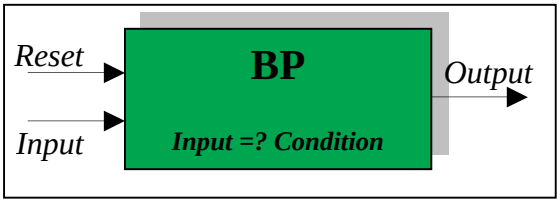

Abbildung 5-11 Breakpoint-Komponente

aktive Reset-Leitung initialisiert den Breakpoint. Das Output-Signal erzeugt bei Erfüllung der gespeicherten Bedingung eine logische ‘*1*’, anderenfalls ist dieses Signal auf einer logischen ‘*0*’. Intern wurde die Funktionalität des BP durch einen eigenen Prozess

umgesetzt, der auf Ereignisse an dem zu überwachenden Signal sowie am *Reset* aktiv reagiert Der Zustand des BPs wird dabei mit Hilfe eines internen Signals (*currentState*) gehalten. Zu Beginn einer Debugger Sitzung werden alle Komponenten der Schaltung, die eine Initialisierung benötigen, über ein zentrales Reset-Signal in einen vordefinierten Zustand versetzt. Im Fall der BPs wird der interne Zustand *(currentState)* auf eine logische '*0*' gesetzt. Solange nun dieser Zustand auf einer logischen '*0*' steht, wartet der BP auf Ereignisse. Sobald der Wert am Eingang mit der im generischen Attribut festgelegten Bedingung übereinstimmt, setzt die Komponente das Output-Signal auf eine logische '*1*' und hält diesen Wert konstant. Erst beim nächsten Reset kann dieser Zustand zurückgesetzt

```
entity MY_BP is
 generic (condition: natural range 0 to 1);
  port(MY_BP_input : in std_logic; MY_BP_reset : in std_logic;
       MY_BP_occured : out std_logic);
end MY_BP;

architecture MY_BP_arch of MY_BP is
 signal currentState : std_logic;
 begin
  bp_process : process(MY_BP_input, MY_BP_reset, currentState)
   begin
    if (MY_BP_reset = '1') then  currentState <= '0'; else
     if (currentState = '0') then
      if (( MY_BP_input = '1' AND condition = 1) OR
          ( MY_BP_input = '0' AND condition = 0) ) then
       currentState <= '1';
      end if; end if; end if;
  end process;
  MY_BP_occured <= currentState;
end MY_BP_arch;
```

Abbildung 5-12: Implementierung eines Breakpoints

werden (vgl. Abbildung 5-12).

Da innerhalb einer Schaltung viele BP-Komponenten existieren können, müssen deren Ergebnisse gebündelt werden. Dazu wurde eine sog. *BP_Collector*-Komponente (Abbildung. 5-13) entwickelt, in der diese Bündelung umgesetzt wurde. Diese

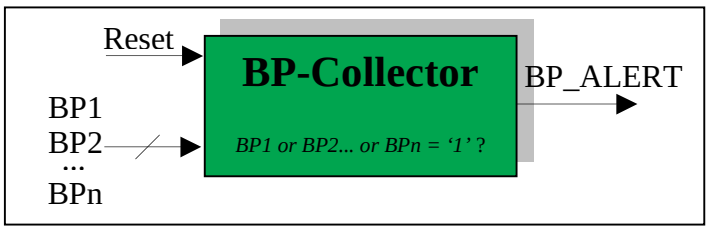

Abbildung 5-13: Breakpoint_Collector

Komponente benötigt zum Zurücksetzen ihrer Logik ein high-aktives Reset-Signal, ein Output-Signal zum Weitergeben des Resultates sowie je ein Input-Signal für den Ausgang jedes BPs. Dieses Zusammenfassen der Breakpoint-Werte ist ebenfalls durch einen eigenen Prozess implementiert. Beim Rücksetzen wird der interne Zustand auf eine logische '*0*' gesetzt. Die Signale die von den einzelnen BPs kommen, werden logisch mit einer *OR*-Funktion verknüpft. Sobald auch nur eines dieser Signal auf einer logischen ‚*1*' steht, wird der interne Zustand der BP_Collector-Komponente sofort auf den konstanten Zustand ‚*1*' gesetzt und kann erst durch ein erneutes Reset zurückgesetzt werden. Gleichzeitig wird dieser Wert an alle Schieberegister geleitet, wodurch sie veranlaßt sind bei der nächsten steigenden Flanke des Taktes ihren Modus auf den *breakpoint mode* zu setzen. Des weiteren erfährt der Benutzer über ein eigenen Pin (*BP_ALERT*) in der TLE, dass ein BP auftrat.

5.2.2 Schieberegister

Die Schieberegister bilden bei der vorliegenden Arbeit die Grundlage zur Auswertung der Schaltung im Simulator. Erst durch sie ist es möglich, diese Arbeit entsprechend der Aufgabenstellung umzusetzen. Diese Register werden durch den NIHD© automatisch an allen externen Signalen der Schaltung integriert. Darüber hinaus ist es für die vorgestellte Methode zwingend notwendig diese Schieberegister auch an alle Flipflops eines Designs anzuschließen, um deren internen Zustand zu speichern. Das Hauptziel dieser Erweiterungen besteht neben der Speicherung von Werten aber auch in Steuerung der ursprünglichen Kernschaltung. Die Speicherung ist implementierungstechnisch durch ein sog. *slice shifting* umgesetzt (siehe. Abbildung 5-14), das bedeutet: Sobald ein neuer Wert gespeichert werden soll, muss die Position *Null* des Puffers durch die Anweisung '*internalBuffer(1 to 15) := internalBuffer(0 to 14)*' freigeschoben werden. Der am Eingang des Registers anliegende Signalwert wird nun an der freien Position *Null* über die Anweisung '*internalBuffer(0) := MY_INPUT_BUFFER_INPUT*' gespeichert. Ein solches Schieberegister arbeitet in drei verschiedenen Modi, dem sog. *standard mode*, den *breakpoint mode* sowie dem *transmission mode*. Die Schnittstellen nach außen umfassen die in Abbildung 5-15 dargestellten Ports. Ein Eingangsport wird benötigt, um anliegende Signalwerte zu verarbeiten, zu speichern und weiterzuleiten. Da es sich bei den Schieberegistern grundsätzlich um Speicherelemente für externe Eingangssignale der Schaltung handelt, ist neben einem Ausgangssignal auch ein *Buffered*-Signal

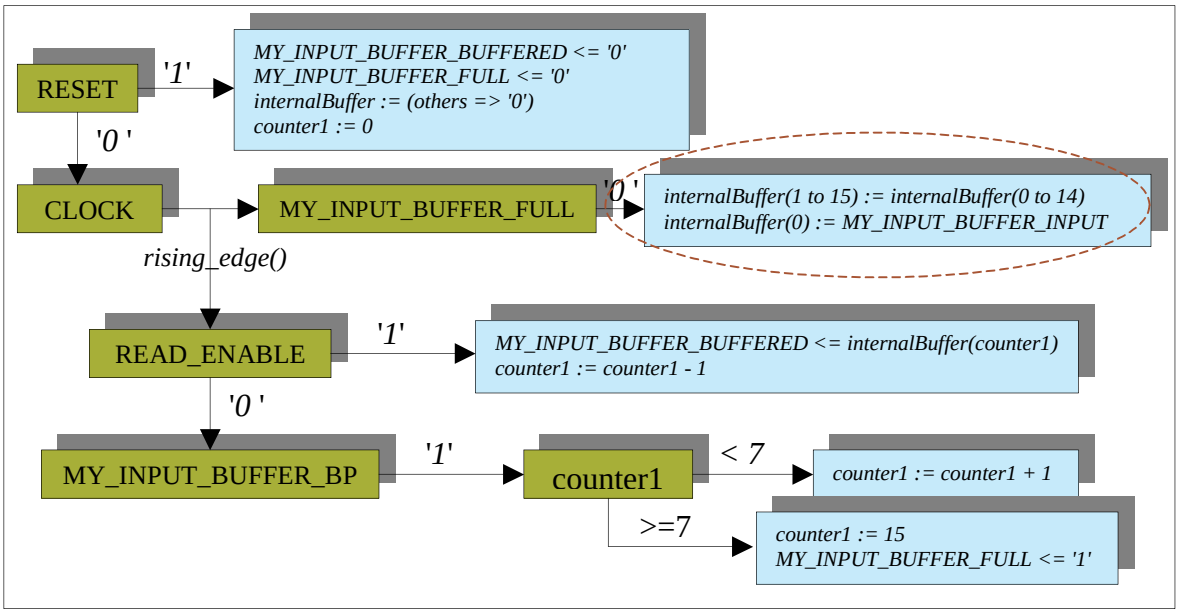

Abbildung 5-14 Arbeitsweise eines 16-Bit breiten Schieberegisters

implementiert. Dieses ist notwendig da eine bidirektionale Verbindung sämtlicher Eingange einer Schaltung einen nicht unerheblichen Aufwand darstellt. Das Ausgabesignal ist deshalb ausschließlich zur Weiterleitung in die ursprüngliche Kernschaltung

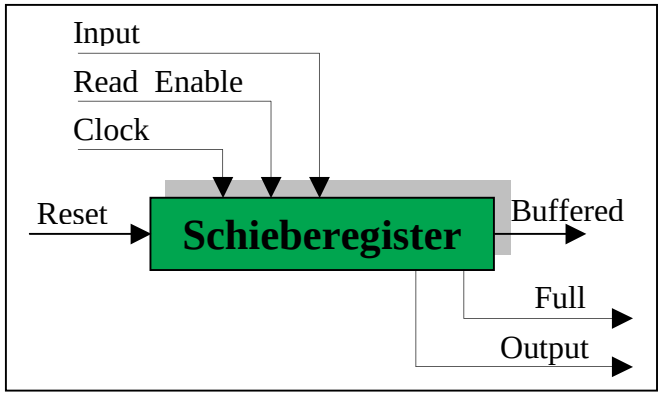

Abbildung 5-15: definierte Schnittstellen eines Schieberegister

vorgesehen, während das *Buffered*-Signal dagegen nur für das Auslesen der gespeicherten Werte benutzt wird. Jedes *Buffered*-Signal ist aus diesem Grund mit einem explizit zur Ausgabe von Werten reservierten Pin in der TLE bzw. dem FPGA verbunden. Anders als die BP-Logik sind die Schieberegister getaktet und benötigen zum Initialisieren ein high-aktives Reset-Signal. Da das aus den einzelnen BPs kommende und in der *BP_Collector-Komponente* berechnete „Alarmsignal“ (engl. *Alert*) über das Auftreten eines BPs informiert, muss auch dieses in sämtlichen Schieberegistern der Schaltung als Eingabeport eingehen. Wenn nun ein Puffer die zuvor festgelegte Anzahl von Werten gespeichert hat,

ist es erforderlich dem Benutzer diesen Zustand anzuzeigen. Dafür besitzt das Register ein weiteres Ausgabesignal (*BUFFER_FULL*) das mit einem eigenen Port in der TLE verbunden ist. Da in einer Schaltung gewöhnlich mehrere Register existieren, werden diese jeweiligen *BUFFER_FULL*-Signale zusammengefasst. In Analogie zu den *BP_Collector*-Komponenten ist eine. *BUFFER_FULL_Collector*-Komponente. Deren Aufgabe besteht in der Verknüpfung jedes *BUFFER_FULL*-Signals der einzelnen Schieberegister mittels einer mehrfachen *AND*-Funktion. Das bedeutet, erst wenn alle Puffer komplett gefüllt sind, wird der Benutzer durch ein externes *BUFFER_FULL*-Signal in der TLE über diesen Zustand der Puffer informiert.

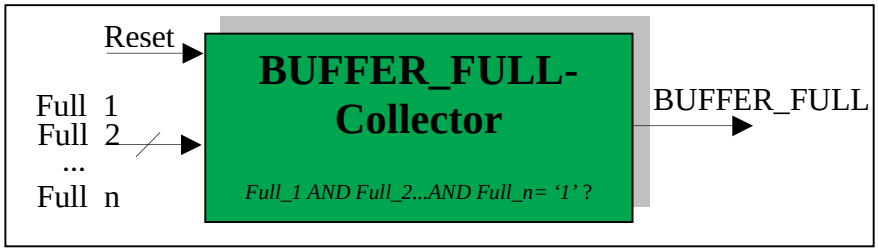

Abbildung 5-16: BUFFER_FULL-Collector

Nachdem alle Puffer gefüllt sind und auf den Auslesevorgang warten, kann der Benutzer diesen Schritt durch das externe Steuersignal *READ_ENABLE* starten.

Wie sieht die Modi eines solchen Schieberegisters im Einzelnen aus? Solange kein Breakpoint auftritt, arbeitet es im *standard mode* völlig trasparent. wobei zu jeder steigenden Taktflanke der am Eingangsport anliegenden Signalwert im internen Puffer gespeichert wird. Neben der Speicherung muss diese Komponente dafür sorgen, dass der Wert ungehindert auch auf ihren Output geleitet wird, wodurch die dahinterliegende Kernschaltung unmittelbar diesen Wert als Stimuli bekommt. Wenn ein Breakpoint auftritt, wechselt das Register zum *breakpoint mode*, in dem primär weiterhin die Werte wie im *standard mode* gespeichert werden, jedoch die Aufzeichnung nach einer gewissen Anzahl von Taktzyklen anhält. Die Anzahl der Werte die nach einem BP noch gespeichert werden dürfen, wird zuvor über ein generisches Attribut im VHDL-Code festgelegt und liegt grundsätzlich bei der Hälfte der gesamten Registerbreite. Für die hier durchgeführten Versuche wird der Einfachheit halber ein Registerbreite von 16 Bit angenommen. Nach Beendigung des Speichervorgangs werden auch keine weiteren Signalwerte mehr in die eigentliche Kernschaltung geleitet, was dazu führt das diese ursprüngliche Schaltung angehalten wird. Mit Hilfe des *READ_ENABLE*-Signal, dass als eigener Eingabe-Pin am FPGA dem Nutzer auf der TLE-Ebene zur Verfügung steht, kann er das Schieberegister

veranlassen in den *transmission mode* zu wechseln. Nach diesem Wechsel liegt an allen Puffern bei jeder steigenden Taktflanke genau einer ihrer gespeicherten Werte in umgekehrter Reihenfolge auf ihrem *BUFFERED*-Port an. Aufgrund der durch NIHD© vorgenommenen Erweiterungen an der SVHDL-Beschreibung, wird jedes dieser Signale auf einen eigenen Ausgabe-Pin des FPGAs gelegt. Über den *SH3*-Prozessor werden nun diese am FPGA anliegenden Werte parallel ausgelesen und im Simulator weiterverarbeitet. Wenn alle Werte ausgelesen wurden, beginnt der Auslesemechanismus durch die implementierte Ringstruktur der Schieberegister erneut von vorn.

5.2.3 Weitere Komponenten

Da bei dieser Arbeit ein von *Xilinx ISE* automatisch erzeugtes VHDL-Format (SVHDL) verwendet wird, ist es erforderlich einige Besonderheiten bei dessen Weiterverarbeitung zu beachten. Bei der Abbildung der vom Benutzer eingegebenen Verhaltensbeschreibung auf die spezifischen Simulationskomponenten von *Xilinx* treten neben leicht nachzubildenden logischen Grundkomponenten wie z.B. *X_ANDx* und *X_ORx*, auch komplexere Komponenten auf (vgl. Abbildung 5-17). Deklariert sind alle *Xilinx*-Komponenten in der

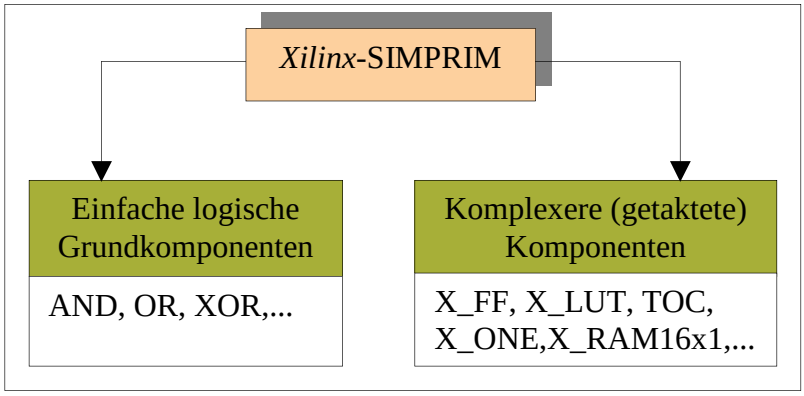

Abbildung 5-17: Einteilung der Xilinx SIMPRIM-Bibliothek

sog. *SIMPRIM*-Bibliothek von *Xilinx*, definiert werden sie auf der Basis von Grundkomponenten der *VITAL*-Bibliothek (*Xilinx*) sowie zugehörigen Timing-Informationen. In der *VITAL*-Bibliothek wird u.a. festgelegt welche Signalverzögerungen (sog. *VitalDelayType*) einzelne Verbindungen innerhalb einer Komponente haben sollen. Außerdem wird der zeitliche Unterschied (engl. *glitch*) zwischen zwei logischen Zuständen bei einer Flanke festgelegt. Als unterste Basiskomponenten existieren u.a. *VitalStateTable*, bzw. *VitalMUX* die eine Art programmierbare Look-up Tabelle, bzw. Multiplexer darstellen.

Zu den spezielleren Komponenten im SVHDL-Code zählen unter anderem TOC (engl. ***T**ristate **O**n **C**onfiguration*) und ROC (engl. ***R**eset **O**n **C**onfiguration*), deren Aufgabe die hardwareähnliche Initialisierung einer Schaltung für Simulationszwecke ist. Umgesetzt wird dieses Verhalten, indem das Ausgangssignal zu Beginn auf einer logischen '0' steht, darauf folgend für eine vordefinierte Zeit auf eine logische '1' wechselt, um abschließend erneut auf eine '0' zu wechseln. Das beschriebene Verhalten wird bei den beiden Komponenten durch zwei verschiedene Signale realisiert. Eine *TOC*-Zelle benutzt dazu das *GTS*-Signal (engl. ***G**lobal **T**ristate **S**ignal*), dass alle Ausgänge des Design in einen hochohmigen Zustand versetzt. Durch das *GSR*-Signal (engl. ***G**obal **S**et/**R**eset*) bringt die *ROC*-Zelle alle Komponenten eines Designs in ihren vordefinierten Zustand. Im Zuge der Weiterentwicklung der *Xilinx*-Simulationsmodelle, insbesondere ab Virtex-FPGA, werden diese beiden Komponenten in der SIMPRIM-Bibliothek durch eine sog. *STARTUP_VIRTEX*-Komponente ersetzt. Sie liefert neben *GTS* und *GSR* auch noch weitere Signale zur Initialisierung. Um die Logik der Schaltung so wenig wie möglich zu verändern, behandelt der NIHD© auch die gerade beschriebenen Komponenten zur Initialisierung eines Designs ohne das sie jedoch bei der weiteren Synthese durch Xilinx beachtet werden, d.h. sie werden wieder wegoptimiert. Da deren interner Aufbau aus den *Xilinx*-Beschreibungen bekannt ist, ist die Umsetzung vergleichbare Zellen recht einfach. Deren ursprüngliches logisches Verhalten muss allerdings wegen den nicht erforderlichen Timing-Informationen für die Synthese geringfügig angepasst werden. Für den zweiten Syntheselauf betrachtet Xilinx ISE die modifizierte Schaltung als Einheit und fügt deshab erneut ROC, TOC, etc. ein, während die durch NIHD©-Komponenten ersetzten ROC und TOC aus dem ersten Syntheselauf wegoptimiert werden.

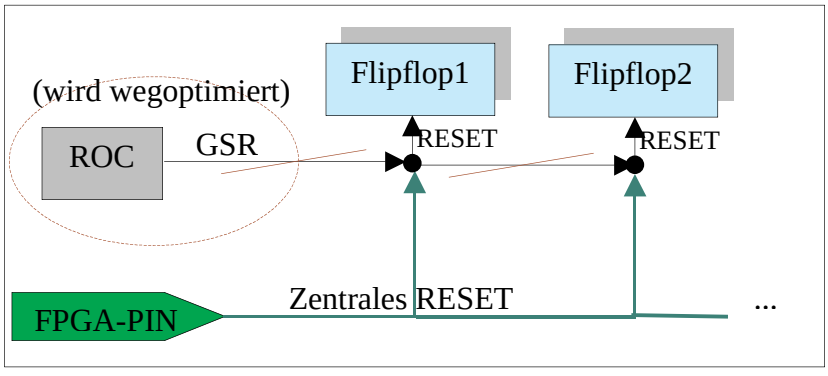

Abbildung 5-18: Entfernung der ROC-Komponente aus dem Design

Neben diesen Initialisierungskomponenten gibt es auch eine Reihe von Zellen, deren Verhalten nicht unmittelbar durch logische Grundfunktionen nachgebildet werden kann.

Dazu zählen u.a. Flipflops (*X_FF*) und Lock-up Tabellen (*X_LUTx*). Die *LUTs* der NIHD©-Bibliothek sind durch zahlreiche Konvertierungen implementiert, so zeigt Abbildung 5-19 z.B. den VHDL-Code für eine 4-fach Lock-up Tabelle.

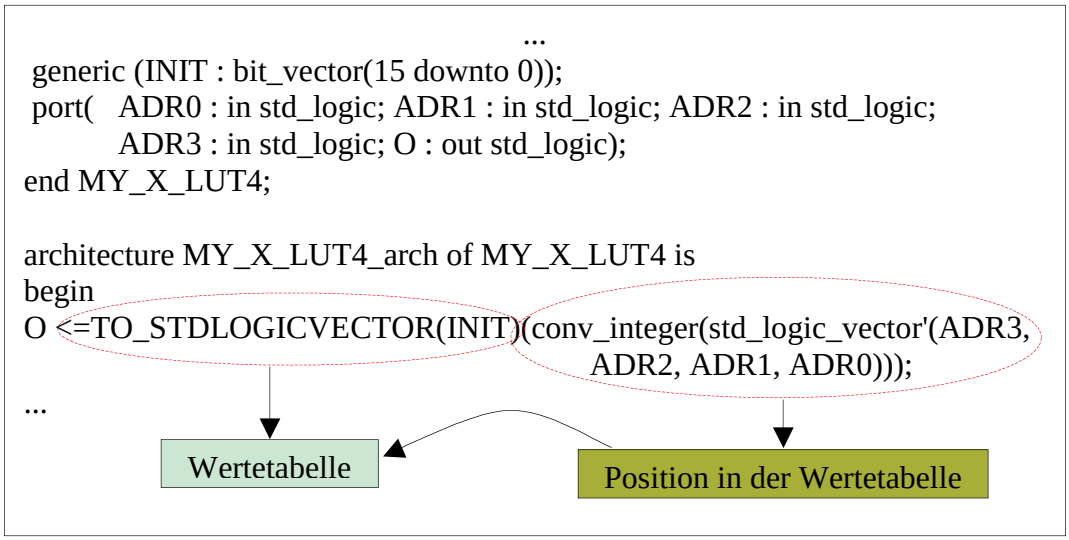

Abbildung 5-19: Umsetzung einer 4-fach LUT in NIHD©

Zuerst werden die Adressleitungen der Komponente in einen *STD_LOGIC_VECTOR* zusammengefasst, um daraus im Anschluss einen Integer-Wert zu generieren. Das beim Bilden einer Instanz anzugebende *INIT*-Attribut für eine 4-fach LUT muss hexadezimal vier Stellen enthalten. Auch dieser Wert wird in einen *STD_LOGIC_VECTOR* umgewandelt und stellt die binäre Wertetabelle dar. Über erzeugten Integer-Wert wählt man die jeweilige Position in der Wertetabelle aus, die als Ergebnis ausgegeben wird.

Bei dem am häufigsten auftretenden Flipflop handelt es sich um einen „*D-Flipflop mit asynchronen Set/Reset und Clock Enable*“. Für die Nachbildung im NIHD© ist zu beachten, dass die verschiedenen Eingänge unterschiedliche Prioritäten besitzen. Die höchste Priorität besitzt das Reset-Signal des Flipflops, im Anschluss daran wird das SET-Signal abgefragt. Nur wenn an diesen beiden Signalen eine logische ‘0’ anliegt, werden die restlichen Signale beachtet, zu denen das *Clock Enable* zählt, dass zum Setzen des Flipflops eine logische ‘1’ haben muss. Nachdem diese Bedingungen erfüllt sind, kann man am Input den zu speichernden Wert angelegen, der schließlich bei der nächsten Taktflanke übernommen wird (vgl. Abbildung 5-20).

Zu den Komponenten deren Notwendigkeit im Rahmen dieser Arbeit umstritten ist, da ihre Funktionen auch ohne eigene Komponenten umsetzbar sind, zählen *X_ONE* sowie *X_ZERO*. Die *X_ONE*-Komponente liefert statisch eine logisch ‘0’, während X_ZERO eine logische ‘1’ erzeugt. Darüber hinaus besitzt die *SIMPRIM* Bibliothek von *Xilinx* noch

zahlreiche weitere Komponenten für Puffer, Schieberegister, etc., die im *Post-Translate-Simulation-Model* auftreten können.

```
                                ...
ffprocess : process (CLK, CE, RST, SET)
begin
 if (RST = '1') then internalState <= '0';
  else
  if (SET = '1') then internalState <= '1';
  else
   if (CE = '1') then
    if rising_edge(CLK) then internalState <= I;
    end if;    end if;    end if;    end if;
 end process;
 O <= internalState;
                                ...
```

Abbildung 5-20: Implementierung eines Flipflops in der NIHD©-Bibliothek

5.3 Konfiguration der Hardware

Die vorliegende Arbeit verwendet aufgrund der guten Steuerungsmöglichkeiten des FPGAs durch den *SH3*-Prozessor sowie wegen der aktuellen Verfügbarkeit das *Spyder*-System als Zielarchitektur. Zur Realisierung der Kommunikation zwischen *SH3*-Prozessor und *Virtex*-FPGA existieren zwei verschiedene Verbindungswege zwischen den jeweiligen Boards. Die Verwendung der sog. *back-plane* als Schnittstelle stellt dabei die erste Variante dar, mit der die gesamte Kommunikation mit dem FPGA abläuft. Die parallele Schnittstelle (technisch als *J9* bezeichnet) hingegen wird ausschließlich für die Konfiguration des FPGAs eingesetzt. Innerhalb der beiden Boards spielt jeweils ein CPLD (*XC95144XL*) eine entscheidende Rolle bei der Umsetzung der Signale. Auf dem *SH3-Board* ist dieses CPLD u.a. über acht Daten- und 24 Adressleitungen mit dem *SH3*-Prozessor verbunden. Neben diesen Leitungen bestehen auch Verbindungen zu der parallelen Schnittstelle des Boards. Dazu zählen laut Abbildung 5-21 die acht Bits des *Dout*-Signals, die einem weitergeleiteten acht Bit breitem Abschnitt des *SH3*-Datenbusses entsprechen. Weiterhin gehören in diese Kategorie ein CS-Signal (*Chip-Select*), ein RW-Signal (*Read/Write*) sowie das ADR-Signal (*Addres-Select*), welches das unterste Bit des *SH3*-Adressbusses reflektiert. Durch ein Flachbandkabel werden diese Signale über das parallele Interface an das *Spyder-Virtex-X2*-Board angeschlossen, auf dem sie direkt mit

einem weiteren CPLD (*XC95144XL*) verbunden sind. Dort werden diese Signale

J9-Pin	SH3-CPLD-Pin	Signal
J9-2	112	Data(0)
J9-3	115	Data(1)
J9-4	117	Data(2)
J9-5	119	Data(3)
J9-6	121	Data(4)
J9-7	125	Data(5)
J9-8	128	Data(6)
J9-9	130	Data(7)
J9-11	111	CS
J9-13	116	ADR
J9-14	139	RW

Abbildung 5-21: Verbindungen zwischen J9 und SH3-CPLD

verarbeitet und ausschließlich zur Steuerung des FPGAs verwendet. Neben dem Flachbandkabel besteht darüber hinaus eine weitere Verbindung zwischen dem CPLD und

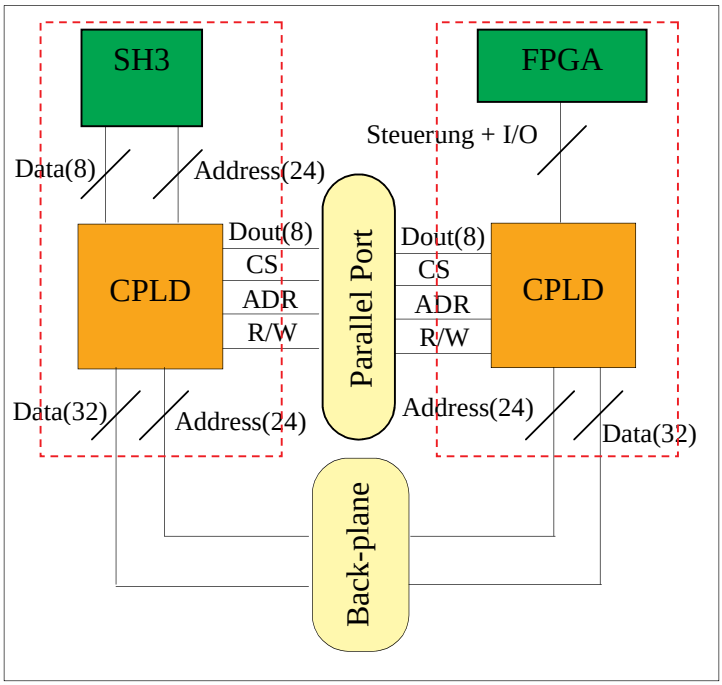

Abbildung 5-22: vereinfachtes Schema der Verbindungen zwischen *SH3* und FPGA

der *back-plane* des *Spyder*-Systems (vgl. Abbildung 5-22). Der Erweiterungsstecker (engl. *Extension-Header*) des *Spyder-Core-P2-Board*s ist dabei an den *Extension-Header 2* des *Spyder-Virtex-X2*-Boards angeschlossen, wodurch der 24-Bit breite Adressbus und der 32-

Bit breite Datenbus des Prozessors mit zahlreichen frei konfigurierbaren Benutzereingabe- bzw. Benutzerausgabe-Pins des FPGAs verdrahtet ist. Praktisch stehen zum Steuern eines FPGAs jeweils nur acht der Datenleitungen und acht der Adressleitungen (Bit 2-9) zur Verfügung. Insgesamt werden alle Kommunikationswege zwischen den beiden Boards immer über die jeweiligen CPLDs durchgeführt. Daneben initialisieren diese CPLDs ebenfalls den sog. *SelectMap-Mode*, der beim Beschreiben des FPGAs mit der *BIT*-Datei verwendet wird.

5.3.1 Initialisierung des *Spyder-Core-P2*-Boards

Die Initialisierungssequenz des *Spyder-Core-P2*-Boards besteht aus mehreren Teilabschnitten. Unmittelbar nach dem Einschalten startet das Board, indem es seine Anfangskonfiguration aus dem integrierten Flash-Speicher ausliest, wodurch der SH3-Prozessor mit seiner Grundfunktionalität geladen wird. Dadurch wird der erweiterte Ladevorgang über das Ethernet (sog. *Net-Boot*) aktiviert, bei dem der Prozessor den gesamten *Kernel* aus einem über das Internet angeschlossenen FTP-Server (engl ***F**ile*

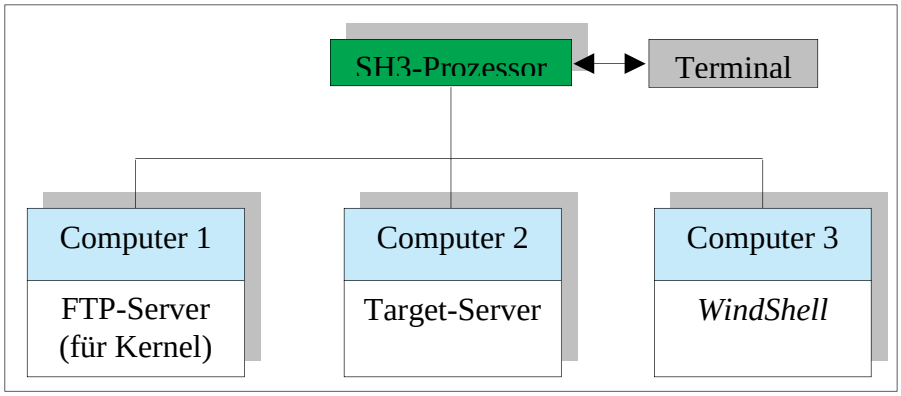

Abbildung 5-23: Netzwerkanbindungen des SH3-Prozessors

***T**ransfer **P**rotocol*) einliest. Derzeit existieren drei verschiedene Anbindungen über diese *Ethernet*-Verbindung, zur Beeinflussung der Funktionalität des Systems. Die erste Variante ist FTP-Server, der den Kernel für den *SH3*-Prozessor bereitstellt. Des weiteren ermöglicht ein *Target-Server*, der auf dem Entwicklungs-PC läuft, die Bereitstellung von Netzlaufwerken über *TCP/IP* und somit auch von weiteren Objektmodulen zum Laden in den *SH3*. Die dritte Möglichkeit ist eine textbasierte Oberfläche (*WindShell*) die dem Benutzer für die Steuerung zur Verfügung steht. Sie erlaubt u.a. das dynamische Laden bzw. Löschen von Modulen in bzw. aus dem Kernel. Alle drei Anwendungen können laut Abbildung 5-23 sowohl auf einem, als auch auf drei unterschiedlichen Computern laufen.

Nachdem dieser Ladevorgang beendet ist, befindet sich auf dem SH3-Prozessor ein voll funktionsfähiger *Kernel* des Echtzeit-Betriebssystems *VxWorks*. Dieser Kernel sowie die in ihm vorhandenen Softwarekomponenten wurden mit Hilfe eines sog. *Cross-Compilers* speziell für diese Zielarchitektur (*SH3*) kompiliert. Durch die Fähigkeit des dynamischen Einbindens (engl. *dynamic linking*) neuer *Objektmodule* ist es möglich, die Funktionalität des SH3-Prozessors während der Laufzeit beliebig anzupassen und neue Module zu kompilieren bzw. zu laden. Die darüber hinaus existierende serielle Schnittstelle auf diesem Board wird zum Anschluss der Terminal-Konsole benutzt.

5.3.2 Laden des FPGAs

Nachdem man eine Schaltung entwickelt hat und sie vom NIHD© entsprechend erweitert ist, möchte man die durch *Xilinx ISE* daraus erzeugte *BIT*-Datei auf das FPGA übertragen und testen. Da auch hierfür die Möglichkeiten des *SH3*-Prozessors Anwendung finden, benötigt man ein eigenes *VxWorks*-Objektmodul, welches das Laden des FPGAs durchführt. In der vorliegenden Arbeit wird das mit Hilfe des entwickelten Moduls *Load_FPGA* durchgeführt. Durch den Cross-Compilers wird die Datei für die *SH3*-Architektur kompiliert und liegt anschließend als Objekt-Datei auf einem der am Target-Server angeschlossenen Netzlaufwerke bereit. Über die *WindShell* lädt man diese Datei mit dem Befehl '*ld Load_FPGA.o*' auf den Prozessor und bindet sie dadurch dynamisch in den bestehenden Kernel ein.

Um nun das FPGA mit einer *BIT*-Datei zu beschreiben wird ausschließlich die parallele Schnittstelle des *Spyder*-Systems verwendet. Seitens des *SH3*-Boards benutzt man die in Abbildung 5-21 aufgezählten Signale. Da der FPGA für seine Initialisierung entsprechend der *Xilinx*-Datenblätter sowohl Steuersignale empfängt, als auch Statusinformationen an den *SH3* sendet, muss die parallele Schnittstelle bidirektional aufgebaut sein. Aufgrund der geringen Anzahl gleichzeitig übertragbarer Signale benutzt man dabei das *ADR*-Signal zur Unterscheidung ob vom FPGA gelesen (*ADR=1*), oder geschrieben (*ADR=0*) wird. Das Objekt *Load_FPGA* auf dem SH3 steuert diese Vorgänge und beginnt zuerst mit der Startsequenz für das Virtex-FPGA. Sobald der Chip initialisiert ist, überträgt das Modul die *BIT*-Datei vom *Target-Server*. Ist der Übertragungsvorgang abgeschlossen, sendet das FPGA das erforderliche Statussignal an das Modul. Ab jetzt steht ein einsatzbereites Design auf dem FPGA zur Verfügung und wartet auf seine Ausführung.

5.3.3 Steuern des Designs auf dem FPGA

Nachdem der Ladevorgang abgeschlossen ist, wird der Parallelport deaktiviert. Die gesamte Kommunikation läuft ab dann über die *back-plane*. Da von den darüber übertragenen Signalen nur acht Adressleitungen und acht Datenleitungen indirekt mit dem FPGA nutzbar sind, sind einige Anpassungen an den Schaltungen erforderlich, um mehr als nur acht Signale austauschen zu können. Die vom NIHD© erzeugte SVHDL-Datei muss dazu in zwei anderen Entities instanziiert werden, bleibt selbst jedoch unverändert. Die äußerste TLE ist *Debug*, diese instanziiert eine weitere mit dem Namen *Dummy* und diese wiederum die vom NIHD© erstellte Datei. Dieser Aufbau ermöglicht es beliebig viele Stimuli gleichzeitig in die Schaltung zu senden, aber auch beliebig viele Werte auszulesen, für die Versuche ist diese Anzahl auf 32 Bit beschränkt (vgl. Abbildung 5-24). Das *CS*-

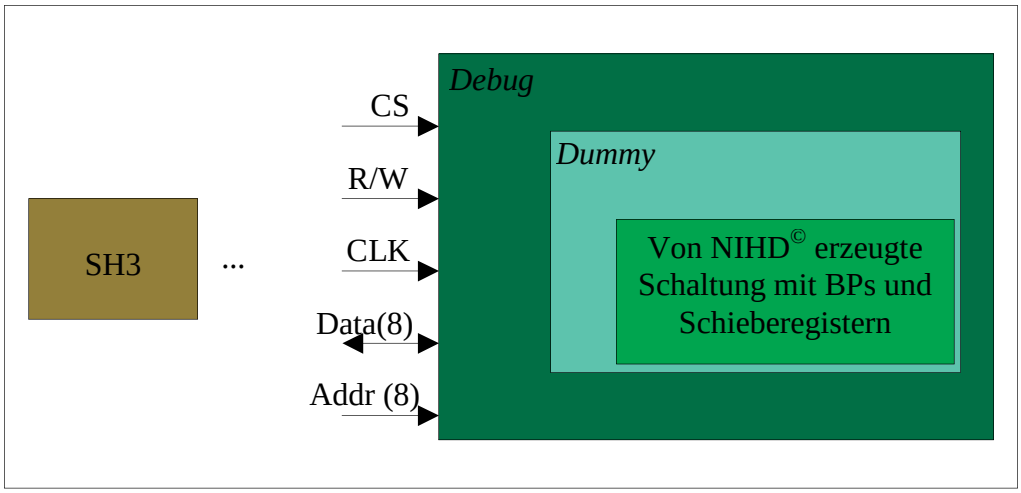

Abbildung 5-24: Aufbau zur Realisierung von je 8 Eingabe- und Ausgabewerten

und das *R/W*-Signal dienen zur Unterscheidung ob man Daten senden oder lesen möchte und werden über den SH3 gesteuert. *CLK* ist das für die Schaltung notwendige Taktsignal. Die acht *Addr*-Signale werden direkt an das „mittlere“ Design (*Dummy*) durchgeleitet. Für die Eingabe- bzw. Ausgabewerte des Kerndesigns benutzt man die acht Datenleitungen. *Debug* ist aus zwei Prozessen aufgebaut, die jeweils in Abhängigkeit des *CS*- bzw. *R/W*-Signals Daten an das FPGA oder an den *SH3*-Prozessor weiterleiten (vgl. Abbildung 5-25). Die mittlere Schaltungslogik (*Dummy*) besitzt als externe Ports das *CS*, das *R/W*, das *CLK*, das *Addr* sowie ein acht Bit breites Eingabe- und ein acht Bit breites Ausgabesignal (vgl. Abbildung 5-26). Intern besteht diese Komponente aus zwei 32 Bit breiten Registern, *data_in* ist für Werte die in die Kernschaltung gehen und *data_out* für Werte die aus dem Chip herausfließen. Mit Hilfe der Signale *CS* und *R/W* unterscheidet man nun ob Daten in

```
                                    ...
write_to_backplane : process(cs, rw)
begin
  if (CS='0' AND RW='1') then
    data(7 downto 0) <= data_out;
  end if;
end process;

write_to_FPGA : process(cs, rw, data)
begin
  if (CS='0' AND RW='0') then
    data_in <= data(7 downto 0);
  end if;
end process;
                                    ...
```

Abbildung 5-25: Implementierung der Debug-Entity

das innere oder in das äußere Design gesendet werden. Mit Hilfe der *Addr*-Leitungen berechnet man die Verschiebung (engl. *Offset*) um die anfallenden Daten von 8 Bit auf die beiden 32 Bit breiten Register abzubilden. Sollte die NIHD©-Schaltung mehr als 32

```
... rein : process(cs, rw, addr_dummy, din0, ...)
  variable offset : integer
  begin
    if  (CS='0' AND RW='0') then
      case addr_dummy is
        when "00000000" => offset:= 0;
        when "00000001" => offset:= 8;
        when "00000010" => offset:= 16;
        when "00000011" => offset:= 24;
                                    ...
hop_in(7 + offset) <= din7
hop_in(6 + offset) <= din6
                                    ...
```

Abbildung 5-26: Implementierung der Dummy-Entity

Eingabe-/Ausgabewerte benötigen, so vergrößert man die Register von 32 Bit auf z.B. 64 Bit und passt die *Offsets* an.

Die Abbildung 5-27 beschreibt das Zusammenspiel der einzelnen Entities im FPGA. Wem der *SH3*-Prozessor CS='0' und R/W='0' setzt, dann werden in der Debug-Entity die extern anliegenden acht Bit Wert in das Register *data_in* übernommen. Gleichzeitig werden acht Bit vom *data_in* Register in das Register *hop_in* in der Entity Dummy kopiert. Die genaue Byte-Adresse innerhalb von *hop_in* berechnet sich aus den acht Addr-Signalen und den damit berechneten Offset.

Sobald R/W auf ’1’ geht schreibt die Debug-Entity die acht Bit aus dem Register *data_out* auf die *back-plane*. Analog zum Lesen von der *back-plane* kopiert die Entity Dummy ein acht Bit Segment aus *hop_out* nach *data_out* in Debug.

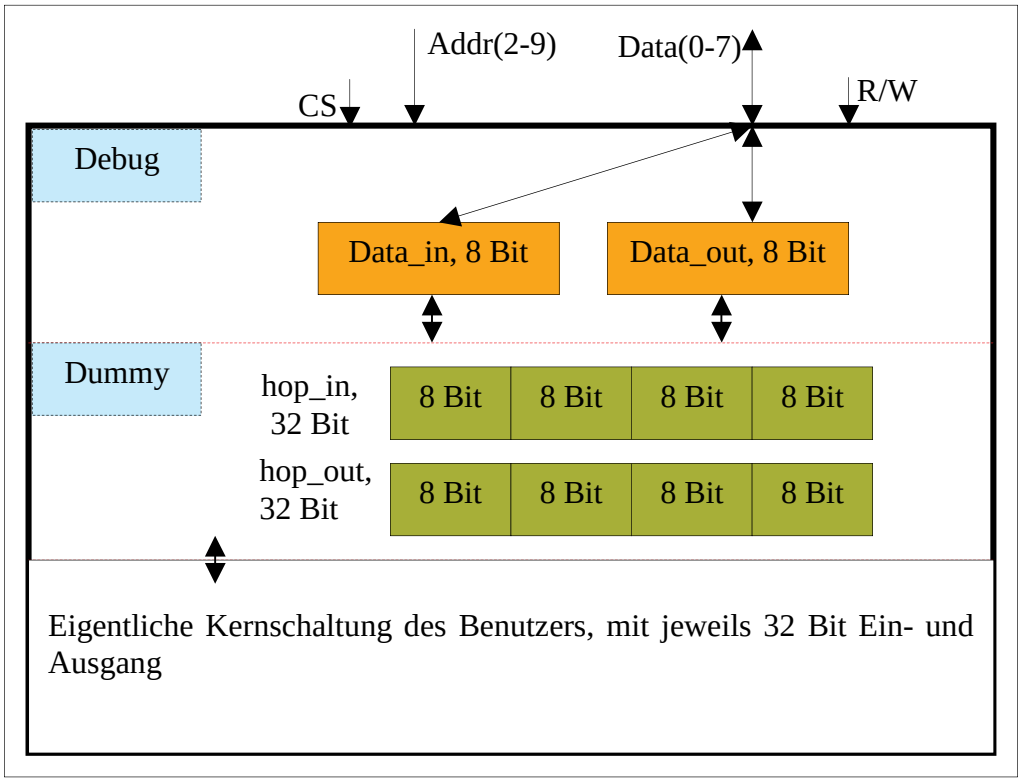

Abbildung 5-27: Interaktion beim Steuern eines Designs

5.3.4 Auslesen des FPGAs

Zum Auslesen der gespeicherten Registerinhalte kommt erneut das *Spyder-Core-P2*-Board zum Einsatz. Ebenso wie es ermöglicht effizient und fehlerfrei stabile Testmuster an das FPGA anzulegen, so ermöglicht der *SH3*-Prozessor auch ein schnelles Auslesen der Pufferwerte.

Nachdem die Schieberegister einer Schaltung ihre maximale Anzahl an Werten gespeichert haben, kann man über das *READ_ENABLE*-Signal diese Puffer veranlassen ihre Werte an den *SH3*-Prozessor zu senden. Bei jeder steigenden Flanke des anliegenden Taktsignals wird aus allen Registern einer der Werte ausgelesen und an das entsprechende externen Signal geleitet. Da auch hierbei in der Regel mehr als acht Werte gleichzeitig gesendet werden, benutzt man ebenfalls die im Abschnitt 5.3.3 erläuterte Technik. So bald die 16 Werte aller Schieberegister im SH3-Prozessor angekommen sind, speichert der Prozessor diese in einer Datei auf einem den angeschlossenen Netzlaufwerke. Das Format der Datei

wird hierbei so formatiert das es anschließend im Simulator als Stimuli benutzt werden kann.

5.4 Auswertung im Simulator

Nach der Formatierung der ausgelesenen externen Eingangswerte und der Werte der Flipflops stehen diese in Form einer *DO*-Datei für die Simulation bereit. In dieser Datei wird den Signalen der zugehörigen Schaltung über das Kommando '*force*' jeweils einer dieser Werte zugewiesen (vgl. Abbildung 5-28). Wenn alle Zuweisungen eines Simulationsschrittes erfolgt sind, setzt *ModelSim XE* mit Hilfe des Befehls '*run*' diese Belegungen um und berechnet, wie beabsichtigt, die fehlenden internen Signale der

```
add list *        // füge alleSignale der Schaltung in den list view
log -r /*         // überwache alle
                                    ...
force RESET 1     // setze RESET auf 1
force CLK 0       // setze CLK auf 1
force UP_DOWN 0   //  setzeUP_DOWN auf 0
run    //aktiviere diese Werte
                                    ...
```

Abbildung 5-28: Beispiel des Aufbaus einer *DO*-Datei

Schaltung in diesem Zeitpunkt. Um dies durchzuführen, muss jedoch zuerst der Simulator über ein weiteres Skript gestartet. Dabei ist es wichtig die Reihenfolge der Befehle genau

```
vlib zaehler_lib
vmap zaehler_lib zaehler_lib
vcom -work zaehler_lib -93 -check_synthesis -source
F:\Lager\test\zaehler_translate_OutputVhdl.vhd
vsim -c -wlf zaehler_dump.wlf zaehler_lib.zaehler(STRUCTURE) <zaehler_command.do
```

Abbildung 5-29: Kommando-Datei zur Durchführung einer Simulation

einzuhalten. Zuerst erstellt man mit *vlib* eine neue Bibliothek, danach kann man durch *vmap* dieser Bibliothek einen anderen Namen zuweisen. Mit *vcom* kompiliert die angegebene SVHDL-Datei, um sie anschließend mit *vsim* und den in der *DO*-Datei angegebenen Werten automatisch zu simulieren (vgl. Abbildung 5-29). Dabei werden die berechneten Ergebnisse in einer *WLF*-Datei gespeichert. In Anschluss daran startet man *ModelSim XE*, lädt im *List*-Fenster diese *WLF*-Datei ein und bekommt eine grafische Übersicht über alle Signalbelegungen zu jedem Simulationszeitpunkt durch die man

schrittweise den Verlauf von Signalen verfolgen kann. Dabei kommt es insbesondere auf die berechneten internen Signale vor, während und nach dem BP an (vgl. Abbildung 5-30).

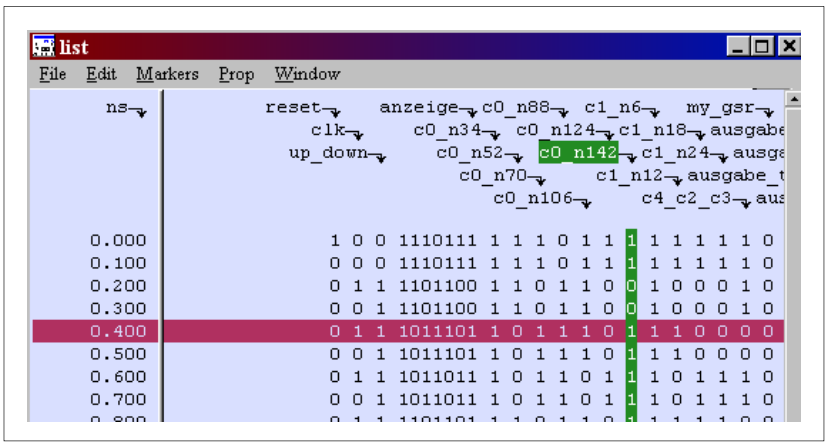

Abbildung 5-30: Auswertung der Signale im List-Fenster

Kapitel 6

Ergebnisse

In diesem Kapitel wird untersucht ob die in dieser Diplomarbeit implementierten Ansätze den gewünschten praktischen Nutzen erbringen. Anhand von einfachen repräsentativen Schaltungen werden zahlreiche Untersuchungen durchgeführt. Bei den Beispielschaltungen wird neben einer allgemeinen ungetakteten Schaltung zum Beweis des Nutzens der Zusatzkomponenten, auch die Funktionsweise bei komplexeren getakteten Designs untersucht.

Zu den bei den Testreihen ermittelten Messwerten zählen u.a. Werte zum zeitlichen Verhalten bei der Schaltungsgenerierung sowie Werte über den gemessenen Mehraufwand durch die verwendeten Zusatzkomponenten. Daneben wird versucht Aussagen über die Auswirkungen auf die Performance einer modifizierten Schaltung zu treffen.

Die dadurch gewonnenen Ergebnisse stellt man abschließend in Bezug zum Umfang der Zusatzlogik. Durch diesen Vergleich der Versuchsergebnisse lassen sich die Vor- und Nachteile des beschriebenen Ansatzes deutlich erkennen.

6.1 Beispielschaltungen

Die Versuche im Rahmen dieser Arbeit umfassen einen *6-fach Parity Checker*, einen *4-Bit Zähler mit Richtungswechsel* sowie ein *Lauflicht für die 7-Segment-Anzeige*. Bei dem *Parity Checker* handelt es sich um ein ungetaktetes Design, das zur Darstellung der Grundfunktionalität der NIHD©-Modifikationen am SVHDL-Code dient. Intern ist es auf

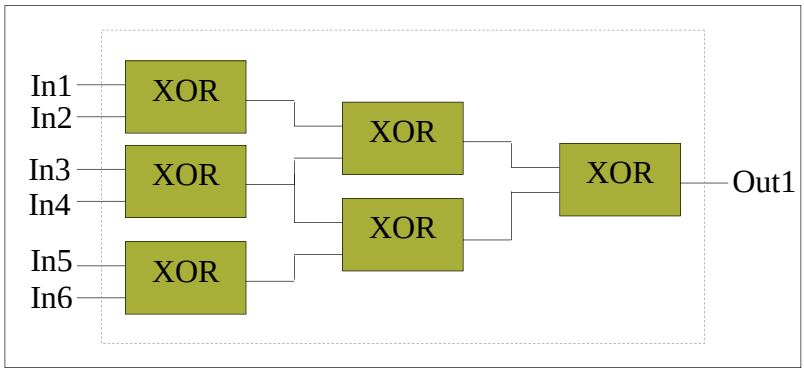

Abbildung 6-1: Aufbau einer 6-fachen Paritäts-Überprüfung

Basis der im VHDL-Standard vorgegebenen *XOR*-Funktion umgesetzt (vgl. Abbildung 6-1). Beim späteren Debugging lässt sich in diese Schaltung leicht ein Fehler integrieren, indem man z.B. eines der XOR-Komponenten durch ein AND ersetzt. Die

```
                                    ...
count_proc : process (CLK, RESET, DIR, Ausgabe_temp)
begin
 if RESET='1' then Ausgabe_temp <= "0000";
  elsif rising_edge(CLK) then
    if DIR = '1' then
      if Ausgabe_temp = "1111" then Ausgabe_temp <= "0000";
      else Ausgabe_temp <= Ausgabe_temp + 1;          end if;
    else
      if Ausgabe_temp = "0000" then Ausgabe_temp <= "1111";
      else Ausgabe_temp <= Ausgabe_temp - 1;  end if; end if; end if;
end process;
Anzeige <= Ausgabe_temp;
                                    ...
```

Abbildung 6-2: Implementierung der Zählerschaltung

Verhaltensbeschreibung des Zählers besteht aus einem einzelnen Prozess der auf Veränderungen des Taktsignals, des asynchronen Rücksetzsignals und des Richtungssignals reagiert (vgl. Abbildung 6-2). Für die Versuche ist diese Schaltung deshalb gut geeignet, weil sie zusätzlich Flipflops enthält, an deren Ausgänge ebenfalls Schieberegister anzubringen sind. Die Umsetzung des Überlaufs bietet zudem eine geeignete Möglichkeit den Code fehlerhaft so zu abzuwandeln, dass man das Verhalten der

NIHD©-Zusatzkomponenten besser beobachten lässt. Die Verhaltensbeschreibung des dritten Versuches besteht aus einem Zustandsautomaten mit sieben Zuständen. Das Lauflicht soll dabei mit jeder steigenden Taktflanke jeweils ein anderes Segment entsprechend des Richtungssignals aufleuchten lassen (vgl. Abbildung 6-3). Der Grund für die Verwendung dieser Schaltung liegt in der Notwendigkeit die korrekte Funktion des Debuggers auch bei den häufig zum Einsatz kommenden Zustandsautomaten nachzuweisen. Dabei kann man recht leicht ein Fehlverhalten hervorrufen, indem man z.B. einen Zustandswechsel falsch programmiert.

```
                                                    ...
type AnzeigeStates is (S1, S2, S3, S4, S5, S6, S7);
signal currentState, nextState : AnzeigeStates;
begin
  SynchState : process(Ext_Clk, Reset, nextState)
  begin
    if Reset = '1' then currentState <= S1;
    elsif Ext_Clk'event AND Ext_Clk = '1' then currentState <= nextState;
    end if;
end process;

ProcState : process(currentState, dir)
begin
case currentState is
when S1 =>
            Anzeige <= "0000001";
            if dir = '1' then nextState <= S2;
            else nextState <= S7;
            end if;
                                                    ...
```

Abbildung 6-3: Implementierung des Lauflichtes

6.2 Beispielsitzungen

Alle drei Schaltungen werden entsprechend des NIHD©-Ansatzes mit Zusatzkomponenten versehen. Die Breite der Schieberegister ist für die praktischen Versuche auf 16 Bit beschränkt. Zum besseren Verständnis des Ablaufs der drei Versuche werden die einzelnen Schritte ausführlich am ersten Beispiel dargelegt.

6.2.1 *6-fache Paritäts-Überprüfung*

Die Verhaltensbeschreibung dieser Schaltung hat den in Abbildung 6-4 dargestellten Aufbau. Um einen kontrollierten BP auszulösen, muss diese Schaltung fahlerhaft gemacht werden. So ersetzt man z.B. die Zeile „*internal3 <= in1 XOR in2*“ durch „*internal3 <= in1 AND in2*“, wodurch der Fehler zwar markant ist, aber nicht sofort auftreten wird. Nach

```
...
architecture Behavioral of xor6 is
  signal internal1, internal2, internal3, internal4, internal5: std_logic;
  begin
    internal1 <= in1 XOR in2;
    internal2 <= in3 XOR in4;
    internal3 <= in5 XOR in6;
    internal4 <= internal1 XOR internal2;
    internal5 <= internal2 XOR internal3;
    out1 <= internal4 XOR internal5;
end Behavioral;
...
```

Abbildung 6-4: Implementierung des 6-fach Paritätstests entsprechend der Abbildung 6-1

der Übersetzung liegt eine rein strukturelle und logisch äquivalente Schaltungsbeschreibung (SVHDL) vor, die vom NIHD© als Eingabeformat benutzt wird (vgl. Abbildung 6-5).

```
...
C44 : X_AND4
  port map (I0 => N_in4, I1 => N29, I2 => N30, I3 => N31, O => N23);

C6_GTS_TRI_7 : X_TRI
  port map (I => C6_GTS_TRI, CTL => NlwInverterSignal_C6_GTS_TRI_CTL, O => out1 );

NlwInverterBlock_C6_GTS_TRI_CTL : X_INV
  port map (I => GTS, O => NlwInverterSignal_C6_GTS_TRI_CTL );
...
```

Abbildung 6-5: Aus Abbildung 6-4 erzeugtes *Post-Translation-Simulation Model* von *Xilinx*

NIHD© verarbeitet diese Datei wie beschrieben, zuerst analysiert der Parser die Struktur und baut diese in Java nach, danach erzeugt die Klasse *VHDLFileCreator* die beiden veränderten SVHDL-Dateien. Diejenige ohne BPs und Schieberegister wartet auf ihren Aufruf im Simulator, während die andere Datei mit den Erweiterungen durchläuft den kompletten *Xilinx* FPGA-Entwurfsvorgang. Um die für das Setzen von BPs relevanten internen Signale zu kennen, sind grobe Funktionstests der fehlerbehafteten Schaltung erforderlich. In diesem Fall ergeben sie ein Fehlverhalten im Zusammenhang mit den externen Signalen *in5* bzw. *in6*. Laut den Informationen über den strukturellen Aufbau des

Design, muss der Fehler deshalb mit den Signalen *n5*, *n17*, *n18* sowie *n33* in Verbindung stehen. Deshalb gibt man im NIHD© die in Abbildung 6-6 aufgeführten BPs ein. Nun wird die *BIT*-Datei für diese Paritäts-Überprüfung auf den *Xilinx*-FPGA geladen. Da sämtliche Übertragungen von Daten nicht direkt angezeigt werden können, wird in dieser Arbeit *ModeSim XE* stellvertretend zur repräsentativen Darstellung der Kommunikationen genutzt.

Internes Signal	BP-Bedingung
n5	0
n17	0
n18	1
n33	0

Abbildung 6-6: gesetzte Breakpoints

Nachdem sich die Schaltung auf dem FPGA befindet wird sie zuerst über das zentrale Rücksetzsignal initialisiert. Den Takt und das READ_ENABLE-Signal sowie die eigentlichen Stimuli für das Design stammen ebenfalls vom *SH3*-Prozessor. Die Signalverläufe vom Einsetzen des *Reset*-Signals bis zum Auftreten des *BUFFER_FULL*-Signals sind in der Abbildung 6-7 veranschaulicht. Da mindestens einer der vier gesetzten Breakpoints eine logische ’1’ lieferte, setzt die Schaltung das BP_ALERT-Signal auf eine logische ’1’, somit lag man mit der Lokalisierung der Fehlerquelle richtig. Sobald die

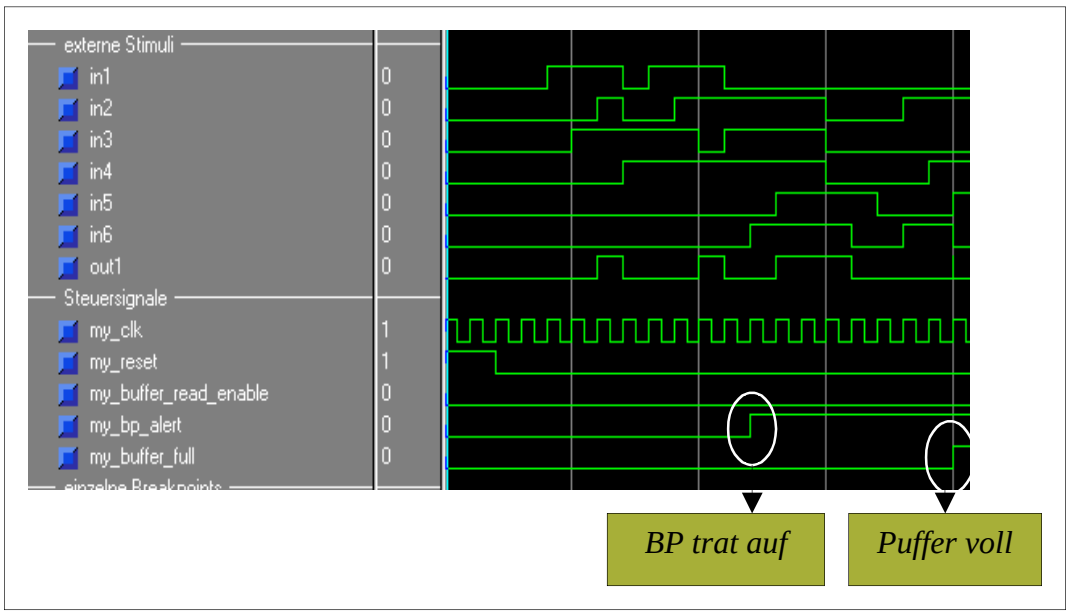

Abbildung 6-7: vereinfachte Darstellung der Kommunikation zwischen SH3-Prozessor und FPGA

Puffer voll sind, warten die Schieberegister auf das *READ_ENABLE*-Signal zum Auslesen

der Puffer. Nach dem Senden dieses Steuersignals durch den *SH3*-Prozessor übertragt der FPGA ab der nächsten steigenden Taktflanke seine gespeicherten Werte (vgl. Abbildung 6-8). Aufgrund der Ringstruktur der Register wiederholt sich der Vorgang solange bis READ_ENABLE wieder auf eine ’0’ gesetzt wird.

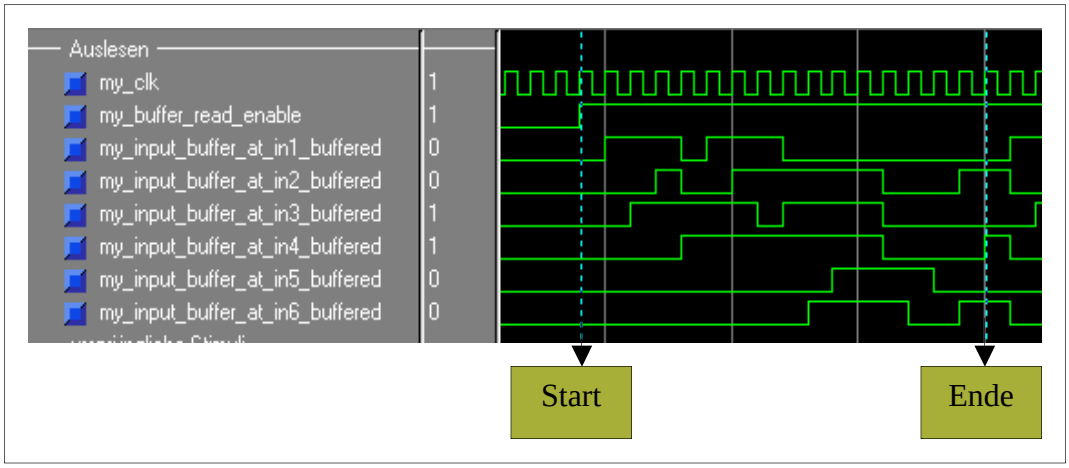

Abbildung 6-8: vereinfachte Darstellung des Auslesevorgangs

Beim Vergleich von Abbildung 6-7 mit 6-8 erkennt man bereits das die gespeicherten Werte den ursprünglichen Stimuli entsprechen. Die Schieberegister somit haben gemäß ihrer Programmierung gearbeitet. Der SH3-Prozessor bereitet diese Daten auf und speichert sie auf einem der angeschlossenen Netzlaufwerke als *DO*-Datei. Die für dieses Beispiel zu Grunde liegende Datei hat den in Abbildung 6-9 gezeigten Aufbau.

```
add list *
log -r /*

force    CLK    0
run
force    in1    1
run
force    in2    0
run
force    in3    0
run
force    in4    0
run
force    in5    0
run              ...
```

Abbildung 6-9: Ausschnitt aus *DO*-Datei

Abschliessend führt man die Kommandodatei zum Starten des Simulationsprozesses aus und übergibt ihm die ausgelesenen Werte sowie die NIHD©-Schaltungsbeschreibung ohne die BPs bzw. Schieberegister. Der Simulator berechnet anhand der Stimuli sämtliche die internen Signale und zeigt ihrenVerlauf im sog. List-Fenster von *ModelSimXE* an. Durch

farbige Kennzeichnungen lassen sich Signalverläufe sehr nachvollziehen (vgl. Abbildung 6-10).

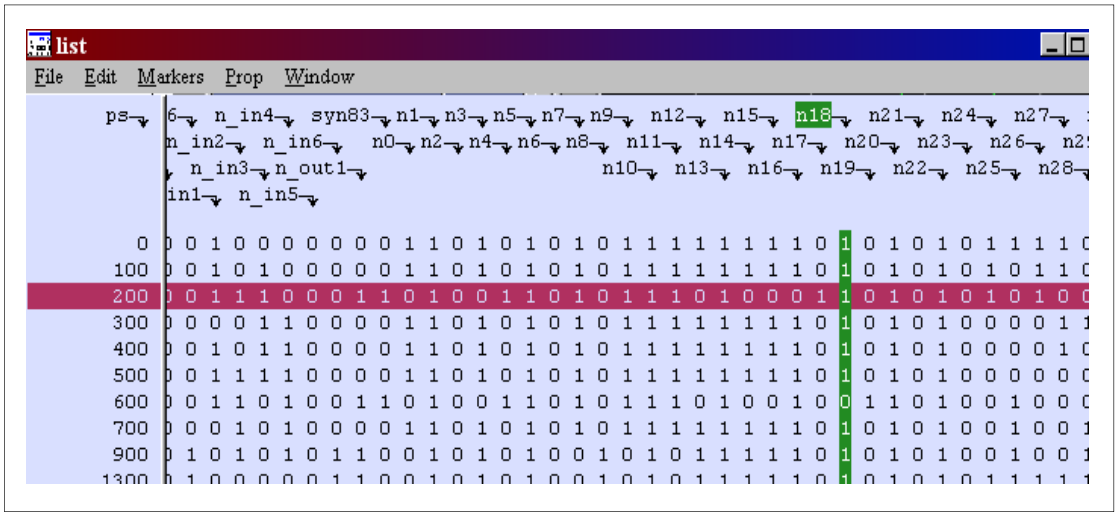

Abbildung 6-10: List-Fenster in *ModelSim XE* mit den Signalverläufen des Parity-Checkers

Mittels dieser gewonnenen Daten lassen sich nun die Werte aller Signale vor, während und nach einem BP näher untersuchen. Durch Rückschlüsse auf die Struktur der Schaltung kann die Fehlerquelle relativ schnell identifiziert und beseitigt werden.

6.2.2 *4-Bit Zähler mit Richtungswechsel*

Der Aufbau dieses Zählers wird in der Abbildung 6-11 gezeigt. Intern besteht er aus einem

```
                                    ...
port( CLK : in std_logic; RESET : in std_logic; DIR : in std_logic;
      Anzeige : OUT std_logic_vector(3 downto 0));
                                    ...
 elsif rising_edge(CLK) then
   if DIR = '1' then
     if Ausgabe_temp = "1111" then Ausgabe_temp <= "0000";
     else  Ausgabe_temp <= Ausgabe_temp + 1;        end if;
   else
     if Ausgabe_temp = "0000" then Ausgabe_temp <= "1111";
     else  Ausgabe_temp <= Ausgabe_temp - 1;
                                    ...
```

Abbildung 6-11: Ausschnitt der Implementierung des Zählers

Prozess der u.a. in Abhängigkeit des Richtungssignals (DIR), auf oder ab zählt. In dem Versuch sind die Vorraussetzungen für den Überlauf abgeändert indem der Zähler nicht erst bei „*0000*“ wieder auf „*1111*“ springt, sondern bereits bei „*0010*“. Die Fehlerbedingung tritt somit nur beim Runterzählen an der Stelle „*0010*“ auf. Technisch werden Zähler grundsätzlich durch Flipflops umgesetzt, daher ist dieses Beispiel auch in

Hinblick auf die Pufferung deren Ausgangswerte von Interesse. Da *Reset* und *Clk* nicht in Schieberegistern gespeichert werden können, schliesst der NIHD© diese nur an den *DIR*-Eingang und das *ENABLE*-Signal. Bei der Untersuchung des Designs stellt sich nur ein Signal als mögliche Ursache für den vermeintlichen Fehler dar, es handelt sich um *SYN86*. Dieses Signal wird dazu mit einem BP verschalten der auf eine logische ’1’ reagiert. Auf dem FPGA zeigt sich danach das in Abbildung 6-12 dargestellte Verhalten.

Anhand der auf dem FPGA durchgeführten Beobachtungen lässt sich bestätigen, das *SYN86* im Zusammenhang mit dem Fehlverhalten stehen muss. Zur genaueren Untersuchung der internen Signale werden zuerst die Puffer mit den gespeicherten Werten der externen Signale und der zugehörigen Flipflop-Werte ausgelesen. Im List-Fenster von

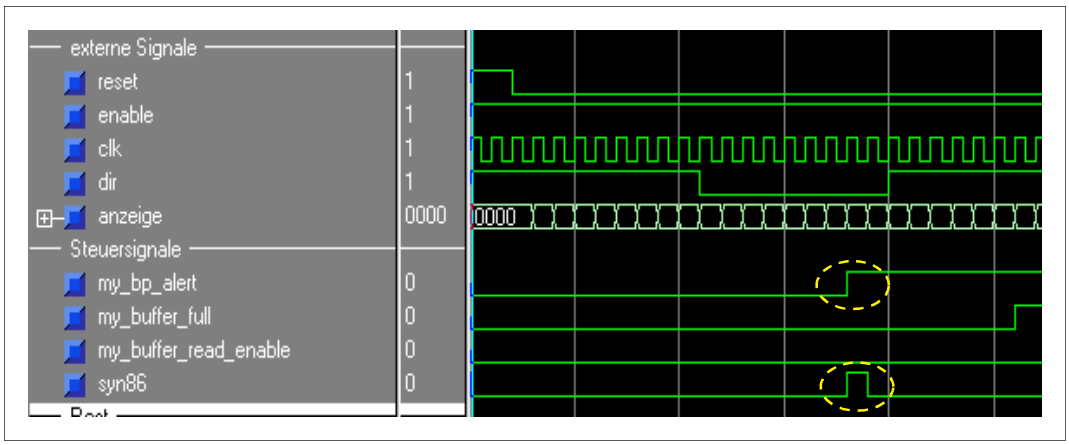

Abbildung 6-12: Verhalten des Zählers bis zum Auftreten des *BUFFER_FULL*-Signals

ModelSim XE lassen sich nun erneut alle berechneten internen Signale auswerten (vgl. Abbildung 6-13).

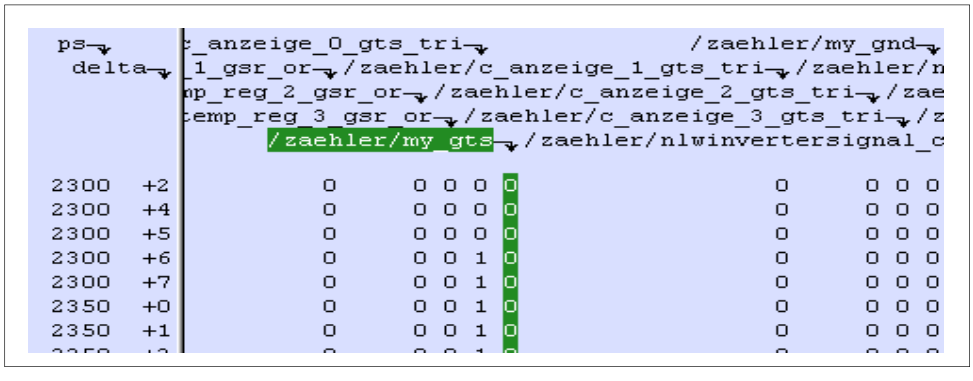

Abbildung 6-13: Auswertung der Signale

6.2.3 *Lauflicht für 7-Segment-Anzeige*

Der dritte Versuch wird analog zu den vorangegangenen durchgeführt. In diesem Beispiel besteht der eingebaute Fehler in einem falschen Übergang vom Zustand 4 in Zustand 6, der Zustand 5 wird bei *DIR='1'* übersprungen. Um das Fehlverhalten zu entdecken wird der BP an das interne Signal *C_ANZEIGE_3_GTS_TRI* gelegt. Zur großen Überraschung stellt man fest das der BP nicht nur bei dieser Fehlerbedingung aktiviert wird. Ein Blick in den Simulator zeigt, dass es im Gegensatz zu den beiden ersten Versuchen kein einziges internes Signal gibt, welches auschliesslich mit dem Fehler in Verbindung steht (vgl. Abbildung 6-14). Deshalb meldet eine BP-Komponente, egal an welcher Leitung, sie angebracht ist, grundsätzlich immer das Auftreten eines Fehlverhaltens, obwohl es sich nicht um den beabsichtigten Auslöser handelt. Diese Tatsache ist insbesondere bei der

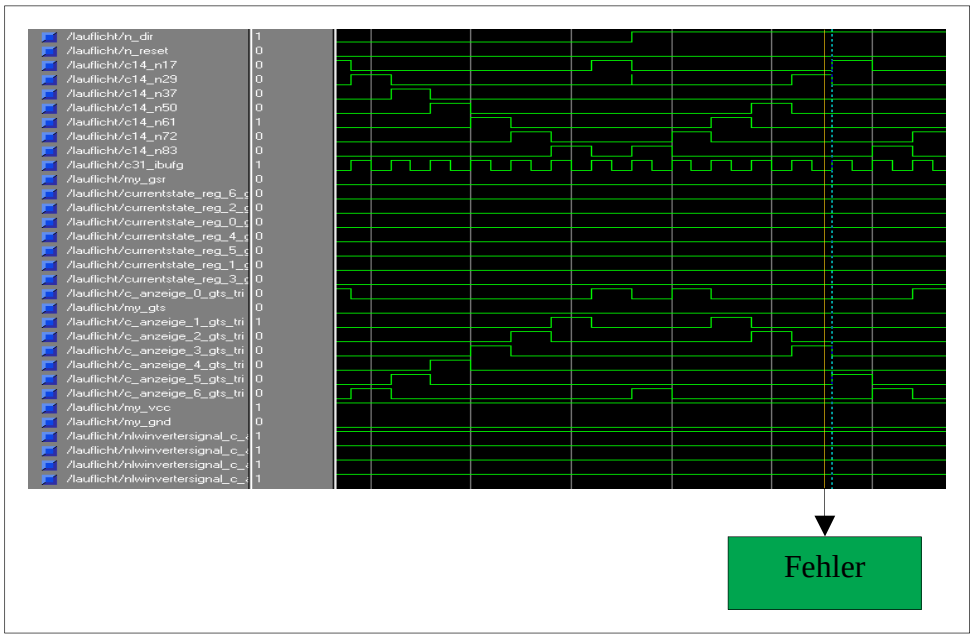

Abbildung 6-14: vereinfachte Signalverläufe der Lauflicht-Schaltung

Weiterentwicklung der vorgestellten Methode zu beachten und zeigt das der bestehende Ansatz um diesen Punkt erweitert werden muss.

6.3 Bewertung

Da die durchgeführten Versuche ausschließlich zum Beweis der Machbarkeit dienen, jedoch keine Rückschlüsse auf die Qualität der erzielten Ergebnisse erlauben, benötigt man weitere Indikatoren um diese besser bewerten zu können. Eine Möglichkeit sind die

Angaben über die Anzahl verwendeter Hardware-Komponenten des Platzierungs- und Verdrahtungsvorgangs (*Place&Route*) sowie des Zuordnungsprozesses (*Mapping*). Bei den hierbei relevanten Daten handelt es sich u.a. um die Bereiche:

- ⇒ Slices
- ⇒ Slice-Flipflops
- ⇒ LUTs mit 4 Eingängen
- ⇒ Eingabe/Ausgabepuffer
- ⇒ Zentrale Taktnetze sowie
- ⇒ Gatter-Äquivalente.

Zur besseren Vergleichbarkeit untersucht man diese Werte unter den selben Randbedingungen, dazu zählt hauptsächlich die Optimierungsstrategie, deren Ziele entweder ein minimaler Platzbedarf, oder eine maximale Geschwindigkeit sind.
Neben dieser Betrachtungsweise lassen sich Schaltungsbeschreibungen ebenfalls nach der maximal erreichbaren Taktfrequenz einteilen. Die *Xilinx ISE* Software bietet jedoch, anders als ihre Vorgänger, keine direkten Möglichkeiten die maximale mögliche Gesamt-Taktrate eines Designs anzuzeigen. Vielmehr ist der Benutzer gezwungen sich diese Daten aus den angegebenen Verzögerungen der einzelnen Signalleitungen selbst zu berechnen. Alternativ dazu kann der Benutzer über eine Umweg diese Ergebnisse dennoch bekommen. Einer der beiden Compiler der innerhalb von *Xilinx ISE* für die zu Grunde liegenden Designs verwendet wird, besitzt eine eigene kleinere Entwicklungsumgebung (*FPGA Express 3.7* der Firma *Synopsys*) in der man die Werte für die Taktfrequenzen direkt ablesen kann. Zusätzlich lassen sich die Betrachtungen auch auf den direkt von *Xilinx* entwickelten Compiler *XST* übertragen.

6.3.1 Verwendete Hardware-Komponenten

Dieser Abschnitt setzt die Daten über verwendete Slices, Flipflops, etc. einer der drei vorgestellten Beispielschaltungen in Beziehung und stellt sie grafisch dar. Diese 6-fach Paritäts-Schaltung verbraucht unter verschiedenen Randbedingungen die in den folgenden Abbildungen gezeigten Ressourcen. Innerhalb einer Optimierungsmethode entsprechen die jeweils ersten Werte der Verwendung des Compilers von *Synopsys*, die dahinter stehenden dem *Xilinx XST*-Compiler. Die Resultate für die Ein/Ausgabepuffer bzw. die zentralen Taktnetze sind für ein Design in allen Versuchen gleich, es sei denn man verändert die

Anzahl der externen Signale in der Schaltungsbeschreibung. Da der *Zähler* und das *Lauflicht* vergleichbare Werte in Bezug auf die Datenreihen des *6-fach Parity Checkers* liefern, wird im Rahmen dieser Arbeit darauf verzichtet, die sehr zeitintensive Berechnung der Datenreihen auch für diese Schaltungen durchzuführen. Stattdessen wird das Hauptaugenmerk auf die primär wichtigen Bestandteile der Designs (NIHD©-Schieberegister sowie BPs) gelegt und deren Einzelergebnisse in Relation gesetzt.
Folgende Datenreihen ergeben sich für den 6-fach Paritäts-Test:

6-fach Parity-Test mit 4 BPs und 8 Bit Puffertiefe				
	Optimierungsmethode			
	Minimaler Platzbedarf		**Maximale Geschwindigkeit**	
Slices	88	104	117	104
Slices Flipflops	96	84	96	90
4-fach LUTs	77	69	78	79
Externe IOBs	17	17	17	17
GCLKs	1	1	1	1
GCLKIOBs	1	1	1	1
Gatter-Äquivalente	1309	1218	1315	1218

FPGA-Express XST

Abbildung 6-15: 6-fach Paritäts-Test mit 4 BPs und 8 Bit Puffer

6-fach Parity-Test mit 4 BPs und 16 Bit Puffertiefe				
	Optimierungsmethode			
	Minimaler Platzbedarf		**Maximale Geschwindigkeit**	
Slices	190	182	190	182
Slices Flipflops	161	144	156	150
4-fach LUTs	156	136	126	130
Gatter-Äquivalente	2143	2281	2137	2280

Abbildung 6-16: 6-fach Paritäts-Test mit 4 BPs und 16 Bit Puffer

6-fach Parity-Test mit 4 BPs und 32 Bit Puffertiefe				
	Optimierungsmethode			
	Minimaler Platzbedarf		**Maximale Geschwindigkeit**	
Slices	327	308	321	314
Slices Flipflops	264	252	264	258
4-fach LUTs	189	203	192	215
Gatter-Äquivalente	3559	3744	3541	3816

Abbildung 6-17: 6-fach Paritäts-Test mit 4 BPs und 32 Bit Puffer

6-fach Parity-Test mit 4 BPs und 64 Bit Puffertiefe				
	Optimierungsmethode			
	Minimaler Platzbedarf		**Maximale Geschwindigkeit**	
Slices	642	567	666	573
Slices Flipflops	468	456	468	468
4-fach LUTs	444	331	474	339
Gatter-Äquivalente	6433	6432	6613	6480

Abbildung 6-18: 6-fach Paritäts-Test mit 4 BPs und 64 Bit Puffer

Die Untersuchungen eines einzelnen Schieberegisters ergeben folgende Datenreihen:

Schieberegister mit 8 Bit Puffer				
	Optimierungsmethode			
	Minimaler Platzbedarf		**Maximale Geschwindigkeit**	
Slices	13	11	13	11
Slices Flipflops	16	14	16	15
4-fach LUTs	10	10	10	10
Gatter-Äquivalente	197	197	197	197
Schieberegister mit 16 Bit Puffer				
Slices	22	21	21	21
Slices Flipflops	26	24	26	25
4-fach LUTs	18	21	18	20
Gatter-Äquivalente	334	373	334	373
Schieberegister mit 32 Bit Puffer				
Slices	42	40	42	39
Slices Flipflops	47	45	47	46
4-fach LUTs	43	43	43	43
Gatter-Äquivalente	718	718	718	718
Schieberegister mit 64 Bit Puffer				
Slices	79	70	79	70
Slices Flipflops	82	80	82	82
4-fach LUTs	86	86	86	68
Gatter-Äquivalente	1229	1202	1229	1213
Schieberegister mit 128 Bit Puffer				
Slices	134	115	145	120
Slices Flipflops	144	142	144	147
4-fach LUTs	129	92	152	96
Gatter-Äquivalente	1926	1917	2065	1955

Abbildung 6-19: Daten eines Schieberegisters mit 8 bis 128 Bit Breite

Wie man erkennt unterscheiden sich die Werte innerhalb einer Stufe nur marginal. Unabhängig vom benutzten Compiler und von der Optimierungsmethode, ergibt sich ein leicht quadratischer Anstieg der belegten FPGA-Ressourcen (vgl. Abbildung 6-20, 6-21), wobei sich auch die Anzahl der logischen Gatter-Äquivalente ähnlich verhält (vgl Abbildung 6-22).

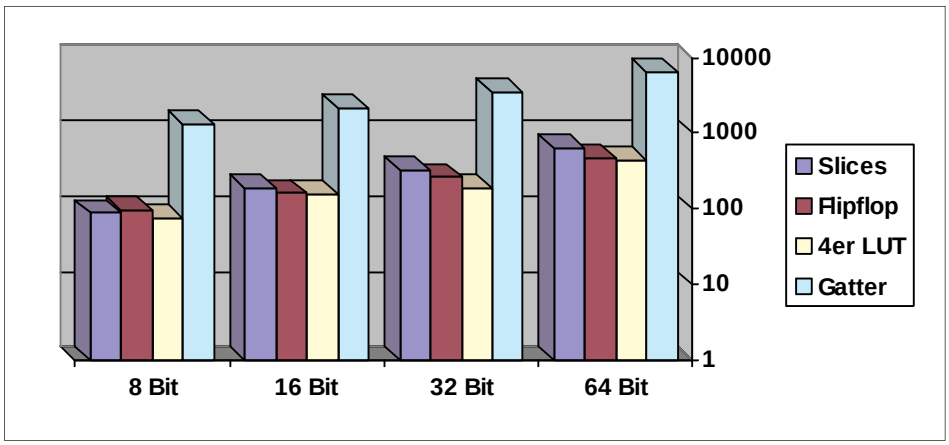

Abbildung 6-20: Übersicht über die Werte des 6-fach Parity Checkers laut Abbildung 6-15 bis 6-18

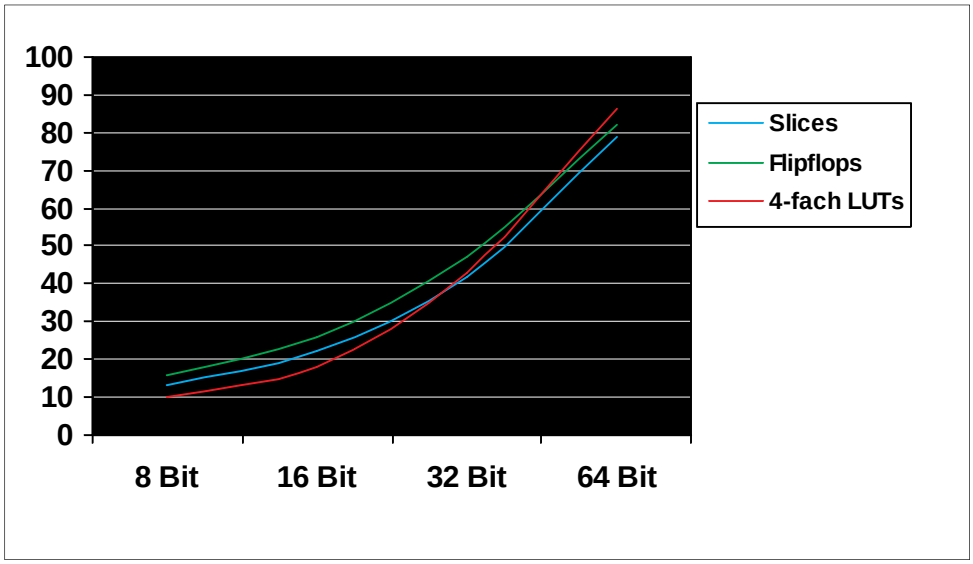

Abbildung 6-21: prinzipieller Verlauf bei einem Schieberegister laut Abbildung 6-19

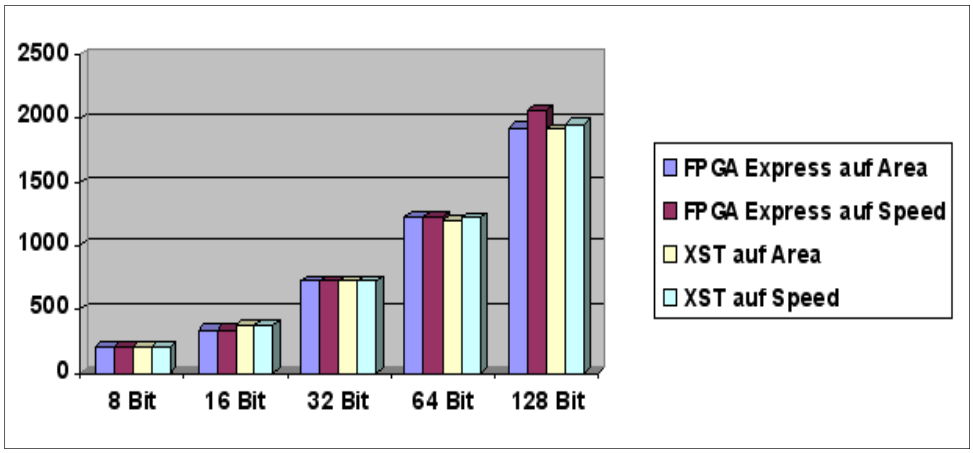

Abbildung 6-22: prinzipieller Verlauf der Anzahl der äquivalenten logischen Gatter bei einem Schieberregister

6.3.2 Vergleich der Taktfrequenzen

Die im Abschnitt 6.3.1 vorgestellten Datenwerte können aber auch hinsichtlich der möglichen Taktfrequenzen untersucht werden. Dabei verwendet man die grafische Oberfläche des Programms *FPGA Express 3.7*, in der man ebenfalls die Möglichkeit hat auszuwählen zwischen minimaler Fläche und maximaler Geschwindigkeit als Ziel für die Optimierungsmethode. Analog zu den Betrachtungen über die verbrauchten Ressourcen eines Chips, sind auch bei diesen Beispielen, wenn überhaupt, nur ein marginale Abweichungen in Bezug auf die gewählte Methode zu sehen. Nachfolgend werden die drei vorgestellten Versuche hinsichtlich der maximal erreichbaren Taktfrequenz des *CLK*-Signal überprüft. Da die nicht modifizierte 6-fach Paritätsschaltung primär keine Taktsignale besitzt, kann für sie kein Ausgangswert angegeben werden. Zusätzlich dazu wird das Verhalten eines einzelnen Schieberegisters unter dem selben Aspekt untersucht (vgl Abbildung 6-23).

6-fach Paritätsschaltung		
	Optimierungsmethode	
	Minimaler Platzbedarf	**Maximale Geschwindigkeit**
Ohne	k.A.	k.A.
8Bit	73 MHz	73 MHz
16 Bit	73 MHz	73 MHz
32 Bit	73 MHz	73 MHz
64 Bit	69 MHz	69 MHz
128 Bit	60 MHz	60 MHz
4 Bit Zähler		
Ohne	94 MHz	94 MHz
8 Bit	73 MHz	73 MHz
16 Bit	73 MHz	73 MHz
32 Bit	73 MHz	73 MHz
64 Bit	69 MHz	69 MHz
128 Btt	60 MHz	60 MHz
Lauflicht		
Ohne	108 MHz	108 MHz
8 Bit	73 MHz	73 MHz
16 Bit	73 MHz	73 MHz
32 Bit	73 MHz	73 MHz
64 Bit	73 MHz	73 MHz
128 Btt	73 MHz	73 MHz
Einzelnes Schieberegister		
8 Bit	60 MHz	60 MHz
16 Bit	60 MHz	60 MHz
32 Bit	60 MHz	60 MHz
64 Bit	60 MHz	60 MHz
128 Btt	60 MHz	60 MHz

Abbildung 6-23: erreichte Taktfrequenzen der Versuche

Die Auswertung dieser Daten (vgl. Abbildung 6-24) zeigt, dass die Größe der Schiebregister kaum Einfluss auf die erreichbaren Taktraten hat. Die durchschnittliche Abweichung der Frequenzen mit und ohne Schieberegister, unabhängig von der Breite, beträgt bei dem Zähler 24,4 MHz, bei dem Lauflicht ist sie mit 35,0 MHz nur geringfügig höher.

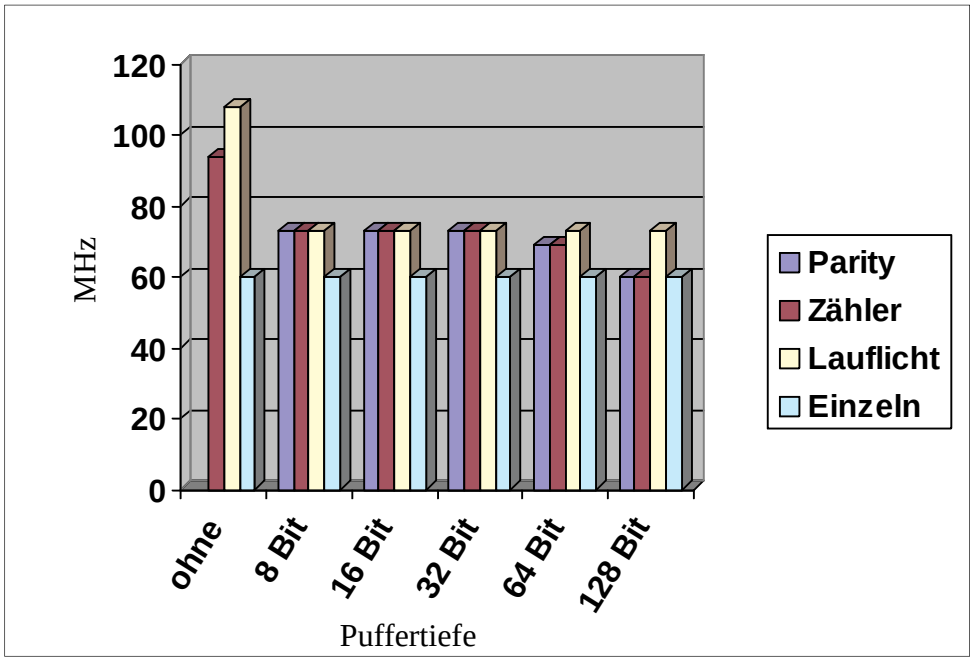

Abbildung 6-24: grafische Darstellung der erreichten Taktfrequenzen

Insgesamt gesehen, sind diese Abstufungen keinesfalls problematisch, da die beiden Optimierungen nur mit den Standardeinstellungen durchgeführt werden. Diese sehen keine Einbindung von Block-RAM-Zellen des Virtex FPGA vor, daneben sind die LUTs nur mit vier der maximal sechs vorhandenen Eingänge belegt, etc. Dahingehend bergen die Synthese-Einstellungen der Compiler noch ein großes Potential für Verbesserungen der Differenzen in den Taktraten.

Grundsätzlich lässt sich sagen: Der hinter dem NIHD© stehende Ansatz ist nicht auf eine bestimmte Taktfrequenz als Obergrenze beschränkt, wobei er selbstverständlich sekundär von den Werten der jeweiligen Optimierungsmethode abhängt. Das bedeutet, rein theoretisch könnte der NIHD© mit jedem beliebigen Takt arbeiten - Praktisch kann er jedoch nur die nach der Platzierung und Verdrahtung noch vorliegenden Werte nutzen. Je effizienter die Compiler in den kommenden Jahren werden, umso schneller kann das Debugging auf Grundlage der in dieser Arbeit vorgestellten Methode durchgeführt werden.

Kapitel 7

Schlussfolgerungen

Abschließend werden in diesem Kapitel die Erfahrungen bei der Realisierung der Aufgabenstellung erläutert. Insbesondere geht es um die Frage, ob es sich lohnt diesen Ansatz weiter zu verfolgen und wenn ja, welche Funktionalität die kommende Version des NIHDs© bieten muss. Aufgrund der gewonnenen Kenntnisse wird außerdem diskutiert inwieweit das derzeitige Ausgangsformat angemessen ist. Daneben wird erörtert welche Vor- und Nachteile eigene Takt- bzw. Rücksetzsignale bieten.

Da eine so komplexe Arbeit nicht ohne Schwierigkeiten abläuft, existiert dafür ein eigener Abschnitt der ausschließlich die aufgetretenen Hindernisse näher beschreibt. Diese Schwierigkeiten umfassen hauptsächlich die Nachbildung von Komponenten des *Post-Translation-Simulation Models* von Xilinx, aber auch Probleme mit den existierenden VHDL-Datenstrukturen.

Zusammenfassend wird die Komplexität des NIHDs© dargestellt, wobei ein besonderes Augenmerk auf die zukünftigen Weiterentwicklungen fällt. Hierbei wird besonders auf die Beziehungen zu dem aktuellen Begriff der *System-Beans* eingegangen. Gerade für die weiteren Entwicklungen im Bereich des Hardware Debuggings werden diese schnell eine Anwendung finden.

7.1 Was haben wir gelernt?

Anhand dieser Arbeit lassen sich eine ganze Reihe von Erkenntnissen für die Weiterentwicklung des Hardware-Debuggings auf der hier beschriebenen Grundlage aufzeigen. Allerdings muss man beachten, dass eines der Hauptziele der vorliegenden Arbeit im Beweis der Funktionsfähigkeit, dem sog. *proof of concept*, des neuartigen und patentierten Ansatzes liegt. Dieses Vorhaben gilt nach den vorliegenden Daten als erfolgreich umgesetzt. Man kann sehen, dass die beschriebene Emulationsplattform mit dem Simulator wie beabsichtigt zusammenarbeitet. Dadurch ist es auch gelungen, einen erheblichen Entwicklungsvorteil zu gewinnen. Wichtig wird in den zukünftigen Weiterentwicklungen u.a. sein, das ein Fehlverhalten nicht mehr nur anhand eines einzelnen Bits definiert wird. Vielmehr wird es in den nächsten Versionen vom NIHD© möglich sein, mehrere zusammenhängende Signalleitungen auf komplexeres Fehlverhalten zu überprüfen. Dazu ist es erforderlich die notwendigen Anpassungen im SVHDL-Code auf einer anderen Ebene durchzuführen.

Obwohl die derzeitige Stufe als Beweismittel angemessen ist, so eignet sie sich für komplexere Anwendungen eher weniger. Die automatisch von *Xilinx ISE* generierte SVHDL-Beschreibung auf Basis der SIMPRIM-Bibliothek beinhaltet wegen der bereits in dieser Stufe des Entwurfsablaufes vorgenommenen Optimierungen nicht mehr den vom Entwickler benutzten Aufbau. Zwar ist das logische Verhalten identisch, jedoch die Umsetzung dessen völlig verschieden. Die nächst höhere Abstraktionsstufe wäre direkt die VHDL-Schaltungsbeschreibung im Quellcode. Bei der Umsetzung einer solchen Beschreibung auf einem bestimmten Schaltkreis sind die dafür verwendeten Schritte allerdings abhängig vom Hersteller der Softwarepakete. Somit kann ein Programm (z.B. NIHD©) anhand einer VHDL-Beschreibung im Quellcode nicht erkennen, welche der zahlreichen Anweisungen letztendlich als Flipflop umgesetzt wird, oder welches interne Signal bei den Optimierungen wegfällt. Die Folge wären unnötig vergrößerte Schaltungsbeschreibungen, da Optimierungen durch die zusätzlichen Komponenten behindert würden.

Deshalb hat sich gezeigt, dass das gegenwärtig verwendete Eingangsformat (SVHDL) für den NIHD© noch nicht optimal ist. Vielmehr muss man ein Format, finden das in seinem Aufbau weitestgehend der Struktur der Ausgangsbeschreibung entspricht, vor allem aber schon erkennen lässt, welche Anweisungen z.B. als Flipflop umgesetzt werden. Darüber hinaus dürfen die internen Signale, die vom Benutzer in seiner Schaltung verwendet

werden, noch nicht optimiert sein. Zusätzlich lässt sich im Voraus nicht sagen, welche der Komponenten der *SIMPRIM*-Bibliothek verwendet werden, da dieser Schritt von zahlreichen Randbedingungen beeinflusst wird. Weil diese *SIMPRIM*-Bibliothek von Xilinx nicht vollständig nachgebildet ist, kann u.U. vorkommen das im SVHDL-Code für den NIHD© eine noch nicht nachgebildete Komponente vorkommt, was derzeit eine Fehlermeldung nach sich zieht.

Ebenso muss man das Festlegen eines Takt- und Reset-Signals völlig neu überdenken. Die derzeitige Verwendung dieser beiden Signale beruht auf einer Vielzahl von Vorraussetzungen. Insbesondere ist es fraglich, ob es genügt, das Taktsignal mit der höchsten Frequenz als zentralen Takt für die NIHD©-Zusatzkomponenten auszuwählen. Zum Einen ist dann dieses Signal nicht überwachbar, zum anderen kann dies zu einer Verschlechterung der Gesamtleistung des Systems führen, da die ursprüngliche Kernfrequenz nicht länger mehr umsetzbar ist. Hierbei ist ebenso wie beim Reset-Signal abzuwägen, ob zusätzliche externe Signale nicht den größeren Nutzen bringen, denn derzeit lassen sich diese beiden Signale nicht überwachen. Deshalb werden auch hierzu weitere Untersuchen durchgeführt.

7.2 Wo waren die Probleme?

Wie bei jeder größeren Arbeit traten neben Erfolgen auch Schwierigkeiten auf. Das wohl gravierendste Problem betraf das Ausgangsformat vom NIHD©. Die Dokumentation der einzelnen Unterkomponenten des *Xilinx*-Entwurfsablaufes lies insbesondere im Bereich der Dateiformat-Beschreibungen des automatisch erzeugten SVHDLs noch Wünsche offen. Gerade bei den Regeln zur Erzeugung der auf der SIMPRIM-Bibliothek basierenden SVHDL-Beschreibung bedurfte es zahlreicher Untersuchungen. Da es für diese Arbeit wichtig war die genaue Implementierung von Komponenten nachzubilden, musste man diese Informationen per Hand aufwendig aus den entsprechenden Quellen extrahieren.

Bei der Umsetzung der einzelnen Typen von *Look-up*-Tabellen war es notwendig eine geeignete Konvertierung der Daten in der richtigen Schachtelung zu implementieren, was einen entsprechenden Zeitaufwand erforderte. Darüber hinaus traten Schwierigkeiten bei der Frage nach der Notwendigkeit von Komponenten auf. Warum soll man, wenn es nur auf das logische Verhalten einer Schaltung ankam, TOC und ROC implementieren? Die Begründung dafür liegt in der Erhaltung der strukturelle Integrität einer

Schaltungsbeschreibung, die so wenig wie möglich verändert werden durfte. In Abhängigkeit von der Zielarchitektur konnte es vorkommen, dass SVHDL-Dateien erzeugt wurden, in denen TOC und ROC gänzlich fehlten. Dafür wurde z.B. eine Komponente mit dem Instanznamen „*C**_0_1*“ integriert. Die Zeichen „**“ stehen für eine vom Design-Compiler willkürlich vergebene Kennung. Wie sollten derartig dynamische Konstrukte automatisch verarbeitet werden? Die Definition dieser Komponente konnte man nur indirekt über die Namen der Pins finden. Sie entsprachen einer sog. *STARTUP*-Komponente, die im Falle eines Virtex als Zielarchitektur als *STARTUP_VIRTEX* bezeichnet wurden. Somit benutzt der NIHD© nun diese Komponente, wenn TOC und ROC nicht vorkamen.

Die nächste Schwierigkeit die sich stellte betraf zwei weitere Komponenten. Es handelte sich dabei um *X_ZERO* sowie *X_ONE*. Ausschließlich für das Simulationsmodell liefert die *X_ZERO*-Komponente eine statische ‘0’, während *X_ONE* eine logische ‘1’ erzeugen soll. So hätte der NIHD© diese Komponenten einsparen können und stattdessen im Simulator, die beiden mit ihren Ausgängen verbundenen Signale, auf zuvor festgelegte logische Pegel setzen können. Wenn die beiden Komponenten erhalten blieben, musste man auch deren logisches Verhalten nachempfinden. Aus den gleichen Gründen wie bei *TOC* und *ROC* wurde deshalb darauf verzichtet *X_ZERO* und *X_ONE* wegzuoptimieren.

Doch auch bei der Realisierung der Flipflops traten Schwierigkeiten auf. Welche Arten von Flipflops traten theoretisch auf? Über Umwege konnte man einen Blick in die Definitionen der SIMPRIM-Bibliothek werfen. Dort befanden sich eine ganzen Reihe von möglicherweise auftretenden Speicherelementen. Es stellte sich jedoch im praktischen Test heraus, dass ausschließlich *D-Flipflops mit asynchronen Reset, Clock Enable, SET und RESET* verwendet wurden. Deshalb wurde vorerst nur diese Art von bistabilen Kippstufen nachgebildet. Da auch die einzelnen internen Werte der Flipflops gespeichert werden sollten, war zunächst unklar wie man diese Werte puffert und anschließend im Simulator den entsprechenden Komponenten wieder zuweisen konnte, ohne Konflikte mit der Berechnungsreihenfolge zu erzeugen. Über die Simulation eines derartigen D-Flipflops lies sich zeigen, dass der interne Zustand des Flipflops gleichzeitig dem Wert am Ausgang der Komponente entsprach. Über diesen Umweg konnte man nun die Initialwerte der Flipflops setzen, ohne die restliche Schaltung zu beeinflussen.

Ebenso groß waren die Probleme bei Fragen, welche die Syntax der SVHDL-Beschreibung betrafen. Dabei war z.B. das Auftreten generischer Attribute mit Timing-Informationen zu nennen, deren Verwendung laut IEEE-Standard zwar zugelassen war, jedoch für das rein

logische Verhalten eine eher untergeordnete Rolle spielte. Dennoch konnte man feststellen, dass der Nutzen insbesondere für spätere Versionen des NIHDs© größer ist, als der Aufwand, diese Attribute zu implementieren, weshalb der NIHD© nun auch diese generischen Parameter erkennt und speichert.

Ein weiteres Problem bei der Implementierung des Debuggers stellten die Initialwerte externer Signale dar, die für Simulationszwecke erlaubt, jedoch bei der Synthese verboten sind. Derartige Initialwerte wurden auch vom NIHD© beachtet, da zum Zeitpunkt der Programmierung die Verwendung eines externen Simulators noch nicht feststand, weshalb sie bei einer Eigenentwicklung hilfreich gewesen wären.

Als aufwändig stellte sich auch die Klärung der Schwierigkeiten mit den externen Reset- bzw. Taktsignalen heraus. Bei der Benutzung eigener Signale war die Überwachung sämtlicher Signale möglich, jedoch wurden zusätzliche Pins am FPGA benötigt. Die Mitbenutzung bereits vorhandener Signale spart Pins, veränderte aber unter Umständen die Gesamt-Performance des Systems. Dennoch birgt die zweite Variante deutliche Vorteile, weshalb diese im NIHD© umgesetzt wurde. Problematisch war weiterhin die Realisierung der verwendeten Datenstrukturen aus der SVHDL-Datei. Bei der Definition der Signale (externe oder interne) kam es vor, dass der Benutzer auf Vektoren zurückgriff, die jedoch während des Design Flows in einzelnen Signale aufgespaltet wurden, wodurch ein Vektor auf *source level* nur schwer in der SVHDL-Beschreibunng wiederzufinden war. Wenn die Vektoren nicht zerlegt wurden, so bedurfte es einer eindeutigen Darstellung im NIHD© für jedes seiner Elemente. Die Lösung hierbei bestand in der Verwendung einer eindeutigen ID. Im NIHD© lagen nun sowohl die VHDL-Vektoren unverändert als eigenständige Objekte vor, als auch ihre aufgespaltenen Einzelsignale. Beide Gruppen wurde als *ID* eine eigene Zahl zugeordnet. Als Alternative dazu hätte man auch nur die Signal-Vektoren benutzen können, wobei sich allerdings durch die zusätzlich dafür entwickelten Methoden die Laufzeit des Programms verschlechtert hätte.

Die Implementierung der beiden Sammelkomponenten *BP_COLLECTOR* bzw. *BUFFER_FULL_COLLECTOR* traten ebenfalls Schwierigkeiten auf. Wie sollte man dort eine beliebige Anzahl von Signalen effizient mit einer einzigen Schleife abfragen? Da die Anzahl der Signale von Design zu Design variieren kann, boten große IF-Schleifen mit ODER- bzw. UND-Verknüpfungen den einzigen Ausweg. Daneben stellten auch RAM-Zellen ein Hindernis dar. In der während des Entwurfsablaufes erzeugten SVHDL-Datei traten unter Umständen auch RAM-Zellen der *SIMPRIM*-Bibliothek auf, wobei diese durch eine Reihe von Flipflops umgesetzt wurden. Deshalb müsste der Benutzer auch an

sie Schieberegister anschließen können. Die notwendigen Vorraussetzungen für die Verwirklichung dieses Vorhabens wurden erst nach der Entwicklung des NIHDs© geklärt, weshalb diese Zellen von der vorliegenden Version nicht behandelt werden. In den praktischen Versuchen spielte diese SIMPRIM-Komponente jedoch keine Rolle, so dass sie für die Testbeispiele nicht relevant war.

7.3 Zusammenfassung und Ausblick

In der vorliegenden Arbeit ist es gelungen, die Rahmenbedingungen zur Umsetzung des patentierten neuartigen Ansatzes für das Hardware-Debugging zu schaffen. Nach der Abgrenzung der Begriffe Simulation, Emulation und Debugging im zweiten Kapitel wurden andere bereits existierende Techniken auf diesem Gebiet betrachtet. Da keine der derzeitigen Methoden ohne größere Nachteile auskommt, bietet das hier beschriebene Vorgehen eine ernsthafte Alternative. Das dabei entstandene Programm NIHD© besteht aus zwei Teilen. Einerseits einem Parser für VHDL, andererseits einem Generator für das modifizierte VHDL. Der Benutzer hat die Möglichkeit eine beliebige VHDL-Beschreibung auf Fehlverhalten hin zu überprüfen. Die notwendigen Anpassungen und Erweiterungen im VHDL-Code erfolgen vollkommen automatisch. Da das benutzte *Spyder*-System die Zielarchitektur dieser Arbeit war, wurde im dritten Kapitel näher auf die zum Einsatz gekommenen Hardware eingegangen. Daneben wurde auf die Anwendung des besonderen SVHDL-Formats sowie dessen Ursprung im Entwurfsablauf näher erläutert.

Die beiden zentralen NIHD©-Komponenten Schieberegister und Breakpoints erlauben es dem Benutzer, jedes interne Signal auf eine logische Bedingung hin zu überprüfen, während die Schieberegister eine bestimmte Anzahl von Werten externer Signale speichert. Anhand dieser gespeicherten Signalwerte und den Flipflops sowie unter Verwendung der originalen VHDL-Beschreibung lässt sich der gesamte Zustand der Schaltung zu einen beliebigen Zeitpunkt exakt nachbilden. Aufgrund der Komplexität der Aufgabe ist diese Art des Hardware-Debuggings bisher noch durch einige Rahmenbedingungen begrenzt. Die Einschränkungen sind spätestens in der nächsten Version des NIHDs© beseitigt. Der Beweis der Umsetzbarkeit ist jedoch schon jetzt durch die Versuche erbracht. Das zeigt, es möglich ist eine Schaltung wie in dieser Arbeit beschrieben zu modifizieren, um sie anschließend effizient auf Fehler hin zu untersuchen.

Der NIHD© bietet also genügend Potential sowohl zu seiner eigenen Optimierung, als auch insgesamt zur Verbesserung des Hardware Debuggings. Zukünftig wird der Benutzer in einer beliebigen VHDL-Ausgangsbeschreibung die Variablen und interne Signale markieren können die für ihn von Interesse sind. Diese Informationen könnten anschließend auf der SVHDL-Ebene wie in der vorliegenden Arbeit beschrieben, verarbeitet werden. Da der Ladevorgang der BIT-Dateien über das Internet erfolgt, ist es sinnvoll den Debugger mit dem Begriff der sog. *System-Beans* zu koppeln. Dabei handelt es sich im Weitesten Sinne um Server-Dienste zum Management von *Embedded Systems*. Diese Dienste umfassen einerseits Überwachungsfunktionen zu Punkten wie Auslastung und Status, andererseits auch Aufgaben zur Entscheidungsfindung. Derartige Entscheidungen werden auf Basis dieser gewonnenen Daten getroffen, wobei es darum geht einen bestimmten Vorgang in Hardware oder in Software auszuführen zu lassen. Der NIHD© kann problemlos in dieser Umgebung interagieren, indem er in mehrere kleine Abschnitte (*Beans*) unterteilt wird, die vom Bean-Server verwaltet und für die Fehlersuche bereitgestellt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[Alt99] Altera Corp., *SignalTAP Embedded Logic Analyzer Megafunction*, http://www.paltek.co.jp/altera/library/ds/ds_signaltap_2e.pdf, 2002

[Ash96] Peter J. Ashenden, *The Designers Guide to VHDL*, 1996

[Bli00] Xilinx Inc., *Xilinx new generation of advanced productivity tools accelerates the design of ten million gate FPGAs*, http://www.xilinx.com/prs_rls/fdnise31i.htm, 2000

[Bou96] A. Auer, R. Kimmelmann, *Schaltungstest mit Boundary Scan*, 1996

[Bri02] Bridges2Silicon Inc., *Intelligent In-Circuit Emulator (IICE)*, http://www.bridges2silicon.com/html/technology.html, 2002

[Car99] C. Nitsch, *Implementierung und Test einer Emulationsplattform für die Hardware- Softwarepartitionierung eingebetteter Systeme*, Diplomarbeit, FZI Karlsruhe, 1999

[Cha97] K. C. Chang, *Digital Design and Modeling with VHDL and Synthesis*, 1997

[Chi02] Universität Hannover, IMS, Projektarbeit Mikroelektronik, *ChipDesign-Verifikation*, 2002

[Date00] J. Haufe, C. Fritsch, M. Gulbins, V. Lück, P. Schwarz: *Real-Time Debugging of Digital Integrated Circuts*, Design Automation and Test in Europe (DATE), Paris, 2000, 235-241

[Dat01] R. Ulrich, W. Grafen, J. Haufe, J. Große, *Debugging of FPGA based Prototypes - A Case Study*, Design Automation and Test in Europe, München, 2001, 109-113

[KHW99] K. Weiß, *Architekturentwurf und Emulation eingebetteter Systeme*, Dissertation, Universität Tübingen, 1999

[Koch98] G. Koch, U. Kebschull, W. Rosenstiel, *Breakpoints and breakpoint detection in source level emulation*, Design Automation of Electrical Systems, 1998

[Lab99] David Van den Bout, *The Practical Xilinx Designer Lab Book*, 1999

[Mod85] R. Walker, D. Thomas, *A Model of Design Representation and Synthesis*, 22. Design Automation Conference, 1985

[Ngd99] Xilinx Inc., Xilinx Software Design Flow, http://www2.informatik.uni-jena.de/~doersing/lehre/ps/a2_1i_dok/data/alliance/dev/fig4.htm

[Pat01a] U. Kebschull, *Verfahren zur schnellen Fehleranalyse digitaler Schaltungen und Schaltungsanordnung zur Durchführung des Verfahrens*, Forschungskontaktstelle Universität, Patentschrift, Az.: 10119170.7, AT.: 08.01.2001, Leipzig, 2001

[Pat01b] U. Kebschull, *Virtuelle Hardwarearchitektur zur Realisierung von Schaltungen auf der Register-Transfer- und Logikebene*, Forschungskontaktstelle Universität, Patentschrift, Az.: 10142553.8, AT: 30.08.2001, Leipzig, 2001

[Pro99] K. Weiß, *Architekturentwurf und Emulation eingebetteter Systeme*, Dissertationsvortrag, FZI Karlsruhe, 1999

[RAW98] A. Kirschbaum, J. Becker, M. Glesner, *A Reconfigurable Hardware-Monitor for Communication Analysis in Distributed Real-Time Systems*, 5. Reconfigurable Architectures Workshop, Orlando, 1998

[Sim02] Universität Kiel, Abt. Informatik, *Emulation vs. Simulation*, http://wwwags.informatik.uni-kl.de/lehre/ws01-02/Seminar/Ausarbeitungen/ChristophUrbanczik, Seminarunterlagen, 2002

[Ter02] Teradyne Inc., *Ai7 - Series Analog Test Instrument Subsystem*, http://www.teradyne.com/prods/cbt/products/library/ai7/ai7_broc_5102.pdf, 2002

[Ulm98] J. Riexinger, *Bewertung von VHDL-Werkzeugen bezüglich Simulations-geschwindigkeit und Syntheseergebnisse auf Grund von verschiedenen Codierstilen*, Universität Ulm, Abt. Elektrotechnik, Diplomarbeit, 1998

[Vir99] Xilinx Inc., Datasheet, *Virtex 2,5V Field Programmable Gate Arrays*, 1999

[Xil02] Xilinx Inc., *Chipscope ILA Software and Cores User Manual*, 2002

[Xil138] Xilinx Inc., *Virtex FPGA series configuration and readback*, XAPP138, 2000

Anhang

In Rahmen diese Diplomarbeit wird darauf verzichtet, den sehr umfangreichen Quellcode des entwickelten NIHDs© sowie sämtliche VHDL-Dateien in Papierform anzuhängen, stattdessen sind sämtliche Daten auf der beiliegenden CD gespeichert.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, November 2002

Jens Frauenschläger