Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Bereitstellung eines kompletten System-on-Chip aus lizenzfreien AMBA 2.0 Komponenten sowie des LEON3-SPARC-Prozessors im Xilinx-EDK

Diplomarbeit

Leipzig, Februar 2008

vorgelegt von

Markus Jäger
geb. am: 04.02.1983
Studiengang: Diplom Informatik

Betreuung: Prof. Dr. Udo Kebschull
Universität Heidelberg

Für meine Mutter

Danksagung

Hier möchte ich allen danken, die mir diese Arbeit ermöglicht haben.
Ich danke Prof. Dr. Udo Kebschull für das aufschlussreiche und anregende Arbeitsthema sowie meinem Betreuer Norbert Abel für seine immerwährende Hilfsbereitschaft und seine Unterstützung bei allen denkbaren Problemen.
Großer Dank gilt auch meiner Mutter, die mir mit ihrer Kraft und Ausdauer jeden Tag aufs Neue ein Vorbild ist.

Abstract

Aufgrund der wachsenden Ressourcen heutiger FPGAs, durch neue technologische Entwicklungen, erschließen sich immer neue Einsatzmöglichkeiten.
Beispielsweise wächst der Wunsch, ein vollständiges System in einem einzigen Chip einzubringen. Die sogenannten **S**ystems-**o**n-**C**hip (kurz SoC) bestehen dabei aus einem Prozessor, einen Bussystem, Schnittstellen zu externen Speichern und anderen Peripheriegeräten.
Die Firma Xilinx bietet mit ihrer Software EDK eine IP-Core Bibliothek an, mit der es möglich ist, ein komplettes SoC für einen FPGA zu synthetisieren. Die Xilinx-IP-Core-Bibliothek benutzt dabei den Soft-Prozessor MicroBlaze als μP. Die IP-Core Bibliothek von Xilinx ist nicht Open-Source und zu ihrer Benutzung werden Lizenzgebühren verlangt.

In dieser Arbeit wird eine neue IP-Core Bibliothek bereitgestellt, welche Open-Source ist und damit frei einsehbar und frei verwendbar ist. Die neue IP-Core Bibliothek wird durch diese Arbeit in den Workflow des Xilinx-EDK eingebunden und ist somit komfortabel benutzbar.
Als Grundlage dient die IP-Core Bibliothek der Firma Gaisler Research, auch genannt **G**aisler **R**esearch **Lib**rary (kurz GRLIB). Die GRLIB besitzt eine Vielzahl von IP-Cores unter denen, für jeden IP-Core der Xilinx Bibliothek, ein Ersatz gefunden werden konnte. Die GRLIB setzt als μP auf den LEON3-Prozessor. Der LEON3-Prozessor wurde nach den Spezifikationen der SPARC entworfen und ist ein höchst flexibler und konfigurierbarer Soft-Prozessor.
In dieser Arbeit wurde weiterhin das „SnapGear“-Linux evaluiert, welches auf dem LEON3-Prozessor mit Komponenten der GRLIB ausgeführt werden kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der praktische Einsatz von **F**ield **P**rogrammable **G**ate **A**rrays (kurz FPGAs) hat in den letzten Jahren weiter zugenommen. Der Grund für diese rasante Entwicklung sind die Grundprinzipien der FPGAs.
So ist ein FPGA rekonfigurierbar. Dadurch kann bei entdeckten technischen Fehler die Funktionalität leicht und schnell korrigiert werden ohne einen Chip oder eine Platine aus dem Gerät auszutauschen. Gleichzeitig liegt auf einem FPGA echte Hardware-Parallelität vor.
Allein durch die Automobilindustrie erfährt der „FPGA-Markt“ ein stetiges Wachstum. Zusätzlich erreichen die FPGAs, mit ihren steigenden Integrationsdichten und Taktfrequenzen, auch in Bereichen der Forschung immer größere Relevanz.

Bei der Betrachtung von FPGAs der Firma Xilinx, lassen sich interessante Trends der Entwicklung feststellen. In den letzten Jahren werden die FPGAs der Familie Virtex-II Pro, welche seit dem Jahr 2002 erhältlich sind, für Forschungszwecke eingesetzt. Die neuen FPGAs der Familie Virtex-5, seit 2006 erhältlich, haben entscheidende Verbesserungen, die ihren Einsatz noch vielseitiger machen.

So gibt es FPGAs der Familie Virtex-5 mit:

- ständig wachsender Kapazität an verwertbarer Logik (Virtex-5 Device: XC5VLX330T mit 51840 Slices)
- einer bis zu 550 MHz bereitgestellten Clock
- auf dem Chip zur Verfügung stehenden Block-RAMs bis zu einer maximalen Gesamtkapazität von 1,3 MB

- zusätzlichen Monitoren, zur Überwachung der Betriebstemperatur und Spannungsversorgung

Vor allem die zusätzlichen Möglichkeiten zur Überwachung des Betriebszustandes und die ständig sinkende Leistungsabnahme, machen den FPGA für eingebettete Systeme zukünftig noch attraktiver.

FPGA Bezeichnung	Anzahl		
	Slices	Block-RAMs	PowerPCs
Xilinx Virtex-II Pro Familie			
XC2VP4	3008	28	1
XC2VP7	4928	44	1
XC2VP20	9280	88	2
XC2VP30	13696	136	2
XC2VP40	19392	192	2
XC2VP50	23616	232	2
XC2VP70	33088	328	2
XC2VP100	44096	444	2
Xilinx Virtex-4 Familie			
XC4VFX12	5472	36	1
XC4VFX20	8544	68	1
XC4VFX40	18624	144	2
XC4VFX60	25280	232	2
XC4VFX100	42176	376	2
XC4VFX140	63168	552	2
Xilinx Virtex-5 Familie			
XC5VLX30T	4800	72	0
XC5VLX50T	7200	120	0
XC5VLX85T	12960	216	0
XC5VLX110T	17280	296	0
XC5VLX220T	34560	424	0
XC5VLX330T	51840	648	0

Tabelle 1.1: Xilinx Virtex Familien und ihre Kenngrößen [Xilfam1], [Xilfam2], [Xilfam3]

Tabelle 1.1 zeigt einen Ausschnitt der verbreitetsten Virtex-Familien. Dabei werden die, für diese

Arbeit relevanten, Kenngrößen (Slices[1], Block-RAMs, PowerPCs) angezeigt. Es ist zu erkennen, dass nicht alle FPGAs mit einem PowerPC Prozessorkerne ausgeliefert werden. Trotzdem sind viele Benutzer bestrebt einen Prozessor auf einem beliebigen FPGA einzusetzen. Auf einem FPGA ohne PowerPC Prozessorkern ist dies nur durch einen Soft-Prozessor möglich.
Durch die ständig wachsende Anzahl der zur Verfügung stehenden Slices, auf einem FPGA, wird es immer lohnender ein komplettes System auf einem einzelnen FPGA unterzubringen. Dieses Konzept nennt sich **S**ystem-**o**n-**C**hip oder System-on-a-Chip (kurz SoC) und umfasst die Einbettung eines Systems mit zugehörigen SoC-Komponenten auf einem konfigurierbaren Chip. Solche SoC-Komponenten sind beispielsweise ein Prozessor, Register, Busse, Schnittstellen zu Speichern und anderen Peripheriekomponenten.

1.1 Motivation

Möchte ein Benutzer einen Prozessor auf einem FPGA einsetzen, auf dem kein kompakter Prozessorkern, wie der PowerPC, vorhanden ist, so kann dies nur mit einem Soft-Prozessor geschehen.
Die Firma Xilinx bietet eine Bibliothek von **I**ntellectual **P**roperties (deut.: geistige Eigentümer, kurz IPs) an. Diese IPs sind einzelne SoC-Komponenten, mit denen es bereits möglich ist ein komplettes SoC zu erstellen. Diese Komponenten werden auch IP-Cores genannt. Ein besonderer IP-Core ist der MicroBlaze-Prozessor. Dies ist ein Soft-Prozessor, der den fehlenden kompakten PowerPC Prozessorkern ersetzen kann. Die Bezeichnung Soft-Prozessor oder Softcore meint, dass ein IP-Core, mit seinem Verhalten, in einer **H**ardware **D**escription **L**anguage (deut.: Hardwarebeschreibungssprache, kurz HDL) vorliegt und nur durch den Prozess der Synthese, mit einer entsprechenden Software, auf einem konfigurierbaren Chip als Hardware betrieben werden kann.
Der Nachteil der Xilinx Bibliothek ist allerdings, dass an die Verwendung deren IP-Cores Lizenzbedingungen geknüpft sind. Diese Lizenzbedingungen machen den Einsatz von Xilinx-IP-Cores, wie dem MicroBlaze-Prozessor, oft unflexibel oder teuer.

Es werden IP-Cores benötigt, welche möglichst flexiblen Lizenzbedingungen unterliegen und in einsehbarem VHDL-Code vorliegen. Mit diesem IP-Cores soll es möglich sein, alle wichtigen IP-Cores der Xilinx Bibliothek zu ersetzen.

1.2 Inhalt der Arbeit

Diese Arbeit beschäftigt sich mit der Erstellung eines kompletten SoC. Das Ziel dieser Arbeit ist, alle SoC-Komponenten in vollständigen, ungeschützten VHDL-Code vorliegen zu haben. Alle

[1] Laut [Xilfam3] ist zu Beachten, dass ab der Virtex-5 Familie ein Slice aus 4 LUTs und 4 flip-flops besteht. Bis zur Virtex-4 Familie waren es 2 LUTs und 2 flip-flops.

Komponenten, einschließlich des Prozessors, werden unter Beachtung der GNU **G**eneral **P**ublic **L**icense (kurz GPL) beliebig veränderbar und frei verwendbar sein. Für ein SoC aus diesen freien Komponenten soll das Betriebssystem Linux nutzbar gemacht werden.

Die Eigenschaft der IP-Cores der GPL zu unterliegen wird in dieser Arbeit mit dem Adjektiv „frei“ bezeichnet.

Zur Synthese des VHDL-Codes werden die Werkzeuge Xilinx ISE 8.2i (kurz ISE) und Xilinx-EDK 8.2i (kurz EDK) genutzt. Die Besonderheit des Werkzeugs EDK ist es, mit der Xilinx-IP-Core-Bibliothek ausgestattet zu sein. Die neuen, freien IP-Cores werden einen vollständigen Ersatz der Xilinx-IP-Core-Bibliothek bilden und werden ebenfalls, wie die IP-Cores der Xilinx Bibliothek, in den Workflow des EDK eingebunden.

Als Quelle freier IP-Cores dient die von der Firma **G**aisler **R**esearch (kurz GR) bereitgestellte **G**aisler **R**esearch **Lib**rary (kurz GRLIB). Diese Bibliothek beinhaltet vollständig zugängliche IP-Cores, die der oben genannten GPL unterliegen. Unter anderem befindet sich in der GRLIB eine VHDL-Entity eines Soft-Prozessors namens LEON3.
Als Ergebnis dieser Arbeit steht die Verbindung der freien GRLIB-SoC-Komponenten mit der Benutzerfreundlichkeit des Xilinx Werkzeugs EDK.

1.2.1 Gliederung der schriftlichen Ausarbeitung

Die schriftliche Ausarbeitung dieser Arbeit ist in folgende Abschnitte gegliedert.

In **Kapitel 1** der Arbeit werden die Lizenzrechtlichen Bedingungen (Kapitelpunkt 1.3) der Xilinx-IP-Cores und der GRLIB-IP-Cores näher erläutert, welche dann die Motivation dieser Arbeit untermauern werden.

Das **Kapitel 2** bereitet die Grundlagen auf, welche benötigt werden, um dem weiteren fachlichen Inhalt der Arbeit folgen zu können und um dargestellte Probleme sowie gewählte Lösungen nachvollziehbar zu machen.
Es wurden spezielle Schwerpunkte gelegt auf allgemeine Möglichkeiten der Realisierung von Systems-on-Chip (Kapitelpunkt 2.1) auf FPGAs und die dazu verwendbaren Werkzeuge von Xilinx. In Kapitelpunkt 2.2.2 werden alle wichtigen EDK-Konstrukte vorgestellt und erklärt, welche später in Kapitel 5, dem eigentlichen Kern der Arbeit, benutzt wurden, um die Einbindung der GRLIB-SoC-Komponenten in das Werkzeug EDK zu realisieren. Dadurch erhält der Leser einen Einblick in die eigentlichen Möglichkeiten sowie das Potential der Software EDK und es werden später geschilderte Probleme besser verständlich.

Kapitel 3 vermittelt einen Einblick in den aktuellen Stand der Technik. Es werden vergangene Arbeiten vorgestellt, in denen die GRLIB zum Einsatz gekommen ist. Dieses Kapitel ordnet die Arbeit in ein technisches Umfeld ein. Kapitelpunkt 3.1 beschäftigt sich ausführlich mit der Gaisler

Research IP Library. In den Kapitelpunkten 3.1.2 bis 3.1.6 wird der Zustand beleuchtet in dem die GRLIB offiziell zu erhalten ist. Es wird ein Einblick in die Verzeichnisstruktur und die Bibliotheken gegeben. Zusätzlich wird der Arbeitsablauf, auch genannt „Workflow“, der GRLIB vorgestellt, wie er von der Firma GR vorgegeben wird um ein SoC mit dem LEON3-Prozessor zu synthetisieren.

In Kapitelpunkt 3.1.5 werden die Modelle der GRLIB im Detail vorgestellt. Als Kern dieser Vorstellung stehen die Designumsetzungen der in der GRLIB verwendeten Busse. Diese sind der **A**dvanced **H**igh-performance **B**us (kurz AHB) und der **A**dvanced **P**eripheral **B**us (kurz APB). Der AHB und der APB bilden das Grundgerüst eines jeden LEON-Systems.
Der LEON3-Prozessor spielt hier eine untergeordnete Rolle, da er nur ein Master des AHB ist. Der LEON3-Prozessor ist für die Grundfunktionalität eines GRLIB Systems unwichtig. Er könnte ebenso gut durch eine einfache Steuereinheit ersetzt werden.

Um die neuen Busse in ihrer Umgebung besser zu verstehen werden Analogien, zum bekannten MicroBlaze-System der Firma Xilinx, deutlich gemacht.
In Kapitelpunkt 3.1.5.4 wird erklärt wie Gaisler Research seiner GRLIB eine technologische Unabhängigkeit verliehen hat und es wird erläutert wie diese Unabhängigkeit in EDK fortgesetzt wird. Die GRLIB ist somit für viele FPGAs verschiedenster Firmen, wie Xilinx und Altera, ausgelegt.

In Kapitelpunkt 3.1.6 werden weitere Werkzeuge genannt und erklärt, welche während des intensiven Umgangs mit einem LEON-System benötigt werden.
Dazu gehört der Compiler, zum Kompilieren nativer C-Programme für den LEON3-Prozessor, der **G**aisler **R**esearch Debug**mon**itor (kurz GRMON) für das Debugging eines auf dem FPGA-Board befindlichen Systems und das Werkzeug SPARC-Simulator (kurz TSIM) zur Simulation eines kompletten LEON-Systems auf der Konsole. Hier werden zum besseren Verständnis wieder Analogien und Vergleiche zum MicroBlaze-System gezogen.

In **Kapitel 4** wird der Ansatz der Arbeit dargestellt. Es wird kurz auf die Ausgangssituation eingegangen und dann alle Ansätze und Vorgehensweisen, zur Erreichung der Ziele dieser Arbeit, vorgestellt und begründet.

Kapitel 5 umfasst die eigentlichen Leistungen der Arbeit. Es wird eine Schilderung der Ausgangssituation vorgenommen und die Ideen zur Umsetzung der Ziele besprochen. In Kapitelpunkt 5.2 wird beschrieben wie die Einbindung eines einzelnen allgemeinen IP-Cores, der GRLIB, in das EDK erfolgt ist. Dazu werden die in Kapitel 2 erklärten Grundlagen benötigt, da die dort aufgelisteten Konzepte benutzt wurden, um eine saubere Einbindung eines IP-Cores in das EDK zu bewirken.

In Kapitelpunkt 5.2 werden bei diesem Prozess aufgetretene Probleme und Besonderheiten besprochen und die Entscheidung zur Lösung begründet und diskutiert. Ein großes Problem hinsichtlich Constraints trat hier bei den IP-Cores Gaisler Research DDR RAM und Gaisler Research Ethernet

auf. Dieses Problem wurde mit Hilfe der **T**ool **C**ommand **L**anguage (kurz Tcl) und Synthese Parametern für das Tool XST gelöst.
Unter dem Begriff „Constraints“ (deut.: Einschränkung) werden hier Bedingungen bezeichnet, welche ein IP-Core bei der Synthese erfüllen muss. Solche Bedingungen sind beispielsweise Forderungen an zeitliches Verhalten der Schaltsignale, auch „Timing“ genannt.

Um eine komplette komfortable Einbindung der GRLIB-SoC-Komponenten in das EDK zu erreichen wurden zusätzlich zwei Bridges konstruiert. Diese „AHB zu OPB“ und „AHB zu LMB“-Bridges sollen hier in ihrem Design ausführlich vorgestellt werden. Zusätzlich werden Gründe für die Erstellung und sich erschließende Möglichkeiten, durch Erstellung dieser Bridges geliefert. Die genaue Beschreibung dieses Sachverhaltes, sowie Begriffserklärungen, befinden sich im Kapitelpunkt 5.4.

In Kapitelpunkt 5.5 werden ausgewählte IP-Cores der GRLIB besprochen, welche verstanden werden müssen, um ein LEON3-System benutzen zu können. Es sollen hier nicht die genauen Funktionsweisen der IP-Cores erläutert, sondern Bedienungsbesonderheiten erklärt werden, welche sich durch die Einbindung der IP-Cores in das EDK ergeben haben.

Das **Kapitel 6** befasst sich mit der Erstellung nativer C-Programme für ein LEON3-System. Speziell der Aufbau eines C-Programms für den LEON3 unterscheidet sich von den Programmen für einen MicroBlaze-Prozessor. Dies liegt nicht zuletzt daran, dass für den LEON3-Prozessor keine so umfangreiche Softwarebibliothek vorhanden ist, wie für den MicroBlaze-Prozessor.
In diesem Kapitel wird das allgemeine Vorgehen erläutert, wie der LEON3-Prozessor Bus-Transfers einleiten kann.

Kapitel 7 beschäftigt sich mit einer Linux Distribution namens „SnapGear“. Das SnapGear-Linux ist speziell zur Ausführung auf einem SoC der GRLIB gedacht und kann mit dem LEON3-Prozessor betrieben werden.
Das SnapGear-Linux bietet eine umfassende Treiberdatenbank speziell für die SoC-Komponenten der GRLIB. Der Kapitelpunkt 7.2.1 beschreibt die Möglichkeiten zur Konfigurationen eines SnapGear-Linux-Kernels für ein beliebig gegebenes LEON3-System.
In Kapitelpunkt 7.2.2 wird beschrieben, wie es theoretisch möglich ist, eigene SnapGear-Linux Treiber, für beliebige LEON3-Systemkomponenten zu erstellen und in SnapGear-Linux einzubinden. Eine praktische Umsetzung dieses Prozesses wurde allerdings in dieser Arbeit nicht durchgeführt.

In **Kapitel 8** werden alle Ergebnisse der Arbeit zusammengefasst. Es werden alle erreichten Ziele aber auch Grenzen dieser Arbeit aufgezählt.
Da der LEON3 ein umfassender Soft-Prozessor ist, wird er, in Kapitelpunkt 8.3, mit dem MicroBlaze-Prozessor hinsichtlich seiner Größe, Performance und benötigten Ressourcen verglichen. Zusätzlich soll im Kapitelpunkt 8.4 ein Performance-Vergleich eines MicroBlaze-SoC und eines LEON3-SoC mit jeweils maximal möglicher Leistung stattfinden.

Durch **Kapitel 9** sollen Lösungsmöglichkeiten gezeigt und Anregungen gegeben werden, in wie weit sich Verbesserungen zukünftig umsetzen lassen. Von manchen Verbesserungen wurde in dieser Arbeit Abstand genommen, da sie teilweise mit Veränderungen des, von Gaisler Research erstellten, VHDL-Codes verbunden sind. Würden diese Änderungen vorgenommen, so müssten umfassende Tests erfolgen, um die Funktionalität und Fehlerfreiheit zu gewähren.
Des weiteren wird die GRLIB ständig aktualisiert und gefundene Bugs werden von der Firma GR ständig behoben. Deshalb soll hier beschrieben werden wie eine neue Version der GRLIB übernommen werden kann und welche Besonderheiten bei diesem Prozess zu beachten sind.

Anhänge

Der **Anhang A** bietet Ergänzungen zum Kapitelpunkt 3.1, welche nicht für den Leitfaden dieser Arbeit wohl aber für den Umgang mit der GRLIB benötigt werden.

Der **Anhang B** widmet sich ausführlich dem LEON3-Prozessor. Es wird allgemein die SPARC-Architektur erläutert, nach der der LEON3 konstruiert ist. Im Anhang B.2 werden viele Fakten des LEON3 beleuchtet, die ein gutes Verständnis des Prozessors erzeugen. Besprochen werden unter anderem die Integer-Unit, die Floating-Point Unit, das Cache-System und die Memory Managment Unit. Besonderes Augenmerk wird auf die Memory Managment Unit des LEON3 gelegt, da diese nicht im MicroBlaze-Prozessor vorhanden ist.

In **Anhang C** soll der sehr umfangreiche LEON3-Prozessor aus der Sicht der eingebetteten Systeme (engl.: embedded systems) betrachtet werden. Als Besonderheit bietet die Firma Gaisler Research äußerst Fehlertolerante IP-Cores an, welche mit Schutz- und Fehlerkorrekturmechanismen versehen wurden, um Betriebsbedingungen stand zu halten bei denen Eingebettete Systeme vorkommen können.
Dazu gehört auch ein fehlertoleranter LEON3 (kurz LEON3FT). In Kapitelpunkt C.2 wird die Echtzeit-Anforderung auf das LEON3-System betrachtet und für den LEON3 erhältliche **R**eal**t**ime**o**perating**s**ystems (kurz RTOS) vorgestellt. Als sehr wichtige Komponente soll hier der Interrupt-Controller des LEON3-Systems in seiner detailierten Funktionsweise besprochen werden.

Anhang D stellt eine Zusammenfassung dar, wie die DVD, zu dieser Diplomarbeit, gehandhabt wird. Die DVD beinhaltet beispielsweise alle in das EDK eingebundenen IP-Cores. Soll ein neues EDK-Projekt mit der GRLIB ausgestattet werden, so können die Daten auf der DVD benutzt werden.

Leitfaden der Arbeit

Kapitel 2 (Grundlagen) und **Kapitelpunkt 3.1** (Die GRLIB) stellen Vorimplementierungen vor, die zu Beginn der Arbeit bereitstanden. Diese Vorimplementierungen werden in **Kapitel 5** (Einbindung der GRLIB in das EDK), dem Implementierungskapitel dieser Arbeit, ausgenutzt, um eine benutzerfreundliche Synthese von SoCs, mit Hilfe der GRLIB-IP-Cores, zu erreichen.
Nachdem die Hardware synthetisiert werden kann, wird in **Kapitel 6** (Native C-Programme für ein LEON3-SoC) erklärt wie erste, einfache Software auf einem LEON3-System ausgeführt werden kann. In **Kapitel 7** (Linux auf dem LEON3-System) wird aufbauend erläutert, wie eine komplexere Software, wie ein Linux-Betriebssystem, auf einem LEON3-System ausgeführt wird.

1.3 Lizenzrechtliche Bedingungen und Probleme

Dieser Kapitelpunkt stellt die Lizenzbedingungen der Xilinx-IP-Core-Bibliothek und der GRLIB gegenüber. Daraus ist erkennbar, warum die GRLIB eine Alternative zu den IP-Cores der Firma Xilinx ist.

1.3.1 Xilinx, Inc. CORE SITE LICENSE AGREEMENT

Die Firma Xilinx stellt mit ihrer Software EDK ein umfangreiches Werkzeug zur Verfügung, um Eingebettete Systeme und SoC Lösungen zu erstellen.
In der Version 8.2i des EDK liegt beispielsweise der Soft-Prozessor MicroBlaze in der Version 5.00.c vor. Aus rechtlichen Gründen veröffentlicht die Firma Xilinx den MicroBlaze allerdings nur in verschlüsseltem VHDL-Code. Andere IP-Cores der „Communication“-Klasse, wie der OPB Ethernet oder der OPB Uart 16550 IP-Core sind, neben den normalen Lizenzbedingungen, zusätzlich mit einem zeitlichen Verfall der Nutzungsrechte versehen.
Grundsätzlich unterliegen alle IP-Cores, welche sich in der Xilinx Bibliothek des EDK befinden, dem Xilinx, Inc. CORE SITE LICENSE AGREEMENT.

Aus akademischen Gründen soll an dieser Stelle ein Auszug aus diesen Lizenzvereinbarungen erfolgen. Es wird allerdings darauf hingewiesen, dass dieser Auszug kein Ersatz gegenüber dem original Dokument darstellt. Ebenfalls sind Ungenauigkeiten in der Formulierung, welche von der Übersetzung herrühren, nicht ausgeschlossen.

Die Firma Xilinx definiert den Begriff „Intellectual Property Rights“ mit den auszugsweisen Punkten:

> (i) Rechte assoziiert mit der Arbeit der Autoren
>
> (iv) Patent- und Designrechte des gewerblichen Eigentums
>
> (v) alle weiteren Rechte am geistigen Eigentum

Dies sind drei Punkte der Begriffsdefinition und deren, im Xilinx Lizenzdokument verwendeten, römische Gliederungspunkte. Es sind noch weitere Punkte genannt, welche hier aber nicht von Belang sein sollen.

Die eigentliche Lizenz umfasst auszugsweise Äußerungen wie:

Nach Zahlung der anwendbaren Gebühren, bewilligt Xilinx dem Lizenznehmer eine nicht-exklusive, nicht übertragbare, widerrufbare Lizenz, welche folgende Genehmigung umfasst:

> (i) Der Lizenznehmer darf das lizenzierte Material (IP-Cores) an den lizenzierten Örtlichkeiten zur Erstellung, Simulation und Implementierung von Chipdesigns benutzen. Dabei sind nur FPGAs der Firma Xilinx zur Konfiguration gestattet.

Es ist also nicht gestattet FPGAs anderer Firmen als Xilinx mit den erstellten Designs aus Xilinx-IP-Cores zu konfigurieren. Dies ist ein grundlegender Unterschied zu den Rechten der GRLIB, die das Ziel haben eine technologische Unabhängigkeit zu erreichen.

Zusätzlich nennt die Firma Xilinx noch Einschränkungen zur Benutzung:

- Die Benutzung des lizenzierten Materials auf nicht Xilinx FPGAs ist verboten.
- Das Erstellen von Kopien, des lizenzierten Materials, ist nur mit starken Einschränkungen gestattet. Weiterführende Erstellungen von Kopien sind nur mit Genehmigung der Firma Xilinx erlaubt.
- Der Lizenznehmer darf kein lizenziertes Material an Dritte weitergeben ohne Genehmigung der Firma Xilinx. Gestattet ist allerdings das Weitergeben von Bitstreams, zur Konfiguration von Xilinx FPGAs und Speicherung, an Dritte.
- Werden Xilinx-IP-Cores mit anderen Technologien, wie Software, dritter Parteien in Verbindung gebracht, ist der Lizenznehmer verpflichtet, Lizenzen Dritter zu beschaffen.
- Es ist dem Lizenznehmer nicht gestattet, ohne Genehmigung der Firma Xilinx, Resultate eines Benchmarks, des lizenzierten Materials zu veröffentlichen.

Es ist zu erkennen, dass erhebliche Einschränkungen bei der Benutzung der Xilinx-IP-Cores vorliegen. Wobei die obige Auflistung, der Lizenzrechtlichen Bedingungen, nicht vollständig ist. Diesen Bedingungen sind in der Forschung sehr hinderlich, da es dort nötig ist, viele Experimente mit Designs durchzuführen, welche empfindlich an die Grenzen der Lizenzrechte stoßen können.

1.3.2 GNU General Public License (GPL)

Eine Alternative stellt hier wiederum die GRLIB dar. Zur verantwortungsvollen Benutzung der GRLIB werden hier einige Auszüge der GNU **G**eneral **P**ublic **L**icense (kurz GPL) diskutiert, welcher die GRLIB unterliegt. Es sei wieder gesagt, dass diese Auszüge kein Ersatz des vollständigen GPL Lizenzdokuments sind. Wenn in diesem Kapitelpunkt von Software gesprochen wird, so ist damit auch die GRLIB gemeint, obwohl mit ihr Hardware synthetisiert wird.

Die GNU General Public License hat es sich zum Ziel gemacht die Freiheiten der Benutzer von Produkten, die der GPL unterliegen, zu bewahren und nicht zu beschneiden. Dadurch wird sichergestellt, dass die Software frei ist für alle ihre Nutzer und Folgenutzer.
Wenn in der GPL von freier Software gesprochen wird, wird die Freiheit des Gebrauches gemeint, nicht aber die Kostenfreiheit. Es ist also erlaubt eine Gebühr auf die Benutzung einer Software zu erheben. Dazu muss die besagte Software im Vergleich zur ursprünglichen GPL Software verändert worden sein.

Einige Richtlinien der GPL sind auszugsweise:

- Benutzer von GPL Software haben das Recht die Software frei zu verwenden und zu verändern, müssen aber die Veränderungen ebenfalls an die GPL binden.
- Auf die Veränderung der vorhergehenden Version muss hingewiesen werden.
- Dem Lizenznehmer ist es erlaubt die Software beliebig zu kopieren und zu verbreiten. Vorraussetzung für diese Erlaubnis ist ein an die Kopie auffällig angebrachter Urheberrechtsvermerk und ein Verzicht auf Garantie.

Diese drei Punkte der GPL haben auch für die vorliegende Arbeit große Bedeutung. Während der Umsetzung der Arbeit war es nötig Veränderungen an der GRLIB vorzunehmen, um die Einbindung der IP-Cores in das EDK fehlerfrei und benutzerfreundlich zu ermöglichen.
Aufgrund der GPL werden somit alle Veränderungen genau beschrieben und deren Folgen deutlich gemacht. Ebenfalls sind die Veränderungen öffentlich einsehbar.

1.3.3 Zusammenfassung

Durch die Ausführungen des Kapitelpunktes 1.3 ist deutlich geworden, dass die GRLIB in Verbindung mit der GPL eine Alternative zu den IP-Cores der Firma Xilinx ist.
Es steht nun der Weg offen die SoC-Komponenten der GRLIB als IP-Cores in das Werkzeug EDK einzubinden und beide Vorteile miteinander zu verbinden.
Die Vorteile der GRLIB sind ihre freien SoC-Komponenten und die Vorteile des Xilinx-EDK sind die benutzerfreundlichen und übersichtlichen Möglichkeiten ein vollständiges SoC zu synthetisieren und die zugehörige Software zu kompilieren.

Das Xilinx-EDK und die GRLIB arbeiten „boardorientiert“. Das bedeutete, dass der Benutzer vor der Erstellung eines SoC entscheiden muss, auf welchem FPGA-Board das Design betrieben werden soll. Durch diese Entscheidung werden beispielsweise im EDK die Pinbezeichnungen festgelegt und SoC-Komponenten, die nicht benutzt werden können, ausgeblendet.
Die GRLIB arbeitet nach dem selben Prinzip. Da die GRLIB allerdings viel weniger FPGA-Boards als das EDK unterstützt wird die Erstellung eines SoC bei Benutzung eines nicht bekannten Boards erheblich erschwert. Dies ist ein weiterer Grund die Vorteile der beiden SoC-Lösungen zu verbinden.

Kapitel 2

Grundlagen

Wurden früher noch einzelne integrierte Schaltkreise auf einer Platine zu einem vollständigen eingebetteten System verbunden, so wird heute das Design des eingebetteten Systems mit einer HDL beschrieben, dann synthetisiert und auf einen einzelnen konfigurierbaren Chip geladen. Ein solches Werkzeug mit dem es möglich ist, den vollständigen Entwicklungsprozess eines SoC durchzuführen ist beispielsweise die Software Xilinx-EDK.
Die Vorteile von SoCs sind ihr geringer Platzbedarf, in den einzelnen Geräten, und die verhältnismäßig kurze Zeit in der sie entwickelt werden können, um dann vermarktet zu werden, auch genannt „time to market“. Die Umsetzung der SoCs in Hardware ist beispielsweise auf CMOS Schaltkreisen oder FPGAs möglich.
Der Zusatz der Rekonfigurierbarkeit von FPGAs hat den Vorteil, dass sich schnelle und einfache Support-Möglichkeiten bieten. Dadurch kann ein Fehler im SoC-Desgin durch eine Rekonfiguration, mit einem im Werk erstellten neuen Bitstream, schnell behoben werden. Auch das aufspielen eines neuen Updates ist meist ohne das Austauchen von Gerätekomponenten möglich.

2.1 System-on-Chip (SoC)

Die Besonderheit der SoCs ist, wie der Name sagt, das vollständige Unterbringen eines ganzen Mikro-Controllers auf einem einzigen Chip. Da in dieser Arbeit mit einem FPGA als Chip gearbeitet wurde, werden im Folgenden nur SoCs auf FPGAs betrachtet.

Zu den Komponenten eines SoC gehören alle denkbaren Komponenten, welche in Hardware realisierbar sind. SoCs bestehen aus dem **Mikrop**rozessor (kurz μP, CPU), einem oder mehreren Bussen und Schnittstellen zu Speichern und **E**ingabe/**A**usgabe (kurz E/A) Komponenten. Gegebenenfalls können sich zusätzliche, kleine aber schnelle, Speichermodule direkt auf dem FPGA

befinden. In diesem Fall wäre nicht nur die Schnittstelle zum Speicher, sondern auch der Speicher „on Chip“.

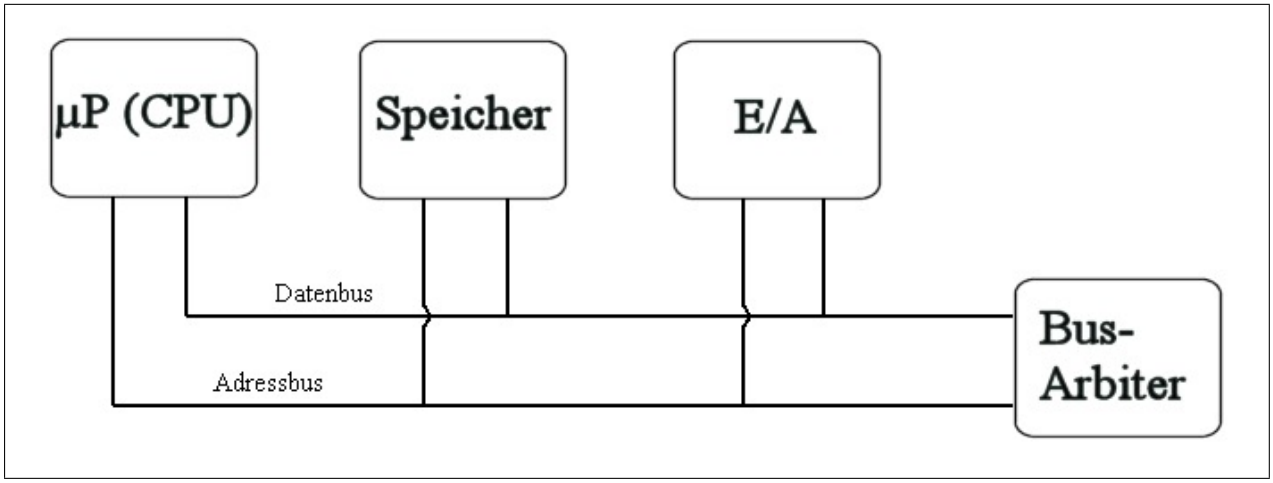

Abbildung 2.1: Blockschaltbild eines SoC

Abbildung 2.1 Zeigt das Blockschaltbild eines typischen SoC, mit allen Standardkomponenten. Die Linien des Daten- und Adressbus bilden den gesamten Bus.

2.1.1 Der Mikroprozessor (μP)

Der Mikroprozessor eines SoC ist wie bei einem **P**ersonal **C**omputer (kurz PC) ein wichtiger Bestandteil des Systems. Um Bezug zu den zukünftig in dieser Arbeit auftauchenden Begriffen zu halten, wäre der μP in der GRLIB, der LEON3-Prozessor und in der Xilinx-IP-Core-Bibliothek der MicroBlaze-Prozessor.
Wie jede CPU, eines Personal Computers, ist der μP in der Lage als zentrale Steuereinheit zu arbeiten. Er liest Befehle, auch Operationen genannt, aus dem Programmspeicher in seine Register, decodiert sie und führt die Befehle aus. Anschließend schreibt der μP berechnete Daten in den Systemspeicher oder führt andere Operationen aus.
Eine wichtige Unterscheidung, welche im letzten Satz geprägt wurde, ist die Unterscheidung zwischen Programmspeicher und Systemspeicher.

Der Programmspeicher ist im allgemeinem Sinn ein Datenspeicher. Er speichert normale Daten welche vom μP als Programmdaten oder Befehle interpretiert werden. Im physikalischen Sinne sind Datenspeicher integrierte Schaltungen, welche aus sich wiederholenden aneinander geketteten Speicherzellen bestehen.

Der Systemspeicher bezeichnet keinen Speicher zum Speichern großer Datenmengen. Zum Systemspeicher gehören auch die Register beliebiger SoC-Komponenten. Diese Register können sich beispielsweise in E/A Komponenten befinden, in Prozessoren oder sogar in Bus-Arbitern. Register werden zur schnellen Kommunikation zwischen den SoC-Komponenten

genutzt. Schreibt der μP ein Datum in das Register einer E/A-Komponente, beispielsweise in die Controllerkomponente für einen Monitor, so weiß die Controllerkomponente, dass sie den Monitor so ansteuern muss, damit er ein Zeichen anzeigt.
Als Systemspeicher wird der gesamte Systemadressraum bezeichnet, über den Zugriff auf alle Register und Datenspeicher erlangt werden kann.

Für den μP ist es somit kein Unterschied, ob er mit einem Datum ein Ergebnis in den Speicher schreiben will oder eine E/A-Komponente steuern möchte.
Für den μP eines SoC gilt, genauso wie für die CPU eines PC, dass er einen komplexen Befehlssatz besitzen kann, ein sogenannter CISC-Prozessor (**C**omplex **I**nstruction **S**et **C**omputer). Dagegen besitzt ein RISC-Prozessor (**R**educed **I**nstruction **S**et **C**omputer) einen kleinen, einfachen Befehlssatz.

Speicher und E/A-Komponenten sind ebenfalls SoC-Komponenten wie der μP. Sie sind über den Bus mit einander verbunden und können so Daten austauschen.
SoC-Desgins werden wie μC-Systeme gehandelt, können allerdings umfangreichere Ausmaße annehmen, vergleichbar mit einem normalen PC. Diese Flexibilität ist meist nur von Einstellungen abhängig, welche vor der Synthese der SoCs im VHDL-Code festgelegt werden müssen.

Eine Vereinfachung eines SoC kann sogar so weit gehen, dass sich in dem SoC kein konventioneller μP mehr befindet, sondern eine rudimentäre Steuereinheit. Für einfache μC reicht solch eine einfache Steuereinheit manchmal aus. Bei eingebetteten Systemen wird eine derartige Vereinfachung gezielt angestrebt, da eine einfache Steuereinheit erheblich weniger Chipfläche und elektrische Leistung benötigt. Außerdem kann eine kleine Steuereinheit viel schneller getaktet werden.
Aus dem Grund der Vereinfachung des μP zur Steuereinheit ist es gerechtfertigt den μP nicht mehr als zentrale Einheit eines SoC zu bezeichnen. An die Stelle der zentralen Einheit tritt in einem SoC oder μC das Bussystem.

2.1.2 Das Bussystem

Ein Bussystem besteht aus dem Bus und dem Bus-Arbiter. In den SoCs dieser Arbeit besteht der Bus aus den Daten- und Adressbus sowie anderen Steuerleitungen. Der Bus wird im wesentlichen durch die Verbindung der SoC-Komponenten bestimmt. Bei der technischen Umsetzung dieser Verbindungen wird von Verdrahtung gesprochen.
Der Bus-Arbiter ist eine kleine Regeleinheit, welche im wesentlichen angibt welche an den Bus angeschlossene SoC-Komponente mit Welcher kommunizieren darf. An das Bussystem eines SoC werden erhebliche Anforderungen gestellt.

Zu diesen Anforderungen gehören:

- Hohe Geschwindigkeit und Datendurchsatzrate von einer zur anderen SoC-Komponente
- Hohe Zuverlässigkeit der Kommunikation und entsprechend schnelle Reaktion beim Auftreten eines Fehlers, während der Kommunikation von SoC-Komponenten
- Zusätzliche Konfigurationsmöglichkeiten zur Festlegung der Daten- und Adressbusbreite sowie Festlegung der „Arbitrierungs-Strategie“[1]
- Hilfreich, bei der Steuerung eines SoC durch den μP, ist auch die Möglichkeit, die Konfigurationsinformationen der SoC-Komponenten über den Bus zugänglich zu machen. Dieser Prozess wird auch Plug and Play genannt.

Die Busse eines GRLIB LEON3-Systems sind der **A**dvanced **H**igh-performance **B**us (kurz AHB) und der **A**dvanced **P**eripheral **B**us (kurz APB). Während die Busse der Xilinx SoC-Komponenten Bibliothek, für den MicroBlaze-Prozessor, **L**ocal **M**emory **B**us (kurz LMB) und **O**n-Chip **P**eripheral **B**us (kurz OPB) genannt werden.

Bus-Kommunikation

Die Kommunikation der SoC-Komponenten über den Bus erfolgt nach einem ganz bestimmten Muster. Eine SoC-Komponente übergibt zur Kommunikation ein Datum an den Bus mit einer zugehörigen Adresse des Systemspeichers, an den das Datum geschrieben werden soll. Die Adresse kann dabei auch im Adressbereich eines Datenspeichers liegen.
Der Bus-Arbiter übernimmt die Daten- und Adress-Information, speichert sie in einem seiner eigenen Register und leitet eine Regelung auf dem Bus ein. Diese Regelung soll bewirken, dass die ursprünglichen Daten an ihrer Zieladresse ankommen und von einer anderen SoC-Komponente empfangen werden.
Das vorgeschriebene Muster der Kommunikation von SoC-Komponente zum Bus und umgekehrt wird als Protokoll bezeichnet. Grundlage der meisten Protokolle ist die Vier-Zyklus-Kommunikation (engl.: four-cycle handshake). Die SoC-Komponenten verfügen dabei nicht nur über eine Daten- und Adressleitung sondern zusätzlich über eine „Anfrage“ und „Bestätigungs“-Leitung.

[1] Unter den Begriff „Arbitrierungs-Strategie“ versteht man einen Entscheidungsalgorithmus, der gewährleistet, dass nach einer „fairen“ Wahrscheinlichkeitsverteilung alle vorhandenen SoC-Komponenten Zugriff auf den Bus erlangen können.

Die Durchführung zur Kommunikation ist folgendermaßen:

Schritt 1: *SoC-Komponente A aktiviert die „Anfrage“-Leitung, wenn sie Daten an eine Adresse senden will und die Daten zur Sendung am Bus bereit liegen.*

Schritt 2: *SoC-Komponente B aktiviert die „Bestätigungs“-Leitung, wenn sie bereit ist Daten anzunehmen.*

Schritt 3: *Ist die Datenübertragung abgeschlossen, d.h. SoC-Komponente B hat die Daten erfolgreich in ein eigenes Register geschrieben, so deaktiviert SoC-Komponente B die „Bestätigungs“-Leitung.*

Schritt 4: *Erkennt SoC-Komponente A die Deaktivierung der „Bestätigungs“-Leitung so deaktiviert SoC-Komponente A die „Anfrage“-Leitung.*

Danach ist der Bus wieder bereit neue Daten zu übertragen.

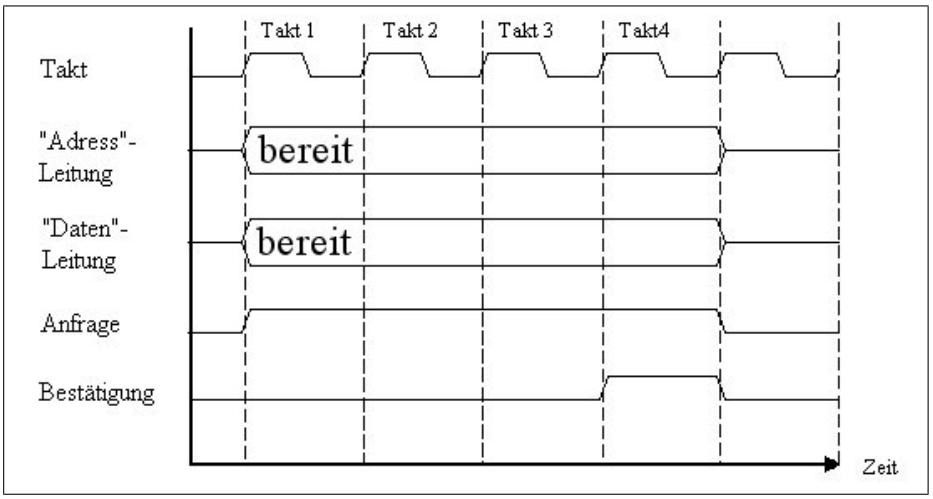

Abbildung 2.2: Timingdiagramm der Vier-Zyklus-Kommunikation

Abbildung 2.2 zeigt eine typische Vier-Zyklus-Kommunikation. Schritt 1 des obigen Kommunikationsmusters geschieht zu Anfang des Takt 1. Schritt 2 zu Beginn des Takt 4. Und die Schritte 3 und 4 geschehen am Ende von Takt 4.

Die gesamte Aufgabe eines Bussystems ist zusammenfassend zu beschreiben als das „Multiplexen“ von Leitungen zwischen den SoC-Komponenten. Ist das Multiplexen geschehen, so signalisiert das Bussystem an die SoC-Komponenten die Bereitschaft zur Datenübertragung.

2.1.3 Werkzeuge zur Erstellung von SoCs

Wie bereits erwähnt ist es nötig SoC-Komponenten in VHDL zu beschreiben. Eine weitere Möglichkeit wäre es die HDL „Verilog“ zu verwenden, was aber in dieser Arbeit nicht durchgeführt

wird, da die GRLIB in VHDL vorliegt.
Das grundlegende Element in VHDL, um Hardware zu beschreiben, ist die Entity (deut.: Entität). Die Entity ist ein VHDL-Konstrukt, das es ermöglicht Verhalten von Hardware modular zu beschreiben, um so eine übersichtliche Zusammensetzung eines SoC zu gewährleisten. Da ein SoC nun aus Entitys zusammen gesetzt wird und ein SoC aus SoC-Komponenten besteht, sind also Entitys mit SoC-Komponenten gleichzusetzen.
Beispielsweise ist der LEON3-Prozessor eine Entity. Dieser Prozessor hat Bestandteile wie die Memory Management Unit (kurz MMU) oder den Cache, welche ebenfalls Entitys sind. Alle Bestandteile eines SoC sind Entitys, neben dem Prozessor auch der Bus-Arbiter, und Schnittstellen zu Speichern. Selbst das SoC an sich, wird letztendlich in einer einzelnen Entity verpackt.

Eine Entity hat grundsätzlich folgenden Aufbau:

```
entity SoC is
  port (
    sys_clk_pin : in std_logic;
    sys_rst_pin : in std_logic;
    led1        : out std_logic;
    button1     : in std_logic
  );
end system;

architecture STRUCTURE of SoC is

signal clk : std_logic;
signal rst : std_logic;

begin

  clk <= sys_clk_pin;
  rst <= sys_rst_pin;
  .
  .
  Prozessor : p1
    port map ( CLK => clk, RST => rst, .. );

  ea_Komponente : ea1
    port map ( CLK => clk, RST => rst, LED => led1,
               Button1 => buton1 );

end architecture STRUCTURE;
```

Abbildung 2.3: Aufbau einer SoC-Entity

Abbildung 2.3 zeigt ein einfaches Beispiel einer SoC-Entity. Der rote Rahmen markiert die Port-Deklarationen der SoC Haupt-Entity. Der grüne Rahmen markiert den Aufbau der SoC-Entity. Blau sind alle Signale markiert. Diese können später den Bus beinhalten. Mit den Deklarationen im schwarzen Rahmen werden zusätzliche SoC-Komponenten (Entitys) in das SoC eingebunden und mit dem Konstrukt „port map“ verdrahtet.

Der Bus, der die Verdrahtung bezeichnet (optional im blauen Rahmen in Abbildung 2.3), ist keine Entity. Das hat die Ursache, dass Kommunikationsleitungen, welche jeweils ein Bit übertragen, in VHDL mit Signalen erzeugt werden. Diese Signale dienen sozusagen als Informationsquelle und Senke. Sie sind vergleichbar mit Variablen eines Softwareprogramms.
Wird ein Signal an einen Eingangsport einer Entity gelegt und bei einer anderen Entity an den Ausgangsport, so generiert die Synthesesoftware eine Leitung zur Übertragung eines logischen Pegels zwischen den beiden Ports.
Wird eine Sammlung von solchen Signalen betrachtet, von der einige Signale in alle SoC-Komponenten münden und einige Signale von den SoC-Komponenten beschrieben werden können, spricht man von einem Bus.

Die Firma Xilinx bietet zwei Möglichkeiten den VHDL-Code, also alle Entitys, zu synthetisieren und auf einem FPGA konfigurierbar zu machen.
Eine Möglichkeit ist es das Werkzeug Xilinx ISE zu verwenden. In diesem Werkzeug ist es dem Autor des VHDL-Codes überlassen, wie er sein System aufbaut. Der Autor schreibt die Entitys des SoC mit einer einfachen Editoroberfläche und ist selbst für die Verbindung der einzelnen Entitys verantwortlich. Es ist also nötig, dass der Autor seine Entitys auf **R**egister-**T**ransfer-Ebene (kurz RT-Ebene) im Verhalten beschreibt. Während des gesamten Entwicklungsprozesses des SoC durchläuft der Autor alle Ebenen des Hardwareentwurfs.
Die Synthese der SoC Haupt-Entity, welche alle Entitys als Untermodule beinhaltet und auch als System-Entity bezeichnet wird, erfolgt dann mit den von Xilinx vorgegebenen Synthesetools.

Eine andere Möglichkeit, ein SoC zu entwerfen, bietet das Werkzeug Xilinx-EDK. Hier beschreibt der Autor das SoC auf Systemebene. Das bedeutet der Autor übernimmt nur die Aufgabe der strukturellen Verknüpfung der SoC-Komponenten.
Im Hintergrund wird letztendlich nichts anderes vorgenommen als eine SoC Haupt-Entity zu erstellen. Diese Entity wird bei der Benutzung des EDK immer System-Entity genannt. In der erstellten System-Entity werden dann lediglich die direkten Unter-Entitys eingetragen und strukturell durch den Bus verbunden.
Da nun aber keine Verhaltensbeschreibung der Unter-Entitys, durch den Autor, notwendig ist wird eine Bibliothek benötigt aus denen die direkten Unter-Entitys, wie der μP, den Schnittstellen zu Speichern oder E/A-Komponenten, entnommen werden. Diese Bibliothek ist die schon erwähnte Xilinx-IP-Core-Bibliothek.
Prinzipiell sind die Arbeitsweisen von EDK und ISE identisch. Als Endprodukt entsteht eine Entity-Hierarchie aus VHDL-Dateien, welche dann in die Synthese gegeben wird. Die Synthese der nun erstellten System-Entity verläuft kaum unterschiedlich zur Synthese mit dem Werkzeug Xilinx ISE.

Die Benutzung der Xilinx Bibliothek erzeugt also eine erhebliche Zeitersparnis bei der Erstellung von SoC-Lösungen. Die Verhaltensbeschreibungen der einzelnen SoC-Komponenten sind durch die einzelnen IP-Cores vorgegeben.

2.2 Das Werkzeug: Embedded Development Kit (EDK)

In diesem Kapitelpunkt werden Betrachtungen durchgeführt welche an einem unverändertem Xilinx **E**mbedded **D**evelopment **K**it (kurz EDK) zu beobachten sind. Es soll also der Zustand des EDK nach einer Installation beschrieben werden, um die Ausgangssituation zu verstehen. Hier werden also noch keine Änderungen durch das Einbinden der GRLIB diskutiert, sondern nur Ansätze gezeigt.

Die Software Xilinx-EDK arbeitet mit einfachen VHDL-Entitys, welche im Hintergrund miteinander verbunden werden. Dieser Vorgang ist allerdings für den Benutzer nicht direkt sichtbar. Die SoC-Entitys der System-Entity, werden zusätzlich in Wrapper (engl.: Verpackung) Entitys eingebettet. Wrapper Entitys sind einhüllende VHDL-Entitys. In ihnen werden die Generic Informationen der IP-Cores fest eingeschrieben, so dass sie zur Synthese bereitstehen.

Zur Erstellung eines SoC werden die nötigen IP-Cores aus dem „IP Catalog“ in das Projekt aufgenommen. Dies kann in einem einfachen Fall ein MicroBlaze-Prozessor, ein OPB Bus, der LMB Bus mit **B**lock-**RAM** (kurz BRAM) und ein GPIO (Schnittstelle zu LEDs und Buttons) sein. Der MicroBlaze-Prozessor würde dann, nach der Synthese, ein einfaches Programm aus dem BRAM über den LMB auslesen. Dieses Programm würde den MicroBlaze-Prozessor beispielsweise anweisen einen Zähler auf den GPIO LEDs, über den OPB Bus, auszugeben.

Seit EDK Version 8.2i ist die **„Bus Interface“-Ansicht** vorteilhaft überarbeitet worden. Dem Benutzer wird nun neben einer Auflistung aller SoC-Komponenten zeitgleich eine Übersicht der entstehenden Busstruktur dargestellt.

Die **„Ports“-Ansicht** stellt alle nach außen sichtbaren Ports einer Entity, also einer SoC-Komponente, dar. Hier kann der Benutzer Entity-Ports mit physikalischen Pins des FPGAs verbinden. Diese Pins sind dann auf dem FPGA-Board mit den LEDs verbunden oder werden als Steuersignale an einen anderen „Hard-Controller“, wie beispielsweise den Ethernet MAC Controller, geleitet. Als „Hard-Controller“ wird hier eine Funktionseinheit von Integrierten Schaltkreisen (kurz ICs) bezeichnet. Dieser Hard-Controller übernimmt dann die detaillierte Ansteuerung der Schnittstelle. Dazu kann zum Beispiel eine Digital-Analog-Wandlung nötig sein.

In der **„Addresses“-Ansicht** hat der EDK Benutzer die Möglichkeit die Systemadressen der einzelnen SoC-Komponenten festzulegen. Schreibt oder liest der μP, während der Programmausführung, an solch eine Adresse, kommuniziert der μP mit der jeweiligen SoC-Komponente.

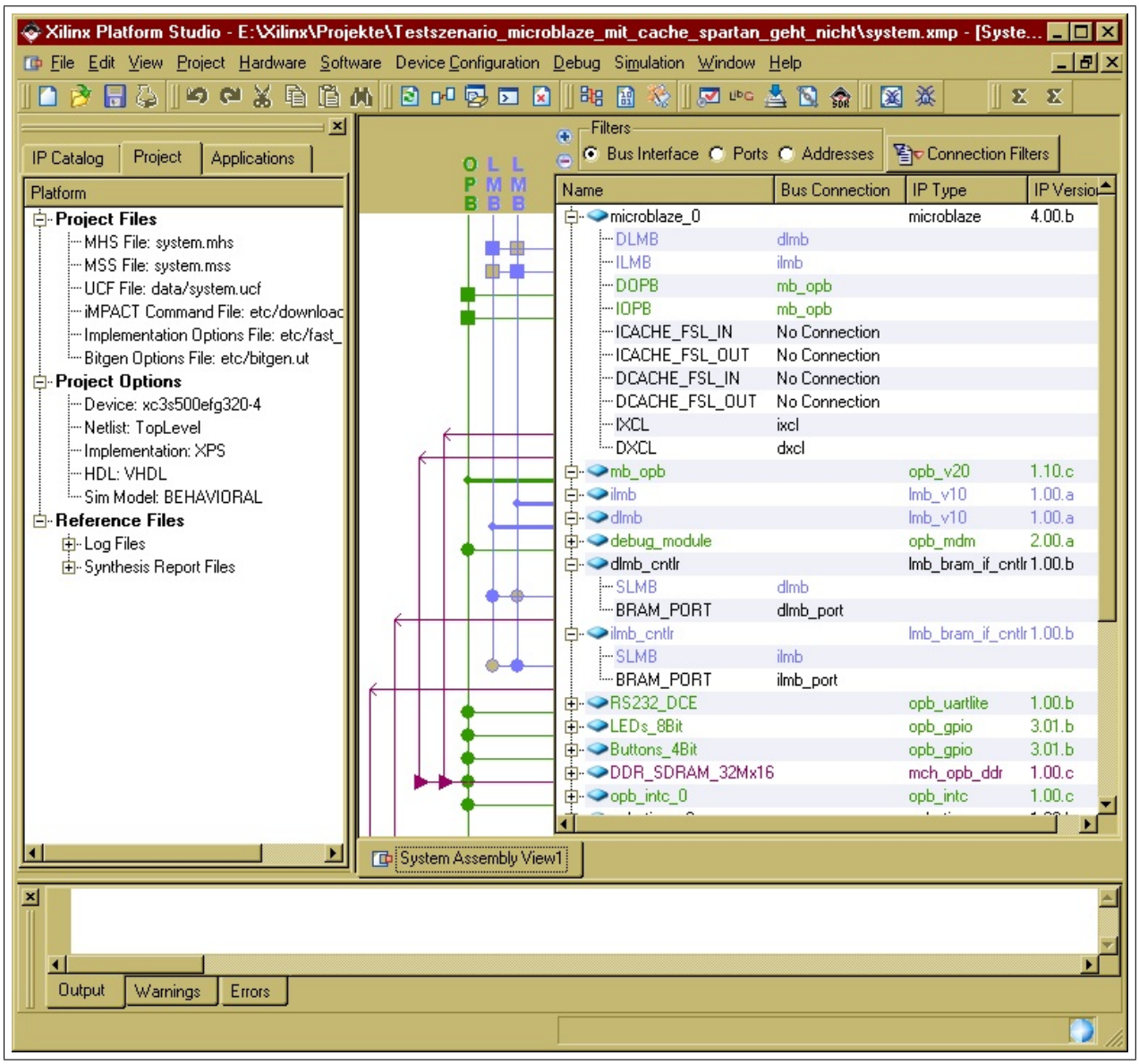

Abbildung 2.4: „Bus Interface“-Ansicht des EDK 8.2i

Abbildung 2.4 zeigt die „Bus Interface“-Ansicht. Rechts ist die Auflistung der SoC IP-Cores zu erkennen und in der Mitte die daraus resultierende Busstruktur.

Klassifikation der IP-Cores

In Kapitelpunkt 2.1 wurde in dieser Arbeit eine Klassifikation der SoC-Komponenten eingeführt. Es wurde grob zwischen Prozessoren, Bussen, Speichern und Schnittstellen unterschieden. Dies ist eine Klassifikation nach Arbeitsweisen oder Verhalten von Entitys. Durch einen Blick auf die „Bus Interface“-Ansicht ist es nun an der Zeit eine praktischere Klassifikation einzuführen.

Ab jetzt werden nur Entitys unterschieden, welche Master oder Slave auf einem Bus sind. Vorab sei noch einmal gesagt, dass alle SoC-Komponenten Entitys im Sinne von VHDL sind.

Bekannte Entitys werden nun wie folgt klassifiziert:

- Ein Prozessor ist eine Entity, welche meist Master auf einem Bus ist.
- Speicher und Schnittstellen sind Entitys, die meist Slaves auf einem Bus sind.
- Ein Bus ist eine Entity, die weder Master noch Slave auf einem Bus ist.

Weiter sei gesagt, dass es von nun an nicht mehr wichtig ist die Bezeichnungen Prozessor, Bus usw. zu verwenden. Alle SoC-Komponenten sind Entitys. Wenn allerdings von der Arbeitsweise oder dem Verhalten der Entitys gesprochen wird, ist es wieder hilfreich auf die alten Bezeichnungen zurückzugreifen.

Die oben genannten Ansichten („Bus Inteface“, „Ports“, „Addresses“), des EDK, ermöglichen es in kurzer Zeit ein SoC benutzerfreundlich zu erstellen. Die Vorteile dieser Ansichten sollen später auch für die GRLIB zur Verfügung stehen.
Eine Ausnahme bildet die „Addresses“-Ansicht. Aus Gründen, welche in Kapitel 5 erörtert werden, war es nicht möglich diese Ansicht für die GRLIB nutzbar zu machen. Die „Addresses“-Ansicht wird also in Zukunft nur für Xilinx-IP-Cores nützlich sein und für GRLIB-IP-Cores verworfen werden.

2.2.1 Arbeitsplan der Software EDK

In diesem Kapitelpunkt soll der Arbeitsplan des EDK genauer erläutert werden. Dies ist zwingend notwendig, um zu verstehen, wo sich alle Ansätze zum Einbinden der GRLIB befinden.
Außerdem sind beim Einbinden der GRLIB Unterschiede im herkömmlichen Workflow entstanden. Mit Unterschieden im Workflow sind Handlungen gemeint, welche ausgeführt werden müssen, um ein GRLIB LEON3-SoC erfolgreich zu synthetisieren. Diese Handlungen sind bei der Arbeit mit den Xilinx-IP-Cores nicht notwendig.

Als Ziel der Hardwaresynthese und Kompilierung der Software steht ein Bitstream, benannt mit „download.bit“. Dieser befindet sich im „[EDK_PROJECT]/implementation“[2] Verzeichnis. Dieser Bitstream beinhaltet die FPGA-Daten zur Realisierung der synthetisierten Hardware und die BRAM-Daten. Die BRAM-Daten werden ebenfalls bei der Bitstream-Konfiguration geschrieben, da sich in ihnen das kompilierte native C-Programm befindet. Der Bitstream „system.bit“, im selben Verzeichnis, beinhaltet nur die Konfigurationsdaten für den FPGA ohne BRAM-Daten.

[2] Mit dem Ausdruck [EDK_PROJECT] wird das Verzeichnis bezeichnet in dem sich das gespeicherte EDK-Projekt befindet.

Zur Erstellung des „download.bit“ Bitstreams muss der EDK-Arbeitsplan erfolgreich abgearbeitet werden. Der EDK-Arbeitsplan wird in vier Arbeitsschritten ausgeführt.

Diese Arbeitsschritte sind:

Arbeitsschritt 1: *Erstellung einer **B**RAM **M**emory **M**ap (kurz BMM) Datei und Abbilden der ausgewählten SoC IP-Cores auf eine VHDL-Code Hierarchie.*

Arbeitsschritt 2: *Das Synthetisieren der SoC IP-Cores sowie der System-Entity. Bei diesem Vorgang werden die Hardware Netzlisten (engl.: Netlists) generiert. Dies geschieht durch die Xilinx Synthesetools (Xst, Place, Route, Trace, Bitgen).*
Als Ergebnis steht der Bitstream „system.bit“.

Arbeitsschritt 3: *Das Kompilieren des nativen C-Programms mit einem, für den Prozessor, gelieferten Compiler. Dies geschieht durch die Xilinx Tools (Libgen, Compiler, Linker).*
*Als Ergebnis steht ein **E**xecutable and **L**inking **F**ormat (kurz ELF), die ELF-Datei des nativen C-Programms.*

Arbeitsschritt 4: *Verbinden der System Netlist mit der ELF-Datei zum vollständigen Bitstream „download.bit“. Dies geschieht durch die Xilinx Tools (Bitinit, Data2Mem)*

Alle 4 Arbeitsschritte laufen in kleinen unterteilten Schritten ab, welche nach dem Konzept der MAKEFILE in der Datei „system.make“ aufgelistet sind. Die Datei „system.make“ befindet sich im „[EDK_PROJECT]“-Verzeichnis. Sie bindet die Datei „system_incl.make“ ein, welche nützliche Informationen über den zu benutzenden Compiler und Benennungen von Verzeichnissen enthält. Damit der EDK-Arbeitsplan für die GRLIB fehlerfrei durchlaufen kann, war ist es nötig Änderungen an der Datei „system.make“ vor zu nehmen.

2.2.1.1 Arbeitsschritt 1 (VHDL-Code der System-Entity generieren)

In Arbeitsschritt 1 werden allgemeine Vorbereitungen getroffen, die der Synthese der System-Entity dienen. Die System-Entity wird nach dem Projektnamen benannt. In dieser Arbeit soll sie mit „system.vhd“ bezeichnet werden. Alle VHDL-Code Dateien mit den Endungen „.vhd“, von denen in Arbeitsschritt 1 gesprochen wird, werden im Verzeichnis „[EDK_PROJECT]/hdl“ gespeichert.

Es werden folgende Operationen ausgeführt:

1. Es wird eine BMM-Datei erstellt. Sie bildet die Daten des kompilierten nativen C-Programms auf den Systemspeicher ab.
2. Für jede SoC-Komponente wird eine Wrapper-Entity erstellt. Diese Wrapper-Entitys binden als Unter-Entitys die Entitys der SoC-Komponenten ein.
3. Es wird eine System-Entity generiert, welche alle Wrapper-Entitys zu einem SoC verbindet.

Operation 1: BMM-Datei

Die BMM-Datei namens „system.bmm“, welche sich im „[EDK_PROJECT]/implementation“-Verzeichnis befindet wird zu Beginn generiert. Die BMM-Datei enthält eine syntaktische Beschreibung in welcher Weise einzelne BRAMs zu einem zusammenhängenden, logischen Datenraum verbunden werden. Die BMM wird vom EDK benötigt, um ein natives C-Programm in die BRAMs zu laden.

```
ADDRESS_MAP microblaze_0 MICROBLAZE 100
  ADDRESS_SPACE lmb_bram_combined COMBINED [0x00000000:0x00001fff]
    ADDRESS_RANGE RAMB16
    BUS_BLOCK
      lmb_bram/lmb_bram/ramb16_s9_s9_0 [31:24] ;
      lmb_bram/lmb_bram/ramb16_s9_s9_1 [23:16] ;
      lmb_bram/lmb_bram/ramb16_s9_s9_2 [15:8] ;
      lmb_bram/lmb_bram/ramb16_s9_s9_3 [7:0] ;
    END_BUS_BLOCK;
    END_ADDRESS_RANGE;
  END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

Abbildung 2.5: Inhalt der BMM eines MicroBlaze-Systems

Ein BRAM hat eine Kapazität von 2 KByte. An der Abbildung 2.5 der BMM-Datei ist zu erkennen, dass, für das native C-Programm, ein Speicher von 8 KByte verwendet werden soll. Dies entspricht einem Adressraum von 0x1FFF Bytes. Insgesamt werden also 4 BRAMs benötigt.
Bei der Einbindung der GRLIB musste sichergestellt werden, dass diese BMM-Datei ebenfalls beim Start des Arbeitsschritts 1 erstellt wird. Dies führte zu dem „BMM-Problem“, welches näher in Kapitelpunkt 5.3.3 beschrieben wird.

Operation 2: Erstellen der Wrapper Entiäten

Die Wrapper-Entitys haben prinzipiell eine Benennung nach dem Muster: „[name]_wrapper.vhd“[3]. Bei der Konfiguration der SoC-Komponenten mittels der EDK GUI, werden die sogenannten Generics festgelegt. Generics sind für eine Entity festgelegte Konstanten mit denen es möglich ist Einstellungen vorzunehmen. In dieser Arbeit wird die Bezeichnung Generic-Konstanten verwendet. Die mit der EDK-GUI festgelegten Generic-Konstanten werden in die Wrapper-Entitys eingebettet und versorgen so die Haupt-Entity eines IP-Cores mit den korrekten Werten.

```
entity leds_4bit_wrapper is
  port (
    OPB_ABus : in std_logic_vector(0 to 31);
    OPB_BE : in std_logic_vector(0 to 3);
    OPB_Clk : in std_logic;
    ..
    GPIO_d_out : out std_logic_vector(0 to 3);
    ..
  );
end leds_4bit_wrapper;

architecture STRUCTURE of leds_4bit_wrapper is

begin
  leds_4bit : opb_gpio
    generic map (
      C_BASEADDR => X"40000000",
      C_HIGHADDR => X"4000ffff",
      C_OPB_AWIDTH => 32,
      C_OPB_DWIDTH => 32,
      C_FAMILY => "virtex2p"
      )
    port map (
      OPB_ABus => OPB_ABus,
      OPB_BE => OPB_BE,
      OPB_Clk => OPB_Clk,
      ..
      GPIO2_d_out => GPIO2_d_out,
      ..
    );
end architecture STRUCTURE;
```

Abbildung 2.6: Beispielausschnitt einer Wrapper-Entity

Abbildung 2.6 zeigt einen Ausschnitt einer typischen automtisch generierten Wrapper-Entity. Der Rote Rahmen markiert den Kernpunkt, die Festlegung der Generic-Konstanten.

Eine bekannte Generic-Konstante heißt „C_FAMILY“. Sie wird durch das EDK automatisch mit einem String gefüllt, der dem verwendeten FPGA-Typ entspricht. Diese Konstante kann dann innerhalb der Entity ausgewertet werden und dadurch das Verhalten der Entity beeinflussen.
Aber nicht alle Generic-Konstanten werden automatisch gefüllt. Viele müssen auch mit „Configure IP ...“[4] festgelegt werden.

[3] Der Ausdruck [name] steht für den Namen der SoC-Komponente

[4] Das „Configure IP ...“-Dialogfeld kann im „IP-Coremenü“ aufgerufen werden. Das „IP-Coremenü“ wiederum erscheint durch Rechtsklicken auf die SoC-Komponente.

Auch die SoC-Komponenten der GRLIB besitzen Generic-Konstanten, welche vor der Synthese festgelegt werden müssen. Dieser Vorgang erfordert, bei der GRLIB, erheblich mehr Kenntnisse als bei der Benutzung der Xilinx-IP-Cores.

Operation 3: Erstellen der System-Entity

Nachdem die Wrapper-Entitys generiert wurden, wird die System-Entity „system.vhd“ erstellt. Sie bindet nun alle SoC-Wrapper-Entitys zu einer Entity zusammen. Abbildung 2.3 aus Kapitelpunkt 2.1.3 zeigt einen äquivalenten Aufbau einer System-Entity. Der Unterschied zur „system.vhd“ ist das im roten Rahmen die Entity „system“ genannt wird und im schwarzen Rahmen die eingebundenen Entitys „[name]_wrapper“ genannt werden. Die beträchtliche Menge an Signalen, welche zur Kommunikation zwischen den SoC-Komponenten verwendet werden, bilden bei der Synthese den Bus.

2.2.1.2 Arbeitsschritt 2 (Hardware Synthetisieren)

Damit als Ergebnis ein Bitstream, zur Konfiguration eines FPGA entstehen kann, sind mehrere Operationen nötig:

1. Synthese der SoC-Komponenten mit dem **X**ilinx **S**ynthese **T**ool (kurz XST)
2. Platzieren der synthetisierten System Netlist auf dem FPGA mit dem Par Tool
3. Routen noch nicht verdrahteter Ports mit dem Par Tool
4. Verifizieren der vorgegebenen Timing, Placing oder Routing Constraints mit dem Trace Tool
5. Erstellen eines System-Bitstreams („system.bit“) mit dem Bitgen Tool

Operation 1: Xilinx Synthese Tool (kurz XST)

Hier findet die eigentliche Synthese der SoC-Komponenten statt. Es wird für jede SoC-Komponente die zugehörige Wrapper-Entity einzeln synthetisiert. Dies geschieht unabhängig von der System-Entity. Bei diesem Vorgang werden NGC-Dateien erstellt, welche im „[EDK_PROJECT]/implementation“-Verzeichnis abgespeichert werden. Die NGC-Dateien sind Hardware Netlists, die das Logische Design in Verbindung mit Constraints für jede einzelne SoC-Komponente enthalten.
Wurde jede SoC-Komponente synthetisiert, erfolgt die Synthese der System-Entity unter Verwendung der NGC-Dateien der SoC-Komponenten.

Dieser Vorgang muss auch mit den IP-Cores der GRLIB möglich sein.

Operation 2: Platzieren (engl.: Placing) mit dem Par Tool

Nachdem die gesamte Netlist des Systems synthetisiert wurde, muss sie auf dem FPGA platziert werden. Dies wird natürlich nur fiktiv erledigt. Dem EDK muss beim Placing bekannt sein, um welchen FPGA es sich genau handelt auf den der entstehende Bitstream geladen werden soll. Das Placing geschieht unter Berücksichtigung spezieller Constraints. Mit diesen Constraints können Timing-Vorgaben berücksichtigt werden. Die Constraints werden in einer Datei namens „system.ucf“ im „[EDK_PROJECT]/data“-Verzeichnis deklariert.
Auf diesen Prozess wurde, durch das Einbinden der GRLIB, kein Einfluss genommen, da nach der Synthese keine Schwierigkeiten mehr aufgetreten sind.

Operation 3: Routen (engl.: Routing) mit dem Par Tool

Wenn das Placing abgeschlossen ist werden noch nicht verdrahtete Ports durch das Routing mit den Pins des FPGA und den Ports anderer SoC-Komponenten verbunden. Auch das Routen erfolgt unter der Berücksichtigung spezieller Constraints. Hier wurde ebenfalls kein Einfluss genommen.

Operation 4: Verfolgen (engl.: Tracing) mit den Trace Tool

Nachdem das Placing und das Routing erfolgreich beendet wurden, werden mit dem Trace Tool die Constraints des Designs verifiziert.
Das Tracing ist vor allem wichtig für timingempfindliche SoC-Komponenten, wie der OPB Ethernet Core oder der OPP DDR RAM Core. Aber auch die GRLIB Ethernet und DDR RAM Cores benötigen spezifische Constraints.

Operation 5: Bitstream Generieren mit dem Bitgen Tool

Aus dem bis hierher gefertigtem Design wird jetzt der Bitstream („system.bit“) generiert. Dieser Bitstream kann mit der Software Xilinx iMPACT auf einen FPGA konfiguriert werden.
Der „system.bit“ Bitstream enthält zu diesem Zeitpunkt noch keine BRAM Daten, so dass kein natives C-Programm auf dem SoC ausgeführt werden kann.

2.2.1.3 Arbeitsschritt 3 (Software Kompilieren)

Zum Kompilieren der naiven C-Programme, welche im EDK Tab „Applications“ aufgelistet sind, werden auch bestimmte Einsprungspunkte in der Datei „system.make“ benutzt. Es sind mehrere

Operationen nötig um aus den C-Dateien eine ELF-Datei zu generieren, die mit der „system.bit“ zur „download.bit“ verschmolzen werden kann.

Diese Operationen sind:

1. Kopieren und Kompilieren der Programm Bibliotheken mit dem Libgen Tool. Diese Bibliotheken werden benötigt, um das „Application“-Projekt (native C-Programm) zu kompilieren
2. Kompilieren des eigentlichen nativen C-Programms mit dem Prozessor Compiler
3. Verlinken der entstandenen ELF-Datei

Operation 1: Bibliotheken kompilieren

Xilinx liefert für seine SoC-Komponenten eine beträchtliche Anzahl von Treiberdateien. Auch für den MicroBlaze-Prozessor gibt es Definitionsdateien. Diese C-Dateien werden kompiliert, um dann später bei der Kompilierung des nativen C-Programms verwendet werden zu können.
Das Kompilieren von Treiberdateien ist für die GRLIB und damit für den LEON3 nicht notwendig. Wie im Kapitel 6 ersichtlich werden wird, sind alle Bibliotheken für das GRLIB-System vorkompiliert und umfangreiche Treiber, wie die der Xilinx Bibliothek, existieren nicht.
Aus diesem Grund wurde die „system.make“ für ein GRLIB Design an diesen Stellen auskommentiert.

Operation 2: Kompilieren mit dem Compiler Tool

Das Kompilieren des nativen C-Programms erfolgt prozessorspezifisch. Es wird eine ELF-Datei erstellt, welche nur von dem zugehörigen Prozessor interpretiert werden kann.
Für den MicroBlaze-Prozessor ist dies beispielsweise ein Ausruf der folgenden Art:

„`$ mb-gcc -O2 TestApp_Peripheral.c -o TestApp_Peripheral/executable.elf`“

Es ist zu erkennen, dass der Compiler „`mb-gcc`“ verwendet wurde. Dies ist ein GNU C-Compiler für den MicroBlaze-Prozessor.

Für ein LEON3-System wurde die „system.make“ an dieser Stelle ersetzt durch:

„`$ sparc-elf-gcc -O2 -msoft-float -c -g TestApp_Peripheral.c -o TestApp_Peripheral/prom.exe`“

Operation 3: Verlinken mit dem Linker Tool

Der Linker hat die Aufgabe, alle kompilierten Programmteile zu einem gemeinsamen Programm zu verbinden. Außerdem hat er die weiter Aufgabe alle Programmsektionen, der ELF-Datei, auf eine bestimmte Speicheradresse abzubilden.
Das ELF besteht aus mehren Programmsektionen, wie die „text“, „data“, „bss“, „dec“ und „hex“ Sektionen. Diese Programmsektionen müssen, vor Ausführung des Programms durch den Prozessor, an den richtigen Adressen im Systemspeicher abgelegt werden. Üblicherweise sollte sich an den ausgewählten Adressen des Systemspeichers ein beschreibbarer Datenspeicher befinden.

Zum korrekten Ablegen der Programmsektionen im Systemspeicher, müssen die Programmsektionen auf die richtigen Stellen im Speicher verlinkt werden. Für das LEON3-System wäre dies mit dem folgenden Aufruf zu bewerkstelligen:

„`$ sparc-elf-ld -T linkprom.ld TestApp_Peripheral/prom.exe -o TestApp_Peripheral/executable.elf`“

Als zu verlinkende Datei wird die ELF-Datei „prom.exe“ verwendet, welche das Ergebnis der Kompilierung (Operation 2) ist. Das übergebene Linkerskript („linkprom.ld“) beinhaltet eine Zuordnung der Programmsektionen auf bestimmte Adressen im Systemspeicher.
Als Ergebnis dieser Operation entsteht die gelinkte ELF-Datei „executable.elf“, welche im Arbeitsschritt 4 benötigt wird.

Eine genaue Behandlung erfolgt wieder im Anhang A.2.1.

2.2.1.4 Arbeitsschritt 4 (Hard- und Software zum Bitstream verbinden)

Für den letzten Arbeitsschritt, zur Erstellung der „download.bit“, werden folgende Dateien verwendet:

- Die „system.bit“, die nach Arbetisschritt 2 die Konfigurationsdaten für die Hardware enthält.
- Die „executable.elf“, die nach Arbeitsschritt 3 den kompilierten, verlinkten Binärcode des nativen C-Programms enthält.
- Die „system.bmm“, welche nach Arbeitsschritt 1 Informationen enthält, wie und wo sich der Programmspeicher der „system.bit“ befindet.

Mit den Tools Bitinit und Date2Mem wird nun eine Verschmelzung der oben genannten Dateien zur „download.bit“ durchgeführt.
An diesem Arbeitsschritt, mussten für die GRLIB keine Änderungen durchgeführt werden. Es war

aber sicherzustellen, dass die oben genannten Dateien, nach Ausführung der Arbeitsschritte 1-3, generiert wurden.

2.2.2 EDK-Konstrukte für IP-Cores

In Kapitelpunkt 2.2.1 wurde der EDK-Arbeitsplan vorgestellt, der durchgeführt werden muss, um aus den Xilinx-IP-Cores ein konfigurationsbereites SoC zu erstellen. Es ist also nun zu Erklären wie die Xilinx-EDK IP-Cores aufbereitet sind, damit sie dem EDK zur Benutzung bereitstehen.

Um die IP-Cores, wie zum Beispiel den IP-Core „opb_uartlite“ in der Gruppe „Communication Low-Speed“, im EDK benutzt zu können, hat sie Firma Xilinx einige Konstrukte eingeführt.
Diese Konstrukte sind hauptsächlich Dateitypen, die jeweils spezifische Informationen über einen einzelnen IP-Core enthalten. Jeder Dateityp hat eine Syntax, die eingehalten werden muss, damit das Xilinx-EDK den IP-Core korrekt erkennt und benutzen kann.

Zur Übersicht sind alle Konstrukte, mit kurzen Erläuterungen, in der folgenden Tabelle 2.1 angegeben. Genaue Erklärungen erfolgen in den darauffolgenden Kapitelpunkten. Für die vollständige Aufzählung, aller Elemente der EDK-Konstrukte, wird auf [platrefdoc] verwiesen. In dieser Xilinx-Dokumentationen werden alle EDK-Konstrukte ausführlich erklärt.

EDK-Konstrukt Bezeichnung	Beschreibung
Microprocessor **H**ardware **S**pecification (kurz MHS)	Die MHS wird in der Datei „system.mhs“ gespeichert. Sie enthält eine Auflistung aller externen Ports und IP-Cores des SoC. Aus der MHS wird die System-Entity erstellt.
Microprocessor **P**eripheral **D**efinition (kurz MPD)	Durch die MPD wird eine Definition eines IP-Cores vorgenommen. Es kann beispielsweise definiert werden, ob der IP-Core Master oder Slave sein soll und auf welchem Bus dies gilt. Es werden auch Syntheseoptionen für den IP-Core festgelegt.
Peripheral **A**nalyze **O**rder (kurz PAO)	Das PAO Konstrukt beinhaltet eine Auflistung aller Hardwarebibliothekdateien, die zur Synthese des IP-Cores benötig werden. Die Hardwarebibliothekdateien werden vor der Synthese der IP-Core Entity analysiert und können dann in den IP-Core eingebunden werden.
Microprozessor **S**oftware **S**pecification (kurz MSS)	Die MSS, genannt „system.mss“, wird benutzt, um alle Betriebsysteme, Treiber und Softwarebibliotheken für alle IP-Cores fest zu legen.
Microprocessor **D**river **D**efinition (kurz MDD)	Das MDD wird benutzt, um zusätzliche Bibliothekdefinitionen, speziell für Prozessoren, vor zu nehmen.
Tool **C**ommand **L**anguage (kurz Tcl)	Die Tcl ist ein Konstrukt, was in Tcl-Dateien zu finden ist. Es ist eine Skript-Sprache, welche benutzt wird, um Generics-Konstanten und Einträge der MHS zu prüfen.

Tabelle 2.1: Übersicht aller, für diese Arbeit relevanten, EDK-Konstrukte [platrefdoc]

Zum besseren Verständnis werden die Erläuterungen der EDK-Konstrukte am konkreten Beispiel des „opb_uartlite“ IP-Cores stattfinden.

2.2.2.1 Das IP-Core Verzeichnis

Bevor alle EDK-Konstrukte, zur Definition eines IP-Cores, erklärt werden, soll eine Übersicht der Verzeichnisse aller EDK-Konstrukte gegeben werden. Dies verdeutlicht den praktischen Einsatz der EDK-Konstrukte und stellt sie in einen besseren Zusammenhang.

Die MHS, mit der Datei „system.mhs“ und die MSS, mit der Datei „system.mss“, werden während des Hinzufügens, von IP-Cores in das SoC, aktualisiert. Sie werden im Verzeichnis „[EDK_PROJECT]“ gespeichert.

Die IP-Cores der Xilinx Bibliothek werden im Verzeichnis „[XILINX_EDK]/hw/XilinxProcessorIPLib/pcores“[5] gepeichert. Die Verzeichnisstruktur eines IP-Cores, für das EDK, wird konkret am Beispiel des „opb_uartlite“ IP-Cores vorgestellt.

IP-Core Verzeichnis

Ein IP-Core Verzeichnis hat eine Benennung nach dem Muster „[name_ip]_v?_??_?“[6].

Unterverzeichnisse

Ein Unterverzeichnis eines IP-Core-Verzeichnisses muss das „**data**“-Verzeichnis sein. Es muss die Konstrukte MPD und PAO mit ihren Dateien enthalten. Das Tcl Konstrukt ist nicht notwendig, kann aber optional dem IP-Core hier hinzugefügt werden. Die Dateinamen der Konstrukte müssen nach dem Muster „[name_ip]_v2_1_0.(mpd|pao|tcl)“ konstruiert sein.

Ein weiteres Unterverzeichnis kann das „**doc**“-Verzeichnis sein. Es beinhaltet Dokumentationen wie das Datasheet eines IP-Cores. Wird eine Dokumentationsdatei nach dem Muster „[name_ip].pdf“ benannt, so kann sie mit dem Menüpunkt „View PDF Datasheet“, in der EDK GUI, geöffnet werden. Der Menüpunkt „View PDF Datasheet“ erscheint mit einem Rechtsklick auf die zugehörigen SoC-Komponente in der „Bus Interface“-Ansicht.

Das letzte Unterverzeichnis muss das „**hdl**“-Verzeichnis sein. Es muss entweder das „vhdl“-Verzeichnis oder „verilog“-Verzeichnis beinhalten. Je nachdem, ob die Verhaltensbeschreibung,

[5] Der Ausdruck [XILINX_EDK] bezeichnet das Installationsverzeichnis der Software Xilinx-EDK auf dem Host PC.

[6] Der Ausdruck [name_ip] steht für die Bezeichnung eines IP-Cores und die vier „?“ stehen für Version, Subversion und Subsubversion. Version und Subversion müssen Ziffern sein und die Subsubversion ein Buchstabe.

des IP-Cores, in VHDL oder Verilog vorliegt, so müssen sich die HDL-Dateien der Verhaltensbeschreibung im entsprechenden Verzeichnis befinden.

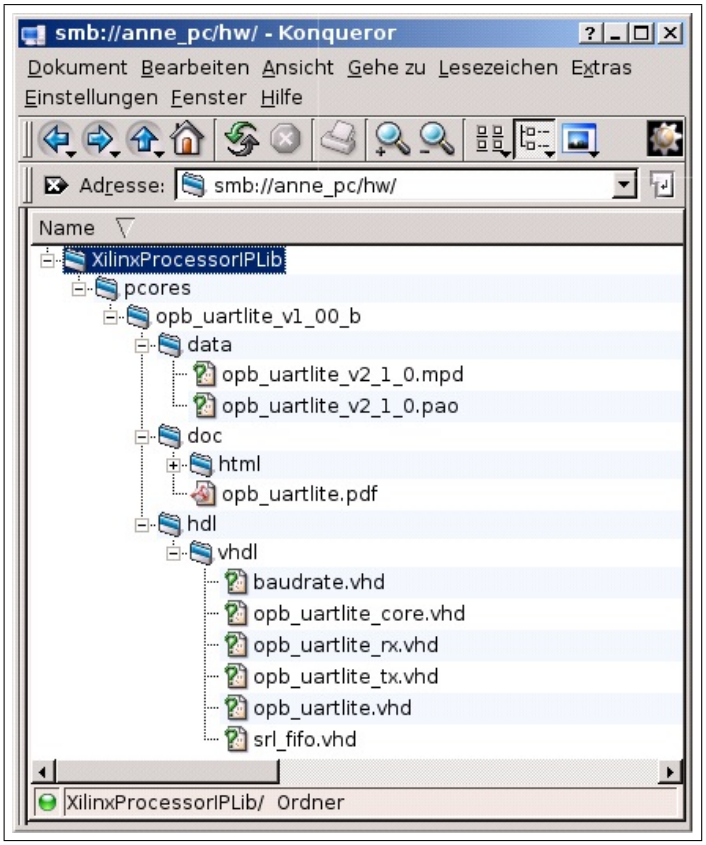

Abbildung 2.7: IP-Core Verzeichnis der OPB Uart Lite

Abbildung 2.7 zeigt die Minimalbesetzung des „opb_uartlite“ IP-Cores Verzeichnisses. Es ist zu erkennen, dass sich im „hdl“-Verzeichnis mehrere IP-Core relevante VHD-Dateien befinden.

2.2.2.2 Microprocessor Hardware Specification (MHS)

Die **M**icroprocessor **H**ardware **S**pecification (kurz MHS) ist eine Zusammenfassung des ganzen, im EDK erstellten, SoC. Die Datei „system.mhs“, welche die MHS enthält wird im Verzeichnis „[EDK_PROJECT]“ gespeichert.

Wird ein IP-Core in das SoC aufgenommen, so wird er automatisch in die MHS eingetragen. Außerdem werden alle Ports der System-Entity mit Signalen assoziiert, die dann in der System-Entity verwendet werden können.

Der Inhalt der MHS wird durch die Informationen bestimmt, die durch die Ansichten des EDK („Bus Inteface“-, „Ports“-, „Addresses“-Ansicht) verändert werden können. Umgekehrt führt eine Bearbeitung der MHS-Datei zu einer Änderung der EDK Ansichten.

```
BEGIN microblaze

        PARAMETER INSTANCE = microblaze_0
        PARAMETER HW_VER = 5.00.c
        BUS_INTERFACE DLMB = dlmb
        BUS_INTERFACE ILMB = ilmb
        BUS_INTERFACE DOPB = mb_opb
        BUS_INTERFACE IOPB = mb_opb
        PORT DBG_CAPTURE = DBG_CAPTURE_s

END

BEGIN opb_uartlite

        PARAMETER INSTANCE = RS232_Uart_1
        PARAMETER HW_VER = 1.00.b
        PARAMETER C_BAUDRATE = 9600
        PARAMETER C_BASEADDR = 0x40600000
        PARAMETER C_HIGHADDR = 0x4060ffff
        BUS_INTERFACE SOPB = mb_opb
        PORT RX = fpga_0_RS232_Uart_1_RX
        PORT TX = fpga_0_RS232_Uart_1_TX

END
```

Abbildung 2.8: Ausschnitt einer MHS

Abbildung 2.8 zeigt eine derzeitige Zusammensetzung eines SoC. Es wird der IP-Core des MicroBlaze-Prozessors und der IP-Core „opb_gpio“ verwendet.

Für jeden IP-Core sind folgende Elemente setzbar:

ELEMENT	**Beispiel**
PARAMETER	„PARAMETER C_BAUDRATE = 9600“
BUS_INTERFACE	„BUS_INTERFACE SOPB = mb_opb“
PORT	„PORT RX = fpga_0_RS232_Uart_1_RX“

Tabelle 2.2: Elemente der MHS

PARAMETER

Das Setzen des „PARAMETER“ Elements bewirkt die Festlegung einer Generic-Konstante. So wird im Beispiel der Abbildung 2.8 die „C_BAUDRATE“-Generic-Konstante mit dem Wert 9600 festgelegt. Dieser Parameter beeinflusst also die zukünftige Baudrate der Uart SoC-Komponente. „PARAMETER“ Elemente können auch benutzerfreundlich durch den „Configure IP ...“ Menüpunkt festgelegt werden.

BUS_INTERFACE

„BUS_INTERFACE“ Elemente legen die Buszugehörigkeit eines IP-Cores fest. Die Abbildung 2.8 zeigt, dass die „opb_uartlite“ SoC-Komponente an den OPB als Slave angeschlossen ist. Der Name der OPB Instanz an den die „opb_uartlite“ angeschlossen ist, lautet „mb_opb“.

PORT

Das „PORT“ Element legt fest, welchen Signale an die Eingangs- oder Ausgangsports des IP-Cores angelegt werden. In Abbildung 2.8 ist zu erkennen, dass der Eingangsport „RX“, der „opb_uartlite“ SoC-Komponente, mit dem Signal „fpga_0_RS232_Uart_1_RX“ verbunden ist. Das Signal „fpga_0_RS232_Uart_1_RX“ kann nun wiederum durch einen Pin des FPGA oder von einem Ausgangport einer anderen SoC-Komponente „getrieben“[7] werden.

Nähere Erläuterungen zu den Bezeichnungen SOPB und MOPB sowie zu den Festlegungen, welche PARAMETER-, BUS_INTERFACE- und PORT-Elemente ein IP-Core besitzt, erfolgen im Kapitelpunkt 2.2.2.3.

2.2.2.3 Microprocessor Peripheral Definition (MPD)

Die **M**icroprocessor **P**eripheral **D**efinition (kurz MPD) ist das wichtigste aller Konstrukte des EDK. An der MPD wird das vielseitige und flexible Potential des EDK deutlich. Die MPD wird für jeden IP-Core einzeln definiert. Sie kann im „IP-Coremenü“, mit dem Punkt „View MPD“ sichtbar gemacht werden.

```
BEGIN opb_uartlite

        OPTION IPTYPE = PERIPHERAL
        OPTION DESC = OPB UART (Lite)
        OPTION CORE_STATE = ACTIVE
        BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE
        PARAMETER C_BASEADDR = 0xFFFFFFFF, DT = std_logic_vector, MIN_SIZE = 0x100
        PARAMETER C_HIGHADDR = 0x00000000, DT = std_logic_vector
        PARAMETER C_BAUDRATE = 9600, DT = integer, DESC = Baudrate
        PORT OPB_RNW = OPB_RNW, DIR = I, BUS = SOPB
        PORT RX = , DIR = I, PERMIT = BASE_USER, DESC = 'Serial Data In'
        PORT TX = , DIR = O, PERMIT = BASE_USER, DESC = 'Serial Data Out'

END
```

Abbildung 2.9: Ausschnitt der MPD des „opb_uartlite“ IP-Cores

[7] Unter dem Ausdruck „treiben“ oder „wird getrieben“ versteht man das Übertragen eines logischen Pegels auf etwas. Dies wird bei Hardware durch Verdrahtung realisiert.

Abbildung 2.9 zeigt einen Ausschnitt der MPD des „opb_uartlite“ IP-Cores. Es werden Syntheseoptionen gesetzt, die Bus-Zugehörigkeit festgelegt und alle E/A-Ports der Haupt-Entity deklariert.

Für jedes IP-Cores sind folgende Elemente in der MDP verwendbar:

ELEMENT	Beispiel
OPTION	„OPTION DESC = OPB UART (Lite)“
BUS_INTERFACE	„BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE“
PARAMETER	„PARAMETER C_BAUDRATE = 9600, DT = integer“
PORT	„PORT OPB_RNW = OPB_RNW, DIR = I, BUS = SOPB“

Tabelle 2.3: Elemente der MPD

OPTION

Die „OPTION“ Elemente, definieren Ausdrücke, die zur benutzerfreundlichen Einbindung der IP-Cores in EDK verwendet werden. Das Beispiel aus Abbildung 2.9 setzt den „DESC“ Ausdruck auf „OPB UART (Lite)“. Dadurch wird im „IP Catalog“, der EDK GUI, eine kleine Beschreibung des IP-Core „opb_uartlite“ angezeigt.

Ein weiterer nützlicher Ausdruck ist „CORE_STATE“. Wird der Ausdruck auf „ACTIVE“ gesetzt, so wird der IP-Core nur bei der ersten Synthese synthetisiert. Da sich die Netlist des IP-Cores nicht unbedingt geändert hat, muss diese, bei einer wiederholten Synthese, nicht neu synthetisiert werden. Verändert sich allerdings eine der Generic-Konstanten hat das eventuell, je nach VHDL-Code, einen Einfluss auf das Verhalten der SoC-Komponente und der IP-Core wird neu synthetisiert.
Wird „CORE_STATE“ allerdings auf „DEVELOPMENT“ gesetzt, so wird der IP-Core bei jeder Synthese neu synthetisiert, da es sich offensichtlich um ein IP-Core handelt, der sich in der Entwicklung befindet und oft geändert wird.

BUS_INTERFACE

Mit dem „BUS_INTERFACE“ Element ist es möglich einem IP-Core eine Zugehörigkeit zu einem Bus zuzuteilen.
Die Abbildung 2.9 der „opb_uartlite“ MPD fügt dem IP-Core ein Businterface mit der Bezeichnung „SOPB“ hinzu. Das Businterface „SOPB“ stellt einen Anschluss an einen OPB als Slave dar. Dies ist an den Deklarationen von Busstandard (BUS_STD) und Bustype (BUS_TYPE), in Abbildung 2.9, erkennbar.

Die bloße Anwesenheit des, in Abbildung 2.9 gezeigten, „BUS_INTERFACE“ Elements, erlaubt es dem EDK-Benutzer die SoC-Komponente an einen OPB an zuschließen. Der Verbindung ist dann in der Busstruktur der „Bus Interface“-Ansicht zu erkennen.

In einem Gedankenspiel könnte man die Zeile:

„BUS_INTERFACE BUS = MOPB, BUS_STD = OPB, BUS_TYPE = MASTER“

der MPD hinzufügen und tatsächlich könnte der Benutzer nun den „opb_uartlite“ IP-Core als Master an den OPB anschließen.

Diese Einstellung hätte natürlich keine schaltungstechnischen Auswirkungen, da einiges mehr notwendig ist, um ein Master auf einem Bus zu sein. Aber es ist eine beachtliche Flexibilität erkennbar, die bei der Einbindung der GRLIB nützlich war.

PARAMETER

Alle mit „PARAMETER“ versehene Ausdrücke sind Definitionen von Generic-Konstanten. Diese Konstanten sind im „Configure IP ...“-Dialogfeld einstellbar.

Abbildung 2.9 zeigt die „C_BAUDRATE“-Generic-Konstante, die auf den Strandartwert „9600“ gesetzt wird. Erfolgt also keine Änderung dieser Konstante, im „Configure IP ...“-Dialogfeld, so kommuniziert die Uartlite Komponente mit 9600 Bauds pro Sekunde.

PORTS

Die mit dem Element „PORTS“ versehene Ausdrücke, kennzeichnen alle Eingangs- und Ausgangsports der IP-Core Entity. Hier können Standards festgelegt werden, mit welchem Signal ein Port getrieben werden soll oder welcher Port zu einem Businterface gehört.

Das Beispiel in Abbildung 2.9 trifft Festlegungen über den Port „OPB_RNW“. Dieser Eingangsport dient zur Kommunikation über den angeschlossenen OPB. Mit dem „OPB_RNW“ Port kann die „opb_uartlite“-Entity auswerten, ob aus ihren Registern gelesen oder in ihre Register geschrieben werden soll.
Der „OPB_RNW“ Port gehört zu einem Bestandteil des OPB-Interface und muss von einem OPB-Slave IP-Core, zur Verfügung gestellt werden. Ist dies nicht der Fall, kann das OPB-Protokoll nicht richtig ausgewertet werden und die Kommunikation schlägt fehl. Wenn ein Port eine solche Businterface Zugehörigkeit besitzt, so wird dessen MPD Element mit dem „BUS“ Ausdruck erweitert.

Mit einem „=“ folgt die Festlegung der Businterface Zugehörigkeit. Das Businterface muss zuvor mit „BUS_INTERFACE“ deklariert worden sein.

Diese eben genannte Erweiterung hat eine entscheidende Wirkung. Durch ein einfaches Verbinden einer SoC-Komponente mit einem Bus in der Busstruktur über die EDK-GUI, werden automatisch alle gekennzeichneten Ports im Arbeitsschritt 1, des EDK-Arbeitsplans, mit dem Bus verbunden.
In der System-Entity wird also ein Signal eingefügt, dass nun den „OPB_RNW“ Port der „opb_uartlite“-Entity treibt und seinen Ursprung in der Entity des OPB hat. Da ein einzelnes Businterface viele Ports umfasst, wird durch die eben genannte Wirkung erheblich Arbeit und Zeit, bei der Erstellung des SoC, gespart.
Auch bei der Einbindung der GRLIB war es so möglich, einen Busstandard zu definieren und alle IP-Cores so einfach anschließbar zu machen.

Wird für einen Port Ausdruck in der MPD kein Standardsignal gewählt, also statt:

„PORT OPB_Clk = xyz“ nur „PORT OPB_Clk = “” “,

so kann dieser Port kein Businterfaceport sein.

Stattdessen wird dieser Port in der „Ports“-Ansicht der EDK GUI sichtbar und kann mit einem Signal manuell nach außen, zu einem Pin, oder innerhalb der System-Entity weiter verbunden werden.

2.2.2.4 Peripheral Analyze Order (PAO)

Die **P**eripheral **A**nalyze **O**rder (kurz PAO) ist ein Konstrukt, das in einer Datei mit der Endung „pao“ gespeichert wird.

```
lib common_v1_00_a pselect
lib opb_uartlite_v1_00_b baudrate
lib opb_uartlite_v1_00_b srl_fifo
lib opb_uartlite_v1_00_b opb_uartlite_rx
lib opb_uartlite_v1_00_b opb_uartlite_tx
lib opb_uartlite_v1_00_b opb_uartlite_core
lib opb_uartlite_v1_00_b opb_uartlite
```

Abbildung 2.10: Ausschnitt der PAO des „opb_uartlite“ IP-Cores

Die PAO-Datei beinhaltet eine Auflistung von VHD-Dateien, welche als Bibliotheken genutzt werden können. Diese Bibliotheken werden in der Reihenfolge analysiert wie sie in der PAO-Datei aufgelistet sind. Erst nachdem dieser Vorgang beendet ist, stehen die VHDL-Entitys zur Synthese

des eigentlichen IP-Cores bereit und können in ihm eingebunden werden. Die Einbindung solcher VHDL-Entitys, in der eigentlichen IP-Core Entity, geschieht mit dem VHDL-Konstrukt „port map“.

Alle VHD-Dateien, die als Bibliotheken zu Verfügung stehen sollen, müssen mit einem „lib“ Element in die PAO-Datei eingetragen werden. Mit Leerstelle getrennt folgt der Name der IP-Core Bibliothek in der die VHD-Datei gesucht werden soll. Es ist also auch möglich, dass ein IP-Core die VHD-Dateien eines anderen IP-Cores nutzen kann. Es wird prinzipiell nur das „hdl“-Verzeichnis des angegebenen IP-Cores durchsucht.
Abermals mit Leerstelle getrennt folgt der Name der VHD-Datei deren Entitys nutzbar sein sollen.

In Abbildung 2.10 ist zu erkennen, dass der „opb_uartlite“ IP-Core eine „opb_uartlite_rx“ VHD-Datei einbinden möchte. In dieser Datei befindet sich die Entity „OPB_UARTLITE_RX“, welche das Verhalten eine FIFO-Datenstruktur beschreibt.
Offensichtlich funktioniert die Kommunikation der Seriellen Schnittstelle mit dem OPB über eine FIFO.

Der LEON3-Prozessor besteht, aus Übersichtlichkeits-Gründen, nicht aus einer Entity. Das bedeutet für die PAO-Datei des LEON3 IP-Cores, dass alle noch benötigten VHD-Dateien dort aufgelistet sind.

2.2.2.5 Microprozessor Software Specification (MSS)

Die **M**icroprozessor **S**oftware **S**pecification (kurz MSS) mit ihrer Datei „system.mss“ existiert in Abhängigkeit der MHS.
Die MSS wird als Eingang für den **Lib**rary **Gen**erator (kurz Libgen) benötigt. In der MSS definiert EDK Assoziationen von Betriebssystemen (engl.: **o**perating **s**ystems, kurz OS), Bibliotheken und Treibern zu jeder einzelnen SoC-Komponente.

```
BEGIN DRIVER

        PARAMETER DRIVER_NAME = uartlite
        PARAMETER DRIVER_VER = 1.01.a
        PARAMETER HW_INSTANCE = opb_uartlite_0

END

BEGIN OS

        PARAMETER OS_NAME = standalone
        PARAMETER OS_VER = 1.00.a
        PARAMETER PROC_INSTANCE = microblaze_0
        PARAMETER STDIN = debug_module
        PARAMETER STDOUT = debug_module

END

BEGIN PROCESSOR

        PARAMETER DRIVER_NAME = cpu
        PARAMETER DRIVER_VER = 1.01.a
        PARAMETER HW_INSTANCE = microblaze_0
        PARAMETER COMPILER = mb-gcc
        PARAMETER ARCHIVER = mb-ar
        PARAMETER XMDSTUB_PERIPHERAL = debug_module

END
```

Abbildung 2.11: Ausschnitt der MSS eines MicroBlaze-Systems

Ein Deklarartionsblock beginnt immer mit dem Schlüsselwort „BEGIN“ und endet mit dem Schlüsselwort „END“. Nach dem „BEGIN“ folgt entweder „DRIVER“, „PROCESSOR“, „OS“ oder „LIBRARY“. Innerhalb der Definitionsblöcke werden ausschließlich „PARAMETER“ Elemente benutzt, um Ausdrücke zu setzen.

DRIVER

Mit diesem Ausdruck wird festgelegt, welcher Treiber für eine SoC-Komponente benutzt werden soll. Ein Treiber ist, in diesem Fall, eine Ansammlung von C-Dateien mit zugehörigen Header-Dateien, welche „IP-Core Funktionen“ beinhalten, mit denen es möglich ist einen IP-Core einfacher und benutzerfreundlicher zu steuern.
Diese „IP-Core Funktionen“ stehen nur zur Verfügung, wenn der Treiber eingebunden wird, also automatisch, wenn die SoC-Komponente im SoC vorhanden ist. Die „IP-Core Funktionen“ können dann innerhalb des native C-Programms benutzt werden.

Der Libgen kompiliert dann alle eingebundenen Treiber, sodass ein erfolgreiches Kompilieren

des nativen C-Programms möglich ist. Der Aufbau eines Treiberverzeichnisses für eine SoC-Komponente wird in Kapitelpunkt 2.2.2.6 behandelt.

In der Abbildung 2.11 ist zu erkennen, dass für den „opb_uartlite“ IP-Core der Treiber „uartlite“ verwendet wird. Dieser Treiber beinhaltet Header-Dateien, die nützliche Definitionen enthalten können. Solche Definitionen können zum Beispiel Fehlercodekonstanten oder Registeradressenkonstanten sein.
Ebenfalls können in den C-Dateien des Treibers Funktionen deklariert sein, welche die Arbeit mit der SoC-Komponente erleichtern. Ein Beispiel solcher C-Funktionen wäre eine Initialisierungsfunktion, die eine Folge von Registerzuweisungen an der SoC-Komponente vornimmt.

PROCESSOR

Die „PROCESSOR“ Blöcke werden speziell für IP-Cores angelegt, die in ihrer MPD mit „OPTION IPTYPE = PROCESSOR“ als Prozessor definiert wurden. Für sie können dann, neben Treibern, Parameter festgelegt werden, wie der zugehörige Compiler oder Archievierer.

OS

Definitionsblöcke mit „OS“ legen für eine bestimmte Prozessorinstanz ein Betriebssystem fest. Dies kann ein einfacher Kernel sein oder ein einzeln stehendes Programm.

In der Abbildung 2.11 ist zu erkennen, dass für den MicroBlaze-Prozessor das Betriebssystem „standalone“ deklariert wird. Das „standalone“ OS ist kein eigentliches Betriebssystem, sondern nur eine Ansammlung von Bibliotheken. Das „standalone“ OS wird benutzt, wenn ein natives C-Programm auf dem SoC laufen soll.

Betriebssysteme werden mit Hilfe des **B**oard **S**upport **P**ackage (kurz BSP) zur Verfügung gestellt. Der Aufbau eines BSP wird in Kapitelpunkt 2.2.2.8 erläutert.

2.2.2.6 Das Treiber-Verzeichnis

Die Treiber für IP-Cores befinden sich in „[XILINX_EDK]/sw/XilinxProcessorIPLib/drivers“.
Für alle IP-Cores für die es lohnenswert ist einige Bibliotheken zur Verfügung zu stellen sind hier die entsprechenden Treiber-Verzeichnisse angelegt.

Unterverzeichnisse

Die beiden wichtigsten Unterverzeichnisse eines Treibers sind das „**data**“-Verzeichnis und das „**src**“-Verzeichnis. Diese müssen existieren. Das Treiber-Verzeichnis hat eine Benennung nach dem Muster „[name_driver]_v?_??_?“[8].

Das „**data**“-Unterverzeichnis muss die **M**icroprocessor **D**river **D**efinition (kurz MDD) enthalten. Die MDD wird in einer Datei gespeichert, deren Name nach dem Muster „[name_driver]_v2_1_0.mdd“ konstruiert sein muss.

Das „**src**“-Unterverzeichnis beinhaltet alle C- und Header-Dateien, welche bei Arbeitsschritt 3, des EDK-Arbeitsplans, vor der Kompilierung des nativen C-Programms kompiliert werden. Die enthaltenen C-Funktionen können dann im nativen C-Programm verwendet werden.

Bei der Einbindung der GRLIB wurde nur ein Treiber für den LEON3 und für die AHB2LMB Bridge angelegt. Dieser Treiber beinhaltet keine C- oder Header-Dateien, sondern nur eine Festlegung zur Benutzung des „standalone“ OS.

2.2.2.7 Microprocessor Driver Definition (MDD)

Die **M**icroprocessor **D**river **D**efinition (kurz MDD) befindet sich im „data“-Unterverzeichnis eines Treiber-Verzeichnisses.

```
BEGIN driver uartlite

        OPTION supported_peripherals = (opb_uartlite opb_jtag_uart opb_mdm);
        OPTION driver_state = ACTIVE;
        OPTION depends = (common_v1_00_a);
        OPTION copyfiles = all;

END driver
```

Abbildung 2.12: Ausschnitt der MDD für den „opb_uartlite“ IP-Core

```
BEGIN driver cpu

        OPTION copyfiles = all;
        OPTION depends = (common_v1_00_a);
        OPTION driver_state = ACTIVE;
        OPTION supported_peripherals = (microblaze);
        OPTION default_os = ßtandalone_v1_00_a";

END driver
```

Abbildung 2.13: Ausschnitt der MDD für den „microblaze“ IP-Core

[8] Der Ausdruck [name_driver] steht für die Bezeichnung des Treibers und die vier „?“ stehen für Version, Subversion und Subsubversion. Version und Subversion müssen Ziffern sein und die Subsubversion ein Buchstabe.

Die MDD beinhaltet nur einen einzigen „BEGIN DRIVER [name_driver]“ - „END“ Definitionsblock. Im Treiber-Definitionsblock werden hauptsächlich Optionen mittels des „OPTION“ Elements gesetzt.

Wichtige Optionen sind:

„supported_peripherals“: Es folgen eine Reihe von IP-Core Bezeichnungen. Existiert ein Treiber, welcher die IP-Core Bezeichnungen „xyz“ an dieser Stelle angibt, so wird, durch einfaches Hinzufügen des IP-Cores, automatisch der Treiber mit der Bezeichnung [name_driver] festgelegt. Diese Festlegung wird in der MSS, des SoC, gespeichert.

An der Abbildung 2.12 ist zu erkennen, dass die SoC-Komponenten „opb_uartlite“, „opb_jtag_uart“ und „opb_mdm“ den gleichen Treiber benutzen. Für diese SoC-Komponenten werden also die gleichen C- und Header-Dateien als Bibliotheken zur Verfügung stehen.

„default_os“: Nach diesem Optionsausdruck folgt die Angabe eines BSP. Mit dem BSP wird eine Art Betriebssystem für einen Prozessor festgelegt.

Die Abbildung 2.13 zeigt, dass für den MicroBlaze-Prozessor standardmäßig das „standalone_v1_00_a“ Betriebssystem vorgesehen ist.

2.2.2.8 Das BSP-Verzeichnis

Die Abkürzung BSP steht für **B**oard **S**upport **P**ackage.
Das BSP-Verzeichnis beinhaltet alle bereitgestellten Betriebssysteme. Das Betriebssystem muss für einen bestimmten Prozessor in der MDD festgelegt werden. Die BSP werden im Verzeichnis „[XILINX_EDK]/ sw/lib/bsp“ aufbewahrt.
Eine nähere Erläuterung des BSP-Verzeichnisses soll hier nicht stattfinden, da diese „rudimentären“ Betriebsysteme zur Erstellung eines SoC für die GRLIB keine Bedeutung haben.
Es sei allerdings erwähnt, dass die Firma Xilinx beispielsweise das BSP „standalone_v1_00_a“ zur Verfügung stellt. Dieses BSP wird für das native C-Programm benutzt. Dadurch wird nur das C-Programm auf dem Prozessor ausgeführt.
Ein weiteres BSP wird „xilkernel_v3_00_a“ genannt. Dieses beinhaltet eine einfache Möglichkeit Threads in Verbindung eines einfachen Schedulers zu benutzen.

2.2.2.9 Tool Command Language (Tcl)

Die **T**ool **C**ommand **L**anguage (kurz Tcl) ist eine Art Skript-Sprache, welche von den EDK Tools ausgewertet werden kann. Sie wird benutzt um Informationen auszuwerten, die in den Konstrukten wie MHS, MSS, MPD und MDD festgelegt wurden.

Hat der EDK-Benutzer die Erstellung seines SoC abgeschlossen, so beginnt er die Synthese. Bevor der Arbeitsplan des EDK in seiner Reihenfolge ausgeführt wird, werden alle gefundenen Tcl-Skripts ausgeführt. Die Tcl-Skripts sind an ihre Dateierweiterung „*.tcl“ erkennbar.
Jeder IP-Core besitzt sein eigenes Tcl-Skript, welches sich im „data“-Unterverzeichnis des IP-Core-Verzeichnisses befindet. Siehe Kapitelpunkt 2.2.2.1.
In jedem Tcl-Skript werden Prozeduren definiert, die speziell auf den IP-Core angewendet werden.

Für diese Arbeit wurden folgende Wirkungen, der Tcl-Skipts, umgesetzt:

- das Überprüfen von Busverbindungen an einzelnen SoC-Komponenten
- das automatische Berechnen und Überschreiben von Generic-Konstanten
- das Erstellen von Constraints-Dateien (UCF-Dateien), zur Festlegung von Bedingungen an einzelne SoC-Komponenten

Nach der Einbindung der GRLIB als EDK-IP-Cores, wurden die Tcl-Skripts so oft wie möglich eingesetzt. Folglich werden dem Benutzer viele Einstellungen abgenommen und die Fehlerauftrittswahrscheinlichkeit wird minimiert.
Es wird zwischen Tcl-Skripts unterschieden, welche auf Informationen der Hardware oder der Software arbeiten. Für diese Arbeit sind nur die Auswertungen der Hardwareinformationen relevant. Die Tcl-Skripts arbeiten auf einer Datenstruktur, welche aus den EDK-Konstrukten, MHS und MPD, erstellt wird. Die MHS und die MPD enthalten nur die hardwarerelevanten Informationen. Die entstehende Datenstruktur wird als vermischte (engl.: Merged) Datenstrktur bezeichnet.

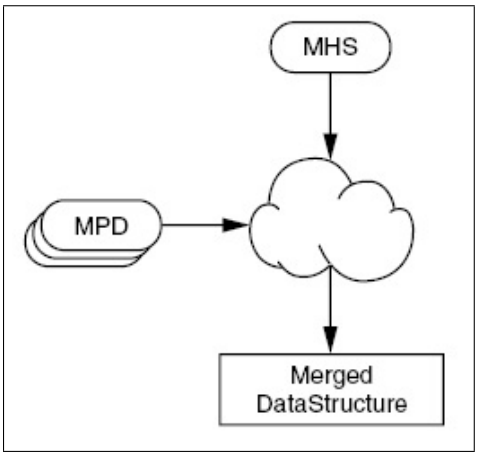

Abbildung 2.14: Generierung der „Merged DataStructure“ aus MHS und MPD [estdoc]

Diese „Merged“-Datenstruktur ist hierarchisch aufgebaut und beinhaltet als oberstes Element den „Merged“ MHS-handle.

Um auf der Datenstruktur zu arbeiten stehen verschiedenste Funktionen zur Verfügung, wie:

Prozedur	Wirkung
xget_hw_name <handle>	Damit kann eine Zeichenkette zurückgegeben werden, die den Namen, der mit „<handle>“ übergebenen Instanz, enthält.
xget_hw_parameter_handle <handle> <parameter_name>	Mit dieser Prozedur ist es möglich den handle eines Parameters zu erhalten, der mit der Zeichenkette „<parameter_name>“ bezeichnet wird.
xget_hw_port_value <handle> <port_name>	Diese Prozedur liefert eine Zeichenkette, mit dem Namen des Signals, welches den Port „<port_name>“ treibt.

Tabelle 2.4: Prozedurenbeispiele der Tcl

Durch die geeignete Wahl und Kombination von solchen Prozeduren ist es möglich alle Informationen aus der MHS oder der MPD zu verarbeiten.

```
proc dsu_check { param_handle } {
  set mhsinst [xget_hw_parent_handle $param_handle]
  set instname [xget_value $mhsinst "PARAMETER" "INSTANCE"]

  puts "============================================================"
  puts "Instance $instname: Look for DSU3-INSTANCE at the AHB..."
  puts "============================================================"

  set dsuif  [xget_value $mhsinst "BUS_INTERFACE" "MDSULEON3"]

  if {[string length $dsuif] == 0} {
    puts "INFORMATION: $instname Processor is not connected
     to a DSU Bus as slave. Return dsu = 0"
    return 0
  } else {
    puts "INFORMATION: $instname Processor is connected
     to a DSU Bus as slave. Return dsu = 1"
    return 1
  }
}
```

Abbildung 2.15: Die „dsu_check“-Prozedur des LEON3 Tcl-Skripts

Die Abbildung 2.15 zeigt ein Beispiel einer Tcl-Prozedur. Sie wird „dsu_check“ genannt und ist Bestandteil des eingebundenen LEON3-IP-Cores. Diese Prozedur überprüft ob eine „Debug Support Unit“ an die konkrete LEON3-Instanz angeschlossen ist und überschreibt dementsprechend die „dsu“-Generic-Konstante.

Kapitel 3

Stand der Technik

Es existieren bereits viele Arbeiten auf dem Gebiet der SoC-Lösungen, welche mit dem LEON2- oder LEON3-Prozessor erstellt wurden. Als Hauptanbieter und Verbreiter wird in diesem Zusammenhang immer wieder die GRLIB genutzt. Die GRLIB setzt grundsätzlich auf die Busse AHB und APB nach der AMBA-Spezifikation. Als Prozessor wird der hauseigene LEON3-Prozessor nach der SPARC benutzt. Ein weiterer Grund für die Verbreitung der GRLIB ist deren Vielzahl von zusätzlichen IP-Cores, welche genutzt werden um Peripheriekomponenten, wie Speicher oder Kommunikationsschnittstellen, anzusprechen.

Vorkommen der GRLIB

Die GRLIB ist eine umfangreiche Zusammenstellung vieler benötigter IP-Cores, so dass die GRLIB heutzutage die wichtigste Zieladresse ist, wenn freie IP-Cores zu Erstellung eines SoC gesucht werden. Die GRLIB wird mit einem eigenen Workflow angeboten, der auf Synthese-Tools, wie die „Xilinx Synthese Tool“ oder „Synplify FPGA“ aufsetzt. Dieser Workflow wird in Kapitelpunkt 3.1.4 (Handhabung der GRLIB) vorgestellt.
Ein weiteres Vorkommen der GRLIB-IP-Cores ist die Firma „Gleichmann Electronics Research“. Gleichmann bietet VHD-Dateien aus der GRLIB an, welche beliebig synthetisiert werden können. Ein eigener Synthese-Workflow wird jedoch nicht angeboten.

Arbeiten mit der GRLIB oder den AMBA-Bussen

In der Arbeit [LeonRob] wurde die GRLIB genutzt, um mit einem SoC einen Hexapod-Roboter zu steuern. Auf dem SoC wurde das „SnapGear“ Linux ausgeführt. Das SnapGear Linux wird ebenfalls auf der Homepage der Firma GR zum freien Download angeboten. In dieser Arbeit wurde der Workflow der GRLIB benutzt, um ein LEON3-System zu synthetisieren.
Die Arbeit [LinuxLeon] evaluiert die Einsetzbarkeit des Linux-Betriebssystems „SnapGear“ auf einem LEON3-System, stammend aus der GRLIB. Auch hier wurde zur Synthese der Workflow der GRLIB verwendet.
Die Arbeit [EvalCPU] beschäftigt sich mit dem Vergleich des LEON2-Prozessors mit dem MicroBlaze-Prozessor. Hier wird speziell auf Größe, Performance und Funktionalität eingegangen. Da diese Arbeit nicht mehr aktuell ist, wurden die Ergebnisse der Arbeit [EvalCPU] teilweise im Kapitelpunkt 8.3 erneuert.

Verbindung des LEON3-Systems mit dem MicroBlaze-System

Heute existieren kaum Ansätze, welche es ermöglichen ein LEON3-System mit einem MicroBlaze-System über eine Bus-Bridge zu verbinden. In [IBMCore] wird von IBM ein IP-Core vorgestellt, welche das PowerPC-System mit dem LEON3-System über eine PLB2AHB-Bridge verbinden kann. Über einen zusätzlichen IP-Core der Xilinx Bibliothek, könnte nun das MicroBlaze-System, mit einer OPB2PLB-Bridge, mit dem LEON3-System verbunden werden. Die PLB2AHB-Bridge von IBM ist leider kostenpflichtig und liegt nicht in einsehbarem VHDL vor.
Aufgrund dieser unkomfortablen existierenden Möglichkeiten, soll die Verbindung, des LEON3-Systems mit dem MicroBlaze-System, in dieser Arbeit ein Hauptbestandteil sein. Zur Umsetzung wurde ein AHB2OPB-Bridge IP-Core entworfen, der den Haupt-Bus des LEON3-Systems (AHB) mit dem Peripherie-Bus des MicroBlaze-Systems (OPB) verbindet. Dadurch kann der LEON3-Prozessor auf direktem Umweg über die AHB2OPB-Bridge Bus-Transfers auf dem OPB auslösen.

Zusammenfassung

Es existieren keine Ansätze, die es ermöglichen die GRLIB mit Prozessor, Bussen und Peripherie-Komponenten in einer benutzerfreundlichen und flexiblen Umgebung wie dem Xilinx-EDK zu nutzen. Dieser Zustand soll mit dieser Arbeit verändert werden.
Zur Weiterbenutzung bereits erstellter IP-Cores für das MicroBlaze-System innerhalb des LEON3-Systems, wurde in dieser Arbeit eine AHB2OPB-Bridge erstellt. Diese Bridge ermöglicht eine, in

der Vergangenheit noch nicht existente, Verbindung des LEON3-Systems mit dem MicroBlaze-System.

Da in den oben genannten Arbeiten die GRLIB der am häufigsten verwendete Ansatz ist, um ein LEON3-SoC mit den zugehörigen AMBA-Komponenten zu synthetisieren, soll im Kapitelpunkt 3.1 (Die Gaisler Research IP Library (GRLIB)) die GRLIB genauer beleuchtet werden.
In Kapitelpunk 3.1.4 soll speziell der Workflow der GRLIB beschrieben werden, mit der die GRLIB veröffentlich wird. Mit diesem GRLIB-Workflow können, auf der Grundlage der XST, LEON3-SoCs synthetisiert werden. Da noch keine Umsetzungen existieren, die es ermöglichen den EDK-Workflow für die GRLIB-IP-Cores zu nutzen, wurde der GRLIB-Workflow in den oben genannten Arbeiten verwendet.

3.1 Die Gaisler Research IP Library (GRLIB)

Nachdem nun im voran gegangenen Kapitel alle EDK-Konstrukte, zur Erstellung eines eigenen IP-Cores, erörtert wurden. Fehlt nur noch ein kleiner aber bedeutender Bestandteil zum vollständigen IP-Core. Bei diesem Bestandteil handelt es sich um die Verhaltensbeschreibung der einzelnen möglichen IP-Cores, die in zugänglichem VHDL-Code vorliegen sollen.
Dazu soll die **G**aisler **R**esearch IP **Lib**rary (kurz GRLIB) benutzt werden. In ihr ist eine derartig Große Anzahl von IPs vorhanden, dass es möglich war, zu jedem wichtigen Xilinx-IP-Core ein GRLIB Pendant für das EDK einzubinden.

Konkrete Beispiele solcher IPs sind:

- Der LEON3-Prozessor ersetzt den MicroBlaze-Prozessor
- Der Advanced High-performance Bus
- Der Advanced Peripheral Bus ersetzt den OPB
- Die APB Uart IP ersetzt die OPB UartLite
- Die AHB DDR RAM IP ersetzt den OPB DDR RAM IP-Core
- Die Gaisler Research Ethernet IP ersetzt den OPB Ethernet IP-Core

Weitere IPs der GRLIB, die neue Funktionalitäten bieten sind:

- Die AHB on Chip RAM IP
- Die AHB on Chip ROM IP
- Eine APB IP für die PS2 Schnittstelle
- Die APB VGA IP zur Ausgabe von Text über die SVGA Schnittstelle

- Die SVGA IP zur Ausgabe von Framebuffer Daten über die SVGA Schnittstelle

Die GRLIB lässt sich auf der Homepage der Firma Gaisler Research herunterladen und war, bei der Erstellung dieser Arbeit, in der Version 1.0.14 Built 2028 erhältlich.
Diese Information ist sehr wichtig, da ständig neue Versionen mit Verbesserungen veröffentlicht werden. Es werden permanent Fehler an alle IPs, vor allem am LEON3, behoben. Dadurch bildet die GRLIB eine sehr kompetente und wertvolle Grundlage zur Erstellung der EDK-IP-Cores.

In diesem Kapitel soll die GRLIB in ihrem Zustand, wie sie auf der Homepage von Gaisler Research veröffentlicht ist, erläutert werden. Dies ist eine wichtige Grundlage, um die Ansätze, zur Einbindung der GRLIB in das EDK, zu verstehen.
Außerdem befinden sich noch weitere IPs in der GRLIB, welche nicht im Verlauf dieser Arbeit eingebunden wurden, da sie nicht primär zur Erstellung eines SoC benötigt werden.
Möchte der EDK Benutzer ein SoC erstellen in dem eine nicht eingebundene Komponente der GRLIB benötigt wird, so ist er gezwungen den Prozess der Einbindung für diese IP durchzuführen.

3.1.1 Überblick

Die GRLIB ist in einer **t**ape **ar**chiver (kurz tar) Datei komprimiert. Wird daraus die GRLIB extrahiert liegt sie in einem einzigen Verzeichnis, mit der Benennung „grlib-gpl-?.?.??-b????“[1], vor.
Die GRLIB ist eine Bibliothek zur Entwicklung von SoC. Sie unterteilt sich in verschiedene Unterbibliotheken, welche von verschiedenen Anbietern stammen. Die SoC-Komponenten, der GRLIB, sind dabei um den AHB angeordnet. Die GRLIB bietet Methoden zur Synthese und Simulation, dabei werden mehrere Werkzeuge unterstützt.
In dieser Arbeit wurde sich auf die Nutzung der Xilinx Werkzeuge ISE 8.2i und EDK 8.2i beschränkt. Es werden allerdings noch weitere Werkzeuge aktiv unterstützt[2], wie die Synthese Werkzeuge:

Altera Quartus, Cadence RTL Compiler (RC), Synplify FPGA, Synopsys Design Compiler (DC)

Und die Simulatoren:

Model*Sim*, Cadence Ncsim, GNU VHDL Simulator (GHDL), Active-HDL VHDL and Verilog simulator, Riviera VHDL and Verilog simulator, Symphony-EDA Sonata VHDL simulator

In der GRLIB wird nicht nur auf dem Gebiet ihrer Verarbeitungswerkzeuge versucht eine Unabhängigkeit zu erreichen, sondern auch auf den Chips auf denen sie konfiguriert werden kann.

[1] Die Fragezeichen des Musters „?.?.??“ stehen für die Version, Subversion und Subsubversion. Die vier aufeinander folgenden Fragezeichen („????“) stehen für die Built-Nummer.
[2] Unter aktiver Unterstützung ist hier die Erstellung von Tool-Skripts und Projektdateien, die mit den einzelnen Werkzeugen geöffnet werden können, gemeint.

So ist es beispielsweise möglich die GRLIB für FPGAs der Firma Xilinx und der Firma Altera zu synthetisieren. Des weiteren ist es auch möglich die GRLIB auf ASICs einzusetzen.

3.1.2 Installation

Bevor die GRLIB benutzt werden kann, muss der Host-PC vorbereitet werden. Auf einem Linux-System müssen folgenden Komponenten installiert sein:

- Eine Shell: Bash oder tcsh
- Das neuste GNU make Tool
- Der neuste **G**NU **C** **C**ompiler (kurz GCC)
- Die Skriptsprache Tcl/tk ab Version 8.4

Auf einem Windows-System muss ein Linux-Emulator installiert werden. Dafür ist „Cygwin“ eine geeignete Wahl. Die für Linux benötigten Komponenten müssen auch im Cygwin installiert werden.

Damit die Software Xilinx-EDK ihren Arbeitsplan ausführen kann, wird mit dem EDK für Windows eine eigene Cygwin Version ausgeliefert.

Zu Beachten 3.1 (Cygwin)

Es ist dringend darauf zu achten, dass bei der Installation des neuen Cygwin, für die Handhabung der GRLIB, keine Vermischung mit der Xilinx-EDK Cygwin Version stattfindet.
Eine Vermischung der beiden Cygwin Versionen würde Fehlfunktionen bei der Ausführung des EDK-Arbeitsplans verursachen.

Die eigentliche Installation der GRLIB ist mit dem entpacken der GRLIB TAR-Datei vollständig ausgeführt.

3.1.3 Verzeichnisstruktur

Nach der Extraktion der GRLIB werden die Hauptverzeichnisse zugänglich, welche sich im Verzeichnis [GRLIB_DIR][3] befinden. Der umfangreiche Inhalt der GRLIB Hauptverzeichnisse wird im Folgenden erläutert.

[3] Der Ausdruck [GRLIB_DIR] bezeichnet das Extraktionsverzeichnis der GRLIB.

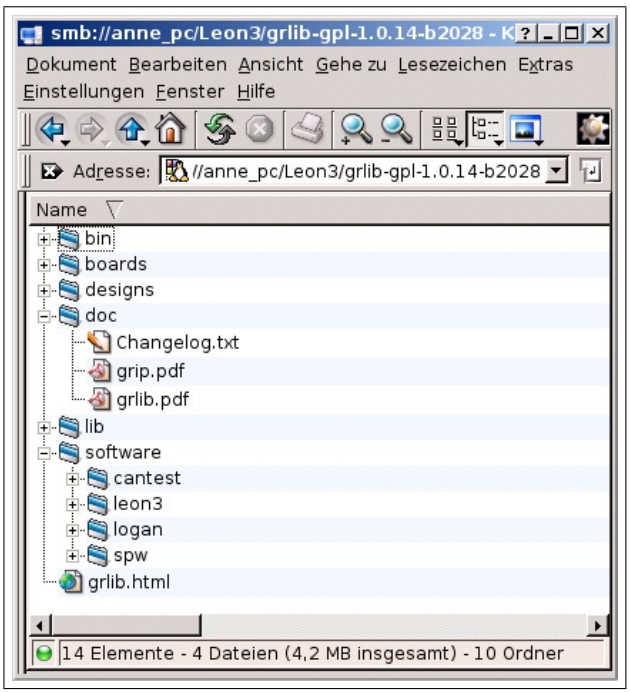

Abbildung 3.1: GRLIB Hauptverzeichnisse

Abbildung 3.1 zeigt die Hauptverzeichnisse der GRLIB 1.0.14. Alle VHD-Dateien der IPs sowie Template-Designs befinden sich im [GRLIB_DIR] Verzeichnis.

3.1.3.1 „bin“-Verzeichnis

Das „bin“-Verzeichnis beinhaltet alle Dateien, die zur Handhabung der GRLIB benötigt werden. Alle Funktionalitäten die mit der GRLIB geliefert werden sind mittels des Konzeptes der MAKEFILE erreichbar. Dazu gehöret beispielsweise die Funktionalität, dass zu jedem GRLIB Design die nötigen Projektdateien erstellt werden können. Diese Projektdateien können dann mit den in Kapitelpunkt 3.1.1 aufgezählten Werkzeugen geöffnet und benutzt werden.
Die GRLIB besitzt eine grafische Oberfläche, mit der eine benutzerfreundliche Konfiguration und Synthese eines SoC möglich sind. Diese Oberfläche ist mit der Tcl/tk geschrieben und befindet sich ebenfalls im „bin“-Verzeichnis.

3.1.3.2 „boards“-Verzeichnis

Die GRLIB, wie in Kapitelpunkt 1.3.2 erwähnt, ist boardorientiert. Das bedeutet, möchte der Benutzer die GRLIB mit ihrem vollen Komfort nutzen, so muss er erst entscheiden auf welchem FPGA-Board das SoC betrieben werden soll. Dies hat Auswirkung auf die Handhabung der GRLIB, welche später beschrieben werden.
Für jedes in der GRLIB unterstützte Board befindet sich ein Unterverzeichnis im „boards“-Verzeichnis, mit der jeweiligen Bezeichnung des Boards. Dieses Unterverzeichnis beinhaltet

boardspezifische Daten, wie die **U**ser **C**onstraint **F**ile (kurz UCF) und eine Include-Datei, für die MAKEFILE im „bin“-Verzeichnis. Die UCF ist eine Ansammlung von Constraints im Textformat. In ihr sind alle Constraints enthalten die das logische Design des SoC betreffen.

Die UCF beinhaltet unter anderen Festlegungen wie:

- Pin-Definitionen für den, auf dem Board befindlichen, FPGA
- Timing Einschränkungen für das gesamte Design des SoC
- Platzierungsbedingungen für einzelne Teile der logischen Schaltung

Das „boards“-Verzeichnis ist für den EDK Benutzer nicht weiter interessant. Es wird nur für das direkte Arbeiten mit der GRLIB benötigt.

3.1.3.3 „design“-Verzeichnis

Im „design“-Verzeichnis befindet sich abermals für jedes Board ein Unterverzeichnis. Die Unterverzeichnisnamen beginnen zusätzlich mit der Zeichenkette „leon3“. Damit soll dem Benutzer deutlich gemacht werden, dass es sich hier um das primäre Verzeichnis zur Synthese des SoC handelt. Diese Unterverzeichnisse beinhalten die Sogenannten Template-Designs (deut.: Schablonen-Designs). Alle generierten oder synthetisierten Dateien werden hier, während der Handhabung der GRLIB, abgespeichert.
Möchte der Benutzer beispielsweise ein LEON3-SoC für das „XUP Virtex-II Pro Development Board“ synthetisieren, so begibt er sich in das Verzeichnis „[GRLIB_DIR]/design/leon3-digilent-xup“ und beginnt dort mit der Handhabung der GRLIB. Die Handhabung wird im Kapitelpunkt 3.1.4 genauer erläutert.

An der Anzahl der Template-Designs Verzeichnisse ist zu erkennen, dass eine Synthese für ein beliebiges Board nicht möglich ist. Eine Abhilfe für dieses Problem bietet das Xilinx-EDK. Kann der Benutzer hier für sein Board ein MicroBlaze-SoC erstellen, so gelingt dies auch für ein GRLIB-SoC. Es müssen nur die GRLIB-IP-Cores in das EDK eingebunden werden.
Viele FPGA-Boardhersteller bieten die entsprechende **X**ilinx **B**oard **D**efinition (kurz XBD) für das Xilinx-EDK an. Jedoch werden von der Firma GR nur wenige Template-Designs der GRLIB hinzugefügt.

Jedes Template-Design beinhaltet folgende grundlegende Dateien:

Datei	Beschreibung
leon3mp.vhd	Diese Datei beinhaltet die Entity des gesamten GRLIB-SoC, sie wird „leon3mp“ genannt. Sie ist vergleichbar mit der Sytem-Enity, welche durch das EDK während seines 1. Arbeitsschritts generiert wird.
config.vhd	In dieser Datei wird ein VHDL-**Package** (deut.: Paket) definiert. In diesem Package werden nur Konstanten initialisiert. Da die GRLIB keine Möglichkeit besitzt eine SoC-Komponente hinzuzufügen oder zu löschen, wie das bei dem EDK möglich ist, bedient sich die GRLIB einer anderen Methode, um diese Möglichkeit umzusetzen. Eine genaue Erklärung findet sich im Paragraph „config.vhd“.
ahbrom.vhd	Diese Datei übernimmt die Aufgabe eines ROM-Speichers für das GRLIB-SoC. Die darin beschriebene „ahbrom“-Entity ist Slave auf dem AHB des SoC und stellt Daten bereit. Eine genaue Erklärung findet sich im Paragraph „ahbrom.vhd“.
testbench.vhd	Mit dieser Datei ist es möglich eine Simulation des GRLIB-SoC in einem VHDL-Simulator durchzuführen.
leon3mp.ucf	Diese Datei ist die, für das Board spezifische, UCF.
Makefile	Diese MAKEFILE ist der Einstiegspunkt für jegliche Handhabungsprozesse mit dem Design-Template. Eine genaue Beschreibung erfolgt in Kapitelpunkt 3.1.4.

Tabelle 3.1: wichtige Dateien einen Template-Designs

- ***„config.vhd“***

Die Datei „config.vhd“ ist ein VHDL-Package. Hier werden ausschließlich Konstanten initialisiert, welche direkte Auswirkung auf die Zusammensetzung des SoC Template-Designs haben.
Für die normale Handhabung der GRLIB besteht keine Möglichkeit, zum Sauberen hinzufügen oder Entfernen von SoC-Komponenten, wie der Benutzer es aus der Arbeit mit dem EDK gewohnt

ist.
Aus diesem Grund werden standardmäßig alle SoC-Komponenten in die „leon3mp“-Entity in Textform eingefügt, welche auf dem Board zur Verfügung stehen. Diese textförmigen Einbindungen der SoC Entitys werden mit dem VHDL-Konstrukt „generate“ umschlossen, um so mit einer Konstante Einfluss auf ihre Einbindung zu erlangen.
Mit der Änderung einer Konstante in der „config.vhd“ kann der Benutzer so bestimmen, ob er die SoC-Komponente in der „leon3mp“-Entity verwenden möchte oder nicht.

```
package config is

-- LEON3 processor core
  constant CFG_LEON3         : integer := 1;
  constant CFG_NCPU          : integer := (1);
-- DDR controller
  constant CFG_DDRSP         : integer := 1;
-- Gaisler Ethernet core
  constant CFG_GRETH         : integer := 0;
-- UART 1
  constant CFG_UART1_ENABLE  : integer := 1;
  constant CFG_UART1_FIFO    : integer := 4;

end;
```

Abbildung 3.2: „config.vhd“ Ausschnitt eines Template-Designs

Abbildung 3.2 zeigt einen Ausschnitt einer „config.vhd“. Der Benutzer möchte offensichtlich einen LEON3-Prozessor, eine DDR RAM Schnittstelle, keinen GR Ethernet IP-Core und eine APB UART benutzen. Die APB Uart Komponente soll eine FIFO der Länge 4 haben.
Möchte der Benutzer nun ein GRLIB-SoC erstellen, ist es nicht notwendig den empfindlichen Text der „leon3mp.vhd“ zu verändern. Auf alle Konfigurationsmöglichkeiten eines GRLIB-SoC erhält der Benutzer mit den Konstanten des VHDL-Packages „config.vhd“ Einfluss.

- ***„ahbrom.vhd“***

Die „ahbrom“-Entity ist bereits eine kleine aber vollwertige SoC-Komponente der GRLIB.
Sie hat das Verhalten eines Datenspeichers, der als Slave an den AHB gebunden ist.
Standardmäßig beginnt der Adresseraum für den Zugriff auf den „ahbrom“ an der Adresse 0x00000000[4].
Beginnt der LEON3-Prozessor nach der Konfiguration seine Arbeit, auf dem FPGA, so sucht er als erstes nach Instruktionen an der Adresse 0x00000000. Legt der Benutzer nun Programmdaten in Textform innerhalb der „ahbrom“-Entity ab, so werden diese nach der Synthese an der Adresse 0x00000000 vorliegen und der LEON3-Prozessor kann somit ein einfaches Programm ausführen.

[4] Die führende Zeichenkette „0x“ an einer Zahl sagt aus, dass die Zahl im hexadezimalen Format angegeben ist.

Die Arbeitsschritte, welche nötig sind um ein beliebiges Programm zu Kompilieren und in eine „ahbrom“-Entity umzuwandeln wird im Anhang A.2.1.3 genau beschrieben.

3.1.3.4 „doc“-Verzeichnis

Das „doc“-Verzeichnis beinhaltet alle mit der GRLIB ausgelieferten Dokumentationen und Datasheets. Für die GRLIB Version 1.0.14 sind die Wichtigsten, die:

„grip.pdf“ [gripdoc] ist eine Dokumentation zu allen GRLIB-IP-Cores. Da diese Datei, beim Umgang mit der GRLIB, immer „griffbereit“ sein muss, wurde sie in das „doc“-Verzeichnis des EDK LEON3 IP-Core-Verzeichnisses kopiert. Dadurch kann sie schnell mit einem Klick auf den Eintrag „View PDF Datasheet“ im „IP-Coremenü“ geöffnet werden.

„grlib.pdf“ [grlibdoc] beinhaltet eine allgemeine Dokumentation über die GRLIB, von der Installation bis zu den Konzepten

3.1.3.5 „lib“-Verzeichnis

Das Library-Verzeichnis, oder „lib“-Verzeichnis genannt, ist der Kern der GRLIB. Hier sind alle IP-Cores mit ihren Verhaltensbeschreibungen in VHDL abgelegt.

Die VHD-Dateien des LEON3-Prozessors befinden sich beispielsweise im Verzeichnis:

„[GRLIB_DIR]/lib/gaisler/leon3“

Die GR Ethernet Schnittstelle befindet sich im Verzeichnis:

„[GRLIB_DIR]/lib/gaisler/greth“

Der AHB, der durch eine einzelne VHD-Datei beschrieben wird, ist die Datei:

„[GRLIB_DIR]/lib/amba/ahbctrl.vhd“

Es ist zu erkennen, dass die Unterverzeichnisse des Library-Verzeichnisses hauptsächlich in Anbieter und Standards unterteilt sind. Die Unterverzeichnisse „esa“, „gleichmann“, „openchips“ und „opencores“ beinhalten beispielsweise VHD-Dateien von anderen Anbietern als der Firma GR. Andere Unterverzeichnisse sind „tech“ und „techmap“. Sie enthalten alle nötigen Pakete und Hardwarebeschreibungen, um die Technologie-Unabhängigkeit zu gewährleisten.

Eine weitere Betrachtung des Library-Verzeichnisses soll hier nicht stattfinden, da die EDK GRLIB-IP-Cores, bei ihrer Synthese, nicht auf dieses Verzeichnis zurückgreifen. Es war notwendig alle GRLIB VHD-Dateien aus dem Library Verzeichnis zu extrahieren und in eine neue Bibliothekenumgebung einzubetten. Dadurch erhielt die GRLIB eine bessere Zuordnung in ihre Verzeichnisse. Diese Prozess wird in Kapitel 5 beschrieben.

3.1.3.6 „software“-Verzeichnis

Dieses Verzeichnis enthält einige Programmbeispiele in der Programmiersprache C. Im Unterverzeichnis „leon3“ befinden sich zum Beispiel Programme zum Testen des LEON3 oder für den Umgang mit der APB Uart oder dem **G**eneral **P**urpose **I/O** Port (kurz GPIO). Der GPIO IP-Core übernimmt die Ansteuerung von LEDs und Push-Buttons.

Eine Aufbereitung der C-Programme wird in Kapitel 6 geschildert. Dort werden die Programmbestandteile eines LEON3-Programms vorgestellt, welche während der Testphase dieser Arbeit entstanden sind. Es wird vor allem deutlich, wie für das LEON3-System Ansteuerungen, der SoC-Komponenten, gelungen sind ohne spezielle Treiber zu verwenden.

3.1.4 Handhabung der GRLIB

In diesem Kapitelpunkt soll beschrieben werden, wie eine Synthese eines GRLIB-Systems, mit dem vorgegebenen Workflow, durchgeführt wird. Dabei werden nur, von der Firma GR zur Verfügung gestellte, Werkzeuge benutzt, es wird aber auch auf Möglichkeiten der Weiterverarbeitung mit dem Xilinx Werkzeug ISE hingewiesen.
Vorraussetzung für das Gelingen der Synthese ist die vorherige Installation einer VHDL Synthese Software, wie der XST. Die XST werden mit der Software ISE geliefert.
Ist die Installation der XST abgeschlossen, so sind die einzelnen Tools systemweit ausführbar. Dies wird von den MAKEFILES der GRLIB ausgenutzt, um eine Synthese in jedem Verzeichnis ausführen zu können.

3.1.4.1 Der Workflow von Gaisler Research

Der Benutzer führt nun folgende Schritte in der Linux-Konsole aus. Für Windows Benutzer erfüllt hier der Cygwin Emulator seinen Zweck.
Es wird darauf hingewiesen, dass nicht alle möglichen Optionen erklärt werden können, dafür wird auf die [grlibdoc] Dokumentation verwiesen.
Bevor der Benutzer die Synthese eines GRLIB-Systems beginnen kann, muss er sich entscheiden auf welchem Board das System betrieben werden soll. Für diese Arbeit stand das Digilent „XUP Virtex-II Pro Development Board“ zur Verfügung. Aus diesem Grund wird der Vorgang an dem „leon3-digilent-xup“ Template-Design erklärt.

Alle Schritte werden also nun im Verzeichnis „[GRLIB_DIR]/designs/leon3-digilent-xup“ ausgeführt.

Zur Erstellung eines Bitstreams aus der GRLIB sind folgende Schritte nötig:

Schritt 1: *Festlegen und Konfigurieren der SoC-Komponenten. Der Benutzer muss vor der Synthese entscheiden, welche SoC-Komponenten und in welchen Konfgiurationen benutzt werden sollen.*

Schritt 2: *Nach der Konfiguration kann die Synthese der GRLIB beginnen. Es wird ein Bitstream ohne BRAM Daten erstellt. Dieser Bitstream ist vergleichbar mit dem Ergebnis „system.bit“ des EDK Arbeitsschritts 2.*

Schritt 1 (Festlegen und Konfigurieren der SoC-Komponenten)

In diesem Schritt werden alle SoC-Komponenten, die in dem GRLIB-SoC vorhanden sein sollen, aus gewählt und konfiguriert.

Der Schritt wird mit dem Konsolen-Aufruf:

```
$ make xconfig
```

gestartet.

Es öffnet sich eine Tcl/tk GUI wie in Abbildung 3.3:

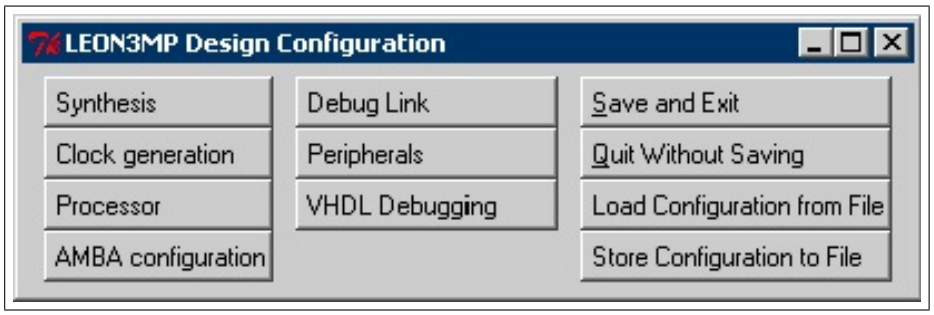

Abbildung 3.3: LEON3MP Design Configuration GUI

In den Sektionen der LEON3MP Design Konfiguration, Abbildung 3.3, können alle SoC-Komponenten hinzugefügt, entfernt und konfiguriert werden.

In der Sektion „**Synthesis**“ kann mit der Option „Target technology“ festgelegt werden, auf welchem FPGA das SoC betrieben werden soll. Für diese Arbeit wurde das SoC für den Xilinx-Virtex2 konfiguriert.

In der Sektion, die mit „**Clock generation**“ bezeichnet ist, wird festgelegt, mit welcher Frequenz das SoC betrieben werden soll. Im Standardfall wäre das 13/20 der Boardfrequenz. Für das XUP Board sind das 65 MHz.

Die „**Processor**“ Sektion umfasst alle Einstellungen des LEON3. Hier kann zum Beispiel die **I**nteger **U**nit (kurz IU), die **F**loating-**p**oint **U**nit (kurz FPU) und die **M**emory **M**anagement **U**nit (kurz MMU) ausgewählt und konfiguriert werden.

Zu Beachten 3.2 (FPU in der GRLIB)

Es ist darauf zu achten, dass die FPU des LEON3-Prozessors deaktiviert wird. Die FPU ist kein Bestandteil der GRLIB und unterliegt einer anderen Lizenz. Für SoCs der GRLIB ist der LEON3-Prozessor nur ohne FPU möglich.
Ist die FPU trotzdem aktiviert, führt dies zu einem Fehler während der Synthese des SoC.

Die „**AMBA configuration**“ Sektion umfasst alle Einstellungen des AHB.
Hier kann:

- der Index des Standard-Master am AHB festgelegt werden
- die Arbitrierungs-Strategie des AHB gewählt werden
- die Startadresse für die den E/A-Bereich des AHB eingegeben werden
- sowie sie Startadresse, des Adressraums, für den APB festgelegt werden

Eine Erklärung der Adressierung innerhalb der GRLIB-SoC-Komponenten wird in Kapitelpunkt 3.1.5.2 durchgeführt.

Die Sektion „**Debug Link**“ behandelt Wege, mit denen es möglich ist ein Debugging des SoC durchzuführen. Es stehen drei Möglichkeiten zur Verfügung:

- über die RS232 Schnittstelle des Boards
- über die JTAG Schnittstelle des Boards
- oder über die Ethernet MAC Schnittstelle des Boards

Das Debugging über die RS232 Schnittstelle setzt das Vorhandensein des „ahbuart“ IP-Cores voraus. Der IP-Core ist nicht sehr groß, d. h. die Fläche der logischen Schaltung umfasst ca. 430 Slices, jedoch ist die Kommunikation langsam.
Eine andere Möglichkeit bietet die Ethernet MAC Schnittstelle. Hier wird der „greth“ IP-Core benötigt. Das Debugging erfolgt schnell und einfach über den Austausch von UDP Paketen. Der „greth“ IP-Core verbraucht allerdings ca. 1800 Slices.

Die letzte Möglichkeit bietet die JTAG Schnittstelle des Boards. Bei einem MicroBlaze-System kann der Debuggingprozess mit der JTAG Schnittstelle über ein Parallel III oder IV Kabel, oder über die USB Schnittstelle stattfinden.

Leider kann das JTAG-Debugging eines GRLIB-SoC nur über das Parallel III oder IV Kabel durchgeführt werden, da die Firma Xilinx die API für ihre USB Treiber nicht veröffentlichen will[5]. Ein Ersatz für schnelles Debugging konnte somit nur die Ethernet Schnittstelle bieten. In dieser Arbeit konnten Debugging Prozesse mit einer Geschwindigkeit von bis zu 70 MBit/s, über die GR Ethernet Schnittstelle durchgeführt werden.
Weitere Einstellungen in der „Debug Link“ Sektion sind die MAC Adresse und die IP Adresse des „greth“ IP-Cores. Diese Einstellungen gelten nur für Debugging Zwecke.

Die „**Peripherals**“ Sektion erlaubt dem Benutzer die Möglichkeit IP-Cores hinzuzufügen, welche Schnittstellen zu Peripheriekomponenten auf dem FPGA-Board darstellen. Zu diesem Komponenten gehören:

- der DDR SDRAM Controller, bindet die „ddrspa“-Entity ein
- der On-Chip ROM/RAM IP-Core, bindet die „ahbrom“- oder „ahbram“-Entity ein
- die Ethernet Schnittstelle, bindet die „greth“-Entity ein
- die Uart, Timer und Interrupt-Controller Komponenten, bindet die „apbuart“, „gptimer“ oder „irqmp“-Entitys ein
- die Keyboard, Maus und VGA Komponenten, bindet die „apbps2“, „apbvga“ oder „svgactrl“-Entitys ein

Eine nähere Erläuterung wichtiger GRLIB-IP-Cores erfolgt in Kapitelpunkt 5.5.

Nach dem alle Konfigurationen abgeschlossen sind, werden die Einstellungen der Tcl/tk GUI mit „**Save and Exit**“ in der Datei „config.vhd“ gespeichert.

Die Ausgabe:

```
$ config.vhd created
```

zeigt, dass nur die Datei „config.vhd“ verändert wurde. Das Einbinden und Konfigurieren des Template-Designs hat also nur Einfluss auf die Konstantenwerte im Paket „config.vhd“. Mit diesem Konstanten werden „gererate“ Konstrukte gesteuert und Generic-Konstanten der IP-Cores festgelegt.

[5] Dies ist eine Information aus der LEON SPARC YAHOO Group von Jiri Gaisler, dem Autor der GRLIB.

```
ua1 : if CFG_UART1_ENABLE /= 0 generate
   uart1 : apbuart                          -- UART 1
     generic map (pindex   => 1, paddr => 1, pirq => 2,
                  console => dbguart, fifosize => CFG_UART1_FIFO)
     port map (rstn, clkm, apbi, apbo(1), u1i, u1o);
   u1i.rxd <= rxd; u1i.ctsn <= '0'; u1i.extclk <= '0';
 end generate;
```

Abbildung 3.4: Ausschnitt der „leon3mp“-Entity, APB Uart Einbindung

In Abbildung 3.4 ist zu erkennen, wenn die „CFG_UART1_ENABLE“ Konstante des „config.vhd“ Package (Abbildung 3.2) entsprechend festgelegt ist, wird die APB Uart SoC-Komponente eingebunden oder nicht eingebunden.

Schritt 2 (Synthetisieren des erstellten SoC)

Dieser Schritt ist vergleichbar mit dem Arbeitsschritt 2 des EDK-Arbeitsplans (Kapitelpunkt 2.2.1.2) und generiert einen Bitstream, welcher in der Datei „leon3mp.bit“ abgespeichert wird.
Es werden die Dateien „leon3mp.vhd“, „config.vhd“, „ahbrom.vhd“ und „leon3mp.ucf“, sowie alle VHD-Dateien des GRLIB „lib“-Verzeichnisses der Synthese übergeben. Der Benutzer arbeitet den Schritt 2 wie folgt ab. Eine Eingabe in der Linux Konsole von:

```
$ make xgrlib
```

öffnet abermals eine Tcl/tk GUI, mit der die Synthese gesteuert werden kann:

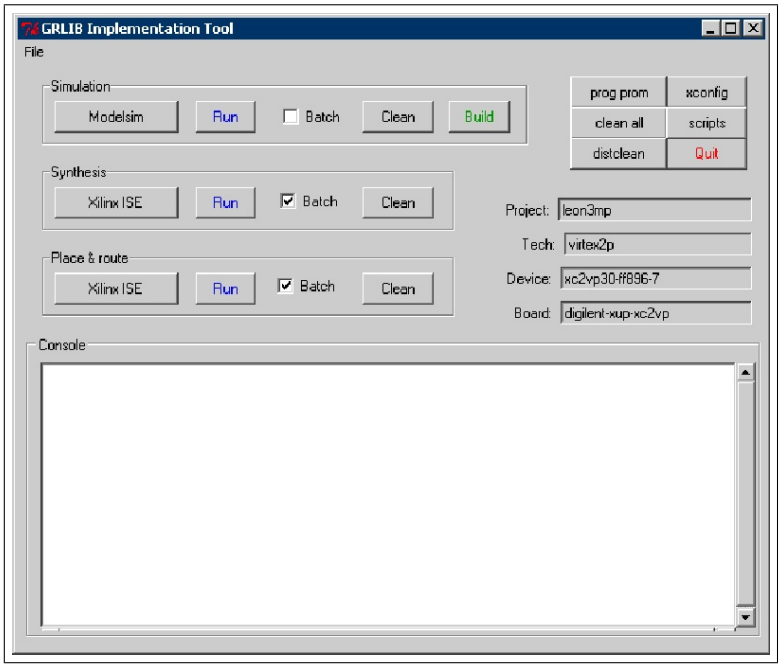

Abbildung 3.5: „GRLIB Implementation Tools“ GUI zur Synthese der eines GRLIB-SoC

Die Abbildung 3.5 zeigt die „GRLIB Implementation Tools“ GUI. Die Elemente der GUI sind in der Abbildung so eingestellt wie sie in dieser Arbeit benutzt wurden. Am rechten Rand wird eine kurze Zusammenfassung des Projektes angezeigt.

Es ist zu erkennen, dass das GRLIB Implementation Tool eine Aufspaltung des Schritts 2 in die Teilschritte „**Synthesis**“ und „**Place & route**“ vornimmt. Bei dem Vergleich der 2 Teilschritte mit dem Arbeitsschritt 2 des EDK-Arbeitsplans fällt auf, dass die Teilschritte „Verifikation der Constraints“ und „Erstellen des Bitstreams“ nicht aufgelistet werden. Die fehlenden Teilschritte werden automatisch nach dem erfolgreichem „Place & route“ Teilschritt ausgeführt.

- ***„Synthesis“***

Als 1. Teilschritt muss die Synthese der „leon3mp“-Entity durchgeführt werden. Dazu wurde in dieser Arbeit die XST verwendet. Im GRLIB Implementation Tool muss nun für die Synthese, mit XST, die Zeichenkette „Xilinx XST“ gewählt werden. Ebenfall muss ein Häkchen bei „Batch“ gesetzt werden, damit alle Ausgaben auf der Konsole des GRLIB Implementation Tools angezeigt werden.
Ein Klick auf den „Run“-Button startet den Teilschritt. Durch einen Klick auf den Button „Clean“ werden alle, bei diesem Teilschritt, erstellten Dateien gelöscht.

- ***„Place & route“***

Nachdem alle Entitys synthetisiert wurden, müssen sie nun auf dem FPGA platziert und verdrahtet werden. Dies ist der 2. und letzte Teilschritt.
Zur Durchführung müssen wieder die XST mit der Zeichenkette „Xilinx ISE“ ausgewählt werden. Die anderen GUI Elemente in der Sektion „Place & route“ verhalten sich analog, wie in der „Synthesis“ Sektion, nur auf den Teilschritt „Place & route“ bezogen.

Sind alle zwei Teilschritte erfolgreich abgeschlossen, so befindet sich nun ein Bitstream namens „leon3mp.bit“ im Template-Design Verzeichnis „leon3-digilent-xup“. Dieser Bitstream kann nun auf dem vorgesehenen FPGA-Board, in diesem Fall das XUP-Board, konfiguriert werden.

Damit ist der von GR vorgesehene Workflow zur Erstellung eines GRLIB-SoC beendet.

3.1.4.2 Der Workflow mit Xilinx ISE

Es gibt eine weitere Möglichkeit eine Synthese des gewünschten Template-Designs durchzuführen. Dieser Weg benutzt aktiv die Xilinx ISE GUI. Alle Konsolenkommandos werden wieder im Template-Design Verzeichnis ausgeführt, also für das XUP-Board im „[GRLIB_DIR]/designs/leon3-digilent-xup“-Verzeichnis. Zur Zusammenstellung und zur Konfiguration des GRLIB-SoC muss der Schritt 1 (Kapitelpunkt 3.1.4.1) ausgeführt werden. Lediglich der Schritt 2 kann nun abweichend, wie folgt ausgeführt werden.

Mit dem Konsolenkommando:

```
$ make scripts
```

werden Projektdateien generiert, die zum Beispiel mit dem Werkzeug Xilinx ISE geöffnet werden können. Bei der Ausführung des Kommandos sollte angezeigt werden, dass ein Scannen des „lib“-Verzeichnisses durchgeführt wird.

Dies ist zu erkennen an der Ausgabe, die in Abbildung 3.6 dargestellt ist:

```
Scanning libraries

    grlib
    synplify
    unisim
    simprim
    dw02
    techmap
    fpu
    gaisler
    esa
    hynix
    micron
    work
```

Abbildung 3.6: Scannen des „lib“-Verzeichnisses zur Erstellung der Projektdateien

Wird diese Ausgabe nicht erzeugt, so kann es sein, dass keine Projektdateien generiert wurden. In diesem Fall sollte das Kommando „`$ make clean`“ ausgeführt werden. Damit wird das Template-Design in seinen Ausgangzustand versetzt und alle generierten Dateien gelöscht.

Wurden die Projektdateien erfolgreich generiert, sollten sich einige „*.npl“ Dateien im Template-Design Verzeichnis befinden.[6] Wurde die Datei „leon3mp.ise“ generiert, dies ist bei der GRLIB 1.0.14 der Fall, so ist dies eine Projektdatei speziell für das Xilinx ISE 7.1.
Werden die Projekte nun im ISE geöffnet ist zu bemerken, dass der Ladevorgang überdurchschnittlich viel Zeit benötigt. Dies liegt daran, dass alle VHD-Dateien aus dem GRLIB „lib“-Verzeichnis als Bibliotheken dem ISE-Projekt hinzugefügt wurden.
Dies muss getan werden, da durch die Einbindung der SoC-Komponenten mittels der „generate“-Konstrukte erst bei der Synthese entschieden wird, welche VHD-Dateien benötigt werden.

Zu Beachten 3.3 (GRLIB Systhese mit Xilinx ISE)

*Bei der Synthese mit dem ISE muss darauf geachtet werden, dass die XST Synthese Option „**-fsm_extract no**“ in den Eigenschaften des „Synthesize - XST“ Prozesses angegeben wird. Dies ist ein ausdrücklicher Hinweis der GRLIB. Dadurch werden keine Finite State Machines extrahiert.*

[6] Auf einem Linux-System sollte die Datei „leon3mp.npl“ geöffnet werden, auf einem Windows-System jedoch die Datei „leon3mp_win32.npl“.

- ***Zusammenfassung***

Es lässt sich über den Workflow mit dem ISE sagen, dass er unakzeptabel langsam ist. Des weiteren muss, bei einer Änderung einer beliebigen Entity, das ganze SoC komplett neu synthetisiert werden.
Dies ist bei der Arbeit mit dem EDK nicht der Fall. Wird hier die Entity einer SoC-Komponente geändert, so muss nur der IP-Core der geänderten SoC-Komponente neu synthetisiert werden. Dies kann bei einem größeren GRLIB-SoC eine Zeitersparnis von 10 bis 20 Minuten bedeuten. Die Zeitangaben haben natürlich eine starke Abhängigkeit vom Host-System.

3.1.5 Design-Konzepte der GRLIB

In diesem Kapitel werden die wichtigsten Design-Konzepte der GRLIB erklärt. Im Mittelpunkt steht dabei das Herzstück eines GRLIB-SoC, der AMBA **A**dvanced **H**igh-performance **B**us (kurz AHB). Dieser Bus ist ein AMBA 2.0 Standard und regelt die Kommunikation zwischen SoC-Komponenten eines GRLIB-Systems. Des weiteren wird der AMBA **A**dvanced **P**eripheral **B**us (kurz APB) besprochen, der als Slave an den AHB angeschlossen wird und die Kommunikation mit den Peripheriekomponenten des SoC übernimmt, die weniger Durchsatzrate benötigen.
Der AHB wird hauptsächlich in Verbindung mit dem ARM Prozessor verwendet.

Man sagt, dass der Bus das Rückgrad des Systems sei. Dieser Ausdruck trifft auf ein GRLIB-SoC vollständig zu. Der AHB bestimmt die maximale Übertragungsrate von Informationen, die auf einem GRLIB-SoC möglich ist. Der LEON3-Prozessor spielt in einem GRLIB-SoC eine untergeordnete Rolle. Er ist nur eine Entity, die als Master an den AHB angeschlossen wird. Zur Untermauerung dieses Zustandes werden zwei Beispiele mit Erläuterungen gegeben:

Beispiel 1: Der Debugging Prozess eines MicroBlaze oder PowerPC Systems erfolgt durch die SoC-Komponenten „**M**icroprocessor **D**ebug **M**odule (kurz MDM)“ oder „JTAGPPC Controller“. Mittels dieser IP-Cores ist es möglich auf den Systemspeicher des jeweiligen Systems zuzugreifen.
Allerdings funktioniert der Debugging Prozess nur bei einem vorhandenen Prozessor.

Dies ist bei einem GRLIB-System nicht der Fall. Das Debugging kann trotz fehlenden LEON3-Prozessor durchgeführt werden. Dies wird möglich, da die drei IP-Cores zum Debugging (AHB Debug Uart, GR Ethernet mit EDCL, AHB JTAG) selbst Masters auf dem AHB sind. Durch diese Verallgemeinerung unterscheiden sich, von der AHB Seite betrachtet, die drei IP-Cores nicht von einem LEON3-Prozessor. Über das AHB-Master-Interface können

die Debug-IP-Cores selbstständig Lese- oder Schreibzugriffe auf dem Systemspeicher ausführen.

Eine genauere Beschreibung der Debug-IP-Cores wird in Kapitelpunkt 5.5 ausgeführt.

Beispiel 2: Als zweites Beispiel soll kurz der GRLIB-IP-Core „svgactrl“ erläutert werden, auch benannt als GRLIB SVGA Controller.
Der GRLIB SVGA Controller benutzt zur Speicherung der Bildpixelinformationen einen Framebuffer. Da der Framebuffer eine Größe von mehreren Megabyte haben kann, muss der GRLIB SVGA Controller auf einen Speicherbereich zurückgreifen, der außerhalb des IP-Cores liegt. Ein möglicher Speicher wäre der DDR SDRAM.

Nun gibt es zwei Möglichkeiten wie die Framedaten aus dem DDR SDRAM in die Register des GRLIB SVGA Controllers gelangen können.

Die **erste Möglichkeit** ist der LEON3-Prozessor könnte die Famedaten aus dem DDR DSRAM holen und an das GRLIB SVGA Kontroller AHB-Slave-Inferface schicken.

Eine **zweite Möglichkeit** wäre, dass der GRLIB SVGA Controller die Framedaten selbst über ein AHB-Master-Interface einliest. Damit würde der LEON3-Prozessor entlastet werden und würde für die Bilddarstellung nicht mehr benötigt.

Die zweite Möglichkeit wurde in der GRLIB umgesetzt. Somit ist die Darstellung von Bildern auf dem Monitor vom Prozessor unabhängig. Letztendlich werden die Framebufferdaten zwar von LEON3 berechnet und im Framebuffer abgespeichert, aber dies könnte auch von einer kleinen, einfachen Steuereinheit erledigt werden. Insgesamt fallen weniger Aufgaben an den LEON3.

Der GRLIB SVGA IP-Core wird in Kapitelpunkt 5.5 genauer betrachtet.

Der AHB und APB sind AMBA 2.0 Standards. Dieser Standard soll in Kapitelpunkt 3.1.5.1 kurz vorgestellt werden. Darauf folg eine Vorstellung der Busse AHB und APB, die in der GRLIB Anwendung gefunden haben.
Es ist wichtig die Funktionsweise der GRLIB Busse zu verstehen, sowie deren Interfaces zu den SoC-Komponenten, da es nötig sein kann weitere IP-Cores der GRLIB, selbst in das EDK einzubinden. Der Vorgang des Einbindens wird in Kapitelpunkt 5.2 behandelt.

3.1.5.1 AMBA Standard

Dieser Kapitelpunkt behandelt einige Auszüge aus der AMBA 2.0 Spezifikation [ambaspec]. Der **A**dvanced **M**icrocontroller **B**us **A**rchitecture 2.0 (kurz AMBA) Standard definiert die folgenden drei Busse:

- den Advanced High-performance Bus (AHB)
- den Advanced System Bus (ASP)
- und den Advanced Peripheral Bus (APB)

Die AMBA Busse werden auf SoCs verwendet. Sie verbinden die SoC-Komponenten und bewerkstelligen deren Kommunikation untereinander.

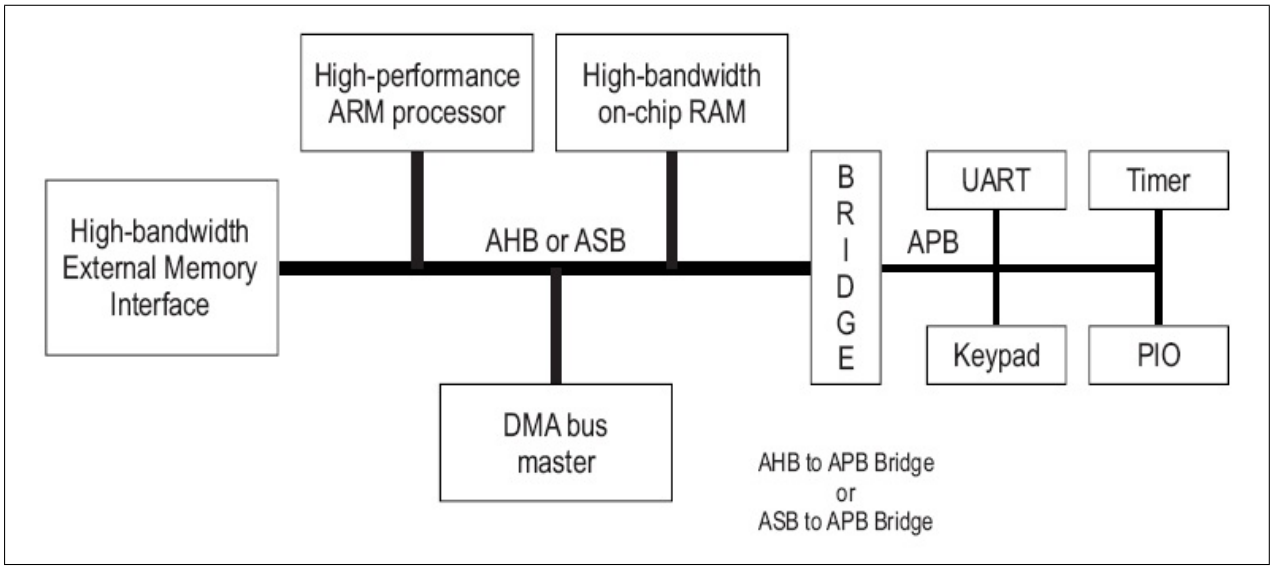

Abbildung 3.7: SoC mit dem AMBA Bussen [ambaspec]

In der Abbildung 3.7 wird eine Übersicht, über alle AMBA Busse, gegeben. Es ist zu erkennen, das der vorgesehene Prozessor der ARM Prozessor ist. Dies muss allerdings nicht allgemein gültig sein.

- ***ASP***

Der ASP ist ein hoch hochleistungsfähiger Bus. Er ist vergleichbar mit dem AHB. Der ASB ist ein Bus der älteren Generation und verfügt nicht über alle Funktionalitäten des AHB. Da der ASP nicht in der GRLIB umgesetzt wurde, soll er in dieser Arbeit nicht weiter behandelt werden. Bei Interesse sei auf die AMBA 2.0 Spezifikation [ambaspec] verwiesen.

Richtlinien

Die AMBA-Spezifikation wurde entwickelt unter der Berücksichtigung der folgenden vier Richtlinien:

- Die Spezifikation soll technologieunabhängig sein und einen hohen Grad an Mehrfachverwendbarkeit haben.
- Es sollen hoch modularisierte Designs erzeugt werden und die Prozessorunabhängigkeit ausgebaut werden.
- Weiter sollen die Ressourcenanforderungen so gering wie möglich gehalten werden und effiziente On-Chip und Off-Chip Kommunikation ermöglich werden.
- Es soll die „Right-First-Time“ Entwicklung von SoCs mit einem oder mehreren CPUs oder Signalprozessoren erleichtert werden.

Die Definition des Begriffes „Right-First-Time“ ist laut des Online Lexikons „onpulson“:

> *„Right-First-Time ist ein Konzeptbestandteil des Total-Quality-Management, bei dem es eine Verpflichtung gegenüber Kunden gibt, keine Fehler zu machen. Dieser Ansatz verlangt von Mitarbeitern auf allen Ebenen, sich dazu zu verpflichten und die Verantwortung zu übernehmen, dieses Ziel zu erreichen. Qualitätszirkel werden bei diesem Verfahren manchmal als Hilfsmittel eingesetzt.“* *[onpulhp]*

Spezifikation

Die Spezifikation der AMBA Busse umfasst folgende Punkte:

a) Technologie-Unabhängigkeit

b) elektrische Charakteristika

c) Timing Spezifikation

- ***a) Technologie-Unabhängigkeit***

Die AMBA-Spezifikation ist eine technologieunabhängige Festlegung von Busprotokollen. Dabei wird der Pegel-Verlauf der einzelnen Bus Ein- und Ausgängsports während eines Bus-Clock-Zyklus betrachtet.

- ***b) elektrische Charakteristika***

Es werden ausdrücklich keine Festlegungen über physikalische Größen (Spannung, Frequenz) der Signale gemacht, da diese Technologie-abhängig sind und vom Herstellungsprozess abhängen.

- ***c) Timing Spezifikation***

Das AMBA Bus Protokoll definiert das Verhalten der Signale während eines Bus-Clock-Zyklus. Allerdings werden keine Festlegungen über den exakten Verlauf der Signale gemacht. Dadurch wird eine maximale Flexibilität gewährt.

Für den Punkt c) der AMBE 2.0 Spezifikation bedarf es einiger zusätzlicher Erklärung.

Der exakte Verlauf der Signalpegel, einer synthetisierten logischen Schaltung, wird durch den Wechsel der logischen Pegel an ihren Ein- und Ausgängen zu einem bestimmten Zeitpunkt festgelegt. Dieser Wechsel geschieht allerdings nicht immer exakt zum Zeitpunkt der auftretenden Taktflanke. Durch die technologische Umsetzung der Schaltung können folgende Erscheinungen auftreten:

- Setup time / Hold time
- Propagation delay time
- Rise time / Fall time

Setup time (t_{setup} oder t_s): Ist die Zeit, die ein Signal A stabil und unverändert vor einem Signal B anliegen muss.

Hold time (t_{hold} oder t_h): Ist die Zeit die ein Signal A, nach einem Wechsel eines Signals B, noch stabil und unverändert anliegen muss.

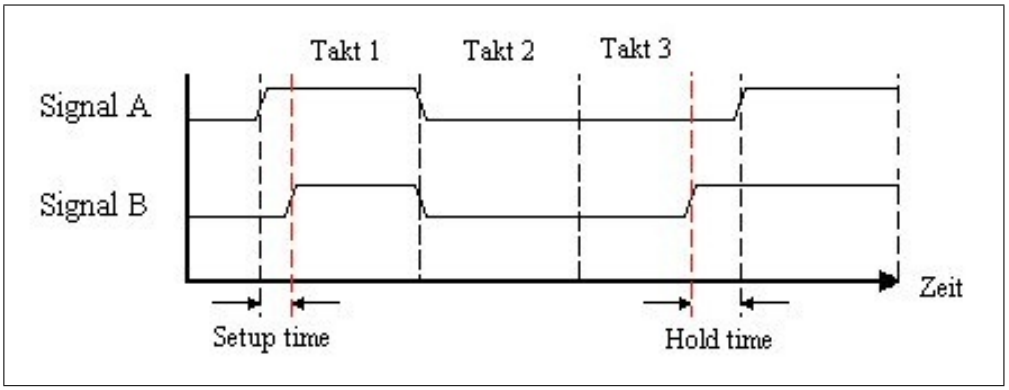

Abbildung 3.8: Setup time und Hold time in einem Timing-Diagramm

Die Setup time und Hold time werden in Abbildung 3.8 in einem Timing-Diagramm veranschaulicht. Sie haben nichts mit Laufzeitunterschieden der logischen Schaltung zu tun. Die Zeiten können lediglich für eine logische Schaltung vorgegeben sein, damit diese korrekt funktioniert. Die Setup time und Hold time sind bei dem GR Ethernet IP-Core für bestimmte Ein- und Ausgänge zu beachten. Die Zeiten werden mittels der UCF geprüft.

Propagation delay time (t_{pd}): Ist die Zeit die verstreicht, vom Anlegen eines stabilen Signals an den Eingang eines Elements bis zur Durchsetzung des Ergebnissignals am Ausgang des Elementes.

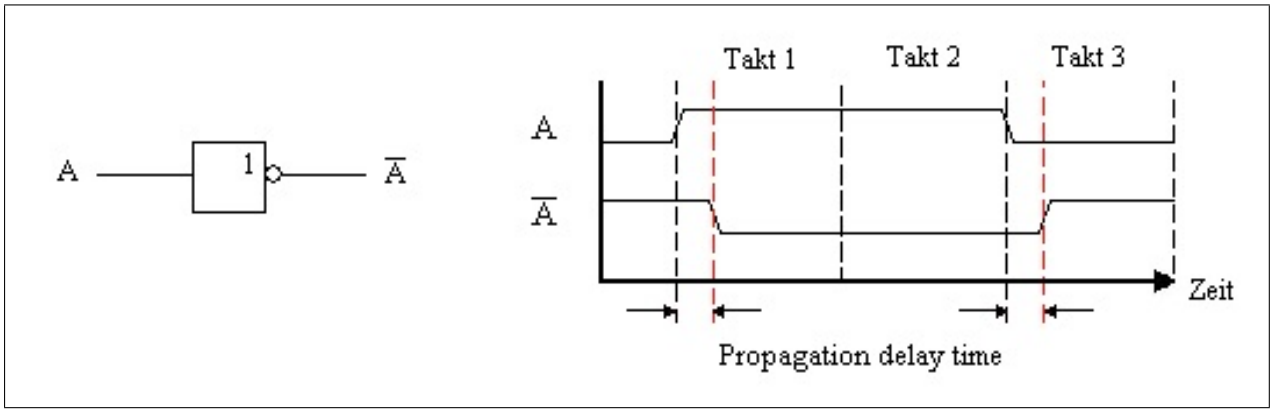

Abbildung 3.9: Propagation delay time, eines NOT, in einem Timing-Diagramm

Die Abbildung 3.9 zeigt die Propagation delay time eines logischen „NOT“. Die Propagation delay time besteht aus der Laufzeit des Signals durch ein Element und der vernachlässigbar kleinen Ausbreitungszeit des Signals. Die Ausbreitungszeit ist durch die relative Elektrische und relative Magnetische Feldkonstante der Verdrahtung bedingt.
Wird beispielsweise innerhalb einer FPGA-Schaltung ein **IO-Buf**fer (kurz IOBUF) verwendet, entsteht eine t_{pd} von 1 bis 2 ns.

Rise time (t_{rise} oder t_r): Ist die Zeit, die von einem Signal benötigt wird um von einem stabilen Low-Pegel auf einen stabilen High-Pegel zu steigen.

Fall time (t_{fall} oder t_f): Ist die Zeit, die von einem Signal benötigt wird um von einem stabilen High-Pegel auf einen stabilen Low-Pegel zu sinken.

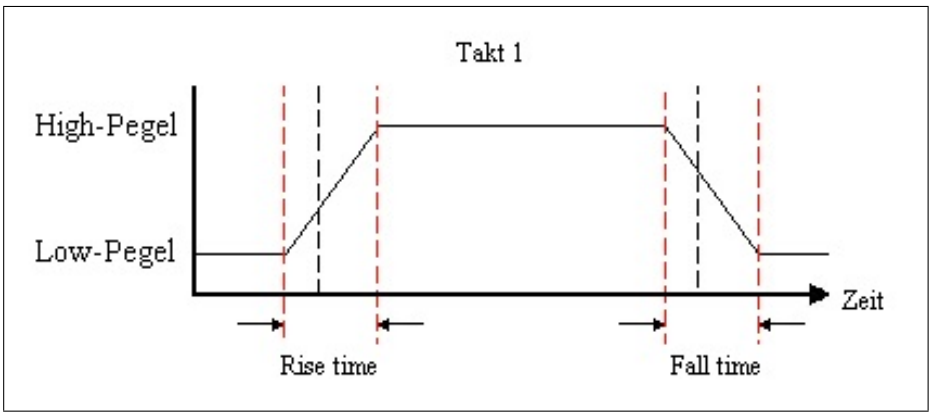

Abbildung 3.10: Rise time und Fall time in einem Timing-Diagramm

Abbildung 3.10 zeigt ein Timing-Diagramm stark vergrößert, so dass die schematische Darstellung der Pegelwechsel-Flanken zu erkennen ist. Der Anstieg der steigenden und fallenden Flanken nicht unendlich steil, sondern erfolgt mit exponentiellem Verlauf. Dadurch benötigen die Signale eine gewisse Zeit, damit sich ihr Pegel so weit angeglichen hat, um als stabil zu gelten.
Aus diesem Grund ist es wichtig bei einer Taktfrequenztransformation, auf einem FPGA, einen **D**igital **C**lock **M**anager (kurz DCM) zu verwenden, da dieser einen optimalen Flankenverlauf gewährleistet.

Die AMBA 2.0 definiert nun allgemeine Festlegungen, über den zeitlichen Verlauf der Signale, um das AMBA Protokoll zu erfüllen. Außerdem werden alle Signale definiert, die ein AHB-Master oder AHB-Slave, sowie ein APB-Slave treiben muss, um vollwertig an den Bus angeschlossen werden zu können. Diese Signale sind bei der Umsetzung in VHDL die Ein- und Ausgangsports einer VHDL Entity.

Die Definitionen der AMBA-Spezifikation sollen nun in den Kapitelpunkten 3.1.5.2 und 3.1.5.3 direkt an den GRLIB Umsetzungen des AHB und APB erklärt werden. Es werden speziell auf Unterschiede zu dem OPB aufmerksam gemacht, da das ein Hauptsystembus eines MicroBlaze-Systems ist.

3.1.5.2 AMBA Advanced High-performance Bus (AHB)

Der **A**dvanced **H**igh-performance **B**us (kurz AHB) ist der Haupt-System-Bus eines GRLIB-SoC. Dieser Kapitelpunkt behandelt eine spezielle GRLIB Umsetzung des AHB, aus der AMBA 2.0 Spezifikation.

Der AHB ist ein hochleistungsfähiger SoC Bus, der maximal 16 Masters und 16 Slaves verbinden kann. Er verfügt über eine Plug and Play Funktionalität und wurde mit einer Interrupt-Steuerung versehen.

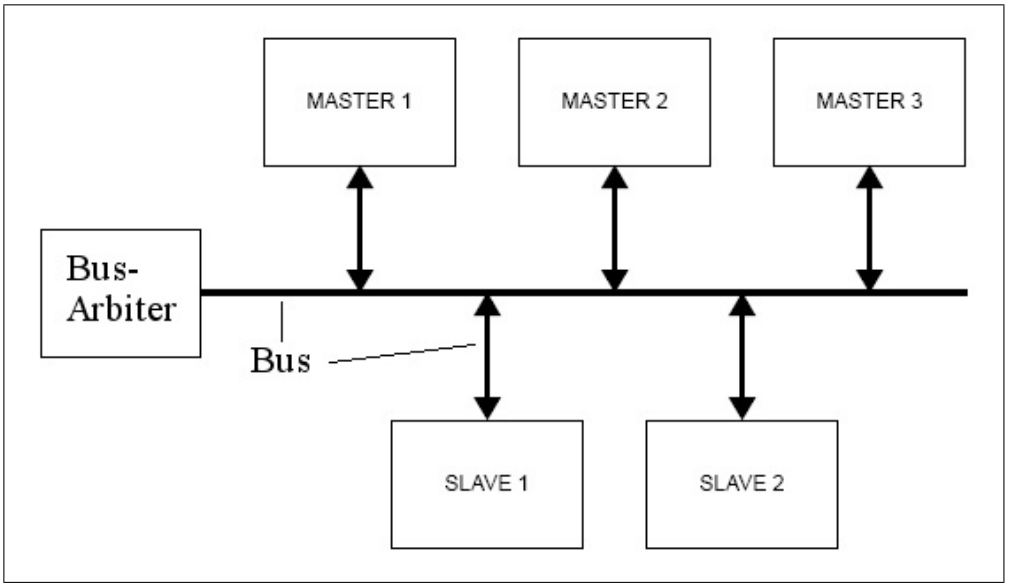

Abbildung 3.11: AHB Schema mit Masters und Slaves [grlibdoc]

Die Abbildung 3.11 zeigt den schematischen Aufbau eines SoC aus 3 Masters und 2 Slaves, die um den AHB angeordnet sind. Die waagerechte Linie sowie die Doppelpfeile sind der Bus, also die Verdrahtung. Der „Bus-Arbiter“ ist auch eine Entity.

Einen besseren Blick auf die Umsetzung zeigt die Abbildung 3.12:

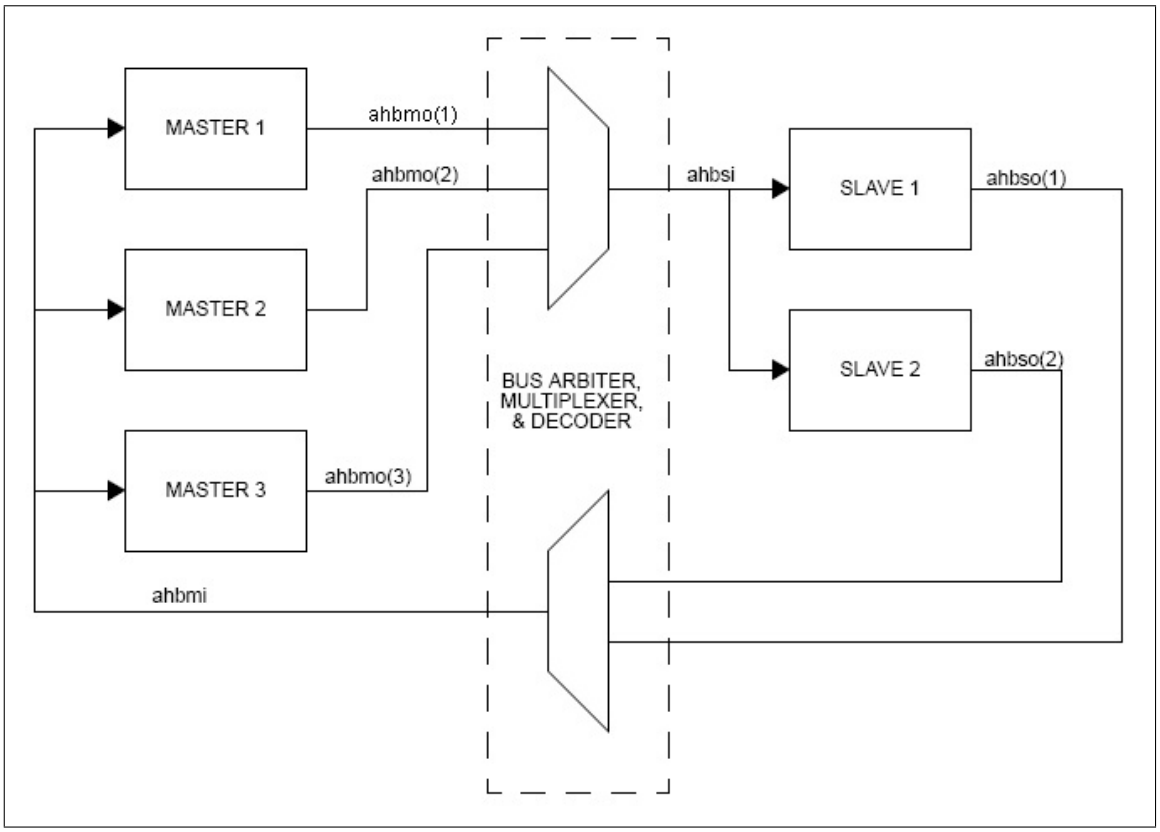

Abbildung 3.12: Verbindungen des AHB [grlibdoc]

In Abbildung 3.12 ist zu erkennen, dass der AHB-Arbiter als Multiplexer dient. Er verbindet den aktiven Master mit einer Slave Komponenten und treibt die Ausgabesignale eines Slave an einen Master weiter. Jede SoC-Komponente, die an AHB angeschlossen werden soll, muss eine bestimmte Menge von Signalen treiben oder lesen, d.h. die Entity der SoC-Komponente muss bestimmte Ports besitzen.

Das AHB-Master-Interface

Die AHB-Master Entity muss die Signalmenge „ahbm“ als Ports einbinden. Die „ahbm“-Signalmenge besteht aus der Vereinigung der Ausgangsportmenge „ahbmo“ und der Eingangsportmenge „ahbmi“. Dabei wird nur eine Signalmenge „ahbmi“ aus dem Bus an alle AHB-Master getrieben. Die Ausgangsportmengen aller Masters („ahbmo“), werden zu einem Signalvektor „ahbmo(*)“ zusammengefasst. Dieser Vektor ist dann eine Signalmenge, die den Eingangsport des AHB treibt.

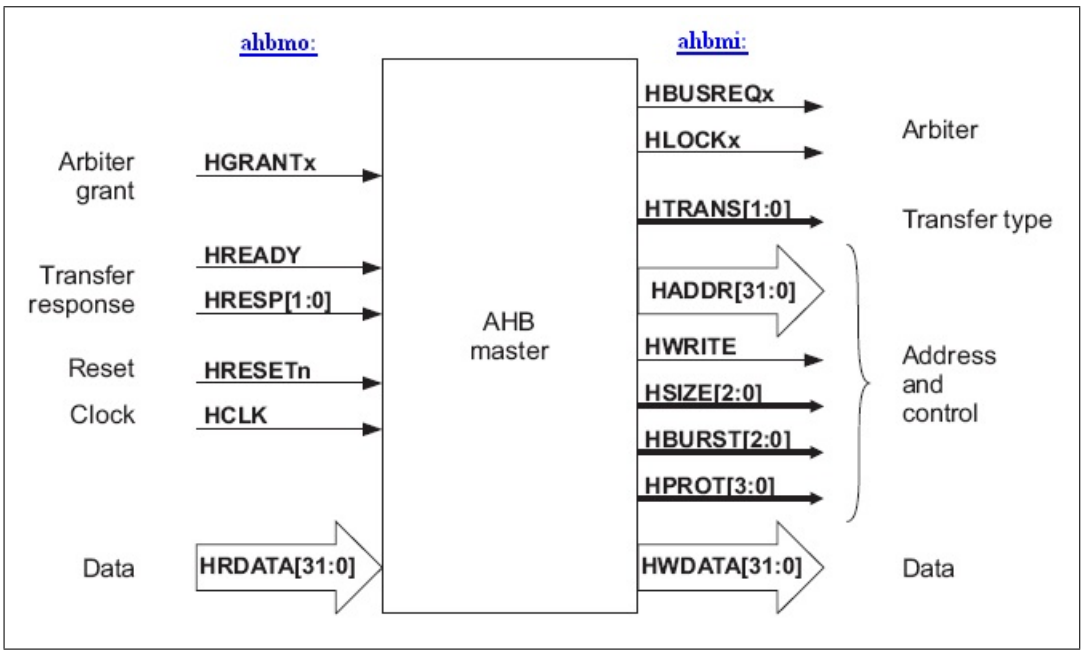

Abbildung 3.13: AHB-Master Eingangs- und Ausgangsports nach [ambaspec]

In der Abbildung 3.13 sind die Signalmengen aufgeschlüsselt. Links die Elemente der Eingangsportmenge („ahbmi“) für einen AHB-Master. Rechts die Elemente der Ausgangsportmenge („ahbmo“) für einen AHB-Master.

Die Namensbedeutungen der Signalmengen sind:

- „**ahbmo**“ - AHB-Master Output
- „**ahbmi**“ - AHB-Master Input

Die genannten Signalmengen werden in dem GRLIB VHDL Paket „[GRLIB_DIR]/lib/grlib/amba/amba.vhd“ als Records „ahb_mst_out_type“ und „ahb_mst_in_type“ definiert:

- ***AHB-Master Output Ports***

```
type ahb_mst_out_type is record
  hbusreq : std_ulogic;
  hlock   : std_ulogic;
  htrans  : std_logic_vector(1 downto 0);
  haddr   : std_logic_vector(31 downto 0);
  hwrite  : std_ulogic;
  hsize   : std_logic_vector(2 downto 0);
  hburst  : std_logic_vector(2 downto 0);
  hprot   : std_logic_vector(3 downto 0);
  hwdata  : std_logic_vector(31 downto 0);
  hirq    : std_logic_vector(NAHBIRQ-1 downto 0);
  hconfig : ahb_config_type;
  hindex  : integer range 0 to NAHBMST-1;
end record;
```

Abbildung 3.14: AHB-Master Output Record der GRLIB

Abbildung 3.14 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „ahbmo“-Signalmenge definiert.

Signalname	Bedeutung
hbusreq	Der Master erwünscht Zugriff auf den Bus. Vergleichbar mit dem „M_request“-Port eines OPB-Master.
hlock	Der Master wünscht den Bus für seine Zwecke zu sperren. Vergleichbar mit dem „M_busLock“-Port eines OPB-Master.
htrans	Der Master führt einen bestimmten Transfer durch (sequentiell, nicht-sequentiell, ...)
haddr	Systemadresse mit der der Master kommunizieren will. Vergleichbar mit dem „M_ABus“-Port eines OPB-Masters.
hwrite	Signalisiert ob der Master lesen oder schreiben will. Vergleichbar mit dem "´OPB_RNW“-Port eines OPB-Master.
hsize	Zeigt an wie viele Bits, der Master, übertragen will. Bis 1024 Bits sind möglich.
hburst	Gibt die Länge eines Burst-Transfers an
hprot	Gibt die Bedeutung der zu transferierenden Daten an (Operationscode, Datenzugriff, ...). Wurde für Komponenten implementiert, die einen bestimmten Schutz der Daten vornehmen wollen.
hwdata	Gibt die zu transferierenden Daten an. In der GRLIB nur 32 Bit, kann aber laut „hsize“ und AMBA-Spezifikation bis 1024 Bit betragen. Vergleichbar mit dem „M_DBus“-Port eines OPB-Masters.
hirq	Ist ein Signalvektor von 32 Bit. Hier kann der Master ein bestimmtes Signal treiben, um einen Interrupt auszulösen. Dieser Port wird nicht in der AMBA spezifiziert.
hconfig	Mit diesem Port übermittelt der Master seine Konfigurationsdaten, wie (Systemadresse, benutzte Interrupt-Nummer, Adressmaske...), an den AHB. Dieser Port wird nicht in der AMBA spezifiziert.
hindex	Übermittelt die Master-Index-Nummer den AHB. Dies ist eine Nummer die dem Vektorindex des „ahbmo(*)“-Signalvektors entspricht, in den die Ausgangsports des Masters münden. Dieser Port wird nicht in der AMBA spezifiziert.

Tabelle 3.2: Signale des „ahb_mst_out_type“ Record und ihre Bedeutung

Tabelle 3.2 zeigt alle Siganle des „ahb_mst_out_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle 3.2 wird mit „ahbmo.*Signalname*“ als Port bezeichnet.

- ***AHB-Master Input Ports***

```
type ahb_mst_in_type is record
  hgrant   : std_logic_vector(0 to NAHBMST-1);
  hready   : std_ulogic;
  hresp    : std_logic_vector(1 downto 0);
  hrdata   : std_logic_vector(31 downto 0);
  hcache   : std_ulogic;
  hirq     : std_logic_vector(NAHBIRQ-1 downto 0);
end record;
```

Abbildung 3.15: AHB-Master Input Record der GRLIB

Abbildung 3.15 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „ahbmi“-Signalmenge definiert.

Signalname	Bedeutung
hgrant	Mit diesem Port signalisiert der AHB-Arbiter dem Master, dass er Zugriff auf den Bus hat. Vergleichbar mit dem „OPB_Mgrant“-Port eines OPB-Master.
hready	Der AHB signalisiert dem Master, dass der Transfer zu einem Slave abgeschlossen ist. Der Port wird durch einen AHB-Slave getrieben.
hresp	Zeigt den Status eines Transfers an (Okay, Fehler, Wiederholen, Spalten). Port wird durch einen AHB-Slave getrieben.
hrdata	Hier werden die Daten eines Slaves während eines Transfers angelegt. Vergleichbar mit dem „OPB_DBus“-Port eines OPB-Master.
hcache	Zeigt an ob die Daten cacheable sind. Dieser Port wird nicht in der AMBA spezifiziert.
hirq	Mit diesem Port kann der Master einsehen, welche Interrupts gerade getrieben werden. Dieser Port wird nicht in der AMBA spezifiziert.

Tabelle 3.3: Signale des „ahb_mst_in_type“ Record und ihre Bedeutung

Tabelle 3.3 zeigt alle Siganle des „ahb_mst_in_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle 3.3 wird mit „ahbmi.*Signalname*“ als Port bezeichnet.

Zu Beachten 3.4 (kein Timeout bei AHB und APB)

Es ist zu erkennen, dass ein AHB-Master kein Signal und keine Codierung des „hresp“-Ports besitz, um einen Transfer-Timeout zu erkennen.
Dies liegt daran, dass der AHB sowie der APB keine Unterstützung für einen Transfer-Timeout besitzen. Sollte ein AHB-Master scheinbar stehen, so kann es also sein, dass immer noch auf ein Transfer-Acknowledgement durch die Überwachung des „hready“ Eingangsports gewartet wird.

- ***Anbindung als AHB-Master***

Soll es nun möglich sein eine Entity als AHB-Master mit dem AHB zu verbinden, so muss die Entity mindestens die folgenden Ports besitzen. Dabei wird ausgenutzt, dass die Signalmengen „ahbmo“ und „ahbmi“ definiert wurden.

```
use grlib.amba.all;

entity ahbmaster is
  generic (hindex : integer := 0);
  port (
    reset : in std_ulogic;
    clk   : in std_ulogic;
    ahbmi : in ahb_mst_in_type;
    ahbmo : out ahb_mst_out_type
  );
end entity;
```

Abbildung 3.16: AHB-Master Definition

Abbildung 3.16 zeigt die Entity Definition eines AHB-Master. Die Ports „ahbmo“ und „ahbmi“ müssen vorhanden sein (roter Rahmen).

Die Verbindung der AHB-Master Entity mit dem AHB erfolgt, durch die Definition der Signalmengen, in einfacher Weise:

```
entity leon3mp is
  port (
    reset : in  std_ulogic;
    clk : in  std_ulogic;
  );
end;

architecture struct of leon3mp is

signal ahbmi : ahb_mst_in_type; --dies ist die horizontale
                                --Linie in Abbildung 4.11
signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
signal ahbsi : ahb_slv_in_type;
signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
begin

ahb : ahbctrl     -- AHB arbiter/multiplexer
  generic map (nahbm => 1, nahbs => 0) --legt die Anzahl der am AHB angeschlossenen
                                       --Masters ("nahbm") und Slaves ("nahbs") fest.

  port map (rst, clk, ahbmi, ahbmo, ahbsi, ahbso); --dies ist die Einmündung der
                                                   --horizontalen Linie in den
                                                   --Bus-Arbiter in Abbildung 4.11
  master0 : ahbmaster
  generic map (hindex => 0)
  port map (rst, clk, ahbmi, ahbmo(0)); --dies sind die Doppelpfeile
                                        --in der Abbildung 4.11
end;
```

Abbildung 3.17: Anschließen eines AHB-Master an den AHB

Die Abbildung 3.17 zeigt, dass der AHB-Master („master0“) an den Vektorindex 0 angeschlossen ist. Die anderen Vektorindizes werden mit der Konstante „ahbm_none“ getrieben. Die Konstante „ahbm_none“ ist ebenfalls in der VHDL AMBA Paket definiert und füllt alle Signale mit logischen Nullen.
Die VHDL-Kommentare in Abbildung 3.17 stellen Assoziotionen mit den Elementen der Abbildung 3.11 her.

Das AHB-Slave-Interface

Die AHB-Slave Entity muss die Signalmenge „ahbs“ als Ports einbinden. Die „ahbs“-Signalmenge besteht aus der Vereinigung der Ausgangsportmenge „ahbso“ und der Eingangsportmenge „ahbsi“. Dabei wird nur eine Signalmenge „ahbsi“ aus dem Bus an alle AHB-Slaves getrieben. Die Ausgangsports aller Slaves („ahbso“) werden zu einem Signalvektor („ahbso(*)“) zusammengefasst. Dieser Vektor ist dann eine Signalmenge, die den Eingangsport des AHB treibt.

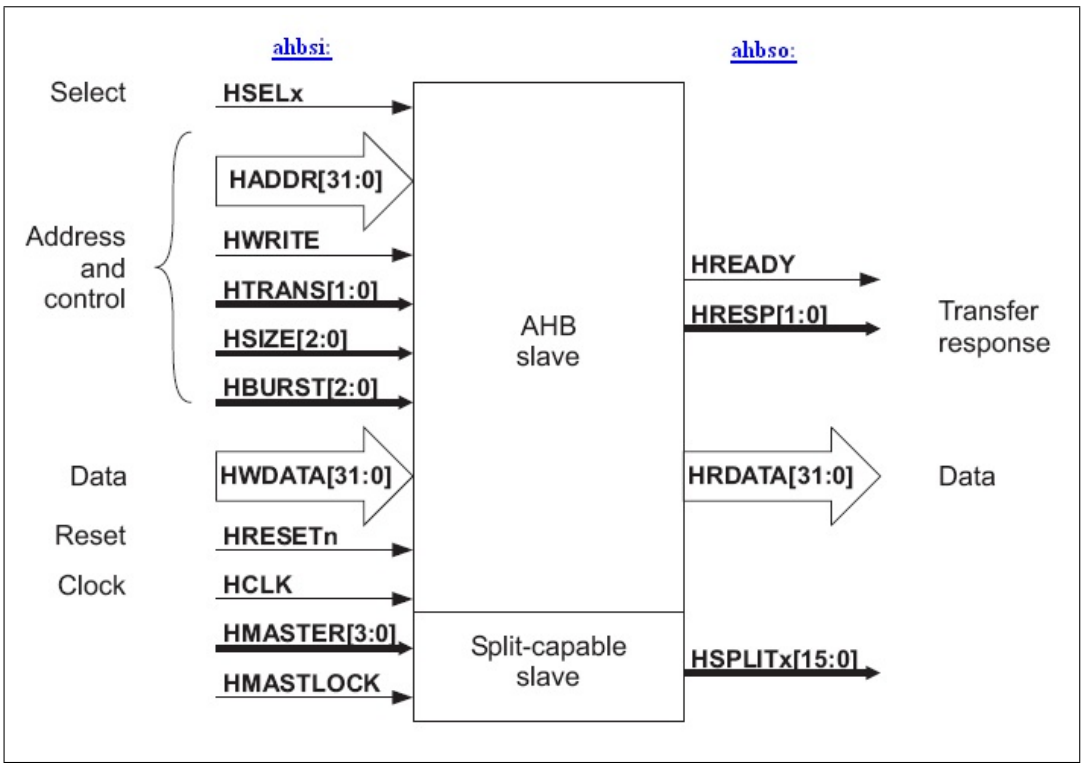

Abbildung 3.18: AHB-Slave Eingangs- und Ausgangsports nach [ambaspec]

In der Abbildung 3.18 sind alle Signalmengen aufgeschlüsselt. Links die Eingangsportmenge („ahbsi“) für einen AHB-Slave. Rechts die Ausgangsportmenge („ahbso“) für einen AHB-Slave.

Die Namensbedeutungen der Signalmengen sind:

- „**ahbso**“ - AHB-Slave Output
- „**ahbsi**“ - AHB-Slave Input

- ***AHB-Slave Output Ports***

```
type ahb_slv_out_type is record
  hready  : std_ulogic;
  hresp   : std_logic_vector(1 downto 0);
  hrdata  : std_logic_vector(31 downto 0);
  hsplit  : std_logic_vector(15 downto 0);
  hcache  : std_ulogic;
  hirq    : std_logic_vector(NAHBIRQ-1 downto 0);
  hconfig : ahb_config_type;
  hindex  : integer range 0 to NAHBSLV-1;
end record;
```

Abbildung 3.19: AHB-Master Output Record der GRLIB

Abbildung 3.19 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „ahbso“-Signalmenge definiert.

Signalname	Bedeutung
hready	Mit diesem Port signalisiert der Slave dem AHB, dass der Transfer abgeschlossen ist. Das Signal des Ports wird an den AHB-Master und an alle Slaves weitergetrieben. Vergleichbar mit dem „Sl_xferAck“-Port eines OPB-Slave.
hresp	Mit diesem Port kann der Status eines Transfers (Okay, Wiederholen, Fehler,...) an den AHB-Master übermittelt werden.
hrdata	Durch diesen Port werden die Daten eines Transfers an den AHB-Master übermittelt. Vergleichbar mit dem „Sl_DBus“-Port eines OPB-Slave.
hsplit	Mit diesem Port bestimmt der Slave welcher AHB-Master einen gespalteten Transfer wieder aufnehmen darf.
hcache	Signalisiert dem verbundenem AHB-Master, ob die transferierten Daten cacheable sind.
hirq	Mit einem Index dieses Ports kann der Slave einen Interrupt auslösen. Dieser Port wird nicht in der AMBA spezifiziert.
hconfig	Mit diesem Port übermittelt der Slave seine Konfigurationsdaten, wie (Systemadresse, benutze Interrupt-Nummer, Adressmaske,...) an den AHB. Dieser Port wird nicht in der AMBA spezifiziert.
hindex	Übermittelt die Slave-Index-Nummer an den AHB. Dies ist eine Nummer die dem Vektorindex des „ahbso(*)“-Signalvektors entspricht, in den die Ausgangsports des Slaves münden. Dieser Port wird nicht in der AMBA spezifiziert.

Tabelle 3.4: Signale des „ahb_slv_out_type“ Record und ihre Bedeutung

Tabelle 3.4 zeigt alle Siganle des „ahb_slv_out_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle 3.4 wird mit „ahbso.*Signalname*“ als Port bezeichnet.

- ***AHB-Slave Input Ports***

```
type ahb_slv_in_type is record
  hsel      : std_logic_vector(0 to NAHBSLV-1);
  haddr     : std_logic_vector(31 downto 0);
  hwrite    : std_ulogic;
  htrans    : std_logic_vector(1 downto 0);
  hsize     : std_logic_vector(2 downto 0);
  hburst    : std_logic_vector(2 downto 0);
  hwdata    : std_logic_vector(31 downto 0);
  hprot     : std_logic_vector(3 downto 0);
  hready    : std_ulogic;
  hmaster   : std_logic_vector(3 downto 0);
  hmastlock : std_ulogic;
  hmbsel    : std_logic_vector(0 to NAHBAMR-1);
  hcache    : std_ulogic;
  hirq      : std_logic_vector(NAHBIRQ-1 downto 0);
end record;
```

Abbildung 3.20: AHB-Master Output Record der GRLIB

Abbildung 3.20 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „ahbsi“-Signalmenge definiert.

Signalname	Bedeutung
hsel	Dieser Signalvektor beinhaltet alle Slave-Select-Signale. Ein Slave überwacht das Select-Signal an seiner eigenen Index-Nummer. Dieser Port wird durch den AHB getrieben. Der AHB berechnet das komplette „hsel“-Signal anhand der Adresse. Vergleichbar mit dem „OPB_select“-Port eines OPB-Slave.
haddr	Dieser Port teil die Adresse dem Slave mit, an die der Transfer gerichtet ist. Dieser Port wird durch einen AHB-Master getrieben. Vergleichbar mit dem „OPB_ABus“-Port eines OPB-Slave.
hwrite	Signalisiert dem Slave, ob der verbundene Master schreiben oder lesen will. Vergleichbar mit dem „OPB_RNW“-Port eines OPB-Salve.
htrans	Signalisiert, welche Art von Transfer (sequentiell, nicht-sequentiell,...) der Master durchführen will.
hsize	Hieran kann der Slave ablesen, wie viele Bits der Transfer umfasst. Die Anzahl wird vom AHB-Master vorgegeben
hburst	Zeigt an, ob der verbundene Master einen Burst-Zugriff durchführen will. Vergleichbar mit dem „OPB_seqAddr“-Port eines OPB-Slave.
hwdata	Durch diese Ports werden die Daten eines Transfers vom AHB-Master an den Slave übermittelt. Vergleichbar mit dem „OPB_DBUS“-Port eines OPB-Slave.
hprot	Zeigt die vom AHB-Master eingestellte Bedeutung (Operationscode, Datenzugriff,...) der Transferdaten an. Damit von Slave Sicherheitsmaßnahmen eingeleitet werden können.
hready	Über diesen Port kann ein Slave feststellen, ob ein anderer Slave einen Transfer beendet hat. Vergleichbar mit dem „OPB_xferAck“-Port des OPB.
hmaster	Über diesen Ports gibt der AHB-Arbiter dem Slave die Master-Index-Nummer an. Dadurch weiß der Slave welchen AHB-Master er später ansprechen soll, wenn er einen Split-Transfer fortsetzen möchte.
hmastlock	Zeigt dem Slave an, dass der AHB-Master einen Lock-Transfer durchführen will.
hmbsel	Zeigt an welche Memory-Bank für den Transfer selektiert wird. Vergleichbar mit dem „OPB_BE“-Port eines OPB-Slave.
hcache	Zeigt an, ob ein Slave einen cacheable Transfer durchführt.
hirq	Zeigt an, welcher AHB-Index-Nummer ein Interrupt ausgelöst wurde.

Tabelle 3.5: Signale des „ahb_slv_in_type“ Record und ihre Bedeutung

Tabelle 3.5 zeigt alle Siganle des „ahb_slv_in_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle 3.5 wird mit „ahbsi.*Signalname*“ als Port bezeichnet.

- ***Anbindung als AHB-Slave***

Soll eine Entity als Slave an den AHB gebunden werden, so muss sie mindestens die Signalmengen „ahbmo“ und „ahbmi“ als Ports besitzen. Dabei wird ausgenutzt, dass die Signalmengen „ahbso“ und „ahbsi“ definiert wurden.

```
library grlib;
use grlib.amba.all;

entity ahbslave is
  generic (hindex : integer := 0);
  port (
    reset : in std_ulogic;
    clk   : in std_ulogic;
    ahbsi : in ahb_slv_in_type;
    ahbso : out ahb_slv_out_type
);
end entity;
```

Abbildung 3.21: AHB-Slave Definition

Abbildung 3.21 zeigt die Entity Definition eines AHB-Slave. Die Ports „ahbso“ und „ahbsi“ müssen vorhanden sein (roter Rahmen).

Die Verbindung der AHB-Slave Entity mit dem AHB erfolgt analog zu einer AHB-Master Entity, in folgender Weise:

```
entity leon3mp is
  port (
    reset : in  std_ulogic;
    clk   : in  std_ulogic;
  );
end;

architecture struct of leon3mp is

signal ahbmi : ahb_mst_in_type;
signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
signal ahbsi : ahb_slv_in_type;
signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
begin

  ahb : ahbctrl      -- AHB arbiter/multiplexer
  generic map (nahbm => 0, nahbs => 1)
  port map (rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

  slave0 : ahbslave
  generic map (hindex => 0)
  port map (rst, clk, ahbsi, ahbso(0));

end;
```

Abbildung 3.22: Anschließen eines AHB-Slave an den AHB

Die Abbildung 3.22 zeigt, dass der AHB-Slave („slave0“) and den Vektorindex 0 angeschlossen ist. Die anderen Vektorindizes werden mit der Konstante „ahbs_none“ getrieben. Die Konstante „ahbs_none“ ist abermals in dem VHDL AMBA Paket definiert und füllt alle Signale mit logischen Nullen.
In der GRLIB Umsetzung des AHB wurden einige zusätzliche Signale für die Bus-Interfaces definiert. Es sollen nun einige wichtige Funktionsweisen des AHB diskutiert werden, dabei wird auf die zusätzlichen Signale eingegangen und auf Unterschiede zu dem OPB aufmerksam gemacht.

- ***Die „hindex“-Konstante***

Wie an den blauen Rahmen in Abbildungen 3.22 und Abbildung 3.21 zu erkennen ist, muss eine GRLIB Entity neben der Anbindung an den AHB noch über eine „hindex“-Konstante verfügen, welche über die Generic-Konstanten eines IP-Cores festgelegt wird. Diese „hindex“-Konstante muss mit dem AHB-Vektorindex übereinstimmten, an den die Entity angeschlossen wurde.
Die „hindex“-Konstante wird verwendet, um den korrekten „hsel“-Signalindex, vom AHB-Slave, zu überwachen und daran zu erkennen wann der AHB-Slave für einen Transfer ausgewählt wurde. Solch eine Konstante ist bei dem bekannten OPB nicht notwendig, da das Ansprechen einer Slave Entity beim OPB anders geschieht.

Eine genaue Erläuterung des Select-Prozesses beim AHB, sowie ein Vergleich zum OPB Select Prozess, erfolgt im Kapitelpunkt 3.1.5.2.

- ***Der „hconfig“-Port***

Dieser Port ermöglicht es den AHB mit der „Plug and Play“ Funktionalität auszustatten. Der AHB-Master und der AHB-Slave können so dem AHB Informationen über ihre Konfiguration mitteilen. Die Konfiguration wird hauptsächlich durch die Entity Generic-Konstanten festgelegt.

Zu den Konfigurations-Informationen einer AHB Entity gehört:

- Anbieter Identifikation
- Device Identifikation
- Device Version
- Device Interrupt

Sowie:

- Adresse im Systemadressraum
- Adressmaske

Eine genaue Erläuterung des „Plug and Play“ Vorgangs erfolgt im Kapitelpunkt 3.1.5.2

- ***Der „hirq“-Port***

Der „hirq“-Port gehört zu den Ausgangsports eine AHB-Master oder AHB-Slave Entity. Durch diesen Port kann die Entity einen für sie zugehörigen Interrupt auslösen.

Nähere Erläuterung der Interrupt-Steuerung beim AHB erfolgt im Kapitelpunkt 3.1.5.2.

- ***Split-Transfer***

Ein Split-Transfer wird ausgelöst, wenn ein Burst-Transfers von einem AHB-Slave unterbrochen wird. Diese Unterbrechung kann beispielsweise ausgelöst werden, wenn die zu übermittelnden Daten noch nicht bereitstehen. Durch diese AHB Funktionalität kann der Bus von uneffizienten „IDLE“ Transfers freigehalten werden und dem AHB-Master, der ein μP sein könnte, mehr Berechnungszeit verschaffen.

Um den Split-Transfer auszulösen muss der AHB-Slave ein entsprechendes „hresp“-Signal anlegen und sich den gerade verbundenen AHB-Master mittels „hmaster“ merken. Ist der AHB-Slave

bereit den Split-Transfer fortzusetzen, so treibt er den „hsplit“-Port mit dem gemerkten „hmaster“ Index und leitet somit einen Transfer ein.

Das AHB-Select-Signal („hsel“)

Das AHB-Select-Signal („hsel“) ist ein Signalvektor aus 16 Signalen. Jeder Slave überwacht den „hsel“-Signalvektor an dem Index, der durch seine „hindex“-Konstante vorgegeben ist. Wird das Signal „hsel(index)“ durch den Bus auf High-Pegel gesetzt, so weiß der Slave mit „hindex = index“, dass er als Ziel eines Transfers ausgewählt wurde.

- ***Selektierung im OPB System***

Die Selektierung eines OPB-Slave funktioniert anders. Hier setzt der OPB-Master seinen „M_select“-Port auf High-Pegel. Der OPB treibt dieses Signal als „OPB_Select“-Signal zu allen OPB-Slaves. Das „OPB_Select“-Signal ist kein Vektor. Zusätzlich wird die Adresse „OPB_ABus“ an alle Slaves übermittel.
Der OPB-Slave entscheidet nun, ob er für einen Transfer ausgewählt wurde. In der VHDL sieht dies wie in Abbildung 3.23 aus, wenn nur die Basisadresse berücksichtigt wird:

```
if (OPB_Select = '1') and (OPB_ABus = C_BASEADDR) then
        Selected<='1';
end if;
```

Abbildung 3.23: Selektierung im OPB-System

Ist das „OPB_Select“-Signal auf High-Pegel dann vergleicht der Slave die angelegte Adresse mit seinem konfigurierten Adressbereich. Dieser wird durch die Generic-Konstante „C_BASEADDR“ festgelegt.

- ***Selektierung im AHB System***

Im AHB System ist ein Vergleich mit der Adresse, von Seiten der Slaves, nicht notwendig. Dies wird durch den AHB-Arbiter durchgeführt.
Da die Konfigurationsdaten durch die „Plug and Play“ Funktionalität an den AHB-Arbiter übermittelt wurden, kann dieser die Adressen aller AHB-Slaves einsehen. Der AHB vergleicht dann, im Falle eines AHB-Master Lese- oder Schreibzugriff, an welchen Slave der Transfer gerichtet ist und setzt den „hsel“-Signalvektor am entsprechendem Index auf High-Pegel.

Dieser VHDL Code ist im AHB („ahbctrl.vhd“) an der Zeile 366 zu finden. Es wird im folgenden nur das notwendigste, in der Abbildung 3.24 eingeblendet:

```
if ((slvo(i).hconfig(j)(31 downto 20) and slvo(i).hconfig(j)(15 downto 4)) =
(HADDR(31 downto 20) and slvo(i).hconfig(j)(15 downto 4))) then

        hsel(i) := '1';

end if;
```

Abbildung 3.24: Selektierung im AHB-System

Das Signal „HADDR“, in Abbildung 3.24 rot markiert, beinhaltet die Adresse, die von einem AHB-Master, zu einem Transfer, ausgewählt wurde. Sie bildet mit der Slave-Adressmaske (in Abbildung 3.24 blau markiert) eine Konjuktion, da nicht immer alle Adressteile von Interesse sein sollen, und wird dann mit der „Plug and Play“ Information aller Slaves verglichen. Wird ein Treffer-Index gefunden so treibt der AHB-Arbiter an diesem Index im „hsel“-Signalvektor den High-Pegel.

Der elementare Unterschied zwischen AHB und OPB ist, dass beim AHB die Selektierung auf der AHB Seite stattfindet und beim OPB auf der Seite des OPB-Slave.

„Plug and Play“

Die „Plug and Play“ Funktionalität des AHB ermöglicht eine hohe Modifizierbarkeit des SoC während der Laufzeit. So ist es zum Beispiel möglich, den verwendeten Interrupt eines Slave nachträglich zu verändern.
Ein anderes interessantes Szenario wäre es, mittels partieller dynamischer Rekonfiguration, unterschiedliche AHB-Slaves an einem AHB-Interface zu betreiben. Die Konfiguration der wechselnden Slaves würde dann automatisch an den AHB-Arbiter übermittelt werden.

Das „Plug and Play“ besteht aus drei Teilen:

- Identifikation der anliegenden SoC-Komponente (für Master und Slave möglich)
- Interrupt-Übermittlung (für Master und Slave möglich, „hirq“-Generic-Konstante)
- Adressen-Übermittelung (nur für Slave möglich, „haddr“-Generic-Konstante)

Will eine Entity ihre Konfigurationsdaten an den AHB-Arbiter übermitteln, so muss sie den „hconfig“-Port treiben. Der „hconfig“-Port besteht aus 8 mal 32 Bit Registern und hat folgenden Aufbau:

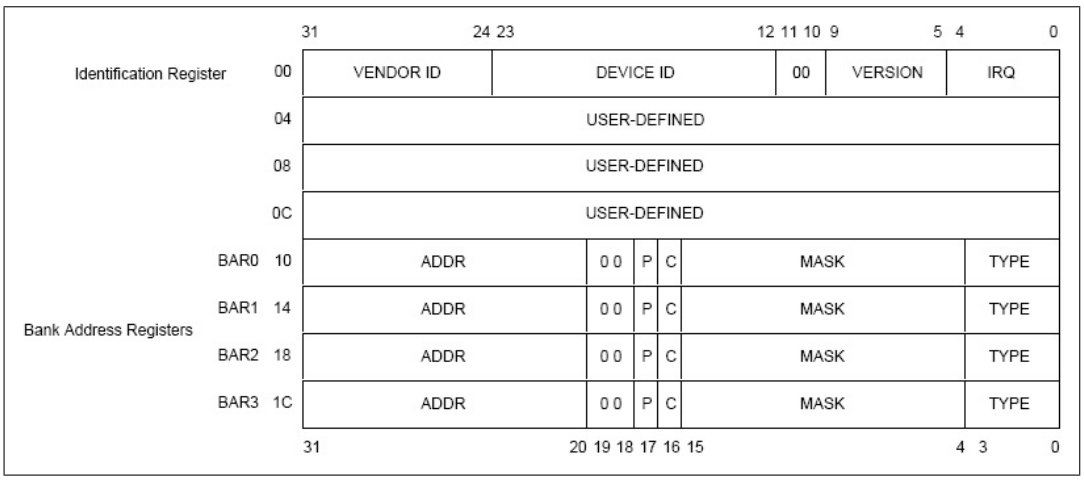

Abbildung 3.25: AHB - „hconfig“-Port Einteilung [grlibdoc]

Das **1. Register** in Abbildung 3.25 ist das Identifikationsregister und beinhaltet die Identifikation des Anbieters der Entity und eine Identifikation der Entity selbst. In den Bits 4 bis 0 wird der verwendete Interrupt angegeben. Das **2. bis 4. Register** sind frei verwendbar. Das **5. bis 8. Register** sind die „Bank Address Registers“. Hier kann ein Slave seine zur Verfügung gestellten Adressbereiche angeben. Die Bits 31 bis 20 beinhalten dabei die Basisadressen („haddr“).
Ein AHB-Slave hat die Möglichkeit mehrere Adressbereiche für sich zu reservieren. Um die Größe eines jeden Adressbereiches festzulegen, muss die jeweilige Adressmaske bestimmt und, in den Bits 15 bis 4, angegeben werden.

Nähere Beschreibung des Adressierungs-Prozesses erfolgt in Kapitelpunkt 3.1.5.2.

Haben nun alle AHB-Masters und AHB-Slaves ihre Konfigurationsinformationen angelegt, so erstellt der AHB, aus den Informationen, eine Konfigurationstabelle. Diese Tabelle besteht aus der Aneinanderkettung aller AHB-Master und AHB-Slave „hconfig“-Signalen.
Die Konfigurationstabelle wird standardmäßig an der Adresse 0xFFFFF000 zur Verfügung gestellt. Diese Adresse setzt sich aus den Generic-Konstanten der Entity „ahbctrl“ wie folgt zusammen:

Startadresse der Konfigurationstabelle =
0x ioaddr & cfgaddr & 00 (Standardmäßig: 0x FFF & FF0 & 00)

Ab der Startadresse 0xFFFFF000 folgen 0x800 Bytes, bis 0xFFFFF7FF, an Konfigurationsdaten der AHB-Masters:

$$0x800Bytes = 2048Bytes = 8\frac{Register}{Master} \cdot 4\frac{Bytes}{Register} \cdot 64Master$$

Es könnten also maximal 64 Masters ihre Konfigurationsdaten ablegen.

Analoges gilt für die Slaves, deren Konfigurationstabellenbereich von 0xFFFFF800 bis 0xFFFFFFFF geht.

$$0x800Bytes = 2048Bytes = 8\frac{Register}{Slave} \cdot 4\frac{Bytes}{Register} \cdot 64Slave$$

Maximal 64 Slaves könnten in diesem Adressbereich ihre Konfigurationen ablegen.

Der Adressierungs-Prozess

Der Adressierungs-Prozess ist der Vorgang, der durchgeführt werden muss, um alle Slaves, die an den AHB angeschlossen sind, mit einer Adresse zu versehen. Dabei dürfen keine Überlappungen, der Adressbereiche im Adressraum des Systems, entstehen. Für AHB-Masters werden keine Adressen vergeben, sondern nur AHB-Index-Nummern.

Die Adressen der einzelnen AHB-Slaves werden mit ihren Generic-Konstanten festgelegt. Jeder AHB-Slave besitzt eine Generic-Konstante namens "haddr". Diese Konstante ist vom Typ Integer im Bereich von 0 bis 4095. Dies entspricht einem hexadezimalen Bereich von 0x000 bis 0xFFF, also einen Signalvektor von 12 Bit.
Leider werden die Adressen der GRLIB-SoC-Komponenten mit Integer-Typen angegeben. Diese Adressen werden in dieser Arbeit in die hexadezimale Schreibweise umgeformt, da diese Schreiweise gewohnter ist und die interne, logische Darstellung der Adressen identisch ist.

Es sei weiter darauf hingewiesen, dass die Generic-Konstante „haddr“ nichts mit der Transferadresse eines AHB-Master „HADDR“ zu tun hat. „HADDR“ wurde im Kapitelpunkt 3.1.5.2 in der Abbildung 3.24 (rot markiert) eingeführt.

Wird für einen Slave „haddr = 0x800“ festgelegt, so lautet die Basisadresse des Slave 0x80000000.

Eine wichtige Feststellung, aus der Abbildung 3.24 ist, dass zur Festlegung des „hsel“-Signalvektors nur die ersten 12 Bits der Adresse des AHB-Slave und der Transferadresse des AHB-Master („HADDR“) benötigt werden.
Dies bedeutet, dass für einen Slave mindestens ein Adressbereich von 0x haddr & 00000 bis 0x haddr & FFFFF reserviert wird. Dies entspricht einer Adressbereichgröße von $0x100000Bytes = 1048576Bytes = 1MByte$.

Möchte der Benutzer einen größeren Adressbereich reservieren, so geschieht dies durch den Einsatz der Adressmaske „hmask“, diese ist ebenfalls eine Generic-Konstante eines jeden AHB-Slave. Standardmäßig ist die Adressmaske auf „hmask = 0xFFF“ gesetzt. Wird sie auf „hmask = 0xFF0“ geändert so steht dem AHB-Slave ein Adressereich von $0xFFFFFFBytes = 16777215Bytes = 16MByte$ zur Verfügung.

Ein Beispiel dafür, wie die Adressierung im Praktischen durchgeführt wird, zeigt folgende Abbildung 3.26:

```
component ahbram
  generic (
    hindex : integer := 0; -- AHB slave index
    haddr  : integer := 0;
    hmask  : integer := 16#fff#);
  port (
    rst   : in std_ulogic;
    clk   : in std_ulogic;
    ahbsi : in ahb_slv_in_type; -- AHB slave input
    ahbso : out ahb_slv_out_type); -- AHB slave output
end component;

ram0 : ahbram
generic map (hindex => 1, haddr => 16#700#, hmask => 16#FF0#)
port map (rst, clk, ahbsi, ahbso(1));
```

Abbildung 3.26: Adressenvergabe an einen AHB-Slave

In Abbildung 3.26 ist die Entity „ahbram“ am „ahbso“ Vektor-Index 1 verbunden. Der Adressbereich beginnt bei 0x70000000, da „haddr = 0x700“ und hat eine Größe von 16 MB, da „hmask = 0xFF0“ (roter Rahmen).

Interrupt-Steuerung

Die Interrupt-Steuerung eines GRLIB-SoC erfolgt über den 32 Bit Vektor „hirq“. Dieser Signalvektor ist ein Ausgangsport einer jeden Entity, die an den AHB angeschlossen ist. Dadurch ist es einfach möglich einen Interrupt auszulösen.

Ein AHB-Slave, der einen Interrupt auslösen will, treibt den „hirq“-Signalvektor wie folgt:

```
ahbso.hirq(hirq) <= '1';
```

Dabei ist der Index „hirq“ eine Generic-Konstante vom Typ Integer, die für jeden AHB-Master oder AHB-Slave festgelegt werden muss.

Der AHB verbindet die Signalvektoren aller AHB-Masters und AHB-Slaves durch eine Disjunktion, d.h. es entsteht ein Ergebnis-Interrupt-Vektor, der alle geworfenen Interrupts des SoC enthält. Dies ist ab der Zeile 498 der AHB Entity „ahbctrl“ zu erkennen und in der folgenden Abbildung 3.27 dargestellt:

```
hirq := (others => '0');
if disirq = 0 then

    for i in 0 to nahbs-1 loop
    hirq := hirq or slvo(i).hirq;
    end loop;
    for i in 0 to nahbm-1 loop
    hirq := hirq or msto(i).hirq;
    end loop;

end if;
```

Abbildung 3.27: Disjuktion aller AHB-Interrupts zum Ergebnis-Interrupt-Vektor

In Abbildung 3.27 ist zu sehen, dass zuerst der Ergebnis-Interrupt-Vektor „hirq“ gelöscht wird. Dann wird, wenn das Interrupt Routen aktiviert ist (disirq = 0), eine Vermischung aller Interrupt-Vektoren mit einem logischen ODER durchgeführt und im Ergebnis-Interrupt-Vektor „hirq“ abgelegt.

3.1.5.3 AMBA Advanced Peripheral Bus (APB)

Der **A**dvanced **P**eripheral **B**us (kurz APB) ist das Peripherie-Bus-System der GRLIB. An den APB werden alle SoC-Komponenten angeschlossen, die keine hohen Übertragungsraten benötigen. Der APB wird an den AHB als Slave angeschlossen. Diese Anbindung wurde bei der Einbindung des APB-IP-Cores berücksichtigt und eine entsprechende Bus-Interface-Deklaration in seiner MPD durchgeführt. Es wird somit keine extra AHB2APB-Bridge benötigt. Die benötigte Logik, welche AHB-Transfers in APB-Transfers umwandelt (Funktionalität einer Bridge) ist bereits im IP-Core des APB enthalten. Es können bis zu 16 Slaves aber keine Masters an den APB angeschlossen werden.

Da die APB-Funktionsprinzipien wie:

- Das APB-Slave-Interface
- Das APB-Select-Signal („psel“)
- „Plug and Play“
- Der Adressierungs-Prozess
- Interrupt-Steuerung

mit den AHB-Funktionsprinzipien stark vergleichbar sind, sind ihre Bedeutungen für den weiteren Verlauf der Arbeit weniger relevant. Später kann es trotzdem der Fall sein, dass zusätzliche APB-Slaves der GRLIB in das EDK eingebunden werden müssen. Aus diesem Grund wird der APB ausführlich im Anhang A.1 (AMBA Advanced Peripheral Bus (APB)), dieser Arbeit, behandelt.

3.1.5.4 Technologie-Unabhängigkeit

Die GRLIB ist mit dem Hintergrund erstellt wurden, technologisch unabhängig zu sein. Das bedeutet es soll möglich sein, ein GRLIB-SoC auf einem ASIC oder einem FPGA umzusetzen ohne dabei gravierende Unterschiede in den VHDL Designs oder im Workflow vorzunehmen. Grundsätzlich gibt es in der GRLIB nur wenige technologisch abhängige Schaltungsteile, die mit unterschiedlichem VHDL Code einhergehen.

Solche technologisch abhängigen Schaltungsteile sind:

- Speicher (On-Chip)
- Pads
- Clock Buffer

Mit Speicher sind nur On-Chip Speicher gemeint. Für die Xilinx FPGAs sind das die BRAMs. Mit Pads sind elementare VHDL Blöcke gemeint, wie IBUFs, OBUFs oder IOBUFs. Clock Buffer sind so zu sagen spezielle Pads, die zur Übergabe von Clock Signalen dienen. Solche Clock Buffer sind zum Beispiel die Elemente BUFG oder BUFGDLL.

Zu Beachten 3.5 (Einbindung von Pads und Buffern)

Pads und Clock Buffer sind Schaltungselemente die eine zusätzliche „Propagation delay time“ erzeugen. Sind solche Elemente in der GRLIB eingefügt, müssen sie bei der Einbindung in das EDK mit übernommen werden. Ein Entfernen dieser Elemente kann das Schaltungs-Timing so empfindlich verändern, dass der IP-Core nicht mehr funktioniert.

Innerhalb der GRLIB sind sogar Unterschiede zwischen einzelnen FPGAs festzustellen. So werden in der GRLIB für einen Xilinx FPGA andere Pad-Konfigurationen benutzt als auf einem Altera FPGA.

Umsetzung

Zur Umsetzung der Technologie-Unabhängigkeit wird wieder das VHDL-Konstrukt „generate“ verwendet. In Abhängigkeit eine globalen Konstante werden unterschiedliche VHDL Schaltungsteile eingebunden, die der korrekten Umsetzung für den speziellen FPGA gerecht werden.

Solche globalen Konstanten sind, in der GRLIB, die:

Generic-Konstante	Anwendung
memtech	für Speichertechnologie
padtech	für Padtechnologie und Clock-Buffer-Technologie

Tabelle 3.6: Technologiekonstanten-Typen

Diese Konstanten sind vom Typ Integer. Ein jeder IP-Core, welcher solche technologisch empfindlichen Schaltungsteile verwendet, besitzt diese Generic-Konstanten. Es ist auch möglich, dass ein IP-Core nur eine der beiden Konstanten besitzt. Dies ist davon abhängig, ob die Speichertechnologie oder die Padtechnologie festgelegt werden muss.

Zur Übersicht folgt ein Ausschnitt des VHDL Pakets „gencomp.vhd“, der alle möglichen Konstantenwerte mit einer bestimmten Technologie assoziiert. Dabei sind die Konstantenwerte für die Speicher- und die Padtechnologie gleichermaßen gültig:

	Technologie		Wert
constant	inferred	: integer :=	0;
constant	virtex	: integer :=	1;
constant	virtex2	: integer :=	2;
constant	memvirage	: integer :=	3;
constant	axcel	: integer :=	4;
constant	proasic	: integer :=	5;
constant	atc18s	: integer :=	6;
constant	altera	: integer :=	7;
constant	stratix	: integer :=	7;
constant	umc	: integer :=	8;
constant	rhumc	: integer :=	9;
constant	proasic3	: integer :=	10;

	Technologie		Wert
constant	spartan3	: integer :=	11;
constant	ihp25	: integer :=	12;
constant	rhlib18t	: integer :=	13;
constant	virtex4	: integer :=	14;
constant	lattice	: integer :=	15;
constant	ut25	: integer :=	16;
constant	spartan3e	: integer :=	17;
constant	peregrine	: integer :=	18;
constant	memartisan	: integer :=	19;
constant	virtex5	: integer :=	20;
constant	custom1	: integer :=	21;
constant	ihp25rh	: integer :=	22;

Tabelle 3.7: Technologiekonstanten-Werte im „gencomp.vhd“-Paket

Wird eine der Generic-Konstanten („memtech“ oder „padtech“) in einem IP-Core der GRLIB, gefunden so muss sie vor der Synthese mit dem richtigen Wert aus der Tabelle 3.7 festgelegt werden. Für nicht, in der Tabelle 3.7, aufgelistete Technologien ist der inferred-Wert (0) zu benutzen.

3.1.6 Weitere Werkzeuge für die GRLIB

In diesem Kapitelpunkt sollen Werkzeuge genannt werden, die für den intensiven Umgang mit der GRLIB unverzichtbar sind. Mit den Erläuterungen in Kapitelpunkt 3.1.4 ist es möglich, ein

GRLIB-SoC zu synthetisieren und einen konfigurierbaren Bitstream zu erzeugen. Allerdings ist damit nur geklärt wie die Hardware eines GRLIB-SoC erzeugt wird.

Zur Kompilierung einer Software, sowie deren Einspeisung in ein GRLIB-SoC, werden zusätzliche Werkzeuge benötigt. Bei diesen Werkzeugen handelt es sich um:

- Einen Software-Compiler für den LEON3-Prozessor
- Den **G**aisler **R**esearch Debug**mon**itor (kurz GRMON) zum Debuggen von laufenden GRLIB-SoCs
- Den SPARC-Simulator (TSIM) zur Simulation eines LEON3-SoCs

In dieser Arbeit wurde der Software-Compiler **B**are-C **C**ross-**C**ompiler (kurz BCC) zur Kompilierung der nativen C-Programme verwendet.
Um ein Linux-Image oder kleinere, mit dem BCC, kompilierte ELF-Programme in ein LEON3-SoC einzuspielen kann das SoC mit dem GRMON debuggt werden und ein Transfer der Programmdaten in den Systemspeicher des LEON3-SoC vorgenommen werden. Der GRMON bietet zusätzliche Möglichkeiten zur Überwachung eines LEON3-SoCs während dessen Betriebs.
Soll ein LEON3-SoC, zum Testen von Software, nur simuliert werden, so kann das Programm TSIM benutzt werden. Dieses Programm simuliert ein Debugging-Zugriff auf ein LEON3-SoC auf der Konsole eines Host-PCs. Dadurch kann ein erster Eindruck über die Programmausführung gewonnen werden.

Da diese Werkzeuge für das Ziel der Arbeit nicht ausschlaggebend, aber für die Arbeit mit einem GRLIB-SoC unverzichtbar sind, sollen sie im Anhang A.2 näher behandelt werden.

3.1.7 LEON3-Prozessor in der GRLIB

Als Soft-Prozessor setzt die GRLIB auf den LEON3-Prozessor nach der **S**calable **P**rocessor **Arc**hitekture (kurz SPARC). Der LEON3 ist ein RISC-Prozessor und besitzt alle Funktionalitäten, welche in der SPARC spezifiziert werden.

Dazu gehört beispielsweise:

- Der typische Befehlssatz eines SPARC-Prozessors.
- Zwei Interfaces für einen Co-Prozessor und eine FPU
- Eine Art Harvard-Architektur
- Zusätzliche Schaltungen zur Integer-Multiplikation und Integer-Division
- Eine Register-Anzahl von 40 bis 520, durch Register-Fenster
- Eine **M**emory **M**anagement **U**nit (kurz MMU)

- Programm- und Daten-Caches

Der LEON3 ist in der GRLIB hoch flexibel implementiert wurden. Es ist also möglich eine beliebige Konfiguration, des LEON3, vor der Synthese festzulegen. Diese Konfiguration wird lediglich mittels unterschiedlicher Festlegung der Generic-Konstanten, der LEON3-Haupt-Entity, durchgeführt. Es ist zum Beispiel möglich die MMU optional einzubinden, den Cache-Speicher zu deaktivieren oder seine Größe zu verändern.
Aufgrund dieser flexiblen Konfigurierbarkeit, besitzt der LEON3 ein hohes Potential. Er kann in kleinen Controller-Systemen oder in größeren Host-Systemen zum Einsatz kommen und ist mit Taktfrequenzen bis zu 100 MHz[7] taktbar.

LEON3-Konfiguration	**Größe**
ohne MMU, ohne Programm- oder Daten Cache, ohne Hardware-Multiplizierer oder Dividierer	ca. 2300 Slices
mit MMU, jeweils 8 kByte Programm- und Daten Cache, mit Hardware-Multiplizierer und Dividierer	ca. 4400 Slices

Tabelle 3.8: LEON3-Konfigurationen und resultierende Größe

Die Tabelle 3.8 zeigt eine Beispielübersicht von möglichen Konfigurationen des LEON3 mit zugehörigen Größen. Eine oberer Größengrenze des LEON3 existiert theoretisch nicht, da alle LEON3-Speicher beliebig vergrößert werden können.

[7] Auf dem FPGA „Virtex-II Pro 30 FF896“ konnte der LEON3 mit bis zu 110 MHz erfolgreich getaktet werden.

- ***Aufbau***

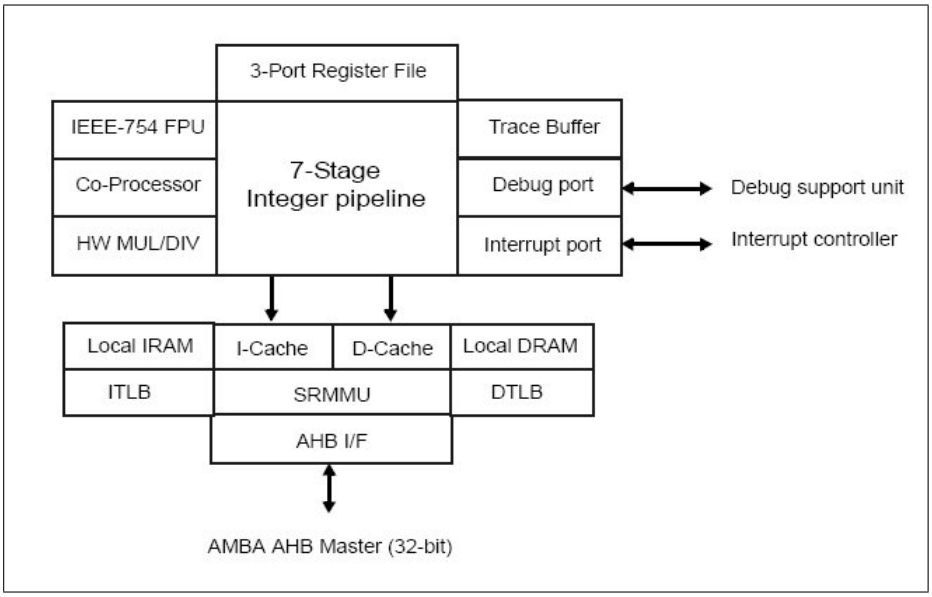

Abbildung 3.28: Elemente des LEON3-Prozessors [gripdoc]

Abbildung 3.28 zeigt eine Gesamtübersicht der Elemente des LEON3-Prozessors [gripdoc]. Es ist zu erkennen, dass sich neben dem AHB-Master-Interface ein Interface für die DSU und den Interrupt-Controller am LEON3 befindet. Jeder LEON3-Prozessor eines SoC wird direkt an die DSU oder an den Interrupt-Controller angeschlossen. DSU und Interrupt-Controller sind eigenständige IP-Cores der GRLIB.

Als besonderes Merkmal besitzt der LEON3 eine MMU, siehe Abbildung 3.28. Dies ist beim MicroBlaze-Prozessor nicht der Fall. Die MMU des LEON3 ermöglicht neue Einsatzgebiete, vor allem im Bereich der Betriebssysteme wie Linux.

Für eine detaillierte Beschreibung des LEON3-Prozessors wird auf den Anhang B (Der LEON3-Prozessor) verwiesen. In diesem Anhang werden vor allem wichtige Generic-Konstanten, der LEON3-Entity, und ihre Wirkungen erläutert. Diese Generic-Konstanten sind für die Arbeit mit dem LEON3 notwendig.

Kapitel 4

Der Ansatz

Die vorliegende Arbeit kann in vier große Phasen eingeteilt werden, welche den umgesetzten Zielen der Arbeit entsprechen.

Die Phasen der Arbeit sind:

1. GRLIB-IP-Cores in das EDK einbinden
2. Testen des LEON3-Systems mit nativer Software
3. Bereitstellung eines Linux für das LEON3-System
4. Verbinden des LEON3-Systems mit dem MicroBlaze-System

4.1 GRLIB-IP-Cores in das EDK einbinden

In der ersten Phase der Arbeit boten sich zwei IP-Core-Quellen für die Ersetzung der Xilinx-IP-Cores an. Die erste Quelle ist die GRLIB. Sie beinhaltet so viele IP-Cores, dass für jeden IP-Core der Xilinx-Bibliothek, ohne Umwege, ein Ersatz gefunden werden konnte. Die GRLIB bietet eine hohe Kompatibilität ihrer IP-Cores untereinander und wird ständig korrigiert und aktualisiert. Ebenfalls wurden, von GR, umfassende Tests mit verschiedenen GRLIB-Konfigurationen durchgeführt.
Eine andere IP-Core-Quelle, ist das OpenCores-Projekt. Das OpenCores-Projekt kann auf der Homepage OpenCores Projekts[1] eingesehen werden.

Hier wird synthetisierbare Hardware in den Beschreibungssprachen VHDL oder Verilog veröffentlicht. Innerhalb des OpenCores-Projekts werden zahlreiche IP-Cores unter Kategorien wie

[1] http://www.opencores.org

μP, Speicher-Controller, Kommunikations-Controller und Busse angeboten. Der Nachteil des OpenCores-Projekts ist, dass dort hauptsächlich Material angeboten wird, welches nicht auf gegenseitige Kompatibilität getestet wurde. Es ist zwar beispielsweise möglich den „OpenRISC 1000“-Prozessor für den „Wishbone“-Bus und den eigentlichen Wishbone-Bus herunterzuladen, doch liegt es beim Anwender selbst die beiden IP-Cores funktionstüchtig miteinander zu verbinden.

Ein Vorteil des OpenCores-Projekt liegt in der Nachbereitung der eingebundenen GRLIB. Von OpenCores könnten dazu einzelne IP-Cores benutzt werden, um die GRLIB zu erweitern oder fehlende IP-Cores zu ergänzen. Beispielsweise besitzt die Open-Source-Variante des LEON3-Prozessors keine FPU. Diese könnte mit einer der zahlreichen FPUs des OpenCores-Projekts ergänzt werden. Hauptsächlich aus Gründen der Kompatibilität und der technischen Pflege der GRLIB, wurde sie in dieser Arbeit als IP-Core-Quelle ausgewählt.

In dieser Phase der Arbeit war es das Ziel die IP-Cores der GRLIB so in das EDK einzubinden, dass sie wie die herkömmlichen IP-Cores, der Xilinx-IP-Core-Bibliothek, aus dem „IP Catalog“ des EDK genutzt werden können.

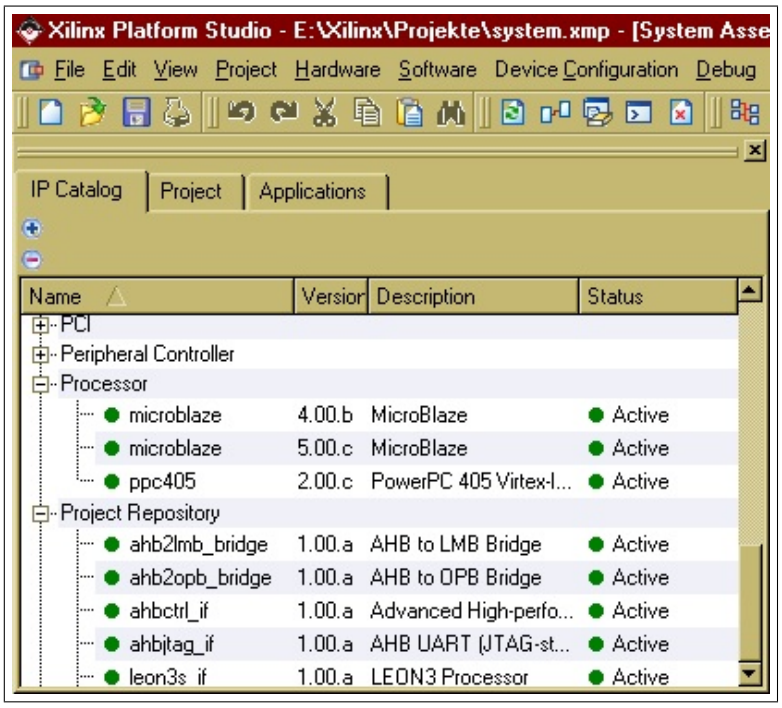

Abbildung 4.1: Der „IP Catalog“ des EDK mit herkömmlichen und neuen GRLIB-IP-Cores

In Abbildung 4.1 ist der „IP Catalog“ des EDK 8.2i zu erkennen. In der Kategorie „Processor“ befinden sich die bekannten IP-Cores für den MicroBlaze-Prozessor und den PowePC-Prozessor. Nach dieser Phase der Arbeit werden alle GRLIB-IP-Cores in der Kategorie „Project Repository“ angezeigt und können von da aus dem SoC-Projekt hinzugefügt werden. Der letzte IP-Core im „IP Catalog“ der Abbildung 4.1 ist beispielsweise der LEON3-Prozessor.

Da das Xilinx-EDK von vielen SoC-Entwicklern genutzt wird und die herkömmlichen Xilinx-IP-Cores mit den, in Kapitelpunkt 2.2.2 vorgestellten, EDK-Konstrukten eingebunden sind, ist es für die optimale Benutzerfreundlichkeit förderlich, wenn die Einbindung der neuen GRLIB-IP-Cores ebenso mit den EDK-Konstrukten durchgeführt wird.

Sind die IP-Cores der GRLIB durch die EDK-Konstrukte einmal eingebunden, so haben sich ihre Wiederverwendbarkeitswerte stark erhöht. Dies liegt daran, dass es nach der Einbindung möglich ist, durch den bloßen Austausch der in dieser Arbeit entstandenen IP-Core-Verzeichnisse ein fremdes EDK auf die Synthese eines LEON3-SoC vorzubereiten. Das EDK definiert klare Schnittstellen zwischen den SoC-Komponenten. Dies macht das „Zusammenstecken“ von SoC-Komponenten einfacher und wenig fehleranfällig. Ohne die Verwendung der EDK-Konstrukte könnten nur die VHD-Dateien der GRLIB ausgetauscht werden.

4.2 Testen des LEON3-Systems mit nativer Software

Nach Abschluss der ersten Phase, kann ein LEON3-SoC mit dem EDK synthetisiert werden. In der zweiten Phase der Arbeit gilt es nun ein LEON3-SoC zu testen. Ein Test kann am einfachsten mit der Hilfe von nativer Software stattfinden. Die native Software wird durch den LEON3-Prozessor ausgeführt und eine entsprechende Reaktion kann an den Peripheriegeräten, wie den LEDs oder der serielle Schnittstelle des FPGA-Boards, beobachtet werden. Sollte ein natives Programm durch das LEON3-SoC erfolgreich ausgeführt werden, so kann davon ausgegangen werden, dass Prozessor, Bus und Peripherie-Komponenten funktionstüchtig sind.
Eine andere Möglichkeit wäre es ein Linux-Betriebssystem in ein LEON3-SoC einzuspielen und dessen Ausführung zu überwachen. Da das Linux-Betriebssystem aber viel komplexer ist als native Software, kann eine Fehlerquelle möglicherweise nicht identifiziert werden, da lediglich der Bootprozess fehlschlägt, man jedoch den Grund nicht erkennt. Mit nativer Software können dagegen gezielt einzelne Peripherie-Komponenten des LEON3-SoC auf ihre Funktionstüchtigkeit überprüft werden.
Aus diesem Grund wurde in dieser Phase die native Software zum Testen benutzt. Um native C-Programme für ein LEON3-SoC zu kompilieren wurde in dieser Arbeit der BCC verwendet. Dieser Compiler bot sich direkt an, da er speziell auf der Homepage der Firma GR, für den LEON3, heruntergeladen werden konnte. In dieser Phase wurde weiterhin die Kompilierung der nativen C-Programme mit dem Arbeitsplan des EDK durchgeführt. Dies erhöht die Wiederverwendbarkeit und die Benutzerfreundlichkeit.

4.3 Bereitstellung eines Linux für das LEON3-System

Ein Betriebssystem auf einem SoC auszuführen ist für viele Anwendungen nützlich oder gar unumgänglich. Mit einem Betriebssystem könnten problemlos mehrere Programme auf einem Prozessor quasiparallel ausgeführt werden. Ein geeignetes Betriebssystem ist, auch für ein LEON3-SoC, das Linux-Betriebssystem. Linux ist ein ausgereiftes, stabiles und ressourcensparendes Betriebssystem. Linux-Image-Dateien für SoCs haben lediglich eine Image-Dateigröße von ca. 2-4 MByte. Aus diesem Grund wurde in der dritten Phase dieser Arbeit ein Linux-Betriebssystem, für die Ausführung auf einem LEON3-SoC, aufbereitet. Eine sehr gute Lösung konnte in dem „SnapGear“-Linux gefunden werden. Es existieren bereits Arbeiten, wie [LeonRob] und [LinuxLeon], in denen das SnapGear-Linux erfolgreich, mit dem LEON3, eingesetzt wurde. Außerdem kann es auf der Homepage der Firma GR heruntergeladen werden und wird regelmäßig aktualisiert.
Grundsätzlich ist die Kompilierung des SnapGear-Linux nicht von der Synthese eines LEON3-SoC im EDK abhängig. Ein laufendes LEON3-SoC kann mit dem GRMON debuggt und ein externes Linux-Betriebssystem eingespielt werden. Da die beiden Workflows (Synthese eines LEON3-SoC und Kompilierung des SnapGear-Linux) vollkommen unabhängig sind, wurden sie auch in dieser Arbeit getrennt aufbereitet. Das heißt, ein LEON3-SoC wird unabhängig von jeglicher Software mit dem EDK synthetisiert. Durch einen anderen Workflow wird das SnapGear-Linux kompiliert und extern, durch das Debugging, in das LEON3-SoC eingespeist.

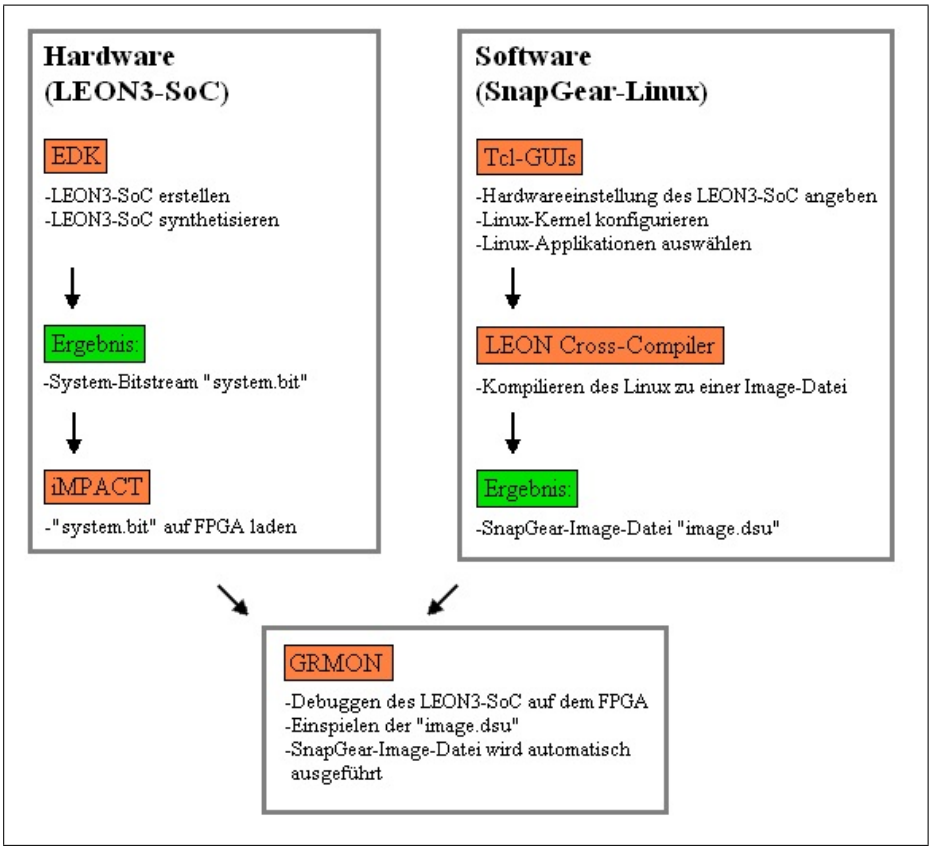

Abbildung 4.2: Workflows: Synthese eines LEON3-SoC und Kompilierung des SnapGear-Linux

Abbildung 4.2 zeigt den Ablauf der beiden Workflows. Im linken Teil der Abbildung 4.2 wird ein LEON3-SoC mit dem EDK synthetisiert. Als Ergebnis entsteht der System-Bitstream „system.bit“. Da für das Linux keine BRAM-Daten gebraucht werden, muss der EDK-Arbeitsplan nur bis zum Arbeitsschritt 2 abgearbeitet werden. Die „system.bit“ kann dann mit der Software Xilinx „iMPACT“ auf einem FPGA konfiguriert werden.

Im rechten Teil der Abbildung 4.2 ist der Workflow, zur Kompilierung des SnapGear-Linux zu erkennen. Mit Hilfe von Tcl-GUIs werden diverse Einstellungen des SnapGear-Linux vorgenommen. Anschließend wird das Linux kompiliert und es entsteht eine Image-Datei „image.dsu“. Diese Image-Datei kann dann mit GRMON in den System-Speicher des konfigurierten LEON3-SoC geladen und ausgeführt werden.

4.4 Verbinden des LEON3-Systems mit einem MicroBlaze-System

Von EDK-Benutzern wurden in der Vergangenheit nicht nur die IP-Cores der Xilinx-Bibliothek verwendet. Viele SoC-Entwickler haben unter dem EDK bereits eigene IP-Cores implementiert, welche neu entwickelte, externe Hardware ansteuern. Diese eigenen IP-Cores wurden so konzipiert, dass sie in Verbindung mit dem MicroBlaze-System funktionieren. Praktisch heißt das, dass die eigenen IP-Cores an den OPB angeschlossen und so vom MicroBlaze-Prozessor kontrolliert werden. Sollten diese eigenen IP-Cores direkt an den AHB des LEON3 angeschlossen werden, so müssten sie aufwendig an den AHB angepasst werden. Daher ist es wünschenswert, dass die entwicklereigenen IP-Cores, auch nach einem Umstieg auf das LEON3-System noch über den OPB angeschlossen werden können. Zu diesem Zweck bietet es sich an, den AHB eines LEON3-Systems mit dem OPB des MicroBlaze-Systems über eine Bridge zu verbinden.

Durch die sog. AHB2OPB-Bridge wird innerhalb eines LEON3-SoC ein Systemspeicher reserviert, der nur für die Kommunikation zwischen dem AHB und dem OPB benutzt wird. Dadurch können IP-Cores für den OPB auch weiterhin benutzt werden und der Umstieg auf das LEON3-System wird erleichtert.

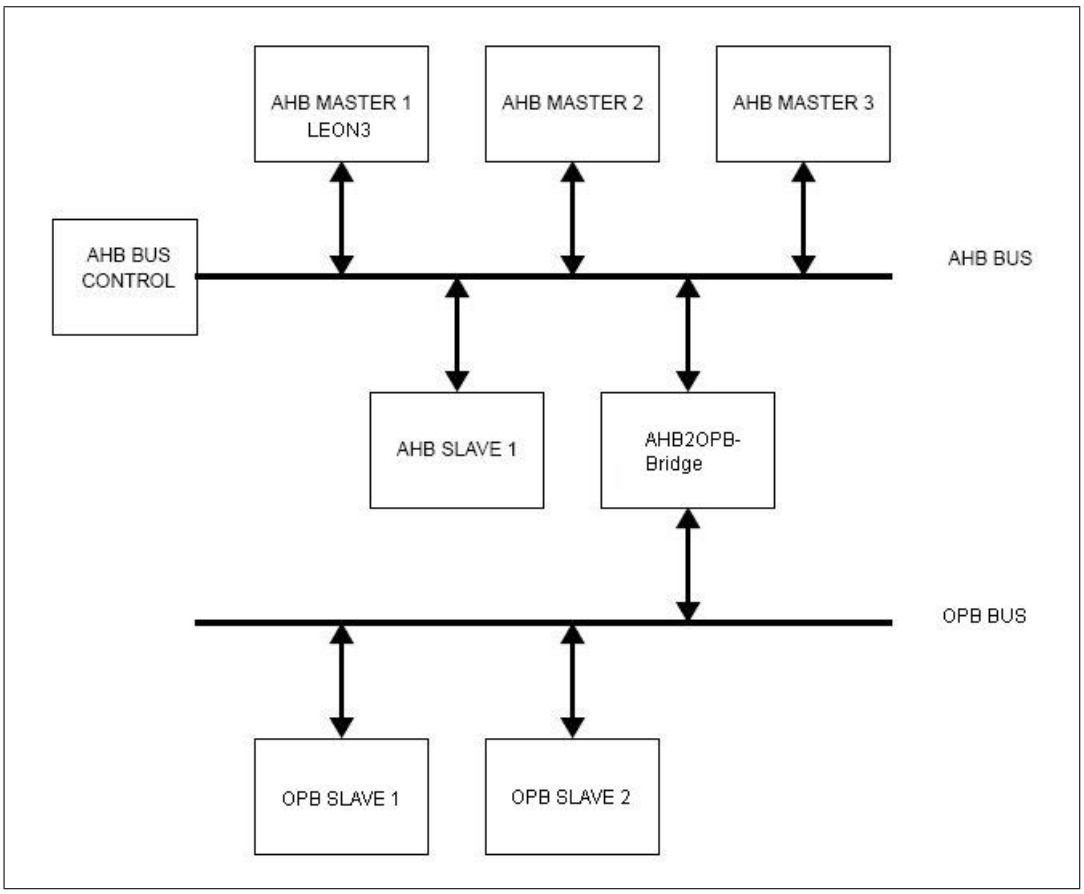

Abbildung 4.3: Verbindung des LEON3-Systems mit dem MicroBlaze-System über die AHB2OPB-Bridge

Die Abbildung 4.3 stellt die Verbindung der beiden Systeme schematisch dar. Es ist zu erkennen, dass der MicroBlaze-Prozessor als OPB-Master nicht mehr benötigt wird. Die Aufgaben des MicroBlaze-Prozessors können vollständig vom LEON3-Prozessor übernommen werden.

Kapitel 5

Einbindung der GRLIB in das EDK

In diesem Kapitel wird erläutert wie alle relevanten IP-Cores der GRLIB in das EDK eingebunden wurden. Das Vorgehen und Entscheidungen gegenüber von Problemenlösungen, bauen hauptsächlich auf den Erläuterungen des Kapitels 2 (Grundlagen) und des Kapitels 3.1 (Die Gaisler Research IP Library) auf.

5.1 Ausgangspunkt und Ansatz

In Kapitelpunkt 2.2.2 wurden alle EDK-Konstrukte erläutert, welche benötigt werden, um einen eigenen IP-Core in EDK einzubinden.

Zu diesen Konstrukten gehören:

- Die MPD, sie beschreibt alle Interfaces und Optionen des IP-Cores.
- Die PAO, ist eine Zusammenfassung aller VHD-Dateien, welche zur Synthese des IP-Cores benötigt werden.
- Die MDD, wird optional angelegt, falls für den IP-Core ein Treiber zur Verfügung stehen soll.
- Die Tcl, kann dem IP-Core hinzugefügt werden, um Generic-Konstanten zu setzen oder automatisch Constraints festzulegen.

Für jeden einzelnen IP-Core der GRLIB galt es, diese Konstrukte zu benutzen um eine optimale Benutzung durch das EDK zu bewirken.
Grundsätzlich wird für die Einbindung eines neues IP-Cores ein neue IP-Core-Verzeichnis angelegt, dass alle Konstrukte in seinen Unterverzeichnissen beherbergt. Die Konstrukte MPD, PAO

und Tcl werden dabei im „data“-Verzeichnis des neuen IP-Core-Verzeichnisses gespeichert, dort werden sie automatisch von den EDK-Tools, welche während des EDK-Arbeitsplans benutzt werden, gesucht.

Die IP-Core-Verzeichnisse der Xilinx-Bibliothek befinden sich im Verzeichnis: „[XILINX_EDK]/hw/XilinxProcessorIPLib/pcores“. In dieses Verzeichnis konnten keine eigenen IP-Cores hinzugefügt werden, da dieser nicht erkannt wurden und dort nicht benutzt werden können. Die Aufnahme neuer IP-Cores und deren Konstrukte muss stattdessen in den Unterverzeichnissen „pcores“, „bsp“ und „drivers“ des „[EDK_PROJECT]“-Verzeichnis geschehen.

Dabei gelten folgende Assoziationen:

Verzeichnis	Beschreibung
[EDK_PROJECT]/pcores	beinhaltet alle IP-Cores und Bibliotheken, die innerhalb des Projektes benutzt werden sollen
[EDK_PROJECT]/bsp	speichert die benötigten BSP. Hier wird ein weiteres Konstrukt namens **M**icroprocessor **L**ibrary **D**efinition (kurz MLD) abgelegt
[EDK_PROJECT]/drivers	bewahrt gegebenenfalls Treiber für einzelne IP-Cores auf. Treiber enthalten jeweils das MDD Konstrukt.

Tabelle 5.1: Für die IP-Cores relevante Verzeichnisse

Wird ein EDK-Projekt geladen, so werden die Verzeichnisse, aus Tabelle 5.1, durchsucht und alle ordnungsgemäß erkannten IP-Cores in den EDK Oberfläche aufgenommen. Gegebenenfalls werden Fehlermeldungen angezeigt, sollte ein IP-Core nicht fehlerfrei erkannt worden sein.
Wird ein IP-Core erfolgreich erkannt, so wird für ihn, ein Eintrag in den „IP Catalog“ in der Kategorie „Project Respository“ angelegt. Die Verzeichnisse, aus Tabelle 5.1, sind trotz ihres abweichenden Aufbewahrungsortes genauso aufgebaut wie in Kapitelpunkt 2.2.2 erläutert.
Die EDK-Konstrukte MHS und MSS müssen nicht bearbeitet werden, da deren Inhalte durch die Arbeit mit der EDK-GUI automatisch angepasst werden.

Nachdem alle IP-Cores eingebunden wurden galt es den Arbeitsplan des EDK nach auftretenden Fehlern zu prüfen und gegebenenfalls Änderungen an diesem vorzunehmen. Der EDK-Arbeitsplan konnte für ein GRLIB-SoC vollständig innerhalb der „system.make“ überarbeitet werden.

5.1.1 Erstellung eines IP-Cores mit dem „Create and Import Peripheral Wizard“

Der herkömmliche Weg einen neuen IP-Core im EDK zu erstellen und ihn im „[EDK_PROJECT]/pcores“-Verzeichnis abzulegen ist der „Create and Import Peripheral

Wizard“. Er kann über den Menüpunkt:

„Hardware“ $\Longrightarrow$ „Create or Import Peripheral“

innerhalb des EDK aufgerufen werden.

Dieser Wizard übernimmt das benutzerfreundliche Anlegen eines IP-Core-Verzeichnisses und passt die EDK-Konstrukte automatisch an. Der „Create and Import Peripheral Wizard“ ist für die Einbindung der GRLIB-IP-Cores nicht geeignet, da folgende Vorraussetzungen erfüllt sein müssen:

- Es können nur Peripheriekomponenten erstellt oder hinzugefügt werden, jedoch keine Prozessoren oder Busse.
- Die erstellten Komponenten müssen an einem Bus mit dem Standard OPB, PLB oder FSL angeschlossen werden. Eine Definition von eigenen Standards ist nicht erlaubt.
- Es wird eine Menge von Ports vorgegeben, welche der neue IP-Core treiben oder lesen kann. Das Hinzufügen eigener Ports ist nicht gestattet.

Diese Voraussetzung sind bei den IP-Cores der GRLIB nicht erfüllt und machten den Wizard, für den Einbindungsprozess unbrauchbar.

5.2 Einbindung eines GRLIB-IP-Cores in das EDK

In diesem Kapitelpunkt werden alle Schritte der Reihe nach erläutert, die für jeden GRLIB-IP-Core abgearbeitet werden mussten, um ihn in das EDK einzubinden. Sollte es nötig sein, einen weiteren IP-Core aus der GRLIB in das EDK einzubinden, so müssen diese Schritte wiederholt ausgeführt werden. Die Schritte umfassen die Anpassungen, der einzelnen EDK-Konstrukte, eines neuen EDK-IP-Cores auf die GRLIB.
Es werden die Anpassungen der EDK-Konstrukte erläutert und auf Probleme hingewiesen. Die Probleme beim Einbindungsprozess werden im Kapitelpunkt 5.3 genauer erklärt und gelöst.

5.2.1 Bibliothek-IP-Cores

Die GRLIB beinhaltet Hunderte von VHD-Dateien, unter denen ein großer Anteil Pakete und Bibliotheken sind. Des weiteren werden in jedem GRLIB-IP-Core Bibliotheken eingebunden, die für dessen Synthese benötigt werden. Zu den wichtigsten Paketen gehört zum Beispiel das „amba“ Paket, das in der Bibliothek „grlib“ aufbewahrt wird. Dieses Paket wird benötigt, um die Master- und Slave-Interface-Definitionen des AHB und APB benutzen zu können.

```
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library techmap;
use techmap.gencomp.all;
```

Abbildung 5.1: typischer Bibliothekenkopf eines GRLIB-IP-Cores

Abbildung 5.1 zeigt einen typischen, minimalen Kopf eines GRLIB-IP-Core. Durch die Deklaration der, in Abbildung 5.1, aufgelisteten Pakete ist es möglich einen IP-Core an den AHB oder APB anzuschließen.
Da die GRLIB unter der Bedingung eingebunden wurde, dass durch diese Arbeit keine Änderungen an den GRLIB VHD-Dateien vorgenommen werden sollten, mussten die GRLIB-Bibliotheken in irgendeiner Weise verfügbar gemacht werden. Dies geschah durch das Hinzufügen von IP-Cores, die als Bibliotheken dienen. Solche IP-Cores werden auch Bibliothek-IP-Cores genannt.

Bibliothek-IP-Cores gehören zu der üblichen Verfahrensweise, welche auch von Xilinx angewendet wird. Ein Beispiel an dem dies deutlich wird, ist der IP-Core „opb_uartlite“. In dessen IP-Core-Unterverzeichnis „data“ befindet sich die PAO benannt mit „opb_uartlite_v2_1_0.pao“. In der PAO wird auf VHD-Dateien verwiesen, die zur Synthese des IP-Cores bereitstehen sollen.

Ein Ausschnitt der „opb_uartlite“ PAO zeigt folgende Zeilen:

```
lib common_v1_00_a pselect
lib opb_uartlite_v1_00_b baudrate
lib opb_uartlite_v1_00_b srl_fifo
```

Abbildung 5.2: Ausschnitt der „opb_uartlite“ PAO

In Abbildung 5.2 ist zu erkennen, dass die VHD-Dateien „srl_fifo.vhd“ und „baudrate.vhd“ zur Synthese des „opb_uartlite“ IP-Cores benötigt werden. Diese befinden sich im „hdl“-Verzeichnis des „opb_uartlite“ IP-Cores. Eine weitere benötigte VHD-Datei ist die „pselect.vhd“ (in Abbildung 5.2 rot markiert). Diese Datei befindet sich allerdings im IP-Core-Verzeichnis „common_v1_00_a“. Dieser IP-Core ist ein reiner Bibliothek-IP-Core, was daran zu erkennen ist, dass dieser IP-Core keine MPD besitzt und somit nicht im „IP Catalog“ zu sehen ist.

Die VHD-Datei „pselect.vhd“ wird nun vom „opb_uartlite“ IP-Core durch die Zeilen:

```
library Common_v1_00_a;
use Common_v1_00_a.pselect;
```

eingebunden.

Mit diesem Weg ist es also möglich GRLIB Bibliothek-IP-Cores zu erstellen und alle darin befindlichen VHD-Dateien für die Synthese im EDK bereitzuhalten. Dabei sind folgende, in Tabelle 5.2 aufgelistete, Bibliothek-IP-Cores entstanden:

„esa, gleichmann, openchip, opencores“ Hierin befinden sich Bibliotheken die von verschiedenen Firmen oder Einrichtungen veröffentlicht wurden.
„fpu“ In dieser Bibliothek werden Kontroller und Schnittstellen für die LEON3 FPU gespeichert, leider nicht die FPU selbst. In dieser Bibliothek könnte eine zukünftige FPU platziert werden.
„gaisler, grlib“ Dies sind Bibliotheken von GR, in der unter anderem der LEON3-Prozessor und andere GRLIB-IP-Cores gespeichert sind.
„tech, techmap“ Enthalten Bibliotheken, die zur Unterstützung der Technologie-Unabhängigkeit benötigt werden.

Tabelle 5.2: EDK Bibliothek-IP-Cores der GRLIB

Wird nun ein GRLIB-IP-Core in das EDK eingefügt so kann er, bei entsprechender Anpassung seiner PAO, auf alle GRLIB-Bibliotheken zugreifen. Die Bibliothek-IP-Cores sind in der „Project Respository“-Kategorie nicht sichtbar, da sie keine MPD besitzen.

5.2.2 IP-Core-Verzeichnis

Für jeden GRLIB-IP-Core musste, zur Einbindung in das EDK, ein extra IP-Core-Verzeichnis angelegt werden. Der Verzeichnisname muss dem Muster „[name_ip]_v?_??_?“[1] entsprechen.
Für die Benennung des IP-Cores mit [name_ip], wurde der Name der obersten IP-Core-Entity gewählt mit einem Zusatz „_if“. Für den LEON3-Prozessor ergibt sich somit ein IP-Core-Name ([name_ip]) von „leon3s_if“ oder für den AHB „ahbctrl_if“.
Der Zusatz „_if“ steht für „**I**nter**f**ace“ und entstand aus einem Problem heraus, dass während der Einbindung der GRLB auftrat. Es wird das „Interface-Entity-Problem“ genannt und wird in Kapitelpunkt 5.3.1 genauer erläutert.

[1] Die Fragezeichen entsprechen der Versionsnummer des Eingebundenen IP-Cores und wurden alle mit „1_00_a“ festgelegt.

Die Namen lassen unter Umständen nicht sofort erkennen, um welchen GRLIB-IP-Core es sich handelt. Allerdings ist diese Wahl optimal, um das entsprechende Kapitel in der GRLIB Dokumentation [gripdoc] sofort zu erkennen.

Die IP-Core-Verzeichnisse der eingebundenen GRLIB-IP-Cores verfügen über das zwingende „data“-Unterverzeichnis in dem die MPD gespeichert wird. Im IP-Core-Unterverzeichnis „hdl/vhdl“ wurden, während der Einbindung, grundsätzlich zwei VHD-Dateien eingefügt. Diese Dateien sind die:

„[name_ip].vhd“ Dies ist die oberste VHDL-Entity, des GRLIB-IP-Cores, direkt aus der GRLIB kopiert.

„[name_ip]_if.vhd“ Dies ist die oberste VHDL-Interface-Entity, des in EDK eingebundenen GRLIB-IP-Cores. Sie bindet als Unter-Entity die „name_ip].vhd“ ein.

Die Interface-Entity musste bei der Einbindung der GRLIB-IP-Cores eingeführt werden, da die verwendeten VHDL-Beschreibungsstile, der GRLIB und des EDK, inkompatibel sind. Diese Diskrepanz musste mit einer Art Adapter-Entity ausgeglichen werden.

Auf das IP-Core-Unterverzeichnis „doc“ wurde bei allen IP-Cores verzichtet, da die Gesamtdokumentation [gripdoc], aller GRLIB-IP-Cores, im „doc“-Verzeichnis des „leon3s_if“ IP-Cores, unter dem Namen „leon3s_if.pdf“ erhältlich ist.

5.2.3 MPD-Anpassungen

Alle Anpassungen in der MPD haben Einfluss auf die Handhabung des IP-Cores innerhalb des EDK und auf die Generierung der Wrapper-Entitys sowie der System-Entity. Die Synthese wird durch die MPD nur indirekt beeinflusst.

5.2.3.1 OPTION

IPTYPE

Für die Synthese und das Funktionieren eines SoC sind nur die Entitys der einzelnen SoC-Komponenten verantwortlich. Die Typen der Entitys (Prozessor, Bus, Bridge, ...) sind theoretisch nicht von Belang.
Eine Ausnahme bildet das EDK. Hier wird zwischen den Typen der IP-Cores unterschieden, da zum Beispiel die „Bus Interface“-Ansicht entsprechend gestaltet werden muss. Grundsätzlich werden Busse und andere IP-Cores in der „Bus Interface“-Ansicht unterschiedlich dargestellt.
Die Festlegung der Typen, einzelner IP-Cores, geschieht mit dem „OPTION“ Elementen, durch die Festlegung des Schlüsselworts „**IPTYPE**“.

Es gibt vier mögliche Werte (Typen), die für die GRLIB-IP-Cores relevant sind:

„IPTYPE“ Wert	**IP-Core Bedeutung**
BUS_ARBITER	Busse
PROCESSOR	Prozessoren
BRIDGE	Bridges die Busse verbinden
PERIPHERAL	andere periphere IP-Cores, meist Slaves an Bussen

Tabelle 5.3: mögliche Typen eines EDK-IP-Cores

Tabelle 5.3 zeigt alle Möglichkeiten des „IPTYPE“ Schlüsselwortes, welche in der MPD eines IP-Cores deklariert werden können. Wird nun ein GRLIB-IP-Core mit dem Typ „BUS“ eingebunden, so würde seine MPD folgende Zeile beinhalten:

```
OPTION IPTYPE = BUS_ARBITER
```

Dadurch wird der IP-Core in der „Bus Interface“-Ansicht mit der typischen Darstellung eines Busses visualisiert.

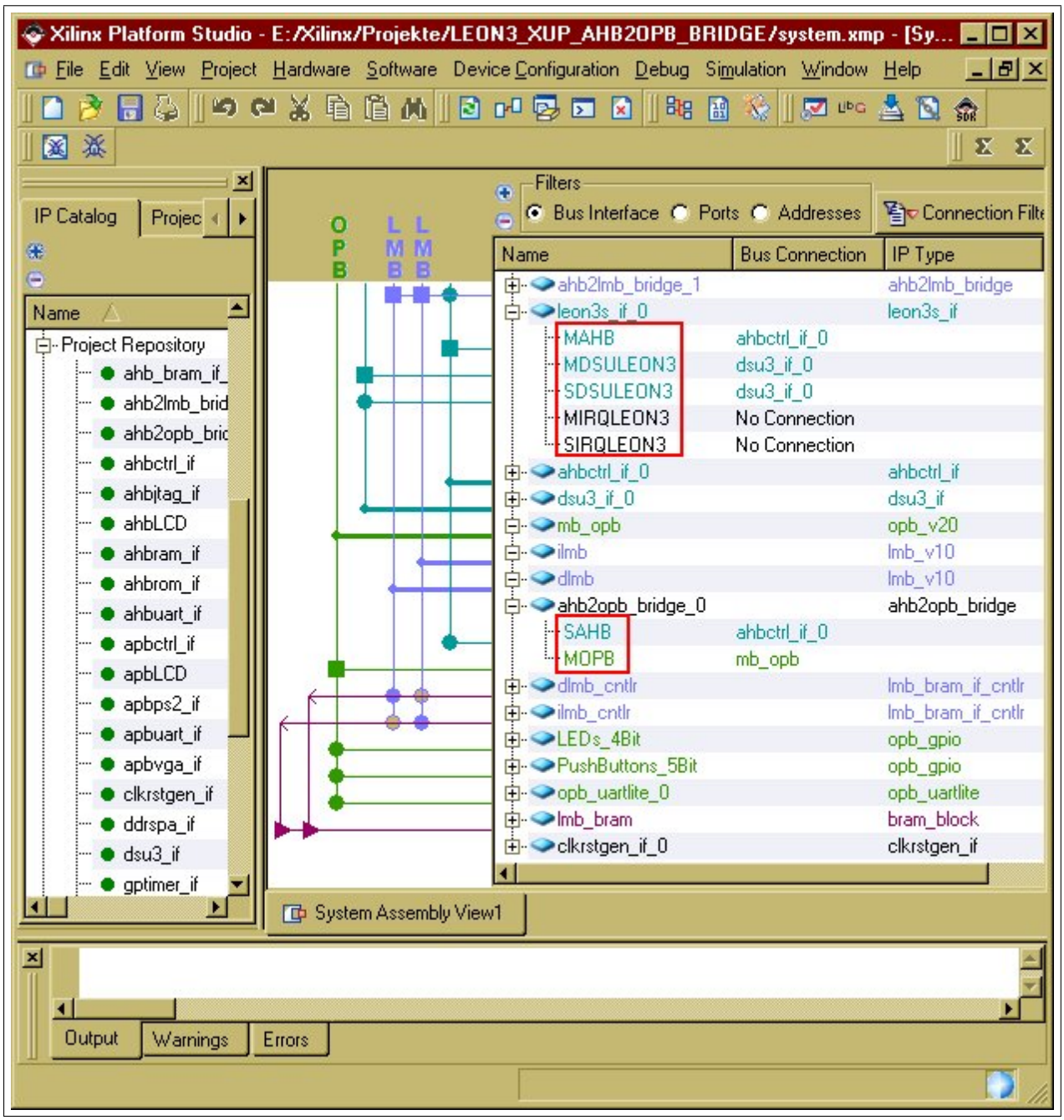

Abbildung 5.3: EDK GUI der GRLIB mit AHB und OPB

Abbildung 5.3 zeigt eine EDK-GUI mit einem GRLIB-SoC. IP-Cores wie der „ahbcrtl_if“ IP-Core sind vom Typ „BUS_ARBITER“ und werden als Bus dargestellt.

Der Typ „BRIDGE“ ermöglicht eine spezielle Behandlung der SoC-IP-Cores. Das EDK erkennt, dass eine Bridge zwei Busse verbindet und kann dadurch feststellen, ob der Prozessor des SoC Zugriff auf einen bestimmten IP-Core vom Typ „PERIPHERAL“ besitzt. Dieser Peripherie-IP-Core muss nicht unbedingt mit dem selben Bus verbunden sein, mit dem der Prozessor verbunden ist.
Wäre die AHB2OPB-Bridge, in Abbildung 5.3 („ahb2opb_bridge“), nicht vom Typ „BRIDGE“,

sondern vom Typ „PERIPHERAL“ so würde sie auch funktionieren, allerdings könnte das EDK nicht erkennen, dass der LEON3 Zugriff auf die „opb_uartlite“ hat.

Bei der Einbindung eines jeden GRLIB-IP-Cores musste herausgefunden werden, welchen Typs der IP-Core ist. Dementsprechend musste sein Schlüsselwort entsprechend festgelegt werden.

BUS_STD

IP-Cores vom Typ „BUS_ARBITER“ können mit dem „BUS_STD“ Schlüsselwort einen eigenen Busstandard definierten. Verfügen andere IP-Cores über ein Bus-Interface mit diesem Standard, so können sie an den Bus angeschlossen werden. Der AHB-IP-Core der GRLIB besitzt beispielsweise in seiner MPD folgende Zeile:

```
OPTION BUS_STD = AHB
```

Dadurch wird der neue Busstandard „AHB“ eingeführt.

Busse der GRLIB verfügen über die Portvektoren:

Portvektoren	Bus
„ahb_mst_out_vector“, „ahb_slv_out_vector“	AHB
„apb_slv_out_vector“	APB

Tabelle 5.4: Identifikation eines Busses aus den GRLIB Portvektoren

5.2.3.2 BUS_INTERFACE

Ein beliebiger IP-Core kann über beliebig viele Bus-Interfaces verfügen. Über diese Bus-Interfaces kann er als Master oder Slave mit einem Busstandard, also einem Bus, verbunden werden. Die Bus-Interfaces, welche beim Einbindungsprozess am häufigsten deklariert wurden, sind mit ihrer MPD-Syntax in Abbildung 5.4 dargestellt:

```
BUS_INTERFACE BUS = MAHB, BUS_STD = AHB, BUS_TYPE = MASTER
BUS_INTERFACE BUS = SAPB, BUS_STD = APB, BUS_TYPE = SLAVE
```

Abbildung 5.4: Bus-interface dar GRLIB-IP-Cores

Für jedes Interface, über das ein IP-Core verfügen soll, musste eine der Zeilen aus Abbildung 5.4 der MPD hinzugefügt werden. Ein IP-Core nach dem Beispiel, aus Abbildung 5.4, könnte

als Master an den AHB und als Slave an den APB angeschlossen werden. Die Bus-Interface-Deklarationen, in Abbildung 5.4, führen zu Anschlussmöglichkeiten, die in der „Bus Interfaces“-Ansicht dargestellt werden (rote Rahmen in Abbildung 5.3).

Jeder GRLIB-IP-Core der über die, in Kapitelpunkt 3.1.5.2 und 3.1.5.3 erläuterten, Ports aus Tabelle 5.5 verfügt, wurde mit einer entsprechenden Bus-Interface-Deklaration ausgestattet.

Ports	Bus-Interface
„ahb_mst_in_type“, „ahb_mst_out_type“	AHB-Master
„ahb_slv_in_type“, „ahb_slv_out_type“	AHB-Slave
„apb_slv_in_type“, „apb_slv_out_type“	APB-Slave

Tabelle 5.5: Identifikation eines Bus-Interfaces aus den GRLIB Ports

5.2.3.3 PARAMETER

Die „PARAMETER“ Elemente der MPD entsprechen den Generic-Konstanten eines IP-Cores. Alle Generic-Konstanten eines GRLIB-IP-Cores mussten mit einem MPD „PARAMETER“ Element aufgeführt werden, um sie mit dem „Configure IP ...“-Dialogfeld festlegen zu können.

Parameterbenennung

Die Benennung der „PARAMETER“ Elemente musste leider im hohen Maße benutzerunfreundlich erfolgen. Dies hatte folgendem Grund.
Um einen Bezug der Parameter, welche im „Configure IP ...“-Dialogfeld einstellbar sind, zu den Generic-Konstanten der IP-Cores herzustellen, mussten die Parameter gleichnamig gewählt werden. Dadurch ist es ebenfalls besser möglich einen Parameter mit seiner Beschreibung in der [gripdoc] Dokumentation zu finden. Um die Benutzerfreundlichkeit der GRLIB-IP-Cores zu erhöhen, wurden in der MPD zusätzliche Bereichsbeschränkungen angegeben.

Die Xilinx-IP-Cores kombinieren diesen Benennungsvorteil mit einer benutzerfreundlichen Beschreibung aller „PARAMETER“ Elemente, in dem ein spezielles XML Dokument namens „[name_ip]_v2_1_0.mui“ im Verzeichnis „[XILINX_EDK]/data/mdtgui/pcores/[name_ip]_v1_00_a/data“ bereitsteht, welches das „Configure IP ...“-Dialogfeld benutzerfreundlich aufbereitet. Es ist in dieser Arbeit nicht gelungen die MUI-Dateien auf die GRLIB-IP-Cores anzuwenden.

Parameter und die Tcl

Ein weiterer Punkt der die Benutzerfreundlichkeit steigert ist das Verlinken eines Parameters mit einer Tcl-Prozedur. Diese Tcl-Prozedur bestimmt den Wert einer Generic-Konstante automatisch. Parameter, die durch eine Tcl-Prozedur automatisch eingestellt werden sollen, müssen mit einem Zusatz an dem entsprechenden „PARAMETER“ Element der MPD versehen werden.

```
PARAMETER NUM = 16, DT=integer, ASSIGNMENT = UPDATE,
SYSLEVEL_UPDATE_VALUE_PROC = update_num
```

Abbildung 5.5: Generic-Konstante mit Tcl-Prozedur

Abbildung 5.5 zeigt den Zusatz (rot markiert), der an einem „PARAMETER“ Element vorgenommen werden muss, um die Generic-Konstante durch eine Tcl-Prozedur berechnen zu lassen.
Der in der Abbildung 5.5 rot markierte Zusatz kennzeichnet, dass der Parameter „NUM“ vor dem Start der Synthese mit dem Rückgabewert der Tcl-Prozedur „update_num“ überschrieben wird.

5.2.3.4 PORT

Jeder GRLIB-IP-Cores besitzt Ports mit denen er logische Pegel liest oder treibt. Damit ein Port mit dem EDK verarbeitet werden kann, muss er in der MPD mit dem Element „PORT“ deklariert sein. Alle Ports eines GRLIB-IP-Cores mussten, bei der Einbindung, mit dieser Methode deklariert werden.

Zu Beachten 5.1 (Ports der GRLIB-IP-Cores)

EDK erlaubt keine Deklaration von Ports, welche ein Record sind. Wurden solche Records in einem GRLIB-IP-Cores gefunden, so mussten diese vollständig in der MPD aufgeschlüsselt werden. In der MPD dürfen nur Ports vom Typ „std_logic“, „std_ulogic“ oder Vektoren davon deklariert werden.

Dieser Sachverhalt führte zum „Interface-Entity-Problem“, da die Bus-Interfaces der GRLIB-IP-Cores grundsätzlich mit Ports realisiert wurden, welche Records sind.

Gehören Ports zu einem Bus-Interface, so können diese leicht, mit der EDK GUI, in der „Bus Interfaces“-Ansicht verdrahtet werden. Dazu muss lediglich ein Zusatz am Ende der MPD Port-Deklaration erfolgen:

```
PORT apbo_prdata = apbo_prdata, DIR = O, VEC = [31:0], BUS = SAPB
```

Der obige Port „apbo_prdata“ ist der Daten-Port des APB-Slave-Interface. Mit dem Zusatz „BUS = SAPB“ wird im Arbeitsschritt 1 des EDK-Arbeitsplans eine automatische Verdrahtung des Ports vorgenommen. Die Zeichenkette „SAPB“ muss dabei ein zuvor deklariertes Bus-Interface sein.

IO-Ports

Manche IP-Cores der GRLIB verfügen über IO-Ports. Dies sind bidirektionale Port. Über sie können logische Pegel gelesen oder geschrieben werden. Das Grundelement um einen IO-Port zu erzeugen ist das Schaltungselement „IOBUF“, dieses hat folgenden Schnittstellen:

```
entity IOBUF is
  port ( O  : out std_ulogic; IO : inout std_logic; I, T : in std_ulogic);
end;
```

Abbildung 5.6: Schnittstellen eines IOBUF

Der IOBUF, in Abbildung 5.6, besitzt einen „IO“ Port vom Typ „inout“. Dieser Port ist der bidirektionale Port. Der „IO“ Port wird mittels eines Ausgangsport „O“ und zwei Eingangsports „I“ und „T“ betrieben. „O“ und „I“ sind die logischen Pegel, welche über die bidirektionale Leitung getrieben werden sollen. „T“ gibt an welcher Pegel gerade auf der IO-Leitung ausgeführt wird („I“ oder „O“).

Es gibt zwei Wege einen IO-Port in einem EDK-IP-Core einzubinden:

1. In der MPD des IP-Cores wird ein fiktiver IO-Port definiert mit dem Zusatz „THREE_STATE = TRUE“:

   ```
   PORT GPIO_IO = , DIR = IO, THREE_STATE = TRUE, TRI_I =
   GPIO_IO_I, TRI_O = GPIO_IO_O, TRI_T = GPIO_IO_T
   ```

   Abbildung 5.7: MPD-Eintrag für einen IO-Port

   Abbildung 5.7 zeigt einen Ausschnitt der MPD. Es wird der „GPIO_IO“ Port des „opb_gpio“ IP-Cores deklariert.
   Dabei müssen die Ports „TRI_*“, welche in Abbildung 5.7 mit „TRI_I“, „TRI_O“ und „TRI_T“ bezeichnet sind, echte Ports des IP-Cores sein.
   Wird nun ein externer Port mit einem fiktiven IO-Port in der „Ports“-Ansicht des EDK verbunden, so fügt das EDK automatisch ein IOBUF Element in die „system.vhd“ ein, das den echten IO-Port erzeugt und die „TRI_*“ Ports des IP-Core treibt.

2. Der IO-Port wird mittels eines IOBUF direkt im inneren eines IP-Cores erzeugt und ein echter IO-Port wird in der MPD des IP-Cores deklariert. Dazu muss der Zusatz „THREE_STATE = FALSE“ verwendet werden:

   ```
   PORT gpio = , DIR=IO, THREE_STATE = FALSE
   ```

   Obige Zeile befindet sich in der MDP des „gr_gpio_if“ IP-Core. Dadurch wird kein IOBUF Element in die „system.vhd“ eingefügt und der IO-Port wird als richtiger bidirektionaler Port behandelt.

Um die Technologie-Unabhängigkeit überall in gleicher Weise durchzusetzen, verwendet die GRLIB den 2. Weg. Dabei hat die Generic-Konstante „padtech“ Einfluss auf die konkrete Umsetzung des IOBUFs.

Zu Beachten 5.2 (Benennung Signal/externer Port bei IO-Port)

Wird ein IO-Port nach dem 2. Weg implementiert so ist folgendes zu beachten.
Der Signalname des IO-Ports muss identisch mit dem externen Port der „system.vhd“ sein. Dies macht sich in der „PORT“-Sektion der MHS bemerkbar:

„PORT“-Sektionen der MHS:
Möglichkeit nach dem 1. Weg:

```
PORT fpga_0_LEDs_4Bit_GPIO_IO_pin = fpga_0_LEDs_4Bit_GPIO_IO, DIR = IO
```

Vorgeschrieben nach dem 2. Weg:

```
PORT fpga_0_LEDs_4Bit_GPIO_IO = fpga_0_LEDs_4Bit_GPIO_IO, DIR = IO
```

5.2.4 PAO-Anpassungen

Ein GRLIB-IP-Core besteht nicht nur aus einer Entity. Es können gegebenenfalls mehrere Unter-Entitys existieren, die wiederum Unter-Entitys einbinden. Alle benötigten Entitys befinden sich letztendlich in einem GRLIB-Bibliothek-IP-Core.

Wird eine Unter-Entity für ein GRLIB-IP-Core benötigt, so musste folgende Zeile der PAO des IP-Core hinzugefügt werden:

`lib [BIBLIOTHEK]`[2] `[VHD-DATEINAME]`[3] `vhdl`

Beispiel: `lib grlib amba vhdl`

[2] Der Ausdruck [BIBLIOTHEK] steht dabei für den Namen eines Bibliothek-IP-Core oder eines anderen IP-Cores.
[3] Der Ausdruck [VHD-DATEINAME] steht für eine VHD-Datei im „hdl/vhdl“ IP-Core-Unterverzeichnis des mit [BIBLIOTHEK] bezeichneten IP-Core.

Zu Beachten 5.3 (Bei GRLIB-IP-Cores Unter-Entitys einbinden)

Werden neue GRLIB-IP-Cores dem EDK hinzugefügt, ist darauf zu achten, dass bei der Synthese alle eingebundenen Unter-Entitys gefunden wurden. Wird eine Unter-Entity bei der Synthese nicht gefunden so wird nicht unbedingt ein Fehler angezeigt. Die Synthese wird dann trotzdem weiter fortgesetzt.

Es muss der Synthesereport, des neuen IP-Core, auf das Vorkommen der folgenden Zeile kontrolliert werden:

```
Instantiating black box module <[component_name]>
```

Wird mit der Zeichenkette [component_name] eine Entity oder VHD-Datei der GRLIB bezeichnet, so wurde diese **nicht** bei der Synthese berücksichtigt und es muss die Deklaration innerhalb der PAO überprüft werden.
Bezeichnet [component_name] allerdings eine elementare Komponente („RAMB16_S36_S36“, „OBUF“, „IBUF“ oder „IBUFG“ usw.), so besteht kein Problem, da diese Komponenten nicht weiter aufgelöst werden können.

Tipp: Die Namen der meisten elementaren Komponenten werden mit der VHDL Syntaxhervorhebung in einer anderen Farbe dargestellt. Es muss allerdings ein Xilinx-Editorfenster benutzt werden.

5.2.5 Tcl-Anpassungen

Die Tcl wird verwendet, um beliebige Generic-Konstanten automatisch durch die EDK-Tools festlegen zu lassen. Die Generic-Konstanten werden in der MPD mit dem „PARAMETER“ Element deklariert.

Je weniger Generic-Konstanten durch den Benutzer eingestellt werden müssen, desto geringer ist die Wahrscheinlichkeit eines Benutzerfehlers. Aus diesem Grund wurde bei der Einbindung der GRLIB darauf geachtet, dass so viele Generic-Konstanten wie möglich durch die Tcl automatisch eingestellt werden.
Mit der Tcl ist es auch möglich zusätzliche Dateien zu erstellen und über deren beliebigen Inhalt zu bestimmen. Dies wurde ausgenutzt, um nötige Constraints für bestimmte IP-Cores in eine UCF-Datei zu schreiben und diese der Synthese zu übergeben. Dadurch wird die „system.ucf“ von IP-Core-spezifischen Inhalten befreit und bleibt übersichtlich. Die „system.ucf“ sollte zum größten Teil nur FPGA-spezifische Informationen beinhalten, wie zum Beispiel die Deklaration von Pins. Gibt es UCF-Inhalte die nicht vom verwendeten FPGA abhängig sind, so sind diese meist IP-Core spezifisch. Diese Inhalte wurden in die Tcl des zugehörigen IP-Cores gebunden.

5.3 Probleme bei der Einbindung und Synthese der GRLIB

Zur Herstellung einer vollständigen Benutzbarkeit der GRLIB mit dem EDK, wurden immer wieder Probleme festgestellt und gelöst. Alle Probleme konnten zwei Quellen zugeordnet werden, welche sich während der Handhabung der GRLIB-IP-Cores mit dem EDK immer wieder eröffneten. Diese Quellen sind:

Quelle 1: *Einbindung eines GRLIB-IP-Cores*

Diese Quelle umfasst alle Probleme, die bei der Einbindung oder bei der Interpretation der IP-Cores aufgetreten sind. Da die GRLIB-IP-Cores geringfügige Unterschiede zu den Xilinx-IP-Cores aufweisen, mussten spezielle Anpassungen vorgenommen werden, um die GRLIB-IP-Cores im EDK optimal nutzen zu können.
Probleme dieser Quelle traten bei der Erstellung eines SoC mit der EDK GUI und während des EDK Arbeitsschritts 1 auf.

Quelle 2: *Synthese eines GRLIB-SoC im EDK*

In dieser Quelle werden alle Probleme zusammengefasst, die einen nicht funktionstüchtigen Bitstream erzeugten oder zum Abbruch der Synthese führten. Aber auch Probleme, die während der Erstellung der ELF-Dateien für das LEON3-System auftraten, werden hier eingeordnet.
Probleme dieser Quelle traten während der EDK Arbeitsschritte 2 bis 4 auf.

Tabelle 5.6: Problemquellen beim Eindindungsprozess

In diesem Kapitelpunkt wird eine Zusammenstellung aller Probleme erfolgen, die aus den Quellen der Tabelle 5.6 stammen. Es wird dabei eine chronologische Ordnung der Probleme vorgenommen, die dem zeitlichen Ablauf der SoC Erstellung gefolgt von der Synthese entspricht.
Zu allen Problemen wird die Entscheidung zur Lösung und die Begründung der Entscheidung diskutiert. Manche Lösungen führten zu geringfügigen Änderungen der GRLIB-VHD-Dateien.

5.3.1 „Interface-Entity-Problem“

Die Ports der Hauptentitäten aller GRLIP-IP-Cores müssen in ihren MPD-Dateien deklariert werden. Dazu zählen alle Bus-Interface-Ports oder Ports, welche die Pins des FPGA lesen oder treiben. Das Xilinx-EDK erlaubt, in der MPD, nur Deklarationen von Ports mit bestimmten Signal-Typen. Zu diesen Signal-Typen gehören nur „std_logic“, „std_ulogic“ sowie die Vektoren „std_logic_vector“ und „std_ulogic_vector“.

Das **Problem** dabei ist, dass die Haupt-Entitys, der GRLIB-IP-Cores, nicht nur Ports dieser Signal-Typen besitzen. Damit wird der Einbindungsprozess eines GRLIB-IP-Cores undurchführbar. Musterbeispiele für andere Ports sind die Bus-Interface Records.

```
signal clkm, rstn: std_ulogic;
signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_vector := (others => apb_none);
signal u1i, dui : uart_in_type;
signal u1o, duo : uart_out_type;

uart1 : apbuart                         -- UART 1
  generic map (pindex   => 1, paddr => 1, pirq => 2)
  port map (rstn, clkm, apbi, apbo(1), u1i, u1o);
```

Abbildung 5.8: Deklaration der APB Uart

Die Abbildung 5.8 zeigt einen Ausschnitt einer „leon3mp“-Entity. In dem GRLIB-SoC aus Abbildung 5.8 wird der „apbuart“ IP-Cores eingebunden. Es ist zu erkennen, dass die „apbuart“-Entity nicht nur normale Ports besitzt, die mit den Signalen „rstn“ und „clkm“ vom Typ „std_ulogic“ getrieben werden, sondern auch Ports von anderen Typen, wie „apb_slv_in_type“ oder „uart_in_type“. Diese nicht gewünschten Signal-Typen sind alle in GRLIB-Paketen definiert und sollen die Benutzerfreundlichkeit der GRLIB erhöhen.
Diese Benutzerfreundlichkeit wird, nach der Einbindung der IP-Cores in das EDK, nicht mehr benötigt, da die Bearbeitung eines GRLIB-SoC nicht mehr in Textform stattfindet.

5.3.1.1 Lösung des „Interface-Entity-Problems“

Das Problem wurde gelöst, indem eine Interface-Entity für jede GRLIB-IP-Core-Haupt-Entity entworfen wurde. Diese Interface-Entity ist nur für GRLIB-IP-Cores nötig, die über Ports verfügen, welche nicht den EDK erlaubten Signal-Typen entsprechen.
Da jeder GRLIB-IP-Core an irgendeinen Bus angeschlossen ist, musste für alle GRLIB-IP-Cores eine Interface-Entity erstellt werden. Wurde ein IP-Core mit einer Interface-Entity ausgestattet, so wurde der Zusatz „_if“ an den Namen des IP-Cores angehängt.

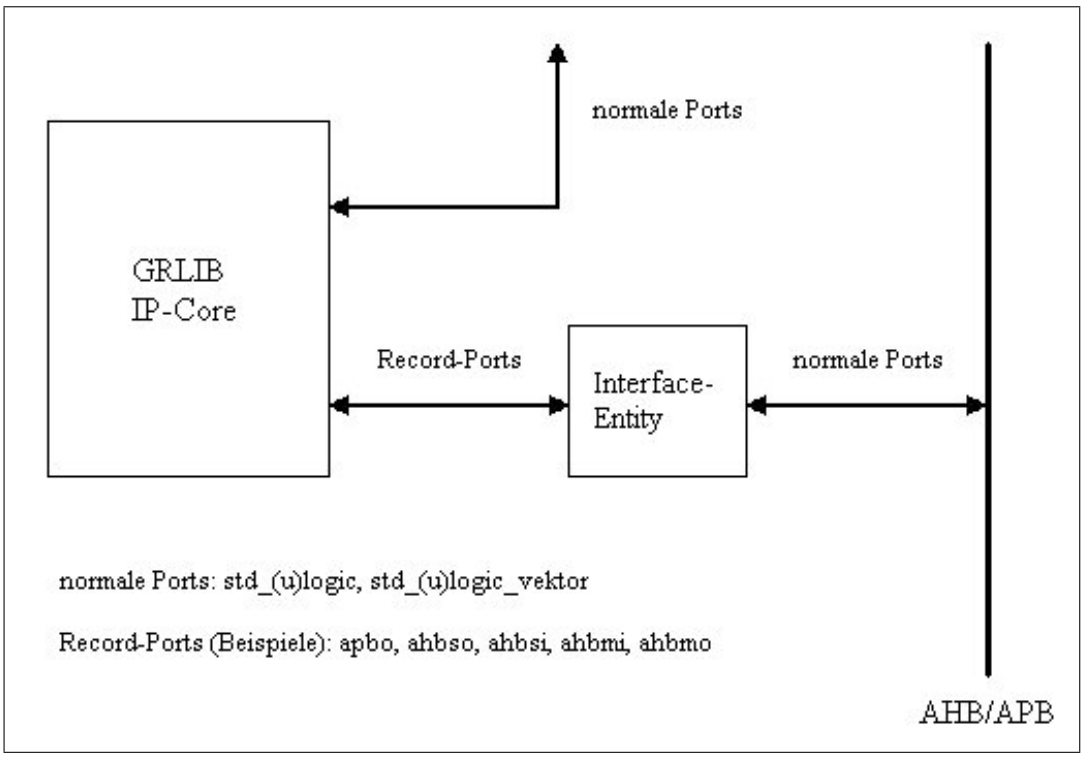

Abbildung 5.9: Schema Interface-Entity eines IP-Core

Abbildung 5.9 zeigt die schematische Stellung der Interface-Entity. Sie übernimmt dabei eine Art Adapterfunktion. Zu ihren Aufgaben gehört das Aufschlüsseln von Record-Ports und das Füllen von Record-Ports. Im folgenden wird zur Verdeutlichung das Beispiel eines Architekturteils einer Interface-Entity angegeben:

```
architecture struct of apbuart_if is

signal uarti : uart_in_type;
signal uarto : uart_out_type;
signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_type;

begin

  led_drive:process(uarti_rxd,uarto.txd)
  begin
    case uarti_rxd is
      when '1' =>
        led_rx<='1';
      when '0' =>
        led_rx<='0';
      when others =>
        led_rx<='0';
    end case;

    case uarto.txd is
      when '1' =>
        led_tx<='1';
      when '0' =>
        led_tx<='0';
      when others =>
        led_tx<='0';
    end case;
  end process;

  uarti.rxd <= uarti_rxd;
  uarti.ctsn <= uarti_ctsn;
  uarti.extclk <= uarti_extclk;

  uarto_rtsn <= uarto.rtsn;
  uarto_txd <= uarto.txd;
  uarto_scaler <= uarto.scaler;

  apbi.psel<=apbi_psel;
  apbi.penable<=apbi_penable;
  apbi.paddr<=apbi_paddr;
  apbi.pwrite<=apbi_pwrite;
  apbi.pwdata<=apbi_pwdata;
  apbi.pirq<=apbi_pirq;

  apbo_prdata<=apbo.prdata;
  apbo_pirq<=apbo.pirq;
  pconO:for i in O to NAPBCFG-1 generate
    apbo_pconfig(i*32+31 downto i*32)<=apbo.pconfig(i);
  end generate;
  apbo_pindex<=conv_std_logic_vector(apbo.pindex,log2(NAPBSLV));

  uart1 : apbuart                         -- UART 1
    generic map (pindex => pindex, paddr => conv_integer(paddr),
    pmask => conv_integer(pmask), console => console, pirq => pirq,
    parity => parity, flow => flow, fifosize => fifosize)
    port map (rst, clk, apbi, apbo, uarti, uarto);

end;
```

Abbildung 5.10: Architekturteil der Interface-Entity des „apbuart_if“ IP-Cores

Die Abbildung 5.10 zeigt den Architekturteil der „apbuart“-Interface-Entity. Im blau umrahmten Teil werden die Eingangsports des „apbuart_if“-IP-Cores in das von GR definierte „apb_slv_in_type“ Record gespeist.
Der braun umrahmten Teil zeigt wie die einzelnen „apb_slv_out_type“-Record-Elemente die Ausgangsports des „apbuart_if“-IP-Cores treiben.
Die eigentliche Einbindung der „apbuart“-Haupt-Entity findet im schwarz umrahmten Teil statt.
Der grün umrahmten Teil zeigt die Aufschlüsselung der übrigen Records. In diesem Fall sind dies die RX und TX Datenleitungen mit zusätzlichen Steuersignalen.

Folgende Vorteile bringt das Konzept der „Interface-Entity“ mit sich:

- Sie ermöglicht eine saubere Anbindung eines IP-Cores an den Bus. Werden also geringfügige Änderungen an der Ansteuerung vorgenommen, müssen diese nicht in der „system.vhd“ geschehen, wie es in der „leon3mp.vhd“ der Template-Designs üblich ist.
- Das Aufschlüsseln und das Füllen von Record-Ports benötigt keine weiteren LUTs, da die Vorgänge nur strukturelle Prozesse sind, welche keine zusätzliche Logik verwenden.
- Mit der Interface-Entity kann ein GRLIB-IP-Core komfortabel mit zusätzlicher Logik erweitert werden, ohne GRLIB VHD-Dateien zu verändern.
- Am rot umrahmten Teil der Abbildung 5.10 ist zu erkennen, dass der „apbuart_if“ IP-Core mit zusätzlichen Signalen und Ports zur Ansteuerung der RX- und TX-LEDs ausgestattet wurde. Diese zusätzliche Logik war in dem ursprünglichen „apbuart“-IP-Core nicht vorgesehen.

Die „leon3mp.vhd" der GRLIB-Template-Designs ist neben der reinen Verdrahtung der SoC-Komponenten, zusätzlich mit „Verwaltungslogik“ durchsetzt. Diese „Verwaltungslogik“ konnte in den Interface-Entitys gekapselt werden, so dass während des Arbeitsschritts 1, des EDK-Arbeitsplans, eine reine strukturelle „system.vhd“ durch das EDK erstellt werden kann.

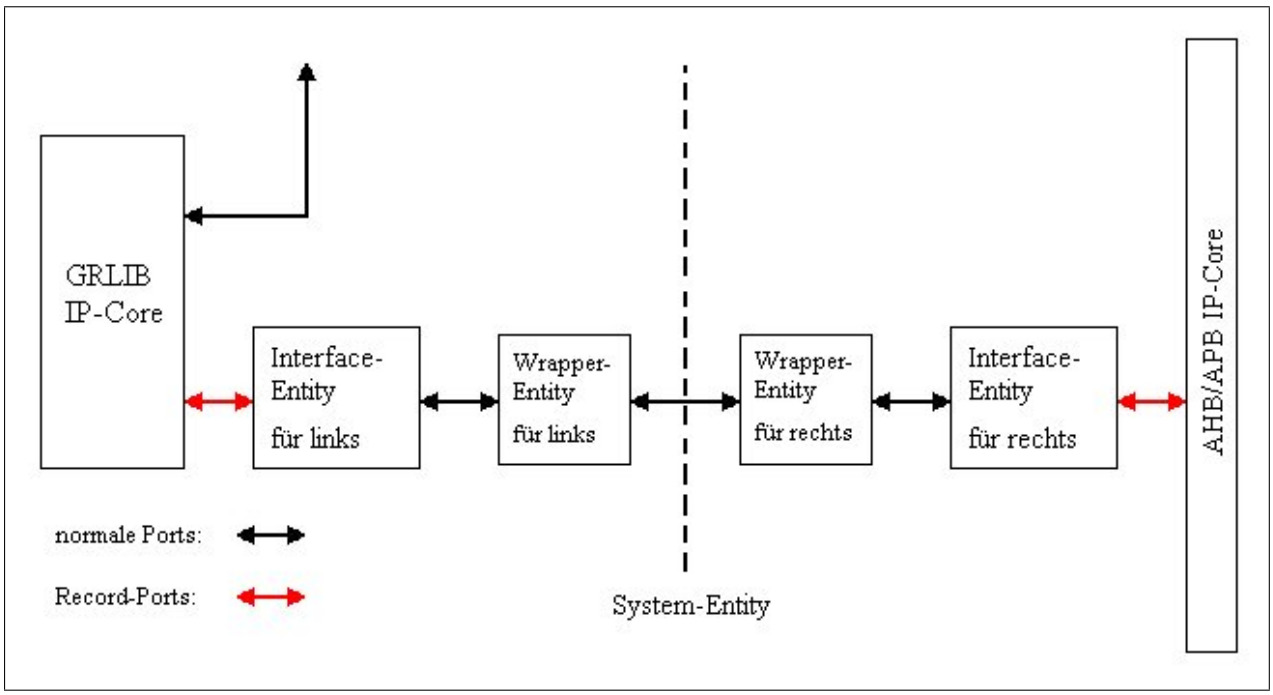

Abbildung 5.11: Schema Interface- und Wrapper-Entitys von IP-Cores

Abbildung 5.11 zeigt ein erweitertes Schema von Abbildung 5.9. Es wurde berücksichtigt, dass der AHB oder APB selbst IP-Cores sind und nach dem Arbeitsschritt 1 des EDK Wräpper-Entitys erstellt wurden. Die zusätzlichen Wrapper-Entitys beinhalten nun wiederum nur die festgelegten Generic-Konstanten. Diese sind ebenfalls reine strukturelle Entitys.

5.3.2 „Bus-Problem“

Ein Bussystem besteht für das EDK aus einem Bus und einem Bus-Arbiter. Der Bus-IP-Core bindet die Bus-Arbiter-Entity ein. Ein Bus-IP-Core, der Xilinx-Bibliothek, wird in seiner MPD mit den Zeilen:

```
OPTION IPTYPE = BUS_ARBITER
OPTION BUS_STD = OPB
```

ausgestattet.

Diese Zeilen veranlassen das EDK den Bus-IP-Core mit einem senkrechten Strich in der „Bus Interface“-Ansicht darzustellen.

Eine **Grundregel** für das EDK lautet: An einen Bus muss mindestens einen Master und ein Slave angeschlossen sein.

Daraus ergibt sich folgendes **Problem**.
Die Grundregel darf mit den eingebundenen GRLIB-IP-Cores nicht verletzt werden. Während des Einbindungsprozesses sind Busse entstanden, welche der Grundregel nicht folgen bräuchten

aber müssen. Aus diesem Grund ergeben sich Besonderheiten für die Busse APB, „irqmp_if“ und „dsu3_if“.

5.3.2.1 „Bus-Problem (APB)“

Wie bereits bekannt, ist der APB ein Slave auf dem AHB. Die Bridge zwischen AHB und APB ist also mit dem APB verschmolzen. Folgerichtig befindet sich das Master-Interface der AHB2APB-Bridge im Inneren des APB dem „apbctrl“-IP-Core. Dieses Master-Interface kann aus dem „apbctrl“-IP-Core nicht herausgetrennt werden ohne den VHDL-Code des APB zu verändern. Es kann also kein „echter“ Master, auf dem APB, in der „Bus Interface“-Ansicht erkennbar sein. Dies verstößt jedoch gegen die oben genannte Grundregel und führt zu einer Fehlermeldung des EDK, beim Start der Synthese.

Lösung des „Bus-Problems“ (APB)

Das Problem konnte gelöst werden, indem ein fiktiver APB-Master geschaffen wurde der folgende Zeile in seiner MPD besitzt:

```
BUS_INTERFACE BUS = MAPB, BUS_STD = APB, BUS_TYPE = MASTER
```

Durch diese Deklaration ist ein IP-Core, der von dem eigentlichen APB-IP-Core verschieden sein muss, an den APB als Master anschließbar. Dies unterdrückt die Fehlermeldung.

Als Inhaber dieser „BUS_INTERFACE“-Deklaration bot sich der IP-Cores des AHB („ahbctrl_if“) an, da der Einsatz eines APB ohne AHB nicht möglich ist.

5.3.2.2 „Bus-Problem“ („irqmp_if“ und „dsu3_if“)

Die Busse „irqmp_if“ und „dsu3_if“ sind keine Busse im herkömmlichen Sinn. Die GRLIB-IP-Cores „irqmp“ und „dsu3“ sind der Multi Prozessor Interrupt-Controller („irqmp“) und die Debug Support Unit („dsu3“).

Der Interrupt-Controller ist ein Slave auf dem APB und überwacht die kombinierten Interruptsignale. Tritt ein Interrupt auf so teilt der Interrupt-Controller dem LEON3 mit das ein Interrupt geworfen wurde. Für diese Mitteilung werden extra Verdrahtungen zwischen LEON3 und Interrupt-Controller benutzt.
Da mit einem AHB bis zu 16 LEON3-Prozessoren verbunden werden können, müssen vom Interrupt-Controller dementsprechend viele Prozessoren-Ports bereit gestellt werden. Dieser Sachverhalt ähnelt einem Bus. Damit alle Prozessoren komfortabel an den Interrupt-Controller angeschlossen werden können, wurde dieser im EDK als Bus eingerichtet.

Ähnlich verhält es sich mit der DSU. Sie ist ein Slave auf dem AHB und kann so Debugging Vorgänge von außen erkennen. Die DSU stellt nun extra Verdrahtungen bereit um jeden einzelnen LEON3-Prozessor in den Debug-Modus zu versetzen oder andere Aufgaben zu übernehmen. Aus diesem Grund wurde die DSU ebenfalls als Bus definiert.

Diese Bus-Definitionen erfüllen vollständig ihren Zweck und ermöglichen eine einfache Handhabung eines LEON3-MPS (lang **M**ulti **P**rozessor **S**ystem). Für den Interrupt-Controller-Bus wurde der Standard „GR_IRQLEON3“ und für den DSU-Bus der Standard „GR_DSULEON3“ gewählt.

Durch die oben genannte Grundregel entsteht allerdings wieder das „Bus-**Problem**“. Es muss mindestens ein Master und ein Slave auf den Bussen GR_IRQLEON3 und GR_DSULEON3 existieren, sonst würde das EDK einem Fehler melden.

Lösung des „Bus-Problems“ („irqmp_if“ und „dsu3_if“)

Unter der Berücksichtigung, dass ein Bus nicht auf sich selbst Master oder Slave sein kann, wurde folgende Lösung getroffen.

Alle LEON3-Prozessoren können Master auf den Bussen GR_IRQLEON3 und GR_DSULEON3 sein. Wenn der Interrupt-Controller und die DSU in einem SoC vorhanden sind, so muss auch mindestens ein LEON3 mit ihnen verbunden sein. Wäre dies nicht der Fall, so könnte der Interrupt-Controller und die DSU aus dem System entfernt werden.
Mit der Masterverbindung jedes einzelnen LEON3-Prozessors sind bereits alle notwendigen Verdrahtungen vorgenommen, es fehlt allerdings noch ein Slave auf jedem Bus.

Jeder LEON3-Prozessor könnte nun auch noch einen fiktiven Slave auf den Bussen GR_IRQLEON3 und GR_DSULEON3 bereitstellen. Dies wäre allerdings bei mehreren LEON3-Prozessoren zu unübersichtlich. Alternativ wurde der AHB wieder dazu benutzt um zwei fiktive Slaves aufzunehmen, jeweils einen für den GR_IRQLEON3 und einen für den GR_DSULEON3.

Als Lösungsergebnis des „Bus“-Problems besitzt der „ahbctrl_if“ IP-Core nun folgende zusätzliche Zeilen in seiner MPD:

```
BUS_INTERFACE BUS = MAPB, BUS_STD = APB, BUS_TYPE = MASTER
BUS_INTERFACE BUS = SDSULEON3, BUS_STD = GR_DSULEON3, BUS_TYPE = SLAVE
BUS_INTERFACE BUS = SIRQLEON3, BUS_STD = GR_IRQLEON3, BUS_TYPE = SLAVE
```

Es wurden somit drei fiktive Bus-Interfaces geschaffen, sodass der EDK-Prüfprozess der Busse APB, GR_IRQLEON3 und GR_DSULEON3 keine Fehler meldet.

5.3.3 „BMM-Problem“

Damit das EDK das native C-Programm in BRAMs laden kann, muss sich der IP-Core „bram_block“, mit den zugehörigen LMB-BAM Controllern und dem LMB, in dem SoC befinden. Um den benutzerfreundlichen Vorteil des nativen C-Programms ebenfalls für das LEON3-System nutzbar zu machen, wurde eine AHB2LMB-Bridge konstruiert. Diese Bridge ermöglicht es den Inhalt des BRAM an einer beliebigen Adresse des LEON3-Systemspeichers bereit zu stellen.

Für das Laden des nativen C-Programms in die BRAMs, wird eine BMM benötigt die während des 1. Arbeitsschrittes, des EDK-Arbeitsplans, automatisch generiert wird. Die konkreten Bedingungen, für die Generierung der BMM sind folgende:

- Es muss ein Prozessor als Master an den LMB angeschlossen sein.
- Der Prozessor muss als MicroBlaze oder PowerPC gekennzeichnet sein.

Aus diesen Bedingungen entsteht folgendes **Problem**.
Würde man den Bedingungen ohne zusätzliche „Tricks“ Folge leisten, so wäre die praktische Umsetzung eine AHB2LMB Bridge-unmöglich. Da nur die AHB2LMB-Bridge ein Master auf dem LMB sein darf, aber kein Prozessor oder MicroBlaze-Prozessor ist. Damit verstößt die AHB2LMB Bridge gegen die zwei Bedingungen.

5.3.3.1 Lösung des „BMM-Problems“

Um die obigen Bedingungen zu überprüfen, bedient sich das EDK der MPD-Inhalte des IP-Cores, welcher als Master an den LMB angeschlossen ist. Ein solcher IP-Core ist in dieser Arbeit die AHB2LMB-Bridge.
Folgende Zeilen befinden sich in der MPD der AHB2LMB-Bridge, welche das Problem lösen:

```
OPTION IPTYPE = PROCESSOR
OPTION SPECIAL = MICROBLAZE
```

Könnte die AHB2LMB-Bridge ordnungsgemäß deklariert werden so wäre sie vom Type eine Bridge, d.h. `(OPTION IPTYPE = BRIDGE)`. Dies musste zu einem Prozessor geändert werden, da ein Prozessor an den LMB angeschlossen sein muss, damit eine BMM generiert wird.
Der Zusatz „`SPECIAL = MICROBLAZE`“ überzeugt das EDK davon, dass es sich bei dem LMB-Master um einen MicroBlaze-Prozessor handelt.
Mit diesen Änderungen wird die BMM ordnungsgemäß generiert und das native C-Programm in den BRAM geladen.

5.3.4 „MHS-Reihenfolge-Problem“

Dieses Problem bezeichnet folgenden Sachverhalt. Die MHS eines SoC beinhaltet alle IP-Cores, welche sich im SoC befinden. Aus dieser MHS wird die „system.vhd“ generiert, die als Haupt-Entity für die Synthese verwendet wird.
Alle IP-Cores werden nacheinander an den Bus angeschlossen und die dazu benötigten Signale werden in der „system.vhd“ generiert. Dabei gilt die Reihenfolge, des Auftretens in der MHS. Die Reihenfolge der IP-Cores in der MHS wird hauptsächlich durch die Reihenfolge des Hinzufügens, der IP-Cores, in das SoC bestimmt. Diese Reihenfolge kann aber auch durch eine manuelle Bearbeitung der MHS verändert werden.

Beispielsweise wird ein Bus-Signalvektor für die Ports „ahbo_hrdata“ aller AHB-Slave-IP-Cores erstellt. Der Bus-Signalvektor hat dabei eine Länge von:

`AHB_DWIDTH` · `C_AHB_NUM_SLAVES` = AHB-Datenbreite · AHB-Anzahl-Slaves
$= 32 \cdot 4 = 128$

In diesem Fall wären 4 Slaves an den AHB angeschlossen und es würde ein Bus-Signalvektor „ahbctrl_if_0_slvo_hrdata“, mit einer Indizierung von „(127 downto 0)“ generiert werden. Alle AHB-Slaves erhalten dann einen bestimmten Bereich des Bus-Signalvektors, auf dem sie ihre zu sendenden Daten treiben können.

```
„Salve 0“ →  ahbso_hrdata => ahbctrl_if_0_slvo_hrdata(31 downto 0)
„Salve 1“ →  ahbso_hrdata => ahbctrl_if_0_slvo_hrdata(63 downto 32)
„Salve 2“ →  ahbso_hrdata => ahbctrl_if_0_slvo_hrdata(95 downto 64)
„Salve 3“ →  ahbso_hrdata => ahbctrl_if_0_slvo_hrdata(127 downto 96)
```

Abbildung 5.12: Bus-Signalvektor Besetzung von Slave 0 - 4

Abbildung 5.12 zeigt eine Zuordnung von 4 AHB-Slaves, die ihre „ahbso_hrdata(31 downto 0)“ Ports in einen großen Bus-Signalvektor („ahbctrl_if_0_slvo_hrdata(127 downto 0)“) treiben. Der Bus-Signalvektor wird dann in den AHB-IP-Core weitergeleitet.
Würde eine andere Reihenfolge der IP-Cores innerhalb der MHS gewählt werden, so wäre das Resultat eine entsprechend Reihenfolge der Bus-Signalvektor-Besetzung.

Regel

Wie bereits bekannt, besitzt jeder Bus-Master oder Slave eines GRLIB-SoC eine „hindex“ oder „pindex“-Generic-Konstante. Diese Generic-Konstante muss mit dem Bus-Signalvektor-Besetzungsindex übereinstimmen, sonst würde bei einer Adressierung durch einen AHB-Master ein falsches Select-Signal erzeugt werden, so dass der anzusprechende Slave nicht ordnungsgemäß selektiert wird.

In den GRLIB-Template-Designs wurde diese Regel immer beachtet. Die Generic-Konstante „hindex“ oder „pindex“ ist immer gleich dem Bus-Signalvektor-Besetzungsindex.

Der Selektierungsprozess wird im Kapitelpunkt 3.1.5.2 für den AHB genau beschrieben. Das „MHS-Reihenfolge“-Problem existiert sowohl für den AHB als auch für den APB.
Zur Lösung des Problems ist der Bus-Signalvektor-Besetzungsindex „i“, in Abbildung 3.24 und Abbildung A.8 genauer zu betrachten.

Problementstehung

Soll das korrekte Select-Signal, durch den Bus-Arbiter, getrieben werden so durchsteift der Bus-Arbiter den „Plug and Play“ Konfigurationsbereich. Wird eine zutreffende Adresse, an einem bestimmten Bus-Signalvektor-Besetzungsindex „i“ gefunden, so wird der selbe Index des Select-Signals mit ’1’ getrieben.
Stimmt die MHS-Reihenfolge allerdings nicht mit den Generic-Konstanten „hindex“ und „pindex“ überein, so lauscht der Bus-Slave auf dem falschen Select-Signalindex und kann nicht auf das, für sich bestimmte, Select-Signal reagieren.

Das „MHS-Reihenfolge“-Problem tritt bei einem OPB nicht auf, da es keine einzelnen Select-Signale gibt. Es wird nur die Adresse und ein allgemeines Select-Signal an alle Slaves angelegt. Die eigentliche Selektierung findet auf Seiten der Slaves statt.

5.3.4.1 Lösung des „MHS-Reihenfolge-Problems“

Das Ziel einer Lösung sollte sein, dass alle GRLIB-IP-Cores während der Bearbeitung des SoC in einer beliebigen Reihenfolge in der MHS auftauchen können und trotzdem ein korrektes Select-Signal, von den Bus-Arbitern des AHB und APB, getrieben wird.

Eine brauchbare Lösung des „MHS-Reihenfolge“-Problems wurde in dieser Arbeit nicht vorgenommen, da die „beste“ Lösung , im aktuellen EDK, mit tiefgreifenden Veränderungen des GRLIB-VHDL-Codes verbunden ist.

Lösungsmöglichkeit 1

Mit einem Tcl-Skript ist es möglich die Reihenfolge der MHS Elemente zu erkennen und ihre genauen Deklarationen auszulesen. Mit einem Tcl-Skript könnte eine Umsortierung der SoC-Komponenten nach den Generic-Konstanten „hindex“ oder „pindex“ stattfinden.

Dazu könnten folgende Tcl-Prozeduren verwendet werden:

Zum Hinzufügen von IP-Cores und deren Elemente in die MHS:

xadd_hw_ipinst, xadd_hw_ipinst_port, xadd_hw_ipinst_busif, xadd_hw_ipinst_parameter

Zum Entfernen von IP-Cores und deren Elemente aus der MHS:

xdel_hw_ipinst, xdel_hw_ipinst_port, xdel_hw_ipinst_busif, xdel_hw_ipinst_parameter

Diese und ähnliche Prozeduren existieren im EDK 7.1 noch nicht. Im EDK 8.2i wurden diese Prozeduren dokumentiert können aber nicht benutzt werden, da sie nicht implementiert wurden. Aus diesen Gründen konnte diese Lösungsmöglichkeit nicht umgesetzt werden.

Lösungsmöglichkeit 2

Das Kernproblem, des „MHS-Reihenfolge-Problems“ besteht darin, dass die Ausgangsports aller Bus-Masters und Slaves in einer falschen Reihenfolge in den Bus-IP-Core münden.
Dieser Sachverhalt ist in Abbildung 3.12 gut zu erkennen. Würde eine falsche Reihenfolge der AHB-Slaves in der MHS vorliegen, so würde in Abbildung 3.12 der „SLAVE 1“ den Record-Port „ahbso(2)“ treiben und der „SLAVE 2“ den Record-Port „ahbso(1)“.
Für die Eingangsports gilt dies nicht, da es nur eine Eingangsportsmenge gibt die in alle Bus-Masters und Slaves getrieben wird.

Die Lösungsmöglichkeit 2 ist eine Umsortierung der Ausgangsports. Aus der Sicht der Busse wären dies die Eingangsports. Diese Lösungsmöglichkeit 2 wurde in dieser Arbeit, innerhalb der Interface-Entitys des AHB und des APB, mit zusätzlicher Logik umgesetzt. Diese Logik kann mit den Generic-Konstanten „CFG_AHBSORT“, für den AHB, und „CFG_APBSORT“, für den APB, aktiviert werden. Standardmäßig ist die zusätzliche Logik deaktiviert, d.h. „CFG_A*BSORT“ = 0.

In der Abbildung 5.13 wird ein Ausschnitt der APB-Interface-Entity gezeigt. Dieser Ausschnitt ist der Kern der Sortierungslogik:

```
procedure fillindex(signal index: out p;signal apbo: in apb_slv_out_vector) is
variable r:integer;
begin
  for i in 0 to NAPBSLV-1 loop
    r:=i;
    for x in 0 to NAPBSLV-1 loop
      if (apbo(x).pindex=i)and(apbo(x).pconfig(1)(0)='1') then r:=x; end if;
    end loop;
    index(i)<=r;
  end loop;
end;

apbo0:for i in 0 to C_APB_NUM_SLAVES-1 generate
      apbo(i)<=apbo_tmp(pindex(i));
end generate;
```

Abbildung 5.13: Sortierungslogik der APB-Interface-Entity

An der umrahmten Zeile ist zu erkennen, dass nicht der Index „i“ des „apbo_tmp“-Signals in das Signal „apbo(i)“, sondern der Index „pindex(i)“. Das Signal „pindex(i)“ wird zuvor mit der Prozedur „fillindex“ ausgefüllt und beinhaltet die korrekte Reihenfolge der Bus-Indizes.
Leider verbraucht diese zusätzliche Logik für die beiden Busse ca. 1400 Slices und ist somit keine akzeptable Lösungsmöglichkeit. Bei Deaktivierung der Sortierungslogik werden keine zusätzlichen Slices verbraucht.

Lösungsmöglichkeit 3

Diese Lösungsmöglichkeit führte zum gewünschten Ziel. Sie hat allerdings den Nachteil, dass der VHDL-Code des AHB und APB verändert werden musste. Diese Änderungen wurden in dieser Arbeit nicht ausgeführt, da sie zur Verifikation ausführliche Tests der Busse benötigen würde.

Folgende Änderungen müssten an den Bussen (AHB, APB) durchgeführt werden:

- Änderungen an der Generierung der „hsel“, „psel“-Signale
- Liest der Bus aus einem Master oder Slave, so könnte am falschen Bus-Signalvektor Besetzungsindex gelesen werden.

Eine Korrektur zur Generierung des „psel“-Signalverktors könnte so erfolgen:

```
if (((apbo(i).pconfig(1)(31 downto 20) and apbo(i).pconfig(1)(15 downto 4)) =
(HADDR(19 downto 8) and apbo(i).pconfig(1)(15 downto 4)))) then
     psel(apbo(i).pindex) := '1';
end if;
```

Abbildung 5.14: Selektierung im APB-System mit Korrektur des „MHS-Reihenfolge“-Problems

Die Abbildung 5.14 im Vergleich zur Abbildung A.8 zeigt die Änderung, mit welcher der APB das korrekte „psel“-Signal treibt und alle Slaves korrekt selektiert werden würden. Es wurde der Index „i“ durch „apbo(i).pindex“ (in Abbildung 5.14 rot markiert) ersetzt.
Das „pindex“-Signal des APB-Slave-Output Records ist nicht in der AMBA spezifiziert und wurde von GR nur aus Simulationszwecken eingeführt. Hier kann es zur Lösung des Problems beitragen.

Diese konkrete Änderung wurde für den APB umgesetzt und seinem „hdl“-Verzeichnis unter dem Namen „apbctrl_beri.vhd“ abgespeichert.

Lösungsmöglichkeit 4

Bei dieser Lösungsmöglichkeit handelt es sich um eine manuelle Lösung. Diese manuelle Lösung wurde in dieser Arbeit ständig durchgeführt, um funktionstüchtige GRLIB-SoCs zu synthetisieren.

Da das „MHS-Reihenfolge-Problem“ nicht gelöst werden konnte ist folgendes zu beachten:

Zu Beachten 5.4 (MHS-Reihenfolge der IP-Cores)

Wird ein GRLIB-SoC mit dem EDK erstellt, dass beliebig viele AHB-Masters, AHB-Slaves oder APB-Slaves enthält, so muss folgendes eingehalten werden:

***Die Reihenfolge** in der AHB-Masters in der MHS auftreten, muss mit der aufsteigenden Ordnung der „hindex“-Generic-Konstante eines jeden AHB-Masters übereinstimmen.*
Der 1. AHB-Master in der MHS muss „hindex = 0“ haben.
Der 2. AHB-Master in der MHS muss „hindex = 1“ haben. usw.

***Die Reihenfolge** in der AHB-Slaves in der MHS auftreten, muss mit der aufsteigenden Ordnung der „hindex“-Generic-Konstante eines jeden AHB-Slaves übereinstimmen.*

***Die Reihenfolge** in der APB-Slaves in der MHS auftreten, muss mit der aufsteigenden Ordnung der „pindex“-Generic-Konstante eines jeden APB-Slaves übereinstimmen.*

Wird dies nicht eingehalten, so funktioniert das, mit dem EDK erstellte, GRLIB-SoC nicht.

Die Umsetzung des Zu Beachten 5.4 kann durch das Editieren der MHS erfolgen.

5.3.4.2 Zusammenfassung des „MHS-Reihenfolge-Problems“

Das „MHS-Reihenfolge-Problem“ wurde in dieser Arbeit nicht, durch einen automatischen Vorgang, gelöst. Es wird vorgeschlagen die Lösungmöglichkeit 3 dieses Problems weiter umzusetzen. Die damit verbundenen Änderungen des VHDL-Codes des AHB und APB müssen allerdings ausgiebig simmuliert und nach der Synthese getestet werden. Da der AHB und der APB grundlegende Elemente eines GRLIB-SoC sind, müssen diese absolut fehlerfrei funktionieren.

5.3.5 „GRMON-APB-Problem“

Nach der Einbindung aller GRLIB-IP-Cores in das EDK ist es gelungen mit dem Programm GRMON den Debugging-Prozess zu starten. Nach der erfolgreichen Verbindung des GRMON mit dem LEON3-System, wird eine kurze Zusammenfassung des SoC angezeigt.

In dieser Zusammenfassung war folgender Fehler zu erkennen:

```
ethernet startup.
GRLIB build version: 2028

initialising ...........
detected frequency: 64 MHz

  Component                          Vendor
  LEON3 SPARC V8 Processor           Gaisler Research
  GR Ethernet MAC                    Gaisler Research
  AHB ROM                            Gaisler Research
  AHB/APB Bridge                     Gaisler Research
  LEON3 Debug Support Unit           Gaisler Research
  DDR266 Controller                  Gaisler Research
  Multi-processor Interrupt Ctrl     Gaisler Research
  General purpose I/O port           Gaisler Research
  General purpose I/O port           Gaisler Research
  Generic APB UART                   Gaisler Research
  Modular Timer Unit                 Gaisler Research
  Multi-processor Interrupt Ctrl     Gaisler Research
  General purpose I/O port           Gaisler Research
  General purpose I/O port           Gaisler Research
  Generic APB UART                   Gaisler Research
  Modular Timer Unit                 Gaisler Research
Use command 'info sys' to print a detailed report of attached cores

grlib>_
```

Abbildung 5.15: Kurzzusammenfassung des GRMON mit Fehler

Die in Abbildung 5.15 rot markierten SoC-Komponenten werden doppelt angezeigt. Es handelt sich dabei nur um Slaves die mit dem APB verbunden sind.

Die Erklärung des **Problems** ist wie folgt:
Der GRMON scannt die Konfigurationsbereiche der beiden Busse (AHB und APB). Alle SoC-Komponenten die darin ihr „pconfig“-Signal abgelegt haben, werden als eigenständige SoC-Komponente erkannt. Trotz dass sich beispielsweise nur ein Multi-Prozessor-Interrupt-Controller an dem APB befindet, werden zwei erkannt und angezeigt. Dies hat schwerwiegende Auswirkungen auf ein Betriebssystem, welches in das GRLIB-SoC geladen wird. Es würde beispielsweise eine APB-Uart-Komponente zweimal erkennen und diese zweimal initialisieren und als Device (ttyS0, ttyS1) bereitstellen. Dies ist nicht akzeptabel.

Das Problem wurde im VHD-Code des APB-IP-Cores, ab Zeile 148, identifiziert:

```
if r.cfgsel = '1' then
     v.prdata := apbo(conv_integer(r.haddr(log2x(nslaves)+2 downto
     3))).pconfig(conv_integer(r.haddr(2 downto 2)));
end if;
```

Abbildung 5.16: Ausschnitt der „apbctrl.vhd“ zum „GRMON-APB“-Problem

Die Variable „nslaves“ (in Abbildung 5.16 rot markiert) steht für die Anzahl der APB-Slaves an dem Bus. Das Register „r.cfgsel“ ist gleich '1', wenn ein Master auf den Konfigurationsbereich zugreift. Dies ist beim Starten des GRMON oder eines Betriebssystems der Fall. Es werden dann keine Registerdaten, der APB-Slaves übergeben, sondern deren „Plug and Play“ Konfigurationsdaten.
Um die richtigen Konfigurationsdaten zu übergeben wird die Scan-Adresse des Masters „r.haddr“ (in Abbildung 5.16 blau markiert) ausgewertet.
Der Fehler war nun, dass die Scan-Adresse unterhalb des „ld(nslaves)+3“-ten Bits zum Vergleich benutzt wurde, um den entsprechenden Konfigurationsbereich auszuwählen.
Da der GRMON oder ein Betriebsystem den vollständigen Konfigurationsbereich scannen müssen, kommt es zu einem Überlauf und der Konfigurationsbereich wird unfreiwillig erneut gescannt. Dies ist allerdings nur bei einer APB-Slave Anzahl von „nslaves ≤ 8“ der Fall, da ab dieser Anzahl ein Bit weniger verglichen wird.

5.3.5.1 Lösung des „GRMON-APB-Problems“

Die Firma GR umgeht in ihren Template-Designs diesen Fehler, indem dem APB immer eine APB-Slave Anzahl von 16 übergeben wird. Dies ist allerdings suboptimal, da eine höhere Anzahl von Slaves mehr Patzverbrauch und niedrigere Taktbarkeit bedeuten.

Die in dieser Arbeit umgesetzte Lösung ist, die Ersetzung der Variable „nsalves“ (in Abbildung 5.16 rot markiert) durch die Konstante „NAPBSLV“. Diese Konstante ist in dem „amba.vhd“-Paket definiert und trägt in der GRLIB Version 1.0.14 den Wert 16.

5.3.6 „FSM-Problem“

Folgender Sachverhalt verbindet sich mit dem „FSM-Problem“. Die Firma GR empfiehlt, für die Synthese ihrer GRLIB, keine **F**inite **S**tate **M**achine (kurz FSM) Extraction zu benutzen. Dieser Empfehlung muss Folge geleistet werden, da sonst ein Funktionieren der IP-Cores nicht garantiert werden kann.
Die FSM-Extraktion kann mit dem Syntheseparameter „fsm_extract“ festgelegt werden.

Das **Problem** entsteht dadurch, dass es für den Workflow des EDK keine herkömmliche Möglichkeit gibt, den Syntheseparameter „fsm_extract“ festzulegen, wie es im ISE der Fall ist.
Für jeden IP-Cores gibt es eine Syntheseparameterdatei namens „[name_ip]_?_wrapper_xst.scr“, welche sich im Verzeichnis „[EDK_PROJECT]/synthesis“ befindet.
In diese Datei könnte der Parameter „-fsm_extract no“ eingetragen werden. Leider wird diese Datei durch das EDK bei jedem Synthesestart neu generiert und überschrieben, so dass der hinzugefügte Parameter gelöscht würde.

5.3.6.1 Lösung des „FSM-Problems“

Damit eine gezielte Deaktivierung der FSM-Extraktion stattfinden konnte, musste der VDHL-Code jeder einzelnen Entity, die FSMs beinhaltet, verändert werden.

Folgende Zeilen wurden den Entitysdeklarationen hinzugefügt:

```
entity greth is

    generic(...)
    port(...);
    attribute fsm_extract : string;
    attribute fsm_extract of greth : entity is no";

end entity;
```

Abbildung 5.17: FSM Extraktion deaktivieren.

In Abbildung 5.17 ist am Beispiel des GR Ethernet IP-Cores die Deaktivierung der FSM-Extraktion rot markiert. War es bei der Einbindung von IP-Cores nötig dieses Attribut dem VHDL-Code hinzuzufügen, so wurden die betroffenen VHD-Dateien dem „hdl“-Verzeichnis des EDK-IP-Cores hinzugefügt. Entsprechend wurde die PAO des IP-Cores angepasst.

Zu Beachten 5.5 (FSM-Extraktion deaktivieren)

Ein IP-Core des EDK wurde nur dann mit dem Parameter „fsm_extract no“ synthetisiert, wenn explizit keine FSM-Extraktionsmeldungen im Synthesebericht vorhanden sind.
Der Synthesebericht wird in der Datei „[name_ip]_?_wrapper_xst.srp“ gespeichert.

5.3.7 „Constraints-Problem“

Für die IP-Cores „GR Ethernet“ und „GR DDR SDRAM“ ist es nötig spezielle Constraints der Synthese zu übergeben, da diese IP-Cores sehr timingempfindlich sind. Mit dem Tcl-Skripts der

einzelnen IP-Cores wurden die Constraints vorerst in eine extra UCF geschrieben und dann entsprechend der gewählten Lösung weiter verarbeitet.

Lösungsmöglichkeit 1

Die extra UCF-Dateien können nach dem Muster „[name_ip]_?_wrapper.ucf“ im Verzeichnis „[EDK_PROJECT]/implementation“ gespeichert werden.

Würde die Zeile:

„`OPTION RUN_NGCBUILD = TRUE`“

der MPD des betreffenden IP-Core hinzugefügt, so würde der IP-Core unter spezieller Berücksichtigung der extra UCF synthetisiert werden.

Diese Lösung wurde zwar umgesetzt, allerdings wird die Wirkung der Lösungsmöglichkeit 1 durch die Lösung des „GR Ethernet-Problems“ unwirksam gemacht.

5.3.7.1 Lösung des „Constraints-Problem“

Die zweite Möglichkeit ist es, die extra UCF-Einträge in die „system.ucf“ zu übertragen. In der „system.ucf“ werden die extra UCF-Einträge, im Arbeitsschritt 2 des EDK-Arbeitsplans, in das System übernommen. Diese Lösungsmöglichkeit wurde in dieser Arbeit umgesetzt.

Die extra UCF-Einträge werden automatisch, durch die „systemLEON.make“, in die „system.ucf“ eingefügt. Die „systemLEON.make“ ist ein Analogon zur herkömmlichen „system.make“. Der Kern dieser Lösung stellt die folgende Zeile 213 der „systemLEON.make“ dar:

„`@cat implementation/*.ucf > implementation/$(SYSTEM).ucf`“

Mit dem obigen Kommando werden die Dateien „system_origi.ucf“ (für die originale „system.ucf“), „greth_if_0_wrapper.ucf“ und „ddrspa_if_0_wrapper.ucf“ zu einer kompletten „system.ucf“ zusammengefügt.

Die Firma GR löst dieses Problem mit einer kompletten „system.ucf“ die für den „normalen“ Benutzer kaum nachvollziehbar ist. Eine Anpassung eines Template-Designs auf ein nicht unterstütztes Board, kann dadurch schnell zu einem Fehlschlag führen.

5.3.8 „GR Ethernet-Problem“

Der GR Ethernet IP-Core der GRLIB („greth_if“) ist extrem timingempfindlich. Trotz der berücksichtigten Constraints, die dank der Lösung des „Constraints-Problems“, auf den IP-Cores angewandt wurden, kam es bei Tests der GR Ethernet Schnittstelle immer wieder zu fehlerhaft übertragenen Paketen.

Das Problem entstand durch den gekapselten Syntheseprozess des EDK. Das EDK synthetisiert zuerst alle IP-Cores der SoC-Komponenten und danach die System-Entity („system.vhd“). Dies führte zu einem minimalen Timingunterschied, da die IP-Cores der SoC-Komponenten als „Black Box“ behandelt werden und nicht weiter mit der System-Entity verschmolzen werden.

5.3.8.1 Lösung des „GR Ethernet-Problems“

Das Verschmelzen der SoC-Komponenten-IP-Cores mit der „system.vhd“ kann durch den Syntheseparameter „`read_cores optimize`“ erzwungen werden. Allerdings werden dann die extra UCF-Constraints, die durch die Lösungsmöglichkeit 1 des „Constraints-Problem“ übernommen wurden, ungültig. Aus diesem Grund musste die zweite Lösungsmöglichkeit des „Constraints-Problem“, in dieser Arbeit, umgesetzt werden.

Zu Beachten 5.6 (Verwendung des GR Ethernet IP-Core)

Um den „greth_if“ IP-Core zu verwenden, muss der Parameter „read_cores optimize“ in die Syntheseparameterdatei der System-Entity eingetragen werden.
Die Syntheseparameterdatei befindet sich unter dem Namen „system_xstLEON.scr“ im Verzeichnis „[EDK_PROJECT]/synthesis“.

Zu Beachten 5.7 (gleichzeitige Verwendung des GR Ethernet und ChipScope IP-Cores)

Durch die Verwendung des Syntheseparameters „`read_cores optimize`“ werden die einzelnen IP-Cores nicht mehr als „Black Box“ behandelt. Dies hat für bestimmte IP-Cores eine schwerwiegende Folge.
Wird beispielsweise der IP-Core „chipscope_opb_iba“ mit diesem Parameter synthetisiert, so wird der IP-Core vollständig wegoptimiert (engl.: trimmed) und kann dadurch nicht mehr benutzt werden.
Die gleichzeitige Funktionstüchtigkeit von GR Ethernet IP-Core und ChipScope IP-Core konnte in dieser Arbeit nicht erreicht werden.

5.3.9 „system.make-Problem“

Um den EDK-Arbeitsplan für die eingebundenen IP-Cores der GRLIB optimal anzupassen, musste eine spezielle „system.make“ erzeugt werden. Die neue MAKEFILE wurde „systemLEON.make“ genannt und wurde im „[EDK_PROJECT]“-Verzeichnis gespeichert.

Zu Beachten 5.8 (MAKEFILE für GRLIB-SoC)

Damit der EDK-Arbeitsplan aus der „systemLEON.make“ gelesen wird, muss sie als „Custom Makefile“ im Menüpunkt:
„Project“ $\Longrightarrow$ „Project Options...“
eingetragen werden.

Die „systemLEON.make“ verwendet das Shell-Skript die Datei „synthesisLEON.sh“, welche im „[EDK_PROJECT]/synthesis“-Verzeichnis abgelegt ist. Dieses Shell-Skript bindet wiederum die Syntheseparameterdatei „system_xstLEON.scr“ ein, welche für das Funktionieren des GR Ethernet IP-Cores entscheidend ist (siehe „GR Ethernet-Problem“).

5.3.9.1 Anpassungen der „systemLEON.make“

Folgende Anpassungen und Veränderungen wurden in der „systemLEON.make“ vorgenommen:

- Anpassung des C-Programm Kompilierungsprozesses auf den LEON3, ab Zeile 143
- Verschmelzungen der extra UCF-Dateien, Zeile 200 und ab Zeile 211
- Einbindung des „`read_cores`“ Syntheseparameters, Zeile 205

Zu Beachten 5.9 (Projektname des nativen C-Programms)

Soll ein natives C-Programm für den LEON3 kompiliert werden, so muss es in der „Applications“-Ansicht des EDK mit dem Projektnamen „TestApp“ versehen werden (siehe Abbildung 6.1).
Eine Unabhängigkeit von diesem Namen konnte in dieser Arbeit nicht hergestellt werden, da die „systemLEON.make“ nicht überschrieben werden darf.

5.4 Verbindung von GRLIB-System und MicroBlaze-System

Dieser Kapitelpunkt stellt IP-Cores vor, die es ermöglichen das lange benutzte MicroBlaze-System mit dem neuen LEON3-System zu verbinden. Dadurch werden die Vorzüge des alten MicroBlaze-Systems für das neue LEON3-System komfortabel einsetzbar.

Zu diesen Vorzügen gehören:

Vorzug 1: *Die Bereitstellung der kompilierten, nativen C-Programme in einem BRAM.*

Vorzug 2: *Die Benutzung von bereits existierenden IP-Cores oder Entitys für den OPB.*

Vorzug 1

Vorzug 1 ist eine Möglichkeit mit der ein Benutzer direkt bei Systemstart ein natives C-Programm auf einem Prozessor ausführen kann. Dazu werden die IP-Cores „bram_block“, „lmb_bram_if_cntlr“ und „lmb_v10“ benötigt.
Ein GRLIB-IP-Core mit vergleichbarer Möglichkeit ist der „ahbrom_if“ IP-Core. Dieser IP-Core beinhaltet die nativen ELF-Programmdaten und wird ebenfalls automatisch als BRAM synthetisiert. Die Benutzung des Xilinx-BRAM-IP-Cores mit dem LMB hat jedoch einen entscheidenden Vorteil.
Verändert man das native C-Programm für ein MicroBlaze-System, so muss dieses neu kompiliert werden. Die entstandene ELF-Datei wird dann mit der „system.bit“ zur „download.bit“ verschmolzen. Es muss also nur der 4. Arbeitsschritt des EDK-Arbeitsplans wiederholt werden.
Wird dagegen das native C-Programm für den LEON3-System verändert, so führt dies zu einer Veränderung des „ahbrom_if“-IP-Cores und damit wird eine vollständig neue Synthese notwendig, um das native C-Programm zu nutzen.

Die Umsetzung des 1. Vorzugs spart also enorme Synthesezeiten ein. Damit der Vorzug 1 für das LEON3-System bereitsteht wurde eine AHB2LMB-Bridge konstruiert.

Vorzug 2

Vorzug 2 entsteht aus der langen Zeit, die das MicroBlaze-System bereits benutzt wird. In der Vergangenheit sind eine Vielzahl von IP-Cores oder Entitys entstanden, welche an den OPB angeschlossen werden müssen. Würde der Benutzer mit dem neuen LEON3-System konfrontiert werden, so könnte er die alten benötigten IP-Cores nicht an den LEON3 anschließen, da das GRLIB-System über keinen OPB verfügt.

Damit der Umstieg auf das LEON3-System so attraktiv wie möglich wird, wurde eine AHB2OPB-Bridge entworfen. Mit dieser Bridge ist es möglich den OPB, des MicroBlaze-Systems, vom MicroBlaze abzukoppeln und diesen über die AHB2OBP-Bridge mit dem AHB zu verbinden.

5.4.1 AHB2LMB-Bridge

Die AHB2LMB-Bridge ist eine Entity ohne FSM. Sie besteht zum größten Teil aus strukturellen Schaltungen. Die AHB2LMB-Bridge ist als Slave an den AHB angeschlossen und als Master an den LMB. Als AHB-Slave treibt die Bridge ebenfalls „Plug and Play“ Konfigurationsdaten. Die Bridge meldet sich am AHB als „ahbrom“-IP-Core an. Diese Wahl wurde getroffen, da die Funktionalität der Bridge dem „ahbrom“ ähnelt.

Bei der Konstruktion des AHB2LMB-Bridge galt der MicroBlaze-Prozessor als Beispiel. Dadurch ist es möglich die AHB2LMB-Bridge als Master an den Instruktion-LMB und als Master an den Daten-LMB anzuschließen. Die Bridge verfügt also über zwei LMB-Master-Interfaces. Die AHB2LMB-Bridge ist in der Abbildung 5.3 als erster IP-Core dargestellt.

Da der LEON3 keine echte Harvard Architektur ist, genügt es nur das Instruktion-LMB-Master-Interface zu treiben. Über dieses werden die Instruktionen aus dem BRAM gelesen aber nicht geschrieben. Sollte sich ein Nutzen für das Schreiben auf den BRAM ergeben, so könnte zusätzliche Logik in der AHB2LMB-Bridge implementiert werden, die es ermöglicht über den Daten-LMB auf dem BRAM zu schreiben. Die benötigten Ports für einen Daten-LMB Lese- oder Schreib-Transfer sind in der Bridge bereits vorhanden.
Der Name des AHB2LMB-Bridge IP-Cores ist „ahb2lmb_bridge“ ohne den Zusatz „_if“. Da es sich bei diesem IP-Core um keinen IP-Core aus der GRLIB handelt, ist keine Interface-Entity notwendig. Die AHB2LMB Bridge existiert in zwei Versionen, die sich „gepipte“ und „ungepipte“ Version nennen. Der Unterschied der Versionen besteht in ihrem Timingverhalten.

„gepipte“ Version

Die gepipte AHB2LMB-Bridge verfügt über einen zusätzlichen Pipeline Zustand für den AHB-Slave Datenport und benötigt 2 Takte, nach Anforderung des AHB-Master, bis die Instruktionen bereitstehen. Nach 4 Takten holt der LEON3 bei der gepipten Bridge die nächste Instruktion. Die gepipte AHB2LMB-Bridge benötigt 20 Slices und ist bis ca. 817 MHz taktbar.

Das Timingverhalten der gepipten AHB2LMB Bridge ist sauberer und zuverlässiger als das der ungepipten Version.

„ungepipte“ Version

Die ungepipte AHB2LMB-Bridge-Version benötigt 1 Takt bis die Instruktionen aus dem BRAM, nach Anforderung des LEON3, am AHB bereitstehen. Der LEON3 holt bereits nach 3 Takten die nächste Instruktion. Die ungepipte Version benötigt 18 Slices. Es ist nicht möglich eine maximale Taktung anzugeben, da die ungepipte AHB2LMB-Bridge keinen getakteten Prozess besitzt.

Mit der Generic-Konstante „**pipe**“ kann bei der AHB2LMB-Bridge zwischen dem „gepipten“ (pipe = 1) und „ungepipten“ (pipe = 0) Design gewählt werden. Standardmäßig wird die Bridge gepipte synthetisiert.

Zu Beachten 5.10 (AHB2LMB-Bridge)

Durch die Einsparung von einem Takt, der ungepipten gegenüber der gepipten AHB2LMB-Bridge, entsteht ein praktischer Performance-Gewinn von ca. 20%, beim einem aus dem BRAM gelesenen Programm.
Um den Performance-Gewinn zu ermitteln, musste der Cache des LEON3 deaktiviert werden, sodass das Programm ständig aus dem BRAM über die AHB2LMB-Bridge gelesen wurde. Bei einem durchgängigen Lese-Transfer aus dem BRAM entsteht ein theoretischer Performance-Gewinn von 25%.

5.4.2 AHB2OPB-Bridge

Die AHB2OPB-Bridge besteht aus einer Entity mit einer FSM. Die Bridge implementiert, zum Zeitpunkt dieser Arbeit, keine FIFOs und kann nur Busse in der gleichen Clock-Domäne[4] verbinden. Die Clocks aller Entitys innerhalb einer Clock-Domäne besitzen also keine ungleichen Taktfrequenzen oder Phasen. Die Bridge benötigt 96 Slices und ist bis ca. 310 MHz taktbar.

Die AHB2OPB-Bridge unterstützt momentan nur Single-Beat-Transfers und keine Burst-Transfers. Würde der LEON3-Prozessor einen Burst-Transfer über die AHB2OPB-Bridge einleiten, so würde der LEON3-System stehen bleiben, da der AHB über keine Timeout-Funktion verfügt.
Damit dies nicht passiert, kann die AHB2OPB-Bridge Burst-Transfers verarbeiten, aber würde, zum Zeitpunkt dieser Arbeit, falsche Daten transferieren.

[4] Mit gleicher Clock-Domäne wird ein Gebiet bezeichnet, welches mit dem selben Clocksignal getrieben wird.

Zu Beachten 5.11 (AHB2OPB-Bridge mit GRMON)

Das GRLIB-SoC-Debugging-Tool GRMON löst, beim Debugging über die Ethernet-Schnittstelle, prinzipiell Burst-Transfers auf einem GRLIB-SoC aus. Dadurch kann die theoretisch maximale Übertragungsrate von 100 MBit/s besser ausgenutzt werden.
Da die AHB2OPB-Bridge zum Entstehungszeitpunkt dieser Arbeit Burst-Transfers nicht fehlerfrei unterstützt, muss dies berücksichtigt werden, wenn mit dem GRMON der Speicherbereich der AHB2OPB-Bridge ausgelesen wird.
Die AHB2OPB-Bridge kann Burst-Transfers verarbeiten und bleibt nicht stehen. Allerdings sind die Burst-Transferdaten nicht korrekt.
Um diesen Fehler zu beheben müsste der „Burst-Pfad“ der AHB22OPB-Bridge-FSM, siehe Abbildung 5.18, überarbeitet und getestet werden.

Die Bridge erkennt einen „Timeout“ auf Seiten des OPB und gibt ihn als Fehler an den AHB zurück. Dies musste so implementiert werden, da der AHB keine Timeout-Funktion unterstützt. Ein „Retry“ kann ebenfalls auf dem OPB erkannt werden. Dieser wird in ein „Retry“ auf dem AHB umgesetzt.

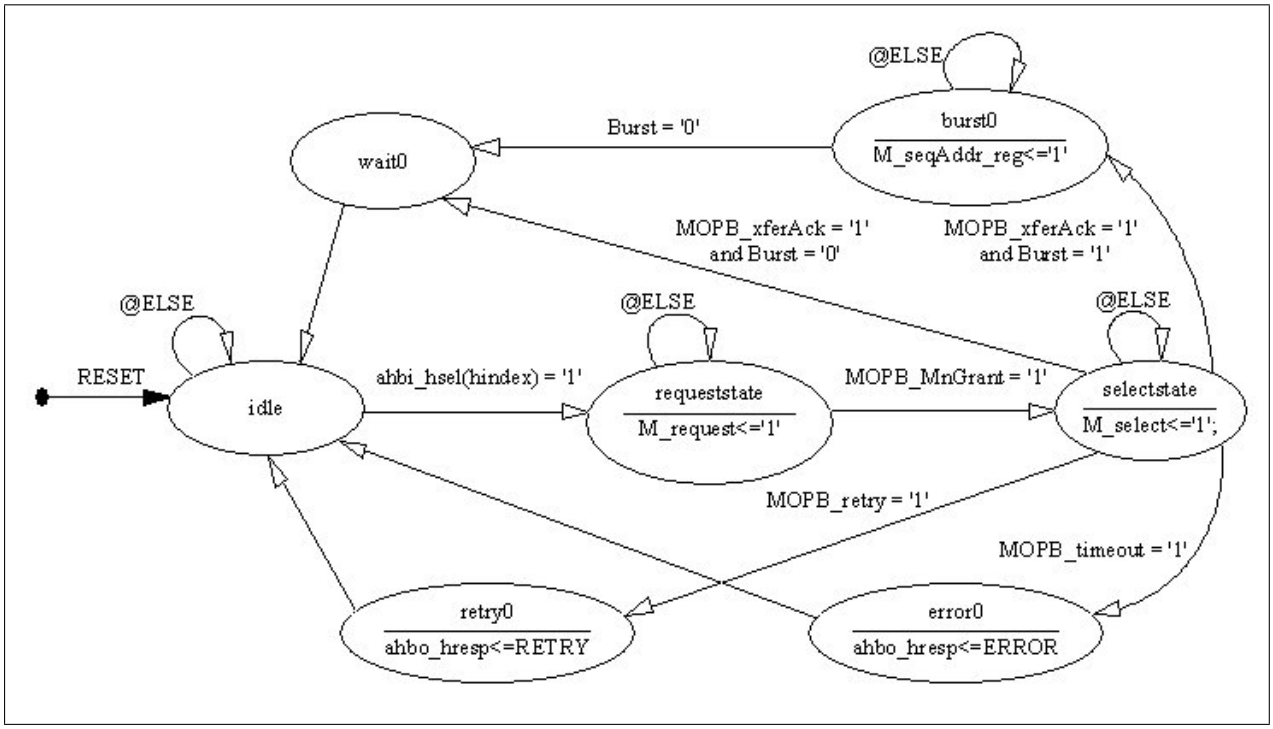

Abbildung 5.18: FSM der AHB2OPB-Bridge

Die Abbildung 5.18 zeigt die FSM der AHB2OPB-Bridge. Wird das SoC in Betrieb genommen befindet sich die Bridge im „idle“ Startzustand. Wird sie als AHB-Slave selektiert, dann wechselt

sie in den „requeststate“ Zustand und initialisiert einen Transfer auf dem OPB.
Erhält die Bridge Zugriff auf den OPB (d.h. MOPB_MnGrant = ’1’) so wechselt sie in den „selectstate“ Zustand und leitet den Transfer ein (Schritt 1 der Vier-Zyklus-Kommunikation, siehe Abbildung 2.2).
Wird ein normaler Transfer auf dem OPB mit MOPB_xferAck = ’1’ erfolgreich beendet (Schritt 3 der Vier-Zyklus-Kommunikation) so wechselt die Bridge in den „wait0“ Zustand und beendet den Transfer auf dem AHB. Danach wechselt die Bridge in den Startzustand „idle“.

5.5 Ausgewählte IP-Cores der GRLIB im EDK und ihre Besonderheiten

Dieser Kapitelpunkt soll dazu dienen, ausgewählte IP-Cores der GRLIB nach ihrer Einbindung in das EDK vorzustellen und Besonderheiten zu erläutern. Die Absichten dieses Kapitelpunktes haben folgenden Hintergrund.
Viele IP-Cores der GRLIB sind in ihrer Funktionswiese und Bedeutung mit denen der Xilinx-Bibliothek vergleichbar. Allerdings gibt es Unterschiede in ihrer Konfigurierbarkeit und in ihrer Funktionalität. Ebenfalls wurden GRLIB-IP-Cores in EDK eingebunden, welche vollkommen neue, nützliche Funktionalitäten bieten, von denen keine vergleichbaren IP-Cores der Xilinx-Bibliothek existieren. Von solchen interessanten IP-Cores soll dieser Kapitelpunkt berichten.

5.5.1 „clkrstgen_if“-IP-Core

Der „clkrstgen_if“ steht für „**Cl**oc**k**-**R**es**e**t-**Gen**erator“. Dieser IP-Core beinhaltet eine Entity, welche während des Einbindungsprozesses entstanden ist.
Diese Entity fasst alle logischen Schaltungen zusammen, welche für die Clock-Generierung und die Reset-Generierung benötigt werden. Dies sind im wesentlichen die GRLIB-Entitys „clkgen“ und „rstgen“. Die logischen Schaltungen zur Clock- und Reset-Generierung sind in den Template-Designs der GRLIB innerhalb der Haupt-Entity „leon3mp.vhd“ verstreut und werden mit dem „clkrstgen_if“-IP-Core zusammengefasst. In Abbildung 5.3 ist der „clkrstgen_if“-IP-Core als letzter in der ED-GUI zu erkennen.

Mit diesem IP-Core kann ein beliebiger Anteil der Boardfrequenz als AHB-Frequenz erzeugt werden. Die AHB-Frequenz ist die Frequenz der Main-Clock des LEON3-Systems.

Zu Beachten 5.12 (BOARD-FREQ Generic des clkrstgen IP-Core)

*Soll ein LEON3-System ordnungsgemäß getaktet werden, so muss die Eingabe der Boardfrequenz mit der Generic-Konstante „**BOARD_FREQ**“, des „clkrstgen_if“-IP-Core, **manuell** erfolgen. Die verwendete Einheit von Frequenzen ist immer „Hz“.*

Die AHB-Frequenz kann dann mit den Generic-Konstanten „CLK_MUL“, als Zähler und „CLK_DIV“, als Nenner des Clock-Faktors beliebig eingestellt werden. Dadurch ergibt sich eine AHB-Frequenz von:

$$\text{AHB-Frequenz} = \frac{CLK\_MUL}{CLK\_DIV} \cdot \text{BOARD\_FREQ}$$

Die Clock-Generierung wird intern mittels eines DCM bewerkstelligt. Dadurch ist der „clkrstgen_if“-IP-Core mit dem „dcm_module“-IP-Core der Xilinx Bibliothek vergleichbar. Der „clkrstgen_if“-IP-Core besitzt zusätzlich eine „DDR_PORT“-Bus-Interface, über das der IP-Core „ddrspa_if“ angeschlossen werden kann. Dadurch werden komfortabel alle benötigten Clock-Signale, über eine „Initiator-Target“-Verbindnug, an den IP-Core des GRLIB DDR SDRAM weitergetrieben.

5.5.2 „ahbuart_if“ und „apbuart_if“-IP-Core

Der „ahbuart_if“ und „apbuart_if“-IP-Core sind vergleichbar mit dem „opb_uartlite“-IP-Core der Xilinx-Bibliothek. Die beiden IP-Cores sind in erster Linie für eine Low-Speed-Kommunikation über die RS323 Schnittstelle gedacht. Dabei sind sie als Slaves an den APB angeschlossen.
Der „ahbuart_if“-IP-Core besitzt zusätzlich die Möglichkeit als vollwertiges Debuggingmodul zu dienen. Zugriff auf ein GRLIB-System erhält die AHB Debug Uart über ein zusätzliches AHB-Master-Interface.

Die Konfiguration der Übertragungsrate (Baudrate) erfolgt bei beiden IP-Cores über Register. Es ist also nur möglich die Baudrate der GRLIB-Uarts über Software einzustellen. Beide Uarts besitzen diesbezüglich ein „Scaler“-Register, welches mit einem bestimmten Wert beschrieben werden muss. Dieses Scaler-Register gilt als Auffüll-Wert für einen Timer, der beim Unterlaufen eine neue Bit-Übertragung einleitet.

Der Wert des Scaler-Registers wird nach folgender Formel explizit berechnet:

$$\text{Scaler-Register} = \frac{\frac{AHB-Frequenz \cdot 10}{Baudrate \cdot 8} - 5}{10} \qquad \text{[gripdoc]}$$

Wird kein Auffüll-Wert in dem Scaler-Register angegeben, so versuchen die GRLIB-Uarts über eine bestimmte Schaltung die angelegte Baudrate selbstständig zu ermitteln. Dies bedarf allerdings oft mehrerer Kommunikationsversuche.

5.5.3 „ahbjtag_if“-IP-Core

Der „ahbjtag_if“-IP-Core ist der eigentliche IP-Core zum Debuggen von FPGA-Board-Systemen. Dieser IP-Core wird als Master an den AHB angeschlossen und kann dadurch jede Art von Transfer auslösen. Als Kernpunkt bindet der „ahbjtag_if“-IP-Core eine „BSCAN“-Komponente ein. Der Ausdruck „BSCAN“ steht für „**B**oundary **Scan**“. Diese Komponente existiert für verschiedenste FPGAs unter den Namen „BSCAN_VIRTEX2“ oder „BSCAN_SPARTAN3“. Wird eine „BSCAN“-Komponente in einer Entity eingebunden, so erhält diese automatisch Zugriff auf die JTAG-Ports des FPGA-Boards.

Verfügt ein SoC einmal über diese JTAG-Ports, so kann über Parallel III, IV oder USB-Kabel der Debuggingzugriff auf das SoC erfolgen. Es wird der zugehörige Treiber auf den Host-PC benötigt. Leider hat die Firma Xilinx die Treiber für den JTAG Zugriff über USB nicht veröffentlicht, so dass der „ahbjtag_if“-IP-Core nur für den Zugriff über das Parallel III oder IV Kabel benutzt werden kann.
Bei Altera FPGAs kann das Debugging zusätzlich über die USB-Schnittstelle mittels „ahbjtag_if“-IP-Core erfolgen, da für Altera FPGAs der Treiber veröffentlicht wurde.

Aufgrund dieses Nachteils wurden zusätzlich die Debugging-IP-Cores „greth“, „ahbuart“ sowie „usbdcl“ von GR entwickelt.

5.5.4 „greth_if“-IP-Core

Dieser IP-Core dient zur Ansteuerung der Ethernet-MAC Schnittstelle eines FPGA-Boards. Optional kann der IP-Core mit einem **E**thernet **D**ebug **C**ommunication **L**ink (kurz EDCL) synthetisiert werden. Der EDCL kann über die Generic-Konstante „**edcl**“ des IP-Core aktiviert oder deaktiviert werden.

Der „greth_if“-IP-Core ermöglicht Ethernet Übertragungsraten mit den Grenzen 10 MBit/s oder 100 MBit/s. Der Modus wird automatisch durch interne Logik erkannt. Für eine Übertragungsrate von 10 MBit/s wird mindestens eine AHB-Frequenz von 2,5 MHz benötigt, während für den 100 MBit/s Modus mindestens 18 MHz AHB-Frequenz benötigt werden. Wird dies nicht eingehalten so kommt es zu Paketverlusten.
Wird dieser IP-Core als Debuggingmodul verwendet so muss der EDCL aktiviert sein und es können die Images von Betriebsystemen mit mehreren 10Mbit/s in den Speicher des FPGA-Boards

geladen werden. Ist der EDCL aktiviert, so muss dem IP-Core eine MAC-Adresse und eine IP-Adresse zugewiesen werden. Dies geschieht über die Generic-Konstanten „**macaddrh_l**“ und „**ipaddrh_l**“. Diese Adressen können selbstverständlich nachträglich, durch das Beschreiben von Registern mittels Software verändert werden.

Zu Beachten 5.13 (MAC- und IP-Adressen Generics des GR Ethernet IP-Core)

Die Angaben von MAC- und IP-Adresse gilt nur für die EDCL-Komponente des IP-Cores. Dadurch ist es möglich mit dem Austausch von UDP-Paketen das Debugging durchzuführen, ohne ein Betriebssystem auf dem FPGA laufen zu lassen.

5.5.5 „ddrspa_if“-IP-Core

Der „ddrspa_if“-IP-Core dient zur Ansteuerung eines DDR SDRAM Moduls und wird als Slave an den AHB angeschlossen. Dies ermöglicht hohe Transferraten zwischen LEON3 und DDR SDRAM.
Der IP-Core wird mit einer separaten Taktung versorgt, die durch das zusätzliche Einbinden von 2 DCMs bewerkstelligt wird. Die Generic-Konstante „**MHz**“ gibt die Board-Frequenz an und wird mit einem Tcl-Skript automatisch festgelegt. Mit den Generic-Konstanten „**clkmul**“ und „**clkdiv**“ können die DCMs des IP-Cores angepasst werden und können eine beliebige Frequenz zum Betrieb des „ddrspa“-IP-Cores erzeugen.

Der „ddrspa_if“-IP-Core besitzt ein zusätzliches Bus-Interface namens „CLKRSTGEN_PORT“. Über dieses Bus-Interface kann der IP-Core über eine Initiator-Target-Verbindung mit dem „clkrstgen_if“-IP-Core verbunden werden. Dadurch werden automatisch alle benötigten Clock-Signale an den „ddrspa_if“-IP-Core getrieben.

5.5.6 „apbvga_if“ und „svgactrl_if“-IP-Core

Die IP-Cores „apbvga_if“ und „svgactrl_if“ sind sehr nützliche SoC-Komponenten. Vorraussetzung zur Benutzung dieser IP-Cores ist allerdings eine auf dem FPGA-Board befindliche SVGA Schnittstelle.

5.5.6.1 „apbvga_if“-IP-Core

Der „apbvga_if“-IP-Core ist ein effizienter Ersatz für den „apbuart_if“-IP-Core und dient als Ausgabe-Komponente für die Kommandozeile.

Der IP-Core teilt den Bildschirm in 80 Spalten mal 37 Zeilen ein und kann in jedem Segment ein Zeichen anzeigen. Dieses Zeichen benötigt einen Speicherplatz von genau einem Byte. Damit für die Kommandozeile eine gewisse Scroll-Funktion möglich ist werden insgesamt 51 Zeilen gespeichert.

Dies ergibt einen gesamten Speicherbedarf, für den Text Framebuffer, von:

Speicherbedarf = $1\frac{Byte}{Zeichen} \cdot 80 Spalten \cdot 51 Zeilen = 4080 Bytes < 4 kByte$

Diese knapp 4 kByte Zeichendaten werden in den 2 BRAMs gespeichert, welche von dem „apbvga_if“-IP-Core zusätzlich eingebunden werden.

5.5.6.2 „svgactrl_if“-IP-Core

Der „svgactrl_if“-IP-Core übernimmt die selbe Funktion, wie der „apbvga_if“-IP-Core, nur für Grafikdaten. Als Framebuffer werden allerdings, je nach Auflösung und Farbtiefe, mehrere MByte Kapazität benötigt. Für eine Auflösung von 1024x786 mit einer Farbtiefe von 32 Bit wären dies:

Speicherbedarf = $1024 \cdot 768 \cdot 4 Byte = 3 MByte$

Der Framebuffer kann an einer beliebigen Stelle des Systemspeichers untergebracht werden. Die Startadresse diesen Speichers kann in einem Register angegeben werden.
Der „svgactrl_if“-IP-Core besitzt ein AHB-Master-Interface, über das der IP-Core einen eigenständigen Lese-Transfer aus dem Framebuffer initiieren kann. Dadurch bleibt der LEON3 unbelastet. Lediglich die Transferdauer der Framebufferdaten über den Bus ist nicht zu vernachlässigen.

Das „SnapGear“-Linux, für den LEON3-Prozessor, kann mit speziellen Treibern für diesen IP-Core kompiliert werden und ist dadurch befähigt eine grafische Ausgabe der Konsole auf einem Monitor zu erzeugen.

5.5.7 „ahbram_if“-IP-Core

Der „ahbram_if“-IP-Core ist ein äußerst bemerkenswerter IP-Core und könnte mit zukünftigen FPGA-Generationen an Bedeutung gewinnen.
Der „ahbram_if“ IP-Core erlaubt es einen On-Chip-Speicher beliebiger Größe auf den FPGA unterzubringen. Der Speicher kann dann über ein AHB-Slave-Interface ausgelesen und beschrieben werden. Die Größe des Speichers kann mit der Generic-Konstante „**kbytes**“ in kByte eingestellt werden. Die Grenze der maximalen Speicherkapazität wird nur durch die Anzahl der BRAMs auf dem FPGA beschränkt. Der IP-Core schließt dann eine Mindestanzahl von benötigten BRAMs

zu einem Gesamtspeicher zusammen. Auf diesen Speicher kann über den gewöhnlichen Systemadressraum zugegriffen werden. Durch die Technologie der BRAMs werden für Lese-Transfers lediglich 1 Takt[5] und für Schreib-Transfers 2 Takte benötigt.

Spätere FPGA-Generationen werden, durch die steigende Integrationsdichte, immer mehr BRAMs enthalten. Dadurch wird es möglich das Betriebssystem eines Systems auf dem Chip selbst zu laden und mit hoher Geschwindigkeit auszuführen. Aufgrund der kurzen Zugriffzeiten könnte der Cache-Speicher eines SoC deaktiviert werden, um so zusätzlichen Konfigurationsraum zu gewinnen.

5.5.8 „apbps2_if“-IP-Core

Der „apbps2_if“-IP-Core wird verwendet, um einem PS2-Anschluss anzusteuern. An diesem kann dann beispielsweise eine Tastatur oder Maus angeschlossen werden. Der IP-Core wird an den APB angeschlossen und kann gegebenenfalls einen Interrupt auslösen, so dass Reaktionen äußerer Peripheriegeräte beantwortet werden können.
Der Kernel des „SnapGear“-Linux kann so konfiguriert werden, dass eine Tastatur oder Maus ohne weiteres benutzt werden kann.

5.5.9 „gptimer_if“-IP-Core

Der „gptimer_if“-IP-Core ist ein Timer-Core und wird an den APB angeschlossen. Der IP-Core ist mit dem Xilinx-IP-Core „opb_timer“ vergleichbar.
Der „gptimer_if“-IP-Core ist mit einem „prescaler“-Timer ausgestattet. Erst wenn der „prescaler“-Wert unterläuft, wird ein „tick“ ausgelöst, welcher den eigentlichen „timer“-Wert dekrementiert. Der „prescaler“- sowie der „timer“-Wert können bis zu 32 Bit lang sein. Dies ermöglicht einen Zeitmessungsintervallbereich von:

$[\frac{1}{AHB-Frequenz}, \frac{2^{64}}{AHB-Frequenz}]$ Sekunden

5.5.10 „logan_if“-IP-Core

Die Bezeichnung „logan“ steht für On-chip **Lo**gic **An**alyzer. Dieser IP-Core ist mit der Funktionalität der ChipScope IP-Cores vergleichbar.
Wird ein „logan_if“ IP-Core in ein GRLIB-SoC eingebunden so können bestimmte Signale überwacht und mit Hilfe von Tools dargestellt werden. Der GRLIB-Logic-Analyzer wird an den APB

[5] Ein Transfer von 1 Takt Dauer entspricht einem Transfer mit 0-waitstates und ein Transfer von 2 Takten Dauer entspricht einem Transfer mit 1-waitstates.

als Slave angeschlossen. Über sein APB-Slave-Interface wird er angesteuert. Zu diesen Ansteuerungen zählen beispielsweise das Aktivieren, Konfigurieren und Auslesen der Analyzer-Daten. Die zu überwachenden Signale werden über einen zusätzlichen Port namens „signals“ in den IP-Core getrieben.

Die Steuerung den „logan_if“ IP-Cores kann vollständig über den GRMON erfolgen. Mit folgenden GRMON Kommandos kann ein Signal einfach analysiert werden:

`grlib> la status` Gibt den Status des „logan_if“ IP-Cores an
`grlib> la arm` Beginnt die Analyse der Signale und speichert sie in einem BRAM ab.
`grlib> la dump` *`filename`* Speichert die aufgenommenen Daten, auf dem Host-PC, in eine Datei namens filename ab. Das verwendet Format ist VCD.

Tabelle 5.7: GRMON Kommandos - Ausschnitt für „logan“ [grmondoc]

Die Analysierten Daten im VCD-Format können dann beispielsweise mit einem Tool „GTKWave“ angezeigt werden.

Kapitel 6

Native C-Programme für ein LEON3-SoC

In diesem Kapitel soll eine kurze Einführung in die Software-Programmierung für ein LEON3-SoC gegeben werden. Dies ist hilfreich, da für das LEON3-System keine so umfangreiche Software-Bibliothek zur Verfügung steht, wie für den MicroBlaze oder PowerPC-Prozessor. Bekannte Befehle wie:

```
int daten = ReadFromGPInput(XPAR_PUSH_BUTTONS_3BIT_BASEADDR);
```

sind in der LEON3 Bibliothek nicht vorwanden.
Mit der obigen Zeile ist es, mit einem MicroBlaze-Prozessor, möglich das 32 Bit Register des „opb_gpio“-IP-Cores auszulesen. In diesem Fall werden die Zustände der Push-Buttons in der 32 Bit Variable „daten“ gespeichert. Der zuvor automatisch definierte Ausdruck „XPAR_PUSH_BUTTONS_3BIT_BASEADDR“ steht für die Startadresse des „opb_gpio“-IP-Cores, der mit den Push-Buttons getrieben wird.

Die Programmiersprache in der die nativen C-Programme geschrieben werden ist „C“. Diverse native C-Programme für diese Arbeit, welche zum Testen des LEON3-System dienten, bestehen aus zwei grundlegenden Operationen. Diese sind:

1. Lesen und Schreiben von Registern einzelner IP-Cores
2. Arithmetische Operationen, welche von dem LEON3-Prozessor intern ausgeführt werden. (Beispiel: „x = x + 3;“)

Punkt 1 der grundlegenden Operationen soll in den Kapitelpunkten 6.1 - 6.3 vorgestellt werden. Die nativen C-Programme, welche während der Arbeit zum Testen der GRLIB-SoCs benutzt wurden sind auf der DVD zur dieser Arbeit abgespeichert, siehe Anhang D.

6.1 Registerdeklaration der IP-Cores

Um Zugriff auf die Register aller IP-Cores, welche mit dem AHB oder APB verbunden sind, zu erhalten, werden Strukturen für jeden IP-Core definiert. Diesen Strukturen muss eine Adresse zugewiesen werden. Diese entspricht üblicherweise der Basisadresse des IP-Cores.
Wird ein Register der Registerstruktur angesprochen, so ist die eigentliche Kommunikationsadresse: Basisadresse + ((Index der Registervariable) · 32 Bit)

```
struct apbuart_reg
{

    volatile int data;      //Index 0
    volatile int status;    //Index 1
    volatile int control;   //Index 2
    volatile int scaler;    //Index 3

};

struct grgpio_reg
{

    volatile int data;
    volatile int out;
    volatile int dir;

};
```

Abbildung 6.1: Definition der Regsiterstruktur

Die Abbildung 6.1 zeigt zwei Strukturen, die 32-Bit-Integer-Registervariablen beinhalten. 32 Bit entspricht genau der Länge eines Registers der GRLIB-IP-Cores. Laut Dokumentation besitzt der „apbuart_if“-IP-Core 4 Register, welche die gleichen Bezeichnungen haben wie die Integer Variablen der „apbuart_reg“-Struktur, in der Abbildung 6.1.
Dies trifft auch für die ersten 3 Register des „grgpio_if“-IP-Cores zu. Dieser wird verwendet um Push-Butttons auszulesen oder LEDs zu steuern. In Abbildung 6.1 wird die Struktur „grgpio_reg“ definiert. Sie erlaubt Zugriff auf die ersten 3 Register des „grgpio_if“-IP-Cores.

Die zusätzliche Bezeichnung „`volatile`“ sagt aus, dass es sich um eine Variable handelt, welche nicht nur durch den herkömmlichen Programmablauf verändert werden kann. Es ist auch möglich, dass der Wert der Variable durch eine Hardware oder einen anderen Thread beschrieben wird. In unserem Fall wäre dies durch eine Hardware.

6.2 Deklaration einer Registerstruktur

Um Register auslesen oder beschreiben zu können, muss nun im eigentlichen Programm ein konkreter Pointer auf eine Registerstruktur deklariert werden. Die dabei verwendete Struktur entspricht üblicherweise der Registerstruktur des IP-Cores, mit welchen kommuniziert werden soll. Möchte ein Benutzer beispielsweise die serielle Schnittstelle des GRLIB-SoC benutzen und die Push-Buttons in seinem Programm einbinden, so müssen folgende Deklarationen stattfinden:

```
int main(void)
{
     struct grgpio_reg *grgpioin = (struct grgpio_reg *) 0x80000100;
     struct apbuart_reg *uart = (struct apbuart_reg *) 0x80000300;
     ...
}
```

Abbildung 6.2: Deklaration der Strukturpointer für IP-Cores

Die Abbildung 6.2 zeigt die Deklaration der Pointer „grgpioin“ und „uart“ auf die Strukturen „grgpio_reg“ sowie „apbuart_reg“. Dem Pointer wird eine Startadresse des GRLIB-Systemadressraums zugewiesen. Diese Adresse muss mit der Systemadresse des jeweiligen IP-Cores übereinstimmen.

Im Fall der Abbildung 6.2 setzen sich die Adressen wie folgt zusammen:

Startadresse des „grgpioin“ Pointers =
0x & haddr (des APB) & paddr (des „grgpio_if“-IP-Cores) & 00 = 0x 800 & 001 & 00

Startadresse des „uart“ Pointers =
0x & haddr (des APB) & paddr (des „apbuart_if“-IP-Cores) & 00 = 0x 800 & 003 & 00

Soll auf IP-Core-Register zugegriffen werden, welche als Slave mit dem APB verbunden sind, so muss die Adresse stets mit der „haddr“-Generic-Konstante des APB beginnen. Darauf folgt dann die „paddr“-Generic-Konstante des gewünschten IP-Cores.

Zu bemerken ist hier, dass die Startadressen der IP-Core-Registers fest im C-Programm vorgegeben sind. Würde ein IP-Core in der Hardware entfernt, vertauscht oder umkonfiguriert werden so müsste das C-Programm angepasst werden. Dies wäre theoretisch nicht nötig, wenn der Systemspeicher des „Plug and Play“ Konfigurationsbereiches ausgelesen und verarbeitet werden würde. Mit dieser Methode könnte das Programm nach der Identifikationsnummer des „apbuart“-IP-Cores suchen und die dort abgelegte Adresse als Startadresse des Pointer verwenden. Dies war allerdings für die einfachen Tests in dieser Arbeit nicht nötig.

6.3 IP-Core Register lesen und schreiben

Die entsprechende Syntax für die Kommunikation mit den IP-Core Register, ist folgende:

```
int main(void)
{

     ... //Pointerdeklarationen siehe Abbildung 6.2
     int y;
     int daten = grgpioin->data;                        //Zeile A
     uart->scaler = (((ahbfreq*10)/(38400*8))-5)/10; //Zeile B
     uart->data = y;                                    //Zeile C

}
```

Abbildung 6.3: Deklaration der Strukturpointer für IP-Cores

Zeile A in Abbildung 6.3 leitet einen Single-Read-Transfer auf dem AHB ein, dieser wird durch seine Adresse (0x80000100) an den APB weitergeleitet. Es soll mit Zeile A der Zustand der Push-Buttons in der Variable „daten“ gespeichert werden.

Zeile B in Abbildung 6.3 startet einen Single-Write-Transfer an die Adresse 0x8000030C. Das „Scaler“-Register des „apbuart_if“-IP-Cores wird auf einen bestimmten Wert gesetzt um eine Baudrate von 38400 Bauds/s zu erreichen.

Zeile C in Abbildung 6.3 startet ebenfalls einen Single-Write-Transfer an die Adresse 0x80000300. Die 8 LSB der „y“ Variable sollen über die serielle Schnittstelle übertragen werden.

Es wurde eine syntaktische Vereinfachung verwendet, denn „uart->data“ steht beispielsweise für „(*uart).data“.

6.4 native C-Programme im EDK

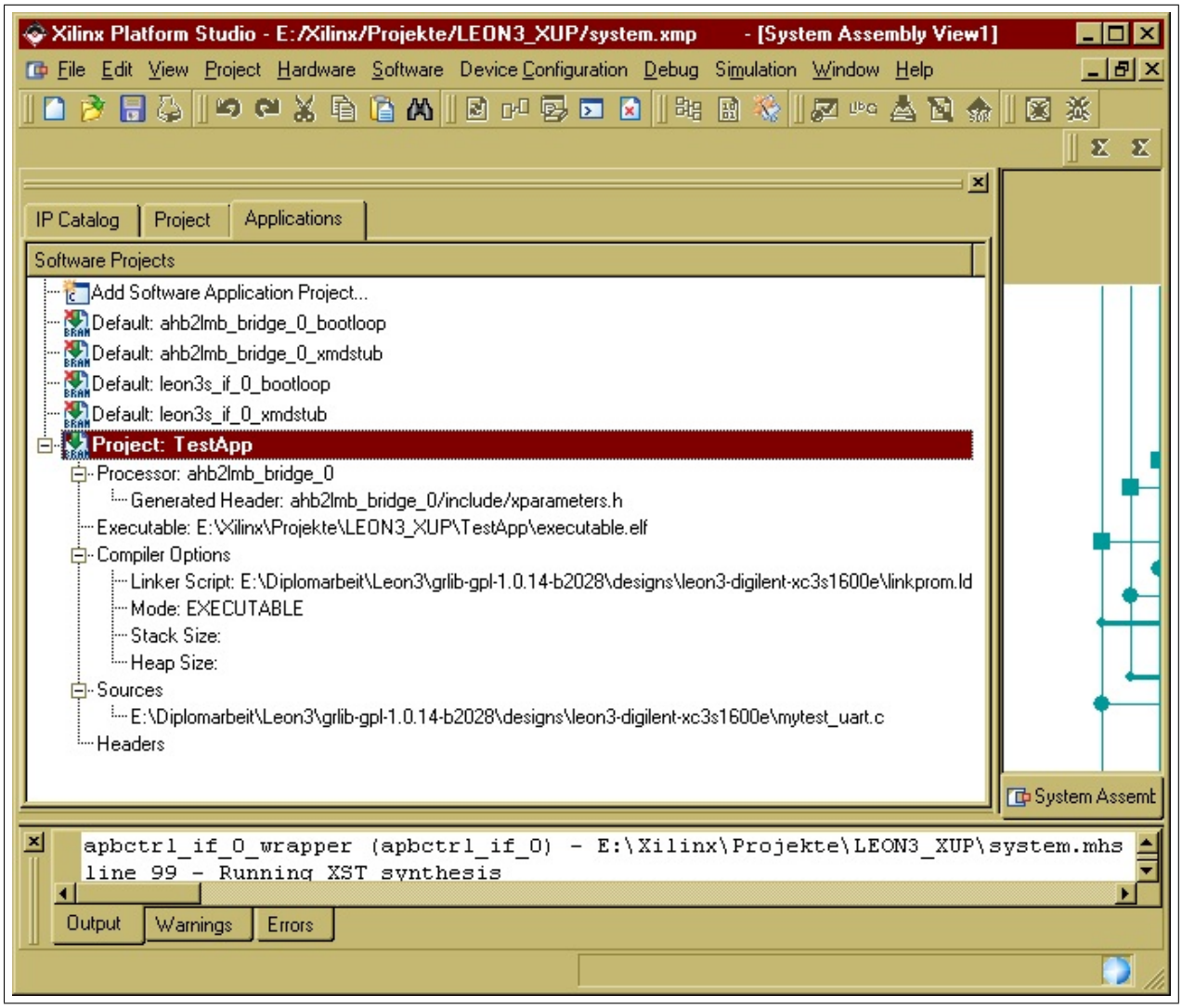

Abbildung 6.4: „Applications“-Tab des EDK in einem GRLIB-SoC

Abbildung 6.4 zeigt den „Applications“-Tab der EDK-GUI. Hier kann das native C-Programm eingetragen und für das SoC konfiguriert werden. Soll ein natives C-Programm für ein GRLIB-System, mit dem LEON3-Prozessor, im Arbeitsplan des EDK kompiliert werden, so sind folgende Punkte zu beachten.

Punkt 1 (Projektname)

Der Projektname des nativen C-Programms muss immer „TestApp“ lauten. Dies ist am selektierten Eintrag in Abbildung 6.4 zu erkennen.
Die Gründe für diese Festlegung sind folgende. Da die „system.make“ für ein GRLIB-System in

die „systemLEON.make“ abgeändert werden musste, entfallen somit alle dynamischen Änderungen, welche das EDK vor dem 1. Arbeitsschritt an der normalen „system.make“ vornimmt.
Zu diesen dynamischen Änderungen gehört die Generierung eines MAKEFILE Einsprungspunkt zur Kompilierung des nativen C-Programms. Dieser Einsprungspunkt ist vom Projektnamen des nativen C-Programms abhängig. Der Vorgang ist in der „system.make“ ab Zeile 138 zu beobachten.

Da die veränderte „systemLEON.make“ nicht mehr durch das EDK dynamisch verändert wird, bleibt der Einsprungspunkt immer gleich. Demzufolge muss auch der Projektname immer gleich lauten.

Punkt 2 (assoziierter Prozessor)

Da die Einstellungen des „Applications“-Tabs Auswirkungen auf die Generierung der BMM haben, muss das native C-Programm dem Prozessor „ahb2lmb_bridge“ zugeordnet werden. Würde dies nicht befolgt, so würde keine BMM generiert werden, da der LEON3-Prozessor nicht direkt an den LMB angeschlossen ist.

Zu Beachten 6.1 (assoziierter Prozessor)

Der assoziierte Prozessor für ein natives C-Programm muss die AHB2LMB-Bridge sein. Abbildung 6.4 zeigt dies deutlich.

Punkt 3 (Linkerskript)

Als Linkerskript muss für das native C-Programm ein benutzerdefiniertes Linkerskript angegeben werden. Für diese Arbeit wurde ein Linkerskript namens „linkprom.ld“ benutzt (siehe Abbildung 6.4). Dieses Linkerskript linkt alle ELF-Sektionen auf die Adresse 0x00000000. An dieser sollte sich auch der Adressbereich der AHB2LMB-Bridge befinden.

Sollen einige ELF-Sektionen in den SDRAM des GRLIB Systems gelinkt werden, so müsste lediglich das Linkerskript verändert werden.

Punkt 4 (Compiler Optionen)

Compiler Optionen können im Dialogfeld „Set Compiler Options“ im Tab „Paths and Options“ vorgenommen werden. In dem Editfeld „Other Compiler Options to Append“ können zusätzliche Compiler Optionen für den BCC angegeben werden.
Da beispielsweise keine FPU für den LEON3 der GRLIB vorhanden ist, muss dort permanent „`-msoft-float`“ eingetragen sein (siehe Abbildung 6.5). Soll beispielsweise ein natives C-Programm die V8 Multiplizierer und Dividierer Hardware benutzen, so muss der Parameter „`-mv8`“ hinzugefügt werden.

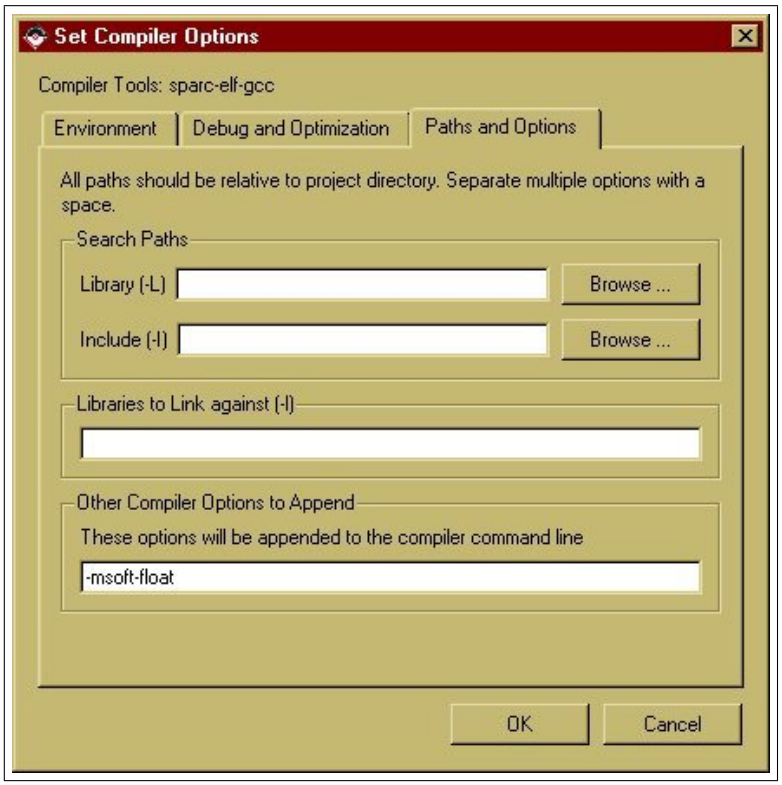

Abbildung 6.5: „Paths and Options“ Tab des EDK 8.2

Abbildung 6.5 zeigt den „Paths and Options“-Tab des „Set Compiler Options“-Dialogfelds. Es ist zu erkennen, dass alle Gleitkommaoperationen emuliert werden sollen.

Kapitel 7

Linux auf dem LEON3-System

Ein weiteres Ziel dieser Arbeit war es, das Betriebssystem Linux für ein LEON3-System verfügbar zu machen. Dieses Ziel konnte mit der Linux-Distribution „SnapGear“ erreicht werden. In diesem Kapitel soll erläutert werden wie ein SnapGear-Linux kompiliert wird und welche Besonderheiten bei der Benutzung des SnapGear-Linux entstehen.
Das SnapGear-Linux wird auf der Homepage der Firma GR immer in der neusten Version bereitgestellt. Die neuste Version integriert ebenfalls die neuste Kernel-Version. In dieser Arbeit wurde das SnapGear-p32 Linux mit dem Kernel 2.6.18.1 verwendet.

Besonderheiten eines Linux für das LEON3-System

Einem Linux, welches auf einem GRLIB-LEON3-System laufen soll, wird eine besondere Möglichkeit geboten. Durch diese Möglichkeit wird der Erstellungsprozess eines Linux, für den LEON3-Prozessor in einem GRLIB-System, erheblich vereinfacht. Diese Möglichkeit bietet die „Plug and Play“ Funktionalität der Busse AHB und APB.

Zur Erstellung eines Linux-Images[1] für den MicroBlaze-Prozessor, in Verbindung mit SoC-Komponenten aus der Xilinx-Bibliothek, ist ein erheblich aufwendigerer Prozess notwendig, als es bei dem SnapGear-Linux der Fall ist. Bei diesem Erstellungsprozess muss die Konfiguration jeder einzelnen SoC-Komponente in das Linux, vor dessen Kompilierung, übertragen werden. Zu diesen Konfigurationen gehören beispielsweise Systemadressen oder Übertragungsraten („opb_uartlite“).

[1] Soll ein Linux-Programm auf einem SoC eingespeist werden, so wird es in einer einzigen ELF-Datei gespeichert. Diese Datei wird auch „Linux-Image“ genannt.

Die Konfigurationen der SoC-Komponenten werden also nicht über den OPB in einem Konfigurationsbereich zur Verfügung gestellt.
Angenommen es würde eine bestimmte Anzahl SoCs, mit dem MicroBlaze-Prozessor und SoC-Komponenten aus der Xilinx-Bibliothek, synthetisiert werden. Alle SoCs würden die selben SoC-Komponenten beinhalten, welche allerdings unterschiedliche Konfigurationen hätten. Dann wären, für das MicroBlaze-System, genauso viele Linux-Images nötig, damit jedes SoC mit einem Linux betrieben werden könnte.

Anders verhält es sich mit einem Linux für die GRLIB mit dem LEON3-Prozessor. Da alle Konfigurationen, der SoC-Komponenten im Konfigurationsbereich des AHB bereitgestellt werden, können diese durch das Linux verwendet werden. Dadurch können die SoC-Komponenten-Konfigurationen während des Bootens des Linux aus dem Konfigurationsbereich gelesen werden.
Angenommen es existierten eine bestimmte Anzahl GRLIB-SoCs mit gleichen SoC-Komponenten aber unterschiedlichen Konfigurationen. In diesem Fall könnten alle SoCs mit dem selben Linux-Image betrieben werden. Eine Voraussetzung ist natürlich, dass alle notwendigen Treiber, für alle verwendeten SoC-Komponenten, in das Linux-Image eingebunden wurden. Das aktuelle SnapGear-Linux enthält immer alle notwendigen Treiber für alle IP-Cores der aktuellen GRLIB und somit auch für alle SoC-Komponenten.

Zu Beachten 7.1 (Systemvoraussetzungen für das SnapGear-Linux)

Um das SnapGear-Linux fehlerfrei ausführen zu können wird ein GRLIB-SoC mit LEON3-Prozessor und einem Timer benötigt. Eine Timer-Komponente bietet die GRLIB mit dem IP-Core namens „gptimer_if“ an.

7.1 LEON Cross-Compiler für Linux

Bevor ein SnapGear-Linux kompiliert werden kann, wird ein Cross-Compiler benötigt. Dieser Cross-Compiler enthält wichtige Bibliotheken und als Hauptbestandteil einen GNU GCC Compiler, sowie einen Linker. Der Cross-Compiler kann auf der Homepage der Firma GR unter dem Namen „sparc-linux-1.0.0.tar.bz2“ heruntergeladen werden. Mit dem Kommando:

```
$ tar -jvxf [path]/sparc-linux-1.0.0.tar.bz2
```
[2]

kann der Cross-Compiler in das aktuelle Verzeichnis extrahiert werden.

[2] Der Ausdruck [path] steht für den Pfad zum Cross-Compiler Archive.

Zu Beachten 7.2 (Linux Cross-Compilers komfigurieren)

Es muss sichergestellt sein, dass der Pfad zum „bin“-Verzeichnis des Cross-Compilers in die „PATH“-Variable exportiert wurde. Dadurch erhält der Compiler-Workflow des SnapGear-Linux ständigen Zugriff auf den Cross-Compiler.

Dies wird mit dem Kommando:

```
$ export PATH=$PATH:[path_cc]/bin
```
[3]

bewerkstelligt.

Der LEON Cross-Compiler steht nun für die Kompilierung eines SnapGear-Linux bereit.

7.2 Das „SnapGear“-Linux

Das SnapGear-Linux kann auf der Homepage der Firma GR heruntergeladen werden. Es ist in Archiven namens „snapgear-???.tar.bz2“[4] enthalten. Im Falle dieser Arbeit handelt es sich um die SnapGear-Linux-Version „p32“. Das Archiv muss in ein beliebiges Verzeichnis entpackt werden. Dies geschieht mit dem Kommando:

```
$ tar -xvjf snapgear-???.tar.bz2
```

Es wird automatisch das Verzeichnis „snapgear-???“[5] erstellt.

Das SnapGear-Linux wird intern mit zwei Linux-Versionen vertrieben, welche optional ausgewählt werden müssen. Die Wahl ist davon abhängig, ob der LEON3 des Systems mit MMU oder ohne MMU synthetisiert wurde.
Ein SnapGear-Linux besitzt 2 Einstellungen, welche vor der Kompilierung festgelegt werden müssen. Diese Einstellungen sind die Version des Linux-Kernels (2.0.x oder 2.6.x) und die Ausstattung des LEON3-Prozessors (mit MMU oder ohne MMU).

[3] Der Ausdruck [path_cc] steht für das Verzeichnis, in dem sich der extrahierte Cross-Compiler befindet.
[4] Die drei „?“ stehen dabei für die Versionsnummer.
[5] Dieses Verzeichnis wird im folgendem mit dem Ausdruck [SNAPGEAR_DIR] bezeichnet.

Die Tabelle 7.1 zeigt eine Übersicht, bezüglich der LEON3 Konfiguration und der Linux Version.

Version	LEON3	
	mit MMU	ohne MMU
Linux 2.0.x	nicht vorgesehen*	vorgeschriebene Wahl
Linux 2.6.x	vorgeschriebene Wahl	läuft nicht

* diese Kombination ist nicht vorgesehen, da das SnapGear-Linux 2.0.x nicht mit der MMU Option kompiliert werden kann. In dieser Arbeit konnte das „Linux 2.0.x ohne MMU“-Image allerdings auf einem LEON3 mit MMU ausgeführt werden, jedoch wurden nicht alle Funktionen des Linux auf Korrektheit getestet.

Tabelle 7.1: SnapGear-Linux-Version für LEON mit MMU und ohne MMU

Die Wahl der Linux-Version hat direkten Einfluss auf die Linux-Kernel-Version. Das SnapGear-Linux unterstützt neben dem LEON3-Prozessor zusätzlich den LEON2-Prozessor. Für den LEON3 wird ein spezieller Multi-Prozessor-Support durch das SnapGear-Linux bereitgestellt, dies gilt nicht für den LEON2.

7.2.1 Der Erstellungsprozess und dessen Werkzeuge

Bevor das SnapGear-Linux kompiliert werden kann, muss es zuerst konfiguriert werden.
Der Erstellungsprozess des SnapGear-Linux wird mit der selben Methodik und Benutzerfreundlichkeit vollzogen, wie es bei dem herkömmlichen Workflow der GRLIB bekannt ist. Der Erstellungsprozess ist aufgrund der „Plug and Play“ Funktionalität der GRLIB-Busse schneller zu bewältigen als bei einem vergleichbaren Linux für den MicroBlaze-Prozessor mit Xilinx-SoC-Komponenten.

Der Erstellungsprozess besteht aus drei Schritten:

Schritt 1: *Festlegung grundlegender Hardwareeinstellungen bezüglich des GRLIB-SoC*

Schritt 2: *Konfiguration des Linux-Kernels. (optional)*[6]

Schritt 3: *Auswahl von bereitgestellten Applikationen, welche für das SnapGear-Linux nutzbar sein sollen. (optional)*

Alle drei Schritte werden ab dem Kapitelpunkt 7.2.1.1 genauer beschrieben.
Als Ergebnis des Erstellungsprozesses steht eine Datei namens „image.dsu“. Diese Datei ist

[6] Die mit „(*optional*)“ gekennzeichneten Schritte des Erstellungsprozesses müssen nicht unbedingt, für ein erfolgreiches Kompilieren, durchgeführt werden. Sie bieten allerdings Einstellungsmöglichkeiten, welche sehr nützlich sind.

ein LEON-Linux-Image im ELF-Format und unterscheidet sich im Prinzip nicht von den ELF-Dateien, welche aus den nativen C-Programmen entstehen. Ein solches Linux-Image wird mit dem GRMON in das SoC geladen und kann dann gestartet werden.
Alle Einstellungen des SnapGear-Linux können, während des Erstellungsprozesses, mit einer GUI vorgenommen werden. Diese GUI ist vergleichbar mit der GUI des GRLIB-Workflows. Damit die GUI genutzt werden kann, muss Tcl/tk 8.4 oder jünger, auf dem Host PC, installiert sein.

7.2.1.1 Schritt 1 (Grundeinstellungen zum SoC)

Der Erstellungsprozess des SnapGear-Linux erfolgt über das Konzept der MAKEFILES. Die ersten Grundeinstellungen können mit einem GUI vorgenommen werden, das gleichzeitig ein Hauptmenü darstellt.

Befindet sich der Benutzer in Verzeichnis „[SNAPGEAR_DIR]“ so genügt das Kommando:

```
$ make xconfig
```

und es öffnet sich das folgende Hauptmenü.

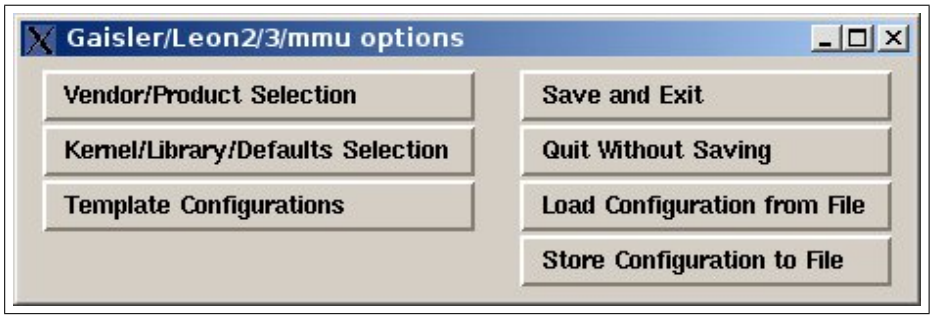

Abbildung 7.1: Hauptmenü des „SnapGear“-Linux Erstellungsprozesses

Um alle Grundeinstellungen vorzunehmen werden zwei Untermenüs angeboten, das „**Vendor/Produkt Selection**“ und das „**Kernel/Library/Defaults Selection**“-Menü.

„Vendor/Produkt Selection“

In diesem Untermenü kann als „Vendor“ der Anbieter „gaisler“ ausgewählt werden. Dies ermöglicht eine Wahl des „gaisler Products“ zwischen LEON2 und LEON3 jeweils mit oder ohne MMU. Unter „Gaisler/Leon2/3mmu options“ können weitere Prozessor- und SoC-spezifische Einstellungen vorgenommen werden. Diese Einstellungen beziehen sich auf die konkrete SoC-Konfiguration für die das SnapGear-Linux kompiliert werden soll.

Zu Beachten 7.3 (Linking-Adresse für SnapGear-Linux)

Im „Gaisler/Leon2/3mmu options“ Menü wird die Systemadresse festgelegt an die das Linux-Image gelinkt wird. Standardmäßig ist diese Adresse 0x40000000.
Da an dieser Adresse das Linux-Image gespeichert wird, wenn es mittels GRMON in das SoC geladen wird. Sollte sich ein „ddrspa_if“-IP-Core mit „haddr = 0x400“ innerhalb des SoC befinden. Ist dies nicht der Fall so kann das Linux Programm nicht gespeichert werden.

„Kernel/Library/Defaults Selection“

In diesem weiteren Untermenü kann unter „Kernel-Version“ der verwendete Kernel für das SnapGear-Linux ausgewählt werden. Die aktuelle Kernel-Version der verwendeten SnapGear-Version wird an dieser Stelle angezeigt. Die Wahl des Kernels muss unter der Berücksichtigung der Tabelle 7.1 geschehen.

Hier können auch die beiden optionalen Schritte des Linux-Erstellungsprozesses aktiviert werden. Für den Schritt 2, welcher zur Konfiguration des Kernels dient, wird der Menüpunk „Customize Kernel Settings“ verwendet. Der Schritt 3, der zur Auswahl der Linux Applikationen dient, kann mit dem Menüpunkt „Customize Vendor/User Settings“ aktiviert werden.

Linux-Image kompilieren und verwenden

Wurden alle ausgewählten Konfigurationen vorgenommen und immer abgespeichert, so werden die Menüveränderungen in verschiedenen „.config“ Dateien abgespeichert. Folgende Kommandos leiten nun die Kompilierung des Linux-Images ein:

```
$ make dep # Ist nur notwendig, wenn der Linux-Kernel 2.0.x im Schritt 1 gewählt wurde
$ make
```

Wurde der Kompilierungsprozess fehlerfrei abgeschlossen, so befindet sich das Linux-Image „image.dsu“ im Verzeichnis „[SNAPGEAR_DIR]/images“. Die Linux-Imagedatei ist ein herkömmliches Programm im ELF-Format. Es kann mit dem GRMON in das SoC geladen und ausgeführt werden.

Dies wird mit folgenden GRMON Kommandos durchgeführt:

```
grlib> load image.dsu
section: .stage2 at 0x40000000, size 10180 bytes
section: .vmlinux at 0x40004000, size 1171520 bytes
section: .rdimage at 0x40139c14, size 1410908 bytes
total size: 2592608 bytes (57.6 Mbit/s)
read 3542 symbols
entry point: 0x40000000

grlib> verify image.dsu # überprüft das korrekte
Laden des Linux-Image
grlib> run
```

Abbildung 7.2: Linux-Image mit GRMON laden

Abbildung 7.2 zeigt die Durchführung des Ladens, Prüfen und Ausführen eines Linux-Images in ein GRLIB-LEON3-System.

7.2.1.2 Schritt 2 (Konfiguration des „SnapGear“-Linux-Kernels)

Für die Ausführung dieses Schritts muss der Menüpunkt „Customize Kernel Settings“, in Schritt 1, aktiviert werden. Wird daruf das Hauptmenü gespeichert und beendet, so erscheint automatisch eine neue GUI. Diese GUI dient nun zur Konfiguration des Kernels.
Dieser Schritt des Linux-Erstellungsprozesses unterscheidet sich jeweils nach der gewählten Kernel-Version.

Konfiguration des Kernels 2.6.x

Die Konfiguration des Kernels 2.6.x umfasst hier:

- Die Auswahl bestimmter Treiber für GRLIB-SoC-Komponente. Solche Treiber sind beispielsweise erhältlich für: „apbuart“, „svgactrl“ oder „apbps2“ (Keyboard oder Maus).
- Konfiguration von Systemeinstellungen. Dies betrifft hier beispielsweise den Prozessor oder Bus.

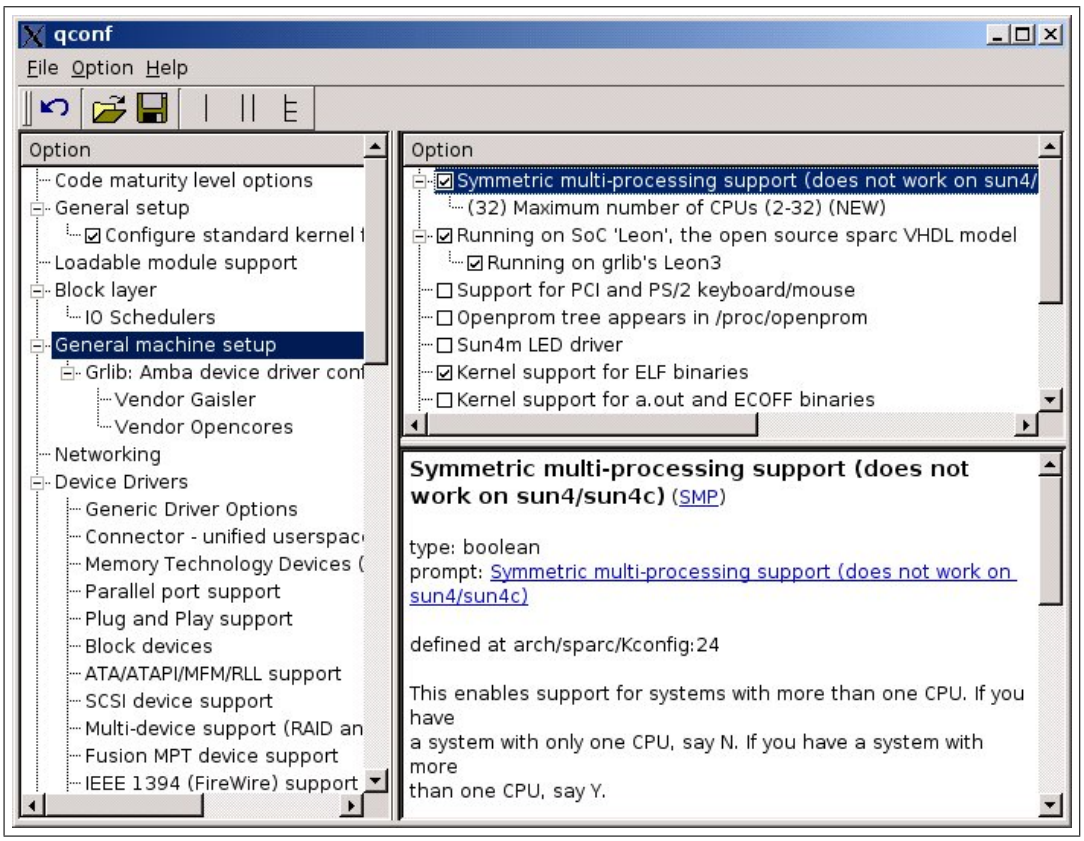

Abbildung 7.3: GUI zur Kernel 2.6.x Konfiguration

Abbildung 7.3 zeigt einen kleinen Ausschnitt der GUI für die Konfiguration des Kernels. Es ist beispielsweise zu erkennen, dass der Kernel 2.6.x für ein Multi-Prozessor SoC kompiliert werden soll.

Konfiguration des Kernel 2.0.x

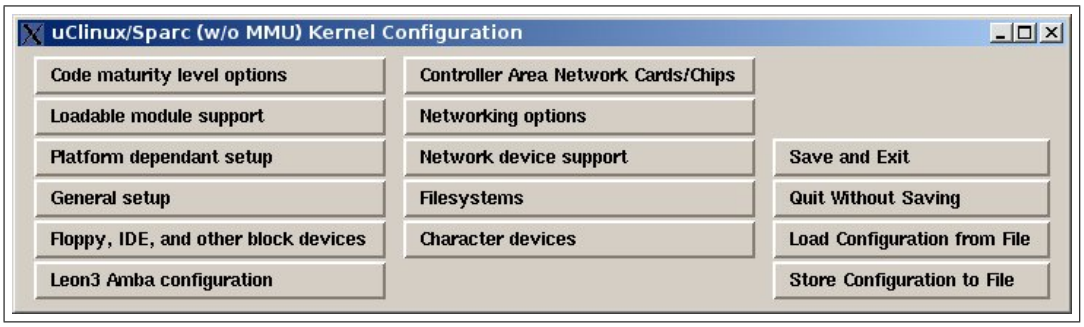

Abbildung 7.4: GUI zur Kernel 2.0.x Konfiguration

Die Konfigurationsmöglichkeiten des Kernels 2.0.x sind nicht so umfangreich wie die Möglichkeiten des Kernels 2.6.x. Es werden nur grundlegende Treiber unterstützt, wie beispielsweise für den „apbuart“ oder den „greth“-IP-Core.

Auf einzelne Konfigurationsmöglichkeiten kann in dieser Arbeit nicht eingegangen werden, da dies zu umfangreich wäre. Für weiteres Interesse wird auf die Dokumentation [snaplinuxdoc].

7.2.1.3 Schritt 3 (Anwendungen für das „SnapGear“-Linux)

Als letzter Schritt des Erstellungsprozesses können Applikationen ausgewählt werden, welche in das Linux-Image eingebunden werden sollen. Um diese Auswahl treffen zu können muss im Hauptmenü, während des Schritts 1, der Menüpunkt „Customize Vendor/User Settings“ aktiviert worden sein.

Folgende Abbildung 7.5 zeigt das „Customize Vendor/User Settings“ Menü für den Kernel 2.6.x:

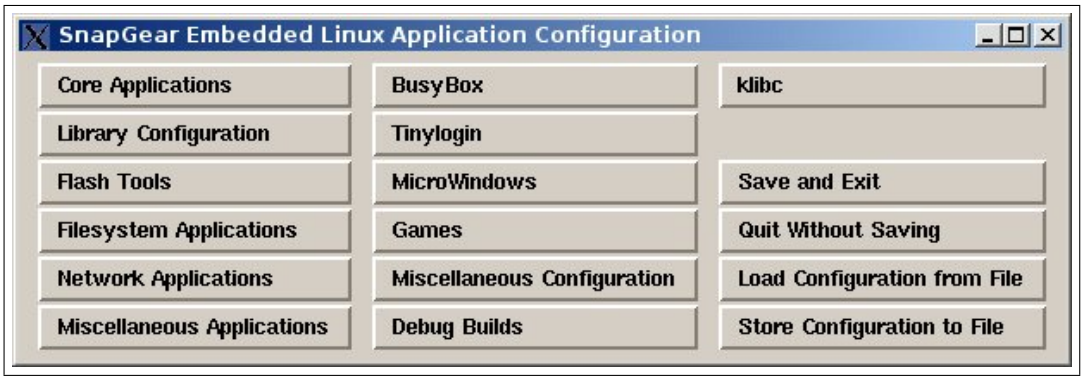

Abbildung 7.5: GUI zur Linux Applikation Auswahl

Alle Applikationen sind im obigen Menü nach Verwendungszweck klassifiziert. So können beispielsweise alle Anwendungen zur Ansteuerung und Konfiguration des Netzwerkes im Menü „Network Applications“ (siehe Abbildung 7.5) ausgewählt werden.
Ein anderes Beispiel ist der Dhrystone Benchmark, mit dem die Leistungswerte für das Linux 2.6.18.1 im Kapitel 8 (Ergebnisse) aufgenommen wurden. Dieser Benchmark kann im Menü „Miscellaneous Applications“ in das Linux-Image eingebunden werden.

Eine sehr interessante Applikation ist das MicroWindows. Das MicroWindows ist ein kleines Fenstersystem, welches in den aktuellen SnapGear-Versionen ständig weiterentwickelt wird. Das MicroWindows Fenstersystem schreibt seine Grafikinformationen direkt in einen Framebuffer. Um diese Anwendung vollwertig ausführen zu können, wird also der „svgactrl_if“-IP-Core sowie dessen Treiber für das SnapGear-Linux 2.6.x benötigt.

7.2.2 Hinzufügen zusätzlicher Treiber

Da das SnapGear-Linux nur über Treiber verfügt, welche die IP-Cores der GRLIB unterstützen, könnte ein Benutzer gezwungen sein zusätzliche Treiber für seine eigenen IP-Cores zu program-

mieren und diese in den Workflow des SnapGear-Linux einzubinden. Dieser Vorgang ist prinzipiell möglich.
Alle Treiber für die GRLIB-IP-Cores befinden sich, beispielsweise für den Kernel 2.6.x, im Verzeichnis „[SNAPGEAR_DIR]/linux-2.6.x/drivers“. Dort könnte ein weiteres Unterverzeichnis angelegt werden, welches den zusätzlichen Treiber beinhaltet.
Die GUI-Menüs des Linux Erstellungsprozesses wurden mittels Tcl-Skripts zusammengestellt. Diese Tcl-Skripts könnten leicht angepasst werden und so einen weiteren Menüpunkt, zur Konfiguration der zusätzlichen Treiber, aufnehmen.

Treiber für OPB-IP-Cores über die AHB2OPB-Bridge

Viele SoC-Entwickler haben, in der Vergangenheit, für ihre eigenen IP-Cores eigene Treiber programmiert. Diese Treiber wurden bisher auf einem MicroBlaze-System umgesetzt. Die eigenen IP-Cores werden an den OPB angeschlossen, der wiederum, über die AHB2OPB-Bridge, mit den LEON3-SoC verbunden ist.
Da das LEON3-System das MicroBlaze-System ersetzen soll, wäre es wünschenswert die eigenen Treiber auf einem LEON3-SoC mit dem SnapGear-Linux einzusetzen.

Die AHB2OPB-Bridge bietet vollständige Transparenz. Es ist also möglich alle OPB-Slaves über eine reservierte Systemadresse mit dem LEON3-Prozessor zu kontaktieren. Die Systemadresse der AHB2OPB-Bridge wird mit ihrer Generic-Konstante „haddr“ beispielsweise auf 0x700 festgelegt. Dadurch erhält der LEON3-Prozessor, ab der Systemadresse 0x70000000, vollkommen transparenten Zugriff auf alle OPB-Slaves. Werden die eigenen IP-Core-Treiber durch das SnapGear-Linux ausgeführt, so können diese die Ansteuerung der eigenen IP-Cores, auf dem OPB, über die AHB2OPB-Bridge problemlos ausführen. Für die AHB2OPB-Bridge werden keine zusätzlichen Treiber benötigt. Für den LEON3 ist sie, bis auf ihre „Plug and Play“ Konfiguration, nicht zu erkennen sein. Es muss lediglich darauf geachtet werden, dass die Systemadresse der eigenen IP-Cores im Adressraum der AHB2OPB-Bridge liegt.
Die eigenen Treiber können, wie zu Beginn des Kapitelpunktes 7.2.2 erläutert, in das SnapGear-Linux eingebracht werden.

Kapitel 8

Ergebnisse

In diesem Kapitel werden alle Ergebnisse dieser Arbeit zusammenfassend vorgestellt und erläutert. Es sei vorab gesagt, dass die praktischen Umsetzungen dieser Arbeit, mit dem „XUP Virtex-II Pro Development Board“ stattgefunden haben. Auf diesem Board ist der FPGA „Virtex-II Pro 30 FF896“ installiert.

Das Board wurde über die JTAG-Schnittstelle mittel des USB-Kabels und der Software „Xilinx iMPACT“ konfiguriert. Es wurde ebenfalls ein Null-Modemkabel verwendet, um Ausgaben der IP-Cores „apbuart_if“ und „opb_uartlite“ an den Host-PC zu senden. Das Debugging wurde für die MicroBlaze-SoC über ein JTAG-USB-Kabel vorgenommen und für ein GRLIB-SoC über ein Null-Modemkabel oder ein Netzwerkkabel. Wurde für ein GRLIB-SoC ein Null-Modemkabel benutzt, so musste der IP-Core „ahbuart_if“ im SoC vorhanden sein. Bei Verwendung eines Netzwerkkabels musste der IP-Core „greth_if“ im SoC vorhanden sein.
Das häufigste Debugging fand über die Netzwerkschnittstelle des XUP-Boards statt. Für die Netzwerkverbindung zum Host-PC wurde ein Crossover Netzwerkkabel verwendet.
Alle Funktionalitäten der IP-Cores „apbvga_if“ und „svgactrl_if“ konnte mit Hilfe eines Monitors, der an die SVGA Schnittstelle des XUP-Boards angeschlossen war, erfolgreich getestet werden.

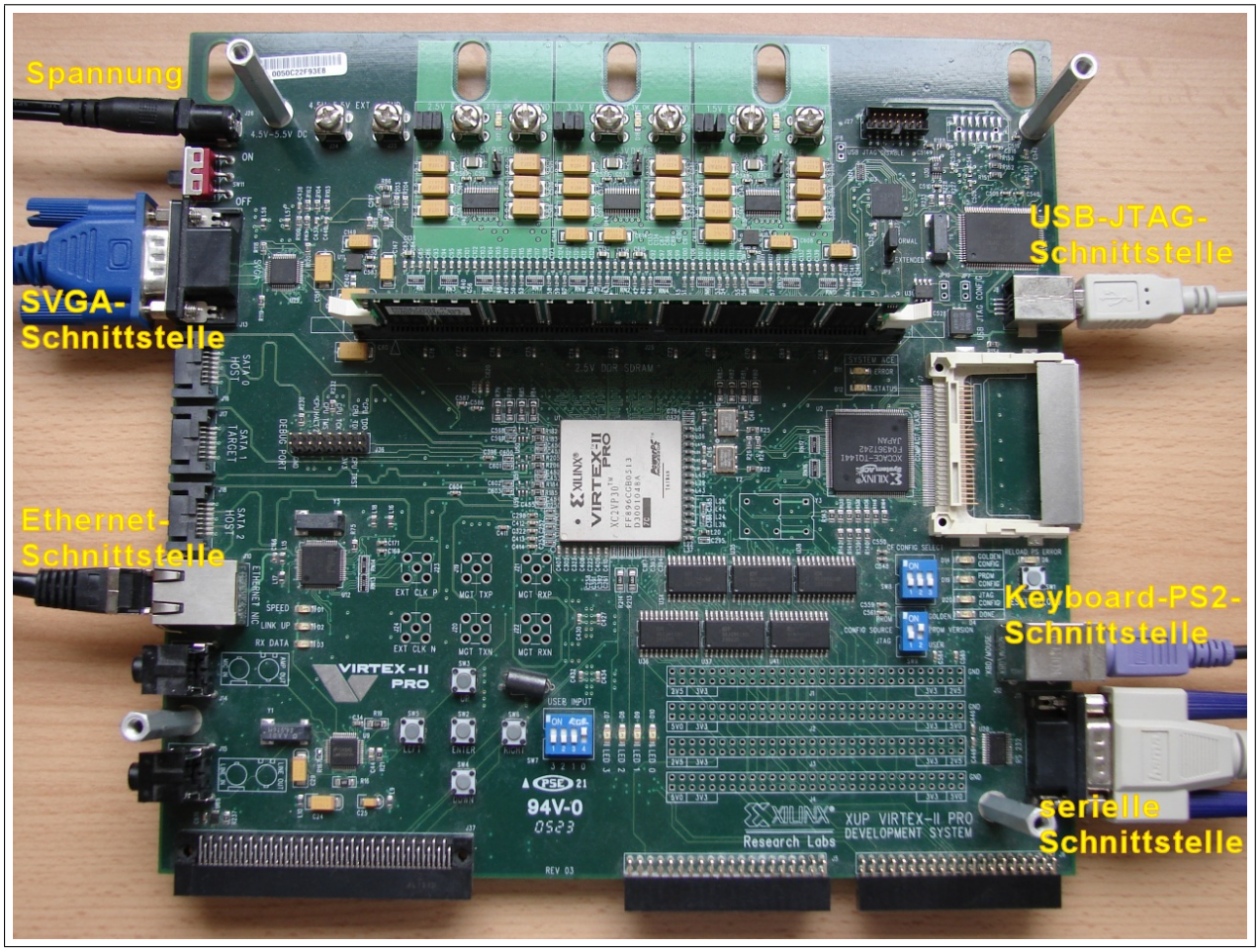

Abbildung 8.1: Das XUP-Board im typischen Versuchsaufbau

Abbildung 8.1 zeigt das XUP-Borad, wie es zum Testen eines GRLIB-SoC benutzt wurde. Die JTAG-USB-Schnittstelle und die serielle Schnittstelle waren bei jedem Test standardmäßig angeschlossen. In der fortgeschrittenen Testphase, wurde das Debugging über den GR Ethernet IP-Core durchgeführt und die Ethernet-Schnittstelle des XUP-Boards wurde benutzt. Als das SnapGear-Linux auf dem XUP-Board aufbereitet wurde, konnte ein Test der Keyboard-PS2-Schnittstelle und der VGA-Schnittstelle stattfinden. Der IP-Core „svgactrl_if“ konnte unter dem SnapGear-Linux getestet werden, indem eine grafische Konsolenausgabe auf einem Monitor erzeugt wurde.

8.1 Erreichte Ziele und Grenzen der Arbeit

Folgende Ziele wurden mit dieser Arbeit erreicht:

Ziel 1: *Es wurden alle notwendigen IP-Cores der Xilinx-Bibliothek durch kostenfreie IP-Cores der GRLIB ersetzt. Darunter gelten speziell die Ersetzungen des MicroBlaze-Prozessors durch den LEON3-Prozessor und des* ***O****n-Chip* ***P****eripheral* ***B****us (kurz OPB) durch den AMBA 2.0* ***A****dvanced* ***H****igh-performance*

Bus (kurz AHB). Als besonderes Merkmal der GRLIB gegenüber der Xilinx-Bibliothek, besitzt der LEON3-Prozessor eine MMU und der AHB eine „Plug and Play“ Funktion.

Ziel 2: *Alle zu ersetzenden GRLIB-IP-Cores wurden in das EDK eingebunden. Zur Umsetzung der Einbindung wurden die Vorteile der EDK-Konstrukte (wie MPD, PAO, MSS, Tcl) auf die IP-Cores des GRLIB angewandt. Dadurch konnte eine hohe Benutzerfreundlichkeit erzielt werden.*

Ziel 3: *Es wurden zusätzliche IP-Cores entwickelt, welche eine Verbindung der Xilinx-Bibliothek mit der neuen GRLIB ermöglichen. Dazu zählen die Brücken IP-Cores AHB2OPB (IP-Corename: „ahb2opb_bridge“) und AHB2LMB (IP-Corename: „ahb2lmb_bridge“). Die AHB2OPB-Bridge bildet eine Verbindung zwischen den Bussen AHB und OPB und die AHB2LMB-Bridge dient als Verbindung der Busse AHB und LMB.*

Ziel 4: *Der Arbeitsplan des EDK wurde an die eingebundenen GRLIB-IP-Cores angepasst. Dadurch ist es mit dieser Arbeit möglich jeden einzelnen EDK-Arbeitsschritt, von der Synthese des SoC bis zum Verschmelzen nativer C-Programme mit dem SoC-Bitstream, an den GRLIB-IP-Cores durchzuführen.*

Ziel 5: *Für ein, mit dem EDK, synthetisiertes GRLIB-SoC, welches den LEON3-Prozessor beinhaltet, konnte eine Linux-Distribution evaluiert und nutzbar gemacht werden. Dieses „SnapGear“-Linux wird mit Kernel-Treibern für alle offiziellen GRLIB-IP-Cores veröffentlicht und kann, dank der „Plug and Play“ Funktionalität des AHB, besonders benutzerfreundlich kompiliert werden.*

Ziel 6: *Zusätzlich wurde ein Vergleich des LEON3-Prozessors mit dem MicroBlaze-Prozessor, hinsichtlich Performance und Ressourcenbedarf, ausgeführt. Der Vergleich wird in Kapitelpunkt 8.3 vorgestellt.*

Folgende Grenzen wurden mit dieser Arbeit erkannt und nicht überschritten:

- *Der Arbeitsplan des EDK konnte nicht perfekt an die GRLIB-IP-Cores angepasst werden. Der Grund ist, dass durch die Firma Xilinx Annahmen und Voraussetzungen festgesetzt wurden, die durch die GRLIB nicht erfüllt werden können. Weicht ein Arbeitsplan des EDK geringfügig vom werkeingestellten Arbeitsplan ab, so können dynamische Entscheidungen nicht mehr getroffen werden. Ein Beispiel bildet der „system.make-Problem“, welches in Kapitelpunkt 5.3.9 erläutert wurde.*
- *Durch die Funktionsweise der GRLIB-IP-Cores konnten einige Arbeitsweisen des EDK nicht angewandt werden. Unter diesen Punkt fällt das „MHS-Reihenfolge-Problem“ so*

wie das „GR Ethernet-Problem“. Außerdem konnte in dieser Arbeit die Funktionalität der „Adresses“-Ansicht der EDK-GUI, für die GRLIB nicht nutzbar gemacht werden.

- *Da ein GRLIB-System mit dem LEON3-Prozessor häufig in Eingebetteten Systemen verwendet wird, wäre es interessant gewesen, in dieser Arbeit, zusätzlich einen Vergleich der Leistungsabnahme eines LEON3-SoC mit eines MicroBlaze-SoC auszuführen.*
  Aufgrund des großen Umfangs dieses Themas wurde der Vergleich nicht durchgeführt.

8.2 Die GRLIB-IP-Cores im EDK

Alle relevanten GRLIB-IP-Cores zur Ersetzung der Xilinx-IP-Cores befinden sich nun in der „Project Repository“-Kategorie der EDK-GUI. Sie können dort ausgewählt und in ein SoC eingefügt werden. Die GRLIB-IP-Cores können benutzt werden wie die IP-Cores der Xilinx-Bibliothek.

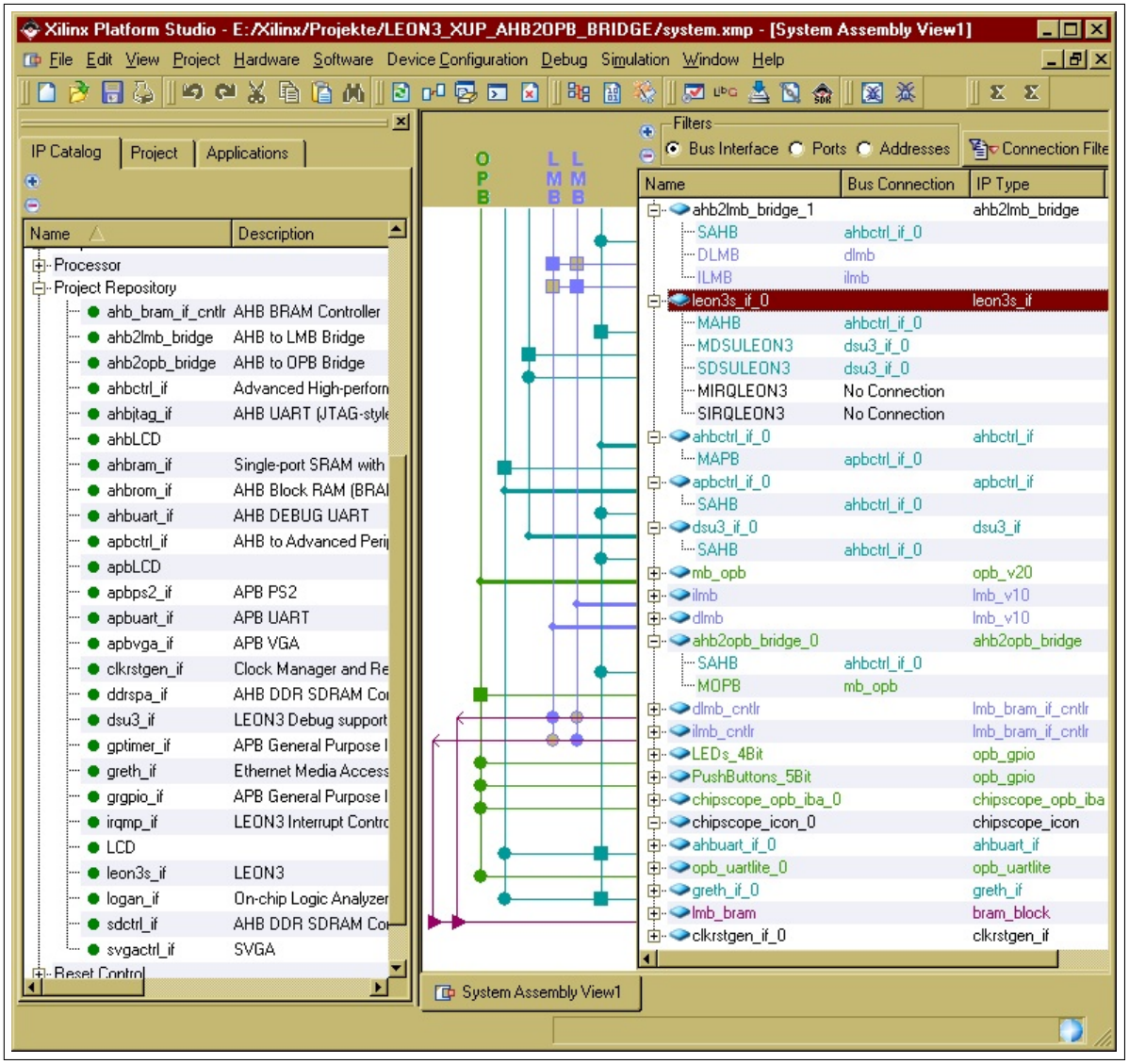

Abbildung 8.2: Die GRLIB-IP-Cores mit einem SoC in der EDK-GUI

In Abbildung 8.2 ist links die „Project Repository“-Kategorie zu erkennen. Dort stehen alle GRLIB-IP-Cores zu Benutzung bereit. Rechts ist die ist ein typischen GRLIB-SoC mit seiner Busstruktur zu erkennen. Mit der gezeigten AHB2LMB-Bridge ist der LEON3-Prozessor an den LMB angeschlossen und mit der AHB2OPB-Bridge werden OPB-IP-Cores, wie „opb_gpio“ oder „opb_uartlite“, mit dem LEON3-Prozessor verbunden.

8.3 Vergleich von MicroBlaze und LEON3

Da der LEON3-Prozessor eine Alternative zum bisher verwendeten MicroBlaze-Prozessor sein soll, wird in diesem Kapitelpunkt ein Vergleich der beiden Prozessoren durchgeführt.
In den vergangenen Kapiteln wurden immer wieder Vergleiche der Funktionsweisen und des Aufbaus der beiden Prozessoren aufgeführt. In diesem Kapitelpunkt soll der jeweilige Prozessor als Einheit aufgefasst werden und ein Vergleich von äußerlich messbaren Größen stattfinden.

Zu diesen Größen gehören:

Performance des Prozessors:

Als weitere Messgröße, zum Vergleich der beiden Prozessoren, soll die Performance mit bekannten Benchmarks gemessen werden.
Da Benchmarks die Leistungsfähigkeit eines Prozessors nur sehr grob bestimmen können, muss eine genaue Wahl des Benchmarks zur Konfiguration des Prozessors stattfinden.

Größe des Prozessors:

Als Größe wird hier die Größe der Prozessor-Entity nach der Synthese aufgefasst. Es soll durch einen Vergleich festgestellt werden, welcher Prozessor wie viele Ressourcen auf einem FPGA verbraucht. Die Größe einer Prozessor-Entity ist dabei die Summe der Größen seiner Unter-Entitys, die wiederum stark von der Funktionalität des Prozessors abhängen.
Von Interesse werden bei den Messungen die benötigten Slices und BRAMs der Prozessor-Entity sein. Alternative Maßeinheiten für die Größe sind zum Beispiel **L**ook-**U**p-**T**able (kurz LUT) oder Gates. In dieser Arbeit wurde sich für die Slices entschieden, da dies eine leicht zugängliche und in der Praxis verbreitete Größe darstellt.

Tabelle 8.1: Messgrößen für LEON3, MicroBlaze 5.00.c Vergleich

8.3.1 Testbedingungen allgemein

Um den MicroBlaze 5.00.c und den LEON3 vergleichen zu können, müssen zuvor Rahmenbedingungen festgelegt werden, welche die Prozessoren vergleichbar machen. Dazu gehören:

- Festlegung der Prozessorkonfiguration

- Welche SoC-Komponenten werden, außer dem Prozessor, am nötigsten gebraucht und in welchen Konfigurationen
- Taktfrequenzen der Systeme
- Aus welchem Speicher bezieht der Prozessor die Testsoftware
- Welche weitere Software wird auf dem Prozessor benutzt (Betriebssystem und Benchmark)

Prozessorkonfiguration

Die Performance und Größen der Prozessoren sind sehr stark konfigurationsabhängig. Aus diesem Grund muss genau überlegt werden mit welchen Konfigurationen die Prozessoren überhaupt vergleichbar sind. Dabei müssen die Konfigurationen so weit wie möglich übereinstimmen, bis auf Funktionalitäten, die durch einen Benutzer nicht beeinflusst werden können.
Alle Konfigurationen, der Prozessoren, können durch bloße Einstellung ihrer Generic-Konstanten, der Prozessor-Entitys, vorgenommen werden. Dazu werden unter anderem auch die Generic-Konstanten verwendet, welche im Anhang B.2 erläutert wurden.

SoC-Komponenten

Da eine Performance nicht messbar ist ohne erkennbare Reaktionen eines Prozessors, die durch den Bus und andere SoC-Komponenten nach außen geleitet werden, müssen geeignete SoC-Komponenten in gleicher Konfiguration entsprechend ergänzt werden.
Bei den Bussen ist zum Beispiel darauf zu achten, dass die Adress- und Datenbreite übereinstimmen. Eine Abweichung hätte Auswirkungen auf die Menge der transferierten Daten pro Takt und würde das Messergebnis verfälschen.
Andere SoC-Komponenten dürfen nur wenn nötig in das SoC eingebunden werden, da sie Auswirkungen auf das Laufzeitverhalten der Prozessorschaltungen haben könnten.

Taktfrequenzen der Systeme

Die Taktfrequenzen der beiden Systeme müssen identisch sein. Zusätzlich muss darauf geachtet werden, dass ein Prozessorsystem nicht mit seiner maximal taktbaren Frequenz betrieben wird, da an solchen Grenzen das Timing instabil sein kann. Ein instabiles Timing könnte zum Beispiel Bus-Transfers erzeugen, die nicht erfolgreich sind und wiederholt werden müssen. Dies würde das Messergebnis verfälschen.
Manche SoC-Komponenten erhöhen die verwendete Haupt-Taktrate um den Faktor 2. Dies kann

mit einem **D**igital **C**lock **M**anager (kurz DCM) bewerkstelligt werden. Jede ähnliche Funktion muss deaktiviert werden.

Programm-Speicher

Beide Prozessoren können ihre Testprogramme aus zwei möglichen Speichertypen lesen. Diese sind die BRAMs oder der SDRAM. Da beide Speichertypen unterschiedliche Latenzzeiten besitzen, bis sie Daten bereitstellen, muss darauf geachtet werden, dass zwei Messergebnisse mit den gleichen Speichertypen gemacht wurden.

Software

Selbstverständlich müssen die beiden Prozessoren mit dem selben Benchmark getestet werden. Allerdings ist schon beim Kompilieren des Benchmarks darauf zu achten welcher Compiler benutzt wurde und mit welchen Optimierungsparametern („-O2“ oder „-O3“).
Manche Benchmarks wurden mit einem zugrunde liegenden Betriebssystem ausgeführt. Da der Einfluss des Betriebssystems schwer zu beurteilen ist, muss es bei jeden Testergebnis angegeben werden.

8.3.2 Testszenario

8.3.2.1 Systemzusammenstellung

Die IP-Core-Zusammenstellungen des MicroBlaze- und LEON3-Systems mussten annähernd gleich sein. Beide Systeme wurden mit 100 MHz betrieben.

MicroBlaze-Prozessor

Das **MicroBlaze** 5.00.c System beinhaltet eine vergleichbare Zusammenstellung. Der Prozessor ist am OPB und LMB angeschlossen.

Am OPB Angeschlossene Komponenten:

- OPB Interrupt-Controller
- DDR SDRAM
- MDM

- zwei OPB GPIO
- OPB Uart Lite
- OPB Timer wird für das „PetaLinux“ benötigt

LEON3-Prozessor

Das GRLIB-System beinhaltet den **LEON3**-Prozessor als zu testende SoC-Komponente. Der Prozessor ist an den AHB angeschlossen.

Am AHB Angeschlossene Komponenten:

- APB („apbctrl_if“)
- Interrupt-Controller („irqmp_if“)
- GR Ethernet („greth_if“) nur für Debugging Zwecke
- DDR SDRAM („ddrspa_if“)
- DSU („dsu3_if“)
- Um eventuell Programme aus den BRAMs zu lesen wurde die AHB2LMB-Bridge („ahb2lmb_bridge“) zusätzlich an den AHB angeschlossen. Sie wurde gepipt eingesetzt („pipe = 1“)

Am APB angeschlossene Komponenten:

- zwei GRGPIO („grgpio_if“)
- APB Uart („apbuart_if“)
- Timer („gptimer_if“) wird für das SnapGear-Linux benötigt

Die Systeme wurden beide mit dem Xilinx-EDK synthetisiert. Es wurde die Syntheseoption „speed“ mit einem Level von 1 gewählt.

8.3.2.2 Systemkonfiguration

Die Systemkonfigurationen betreffen die Festlegungen der Generic-Konstanten der einzelnen IP-Cores.

MicroBlaze-Prozessor

Damit der MicroBlaze-Prozessor mit dem LEON3 vergleichbar ist musste die FPU sowie der Barrel-Shifter deaktiviert werden. Der Hardware Multiplizierer sowie der Dividierer sind mit der V8-Hardware des LEON3 vergleichbar und wurden gleichberechtigt behandelt.
Der Programm- und der Daten-Cache wurde, wie beim LEON3, mit einer Größe von jeweils 8 kByte gewählt und mit einer Cache-Line-Größe von 32 Byte.

LEON3-Prozessor

Der Programm- und Daten-Cache des LEON3 wurde jeweils auf 8 kByte eingestellt. Eine Cache-Linie hat jeweils eine Größe von 32 Byte. Die Überschreibungsstrategie aller Cache-Speicher ist LRR. Bei einer Umstellung auf LRU konnte eine Performance-Verlust von 2% gemessen werden.
Damit der LEON3 mit dem MicroBlaze vergleichbar ist muss die MMU deaktiviert sein. Sollte sie für spätere Testergebnisse aktiviert werden, so besitzt sie einen separaten Instruktion- und Daten-TLB mit jeweils 8 Einträgen und der Überschreibungsstrategie LRU.
Wenn der V8 Multiplizierer und der Dividierer des LEON3-Prozessors aktiviert wurden, so geschah dies in ihrer gepipten Version („v8 = 2“).

Andere Komponenten

Die Uart IP-Cores des LEON3 und des MicroBlaze-Systems wurden mit jeweils 38400 Bauds/s betrieben. Die DDR SDRAM Komponenten der beiden Systeme wurden jeweils mit 100 MHz betrieben. Dies entspricht der Main-Clock Frequenz.
Weiter musste darauf geachtet werden, dass der „Burst Support“ des OPB DDR SDRAM für das MicroBlaze-System aktiviert war, da standardmäßig vom LEON3 Burst-Transfers auf dem AHB DDR SDRAM ausgeführt werden.
Um den OPB mit dem AHB vergleichbar zu machen, musste die Option „Use Only LUTs for OR Structure“ des OPB aktiviert werden.

Variationen der Systemkonfiguration

Um mehrere Messergebnisse für beide Systeme zu erhalten wurden folgende Variationen der Konfiguration durchgeführt:

- Es wurde der Cache gleichzeitig bei beiden Systemen aktiviert oder deaktiviert

- Es wurden die Hardware Multiplizierer und Dividierer gleichzeitig bei beiden Systemen aktivier oder deaktiviert.

Für das LEON3-System bedeutet das eine Variation der Generic-Konstanten von „v8 = 0“ oder „v8 = 2“. Für das MicroBlaze-System bedeutet das eine Variation der Generic-Konstanten von „C_USE_HW_MUL/C_USE_DIV = 0“ oder „C_USE_HW_MUL/C_USE_DIV = 1“.

8.3.2.3 Benchmarks

Um alle Testergebnisse in dieser Arbeit zu erhalten wurde der Benchmark „Dhrystone 2.1“ verwendet.

Dhrystone 2.1

Der Dhrystone-Benchmark misst die Integer-Performance des Prozessors. Seine Testoperationen bestehen hauptsächlich aus Integer-Operationen und Operationen auf Zeichenketten.
Die Messergebnisse der Performance werden in **D**hrystone **I**terationen **p**ro **S**ekunde (kurz DIPS) angegeben. Bei Start des Benchmarks muss eine Anzahl von Iterationen eingegeben werden, die der Benchmark insgesamt durchlaufen soll. In dieser Arbeit sind es immer 400000 Iterationen gewesen.
Es wird die benötigte Zeit für die 400000 Iterationen gemessen und in den Wert Dhrystone Iterationen pro Sekunde (auch genannt Dhrystones pro Sekunde) umgerechnet. Der Dhrystone 2.1 Benchmark wurde in dieser Arbeit ohne „register“ Attribut kompiliert.

8.3.3 Testergebnisse des Dhrystone 2.1 Benchmark

Der Dhrystone-Benchmark wurde für jedes System mit einem Betriebssystem als Grundlage ausgeführt. Für den LEON3-Prozessor war dies das „SnapGear“-Linux mit dem Kernel 2.0.x und für den MicroBlaze-Prozessor war dies das „PetaLinux“ mit dem Kernel 2.6.x.
Das SnapGear-Linux wurde mit dem Sparc-linux-gcc 3.2.2 und Optimierung „-O2“ und das PetaLinux mit dem gcc 3.4.1 und Optimierung „-O2“ kompiliert.

Für den LEON3-Prozessor konnte nicht die Kernelversion 2.6.x verwendet werden, da diese eine MMU benötigt. Die MMU des LEON3 wurde für diesen Test deaktiviert.

8.3.3.1 MicroBlaze 5.00.c Prozessor

DIPS (Kernel 2.6.x)	Hardware MUL/DIV aktiviert	Hardware MUL/DIV deaktiviert
mit Cache	86767,9	67911,7
ohne Cache	5711,0	4653,9

Tabelle 8.2: DIPS des MicroBlaze 5.00.c Prozessors mit „PetaLinux“ Kernel 2.6.x

Tabelle 8.2 zeigt eine Übersicht der Dhrystone Ergebnisse des MicroBlaze-Prozessors in verschiedenen Konfigurationen.

Slices	Hardware MUL/DIV aktiviert	Hardware MUL/DIV deaktiviert
mit Cache	1491	1413
ohne Cache	1289	1224
BRAMs		
mit Cache	10	10
ohne Cache	0	0

Tabelle 8.3: Ressourcenbedarf des MicroBlaze 5.00.c Prozessors

Tabelle 8.3 zeigt eine Übersicht der verbrauchten Ressourcen des MicroBlaze-Prozessors in verschiedenen Konfigurationen.

8.3.3.2 LEON3-Prozessor

DIPS (Kernel 2.0.x)	V8 MUL/DIV aktiviert	V8 MUL/DIV deaktiviert
mit Cache	86580,1	57306,6
ohne Cache	10325,2	10998,1

Tabelle 8.4: DIPS des LEON3-Prozessors mit SnapGear-Linux-Kernel 2.0.x (ohne MMU)

Tabelle 8.4 zeigt eine Übersicht der Dhrystone-Ergebnisse des LEON3-Prozessors in verschiedenen Konfigurationen.

Slices	V8 MUL/DIV aktiviert	V8 MUL/DIV deaktiviert
mit Cache	3083	2688
ohne Cache	2725	2291
BRAMs		
mit Cache	12	12
ohne Cache	2	2

Tabelle 8.5: Ressourcenbedarf des LEON3-Prozessors (ohne MMU)

Tabelle 8.5 zeigt eine Übersicht der verbrauchten Ressourcen des LEON3-Prozessors in verschiedenen Konfigurationen.

8.3.4 Auswertung der Testergebnisse

Es folgt eine Auswertung der Testergebnisse. Es werden die Performance- und Größen-Ergebnisse der Prozessoren und die Größen-Ergebnisse der gesamten SoCs vorgestellt.

8.3.4.1 Performance und Größen-Ergebnisse von MicroBlaze und LEON3

Alle Messwerte, zur Erstellung der Diagramm-Abbildungen, wurden dem Kapitelpunkt 8.3.3 entnommen.

Performance

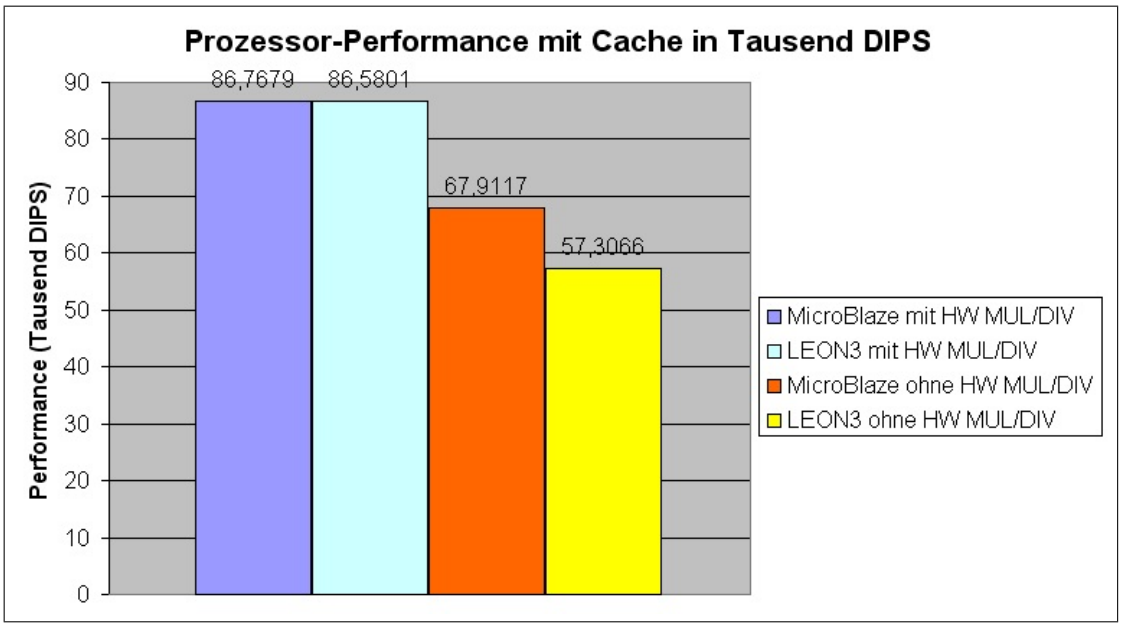

Abbildung 8.3: Performance von MicroBlaze und LEON3 jeweils mit Cache

In der Abbildung 8.3 ist zu erkennen, dass die Performance des LEON3-Prozessors immer unterhalb der Performance des MicroBlaze-Prozessors liegt. Mit zusätzliche Logik zur Integer-Multiplikation und Integer-Division sind die Performance-Ergebnisse der beiden Prozessoren fast gleich und ohne die zusätzliche Logik beträgt die Performance des LEON3 84% von der des MicroBlaze-Prozessors.

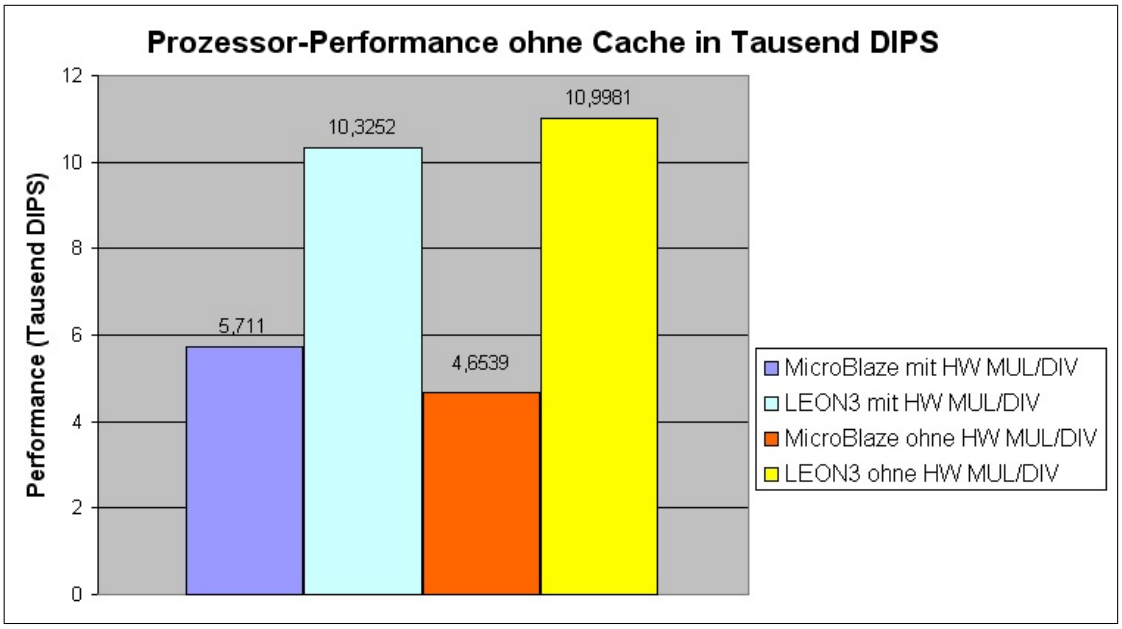

Abbildung 8.4: Performance von MicroBlaze und LEON3 jeweils ohne Cache

In der Abbildung 8.4 sind die Performances der Prozessoren ohne Cache-Speicher dargestellt.

Wenn beide Prozessoren ohne Cache-Speicher synthetisiert wurden, so ist die Performance des LEON3 größer als die Performance des MicroBlaze. Der MicroBlaze schafft rund 50% weniger DIPS als der LEON3-Prozessor. Wird ohne Cache-Speicher die zusätzliche Integer-Logik deaktiviert, so fällt die Performance des MicroBlaze um rund 19%. Die Performance des LEON3 bleibt dagegen, im Rahmen der Messungenauigkeit, konstant.

Größen-Ergebnisse

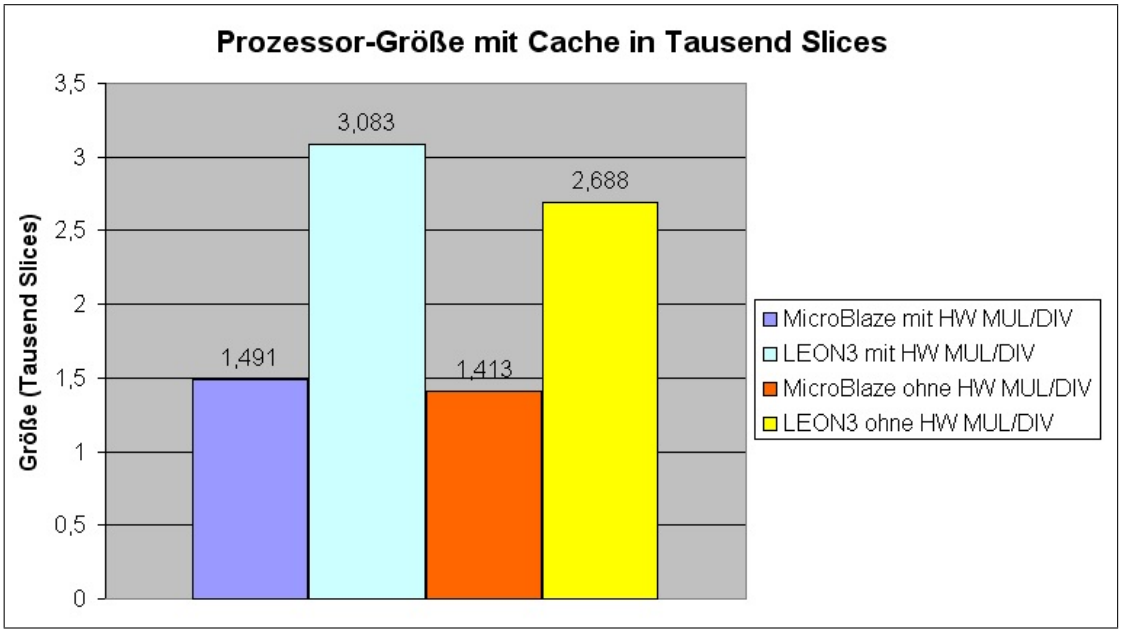

Abbildung 8.5: Größe von MicroBlaze und LEON3 jeweils mit Cache

Aus Abbildung 8.5 ist zu erkennen, dass der LEON3-Prozessor rund doppelt soviel Slice-Ressourcen benötigt, wie der MicroBlaze-Prozessor. Weiter ist zu erkennen, dass die Integer-Logik des MicroBlaze nur 78 Slices der benötigten gesamt Ressourcen ausmacht. Dies entspricht rund 5%. Beim LEON3 werden für die Integer-Logik rund 400 Slices benötigt. Dies entspricht rund 13% der Gesamtgröße.

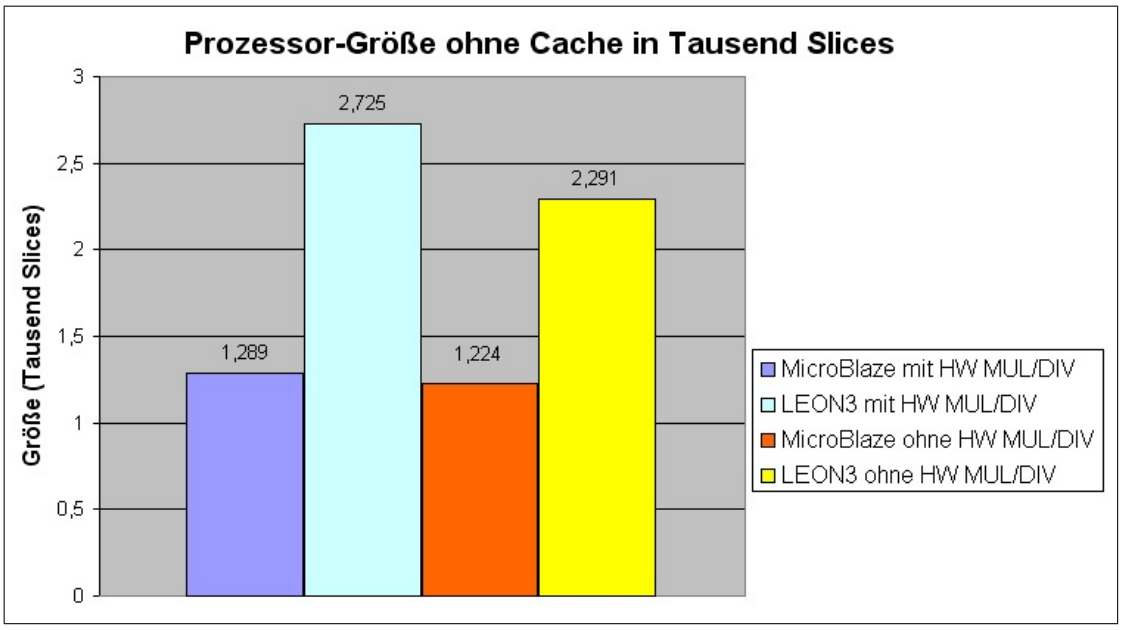

Abbildung 8.6: Größe von MicroBlaze und LEON3 jeweils ohne Cache

Im Vergleich zur Abbildung 8.5, können aus Abbildung 8.6 die benötigten Slice-Ressourcen für die Cache-Logik eines Prozessors ermittelt werden. Beim MicroBlaze werden im Mittel 195 Slices für 8 kByte Cache benötigt. Beim LEON3 werden dagegen im Mittel 370 Slices benötigt. Dies entspricht rund doppelt soviel Ressourcen wie beim MicroBlaze. Bei beiden Prozessoren wurden jeweils 8 kByte Programm- und 8 kByte Daten-Cache verwendet.

8.3.4.2 Größen-Ergebnisse von MicroBlaze-SoC und LEON3-SoC

Die SoCs der Testszenarien aus Kapitelpunk 8.3 wurden durch das EDK synthetisiert, platziert und verdrahtet. Dabei wurde, wie bereits erwähnt, die Optimierung auf „Speed“ angewendet. Es folgt nun eine Darstellung der jeweiligen Größe vom MircoBlaze-SoC und vom LEON3-SoC, gefolgt von spezifischen Daten.

Um die folgenden Größen-Ergebnisse eines LEON3-SoC mit einem MicroBlaze-SoC vergleichbar zu machen ist eines zu beachten. Ein LEON3-SoC kann nur „schnell“ debuggt werden, wenn es den GR Ethernet IP-Core beinhaltet, so ist es auch bei den Testszenarien. Bei einem MicroBlaze-SoC ist dies allerdings nicht notwendig, da dieses über JTAG mit einer „BSCAN“-Komponente debuggt werden kann.
Aus diesem Grund wurde der GR Ethernet IP-Core aus dem LEON3-SoC entfernt. Das LEON3-Testszenario, aus Kapitelpunkt 8.3, ist also nur um die GR Ethernet IP-Core benötigten Ressourcen größer als das hier ausgeführte LEON3-SoC.

Die benötigten Ressourcen des GR Ethernet IP-Cores sind[1]:

Number of Slices:	1789	out of	13696	13%
Number of BRAMs:	6	out of	136	4%

[1] Prozentangaben von Ressourcen sind auf den „Virtex-II Pro 30 FF896“ bezogen

Es ergeben sich folgende Größen-Ergebnisse:

MicroBlaze-SoC:

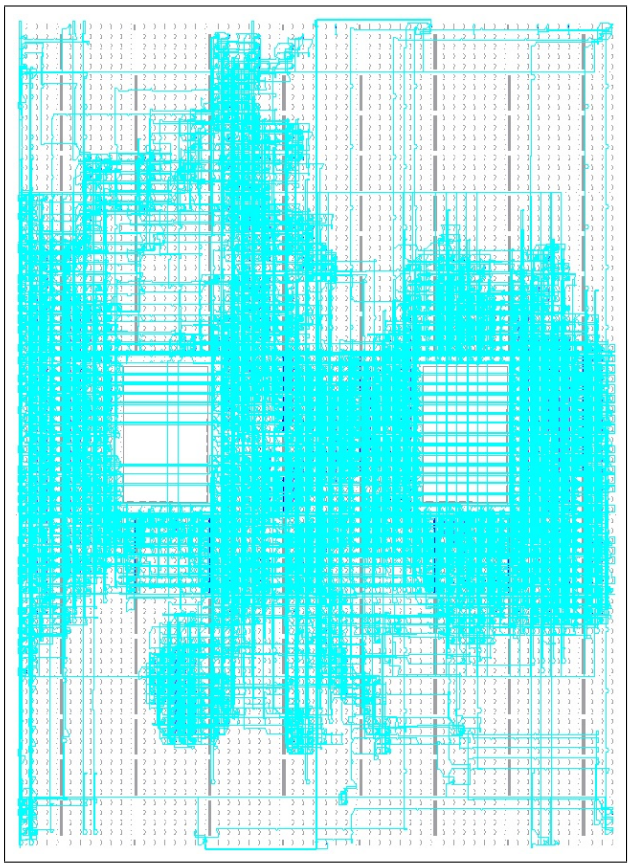

Abbildung 8.7: FPGA-Editor-„chip graphics“ des MicroBlaze-SoC

Number of DCMs:	2	out of	8	25%
Number of RAMB16s:	14	out of	136	10%
Number of SLICEs:	3815	out of	13696	27%

SLICEs microblaze:	1491
SLICEs opb_v20:	121
SLICEs opb_intc:	91
SLICEs opb_dmd:	67
SLICEs opb_gpio:	28
SLICEs opb_uartlite:	51
SLICEs mch_opb_ddr:	1867
SLICEs opb_timer:	269

„opb_dmd“, „opb_uartlite“ sind eigentlich nicht mit „dsu3_if“, „apbuart_if“ vergleichbar, da ihre Funktionalitäten weniger umfangreich sind.

LEON3-SoC:

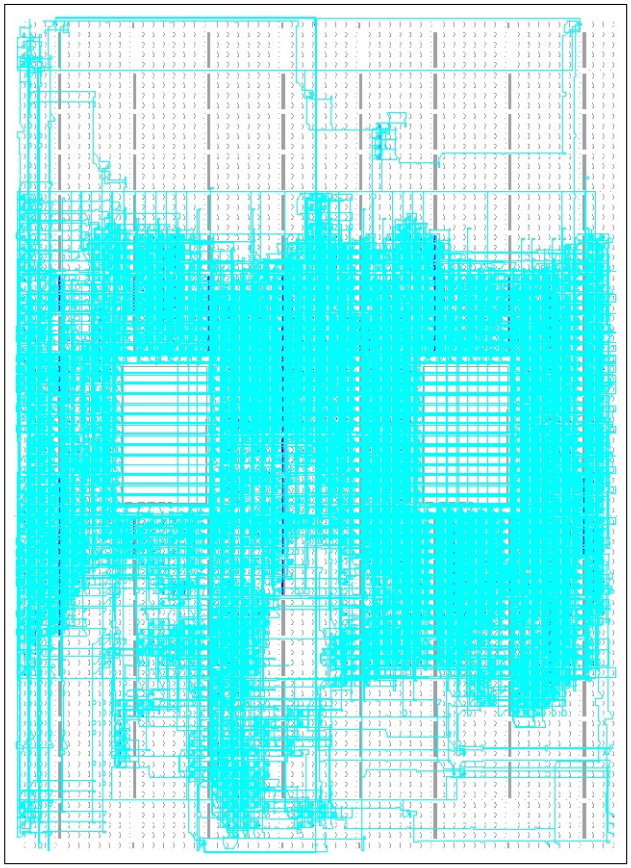

Abbildung 8.8: FPGA-Editor-„chip graphics“ des LEON3-SoC

Number of DCMs:	3	out of	8	37%
Number of RAMB16s:	26	out of	136	19%
Number of SLICEs:	4855	out of	13696	35%

SLICES leon3_if:	3083
SLICES ahbctrl_if:	691
SLICES apbctrl_if:	261
SLICES irqmp_if:	153
SLICES dsu3_if:	458
SLICES gpio_if:	13
SLICES apbuart_if:	211
SLICES ddrspa_if:	501
SLICES gptimer_if:	508

Für ein LEON3-SoC mit Debugging über den GR Ethernet IP-Core werden zusätzlich 1789 Slices benötigt.

Aus der obigen Aufschlüsselung ist zu erkennen, dass das LEON3-Prozessor rund doppelt so viele Slice-Ressourcen benötigt, wie der MicroBlaze-Prozessor. Weiter fällt auf, dass der cachebare „OPB DDR RAM“-IP-Core rund das 3,7-fache an Slice-Ressourcen mehr benötigt, als der vergleichbare „GR DDR RAM“-IP-Core (ddrspa_if). Dies ist bemerkenswert, da der „GR DDR RAM“-IP-Core eigentlich viel flexibler konfigurierbar ist und sogar seine Clock-Versorgung mit zwei DCMs selbst organisiert.

8.4 Maximal-Performance vom MicroBlaze und LEON3

Viele SoC-Entwickler interessiert in der Praxis weniger der Vergleich verschiedener Prozessoren unter gleichen Voraussetzungen. Es kann auch gewünscht sein, den leistungsstärksten Prozessor zu finden, der mit all seinen Optimierungen und Ausreizungen existiert. Aus diesem Grund, sollen die Prozessoren, MicroBlaze und LEON3, in diesem Kapitelpunkt mit ihrer maximal möglichen Performance vorgestellt werden. Es muss dazu gesagt werden, dass der hier aufgeführte Vergleich extrem davon abhängig ist, unter welchen Bedingungen er stattfindet. Zu diesen Bedingungen gehören beispielsweise:

- verwendeter FPGA
- Spannungsversorgung
- Betriebstemperatur
- Syntheseoptionen

verwendeter FPGA

Alle FPGAs haben unterschiedliche Größen und Verteilungen ihrer konfigurierbaren Schaltkreise. Hat ein FPGA stark begrenzte Ressourcen, so sind die XST bei einem größeren SoC gezwungen längere Zeit in die Platzierungsoperation (EDK-Arbeitsschritt 2) zu investieren, um eine erfolgreiche Platzierung auf dem FPGA zu ermöglichen. Dies hat Einfluss auf die maximale Taktbarkeit des SoC.

Für die Tests in diesem Kapitelpunkt wurde das XUP-Board mit dem „Virtex-II Pro 30 FF896, Speed Grade: -7“ FPGA verwendet.

Spannungsversorgung

Da die SoCs am Rande ihrer Leistungsfähigkeit arbeiten, werden besonders stabile und saubere Taktpegelverläufe benötigt. Werden diese nicht geliefert, so kann das SoC abstürzen. Die Erhöhung der Betriebsspannung kann in solchen Grenzfällen einen stabileren Taktpegelverlauf erzeugen und so die maximale Taktbarkeit erhöhen. Eine Einstellung der Betriebsspannung für bestimmte Taktsignale könnte in der UCF eines SoC, durch die Wahl geeigneter PADs, vorgenommen werden.

In dieser Arbeit wurden keine Änderungen der Betriebspannungen vorgenommen. Alle Betriebsspannungen wurden so belassen, wie sie durch die UCFs von Xilinx oder von GR vorgegeben wurden.

Betriebstemperatur

Die Betriebstemperatur eines FPGA hat Einfluss auf die maximale Taktbarkeit seiner konfigurierten Schaltungen. In der Regel gilt, dass bei niedrigerer Betriebstemperatur eine höhere Taktrate erzielt werden kann.

Die Betriebstemperatur wurde in dieser Arbeit nicht berücksichtigt. Es sei allerdings erwähnt, dass spätestens 20 Sekunden nach Konfiguration eines SoC auf dem FPGA, die Performance gemessen wurde. Durch diese geringe Zeitspanne hat sich der FPGA nur relativ geringfügig erwärmt.

Syntheseoptionen

Die Syntheseoptionen der XST haben Einfluss auf die maximale Taktbarkeit eines SoC. Die XST können beispielsweise auf minimal benötigte Fläche oder maximale Geschwindigkeit eines SoC optimieren. In dieser Arbeit wurden immer folgende ausschlaggebende Syntheseoptionen bewählt:

```
-opt_mode speed
-opt_level 1
```

8.4.1 Testergebnisse

Vorgehen zur Leistungssteigerung

Der Ausgangspunkt dieser Tests war das jeweilige Testszenario, für den MicroBlaze oder LEON3, aus Kapitelpunkt 8.3. Von diesen aus wurde in erster Linie die Taktfrequenz des jeweiligen SoC maximal erhöht. Wurde die Taktfrequenz zu hoch, so reagiert das SoC nicht mehr. Ausgehend von

diesem lokalen Maximum wurde die Größe des Cache-Speichers variiert, um zusätzliche Leistung aus dem SoC zu erhalten.
Weiter wurde versucht die Taktfrequenz des DDR-Speichers sowie der zugehörigen Schnittstellenkomponente zu erhöhen, um eine Performance-Steigerung zu erreichen.

LEON3-Prozessor

Die MMU des LEON3 war für die Messungen immer aktiviert. Eine Deaktivierung der MMU brachte, unter dem Linux-Kernel 2.0.x, keine Performance-Änderung. Für den LEON3 zeigten sich 8 kByte Cache (für Programm und Daten) als optimal. Mit 4 kByte waren Leistungseinbußen festzustellen und mit 12 oder 16 kByte war die maximale Taktfrequenz zu niedrig.

Eine maximale Performance ergab sich für das LEON3-SoC bei:

- AHB-Frequenz von 110 MHz
- V8 Hardware Multiplizierer und Dividierer aktiviert
- jeweils 8 kByte Cache mit Ersetzungsstrategie LRR
- DDR-RAM-Taktfrequenz von 125 MHz

Die maximale Performance wurde mit dem Linux-Kernel 2.0.x mit 94339.6 DIPS gemessen.

DIPS	**Linux-Kernel 2.6**	**Linux-Kernel 2.0.x**
LEON3 (maximale Performance)	87912,1	**94339,6**
DDR-RAM-Taktfrequenz: 100 MHz	88105,7	90909,1
Cache-Ersetzungsstrategie: LRU und DDR-RAM-Taktfrequenz: 100 MHz	86393,1	89285,7
jeweilige Cache-Größe: 4 kByte und DDR-RAM-Taktfrequenz: 100 MHz	77669,9	89887,6

Tabelle 8.6: LEON3-Maximal-Performance mit Variationen abweichend von der Maximal-Performance-Konfiguration

Die Maximal-Performance-Konfiguration des LEON3-SoC benötigte 9051 Slices auf dem FPGA. Davon wurden 4347 Slices vom LEON3-Prozessor benötigt. Für den Fall der maximalen Performance ergibt sich für den LEON3-Prozessor ein theoretische Proportionalitätsfaktor von 857,6 DIPS/MHz.

MicroBlaze-Prozessor

Eine MMU existiert für den MicroBlaze-Prozessor nicht. Die maximale Performance wurde beim MicroBlaze mit jeweils 8 kByte Cache Programm- und Daten-Cache festgestellt. Eine Erhöhung des Cache-Speichers brachte keinen weiteren Performance-Gewinn. Der Dhrystone-Benchmark wurde wieder unter dem „PetaLinux“-Kernel 2.6.x ausgeführt.

Eine maximale Performance ergab sich für das MicroBlaze-SoC mit:

- OPB-Frequenz von 100 MHz
- Hardware Multiplizierer und Dividierer aktiviert
- jeweils 8 kByte Cache
- DDR-RAM-Taktfrequenz von 100 MHz
- OPB-DDR-RAM Burst-Support aktiviert

Vor Angabe der Performance-Ergebnisse sei folgendes erwähnt. Es existierten in dieser Arbeit zwei MicroBlaze-SoCs. Eins mit 100 MHz, für einen fairen Vergleich mit dem LEON3-SoC (siehe Kapitelpunkt 8.3.2) und mit 50 MHz zur Ermittelung des Frequenzeinflusses auf die Performance des Prozessors. Beide SoCs wurden mit dem „Base System Builder wizard“ erstellt. Die Einstellung der OPB-Frequenz wurde mit der „Processor-Bus clock frequency“-Angabe durchgeführt. Diese Einstellung hat Auswirkung auf die Clock-Frequenz des OPB. Der MicroBlaze-Prozessor selbst benutzt wiederum die Clock-Frequenz des OPB für seine eigene Clock-Versorgung.

DIPS	**mit 100 MHz**	**mit 50 MHz**
MicroBlaze (maximale Performance)	86767,9	86580,1

Tabelle 8.7: MicroBlaze-Maximal-Performance im Vergleich zum 50 MHz Testszenario

Für den Fall der maximalen Performance ergibt sich für den MicroBlaze-Prozessor ein theoretische Proportionalitätsfaktor von 865,801DIPS/MHz.

Auswertung

Aus Tabelle 8.7 ist zu erkennen, dass sich die Performance nach einer Halbierung der OPB-Frequenz praktisch nicht gesenkt hat. Dieser Sachverhalt kann mit den EDK-Projekten „Testszenarion_microblaze_mit_cache_100MHz“ und „Testszenarion_microblaze_mit_cache_50MHz“, welche sich auf der DVD zu dieser Arbeit befinden, nachvollzogen werden (siehe Anhang D).

In dieser Arbeit konnte nicht festgestellt werden, warum keine Performance-Senkung zu erkennen war, obwohl sich die OPB-Frequenz halbiert hat.

Insgesamt ist festzustellen, das die maximal mögliche Performance des LEON3-Prozessors, mit 94339.6 DIPS, höher liegt als die maximal mögliche Performance des MicroBlaze-Prozessors, mit 86767.9 DIPS. Die maximale Performance des LEON3-Prozessors ist rund 9% höher als die des MicroBlaze-Prozessors.

Kapitel 9

Ausblick

In diesem Kapitel sollen Weiterentwicklungen der vorliegenden Arbeit angesprochen werden. Diese Weiterentwicklungen sind allgemein gefasst und müssen nicht auf die Vorgehensweise dieser Arbeit angepasst sein.
In dieser Arbeit wurden die IP-Cores der GRLIB in das EDK eingebunden. Durch die Lösung des „Interface-Entity-Problems“ konnte das Einbinden reibungslos erfolgen. Lediglich die Synthese der IP-Cores mit dem EDK ist, durch die relativ unflexible Oberfläche des EDK sowie des EDK-Arbeitsplans, mit bleibenden Problemen behaftet. Es ist zukünftig denkbar, dass die eingebundenen GRLIB-IP-Cores aus ihrer momentanen EDK-Umgebung extrahiert werden, um in einer weitaus flexibleren Umbebung synthetisiert zu werden. In dieser neuen Umgebung könnten zudem Probleme mit der GRLIB speziell gelöst werden.
In der Tat existiert bereit eine solche Umgebung. In der Arbeit „**System Specification of Embedded Systems in Java for Synthesis**“ an der Universität-Heidelberg wurde eine Art „Java-GUI“ entwickelt, welche auf den XST aufsetzt. Mit dieser Java-GUI können die herkömmlichen EDK-Konstrukte (wie MPD, PAO oder Tcl) gelesen und ausgewertet werden. Es ist also möglich in der Java-GUI die GRLIB-IP-Cores, mit ihren EDK-Konstrukten, einzubinden und mit einem flexiblen Arbeitsplan deren Synthese zu verbessern. Mit der Java-GUI könnte der in dieser Arbeit aufgetauchte „system.make-Problem“ vollständig behoben werden. Ebenfalls die Lösungen der Probleme „GR-Ethernet-Problem“ und „FSM-Problem“ könnten mit dieser flexiblen Oberfläche problemlos und benutzerfreundlich umgesetzt werden. Speziell das „MHS-Reihenfolge-Problem“ der GRLIB-IP-Cores kann mit der Java-GUI von vorneherein ausgeschlossen werden. Dazu müsste die Java-GUI bei der Erstellung der System-Entity lediglich darauf achten, dass die IP-Cores innerhalb der MHS nicht in der Reihenfolge ihres Vorkommens eingefügt werden, sondern in der Reihenfolge ihres Bus-Indizes (hindex, pindex).

Update der GRLIB-IP-Cores

Die eingebundenen IP-Cores der GRLIB unterliegen, wie die GRLIB selbst, Neuerungen und Verbesserungen. In dieser Arbeit wurden die IP-Cores der GRLIB-Version 1.0.14-b2028 in das EDK eingebunden. Allerdings werden ständig Bugs behoben und neue Funktionalitäten der GRLIB hinzugefügt. Erscheint eine neue Version der GRLIB, so könnte diese unkompliziert in das EDK übertragen werden. Es müssten lediglich die, in dieser Arbeit entstandenen, Bibliothek-IP-Cores aktualisiert werden. Da die Bibliotheken der GRLIB in dieser Arbeit in die Bibliothek-IP-Cores übernommen und nicht in alle IP-Cores verteilt wurden, kann eine Aktualisierung durch Überschreiben problemlos stattfinden.
Sollten die Änderungen einer neuen GRLIB-Version jedoch darin bestehen, das zusätzliche Ports den IP-Cores hinzugefügt wurden, so hätte dies Auswirkungen auf die Interface-Entitys und die MPD-Konstrukte der GRLIB-IP-Cores im EDK. In diesem Fall könnte sich eine zukünftige Automatisierung als nützlich erweisen.

Ausbauen der AHB2OPB-Bridge

Eine weitere Verbesserungsmöglichkeit dieser Arbeit, wäre die Aufbereitung der AHB2OPB-Bridge. Da die besagte Bridge noch nicht mit einem vollwertigem Burst-Modus implementiert wurde, könnte dies bei Bedarf geschehen. Es könnte ebenfalls hilfreich sein, wenn die AHB2OPB-Bridge zusätzliche „Plug and Play“-Informationen weiterleiten würde, da diese in den AHB übertragen werden könnten und so die Treiberprogrammierung erleichtern.

Echtzeit durch ein LEON3-SoC

Ein LEON3-SoC mit dem SnapGear-Linux ist nicht automatisch echtzeitfähig. Diese Fähigkeit ist allerdings für die Verwendung eines LEON3-SoC als eingebettetes System wünschenswert. Zur Echtzeitfähigkeit eines LEON3-SoC müsste ein „nicht-Linux“-Echtzeit-Betriebssystem auf dem LEON3-SoC ausgeführt werden, oder das SnapGear-Linux zur Echtzeitfähigkeit erweitert werden. Da das SnapGear-Linux Open-Source und sehr konfigurierbar ist, würde sich die zweite Möglichkeit, der Echtzeit-Erweiterung des SnapGear-Linux, anbieten.
Zur Ausführung dieser Erweiterung könnte das, im Anhang C.2, vorgestellte **R**eal **T**ime **A**pplication **I**nterface (kurz RTAI) genutzt werden.

Anhang A

Anhang zur GRLIB

A.1 AMBA Advanced Peripheral Bus (APB)

Dieser Kapitelpunkt behandelt die spezielle GRLIB-Umsetztung des APB, aus der AMBA 2.0 Spezifikation. Der APB kann bis zu einem Master und 16 Slaves verbinden. Dies ist ein Unterschied zum OPB. Der OPB kann bis zum 16 Masters anschließen. Da aber im LEON3-System zwischen hochperformanten Prozessor-Bus (AHB) und Peripherie-Bus (APB) unterschieden wird, benötigt der APB nur einen Master. Der APB wird an einem AHB über eine AHB2APB Bridge angeschlossen.

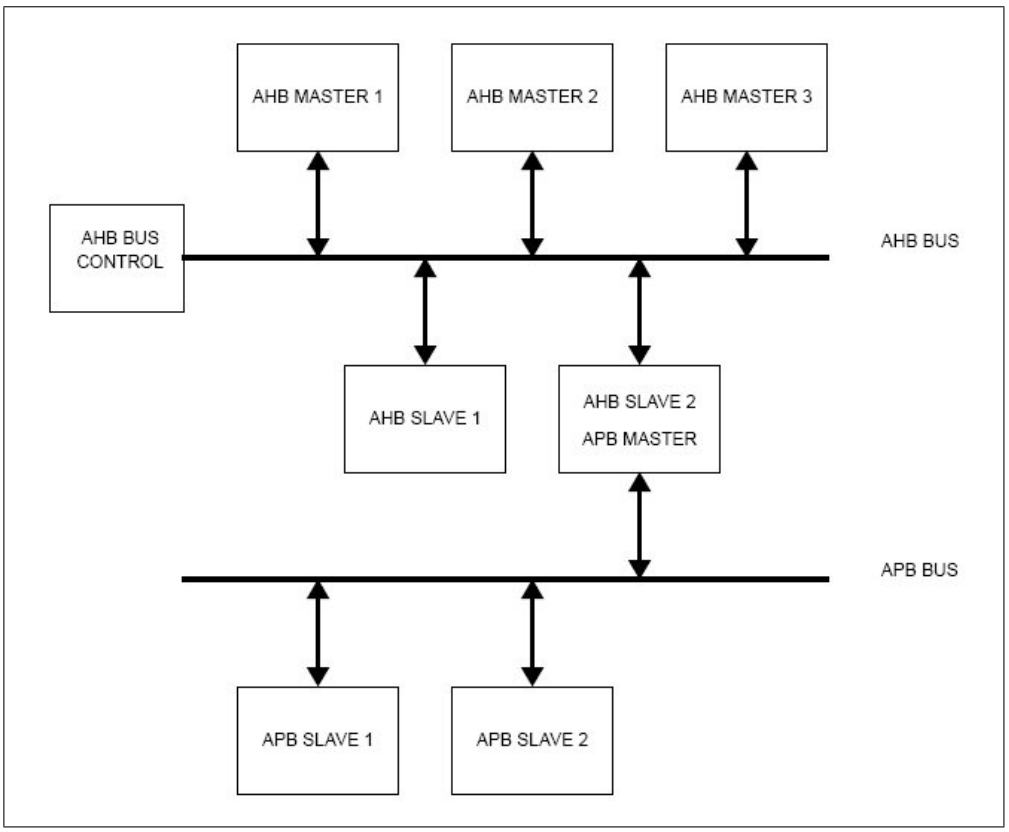

Abbildung A.1: Schema eines AHB und APB Systems [grlibdoc]

Die Abbildung A.1 zeigt ein komplettes AHB und APB System. Es können auch mehrere APB an einem AHB angeschlossen werden, dazu werden lediglich mehrere Bridges benötigt.

Eine Bridge existiert in der GRLIB nicht als einzelner IP-Core. Die AHB2APB Bridge wurde mit dem APB verschmolzen, dadurch wurde das überflüssige APB-Master-Interface auf dem APB eingespart. Daraus entsteht eine neue Art von Bus-Entity. Der neue APB ist ein Bus und gleichzeitig ein Slave auf dem AHB. Die neue APB-Entity ist in der Datei „apbctrl.vhd“ gespeichert.
Da die verschmolzene Bridge bereits ein Master auf den APB darstellt, ist es nicht möglich einen weiteren APB-Master anzuschließen.

Einen besseren Blick auf den APB wird durch die Abbildung A.2 gegeben:

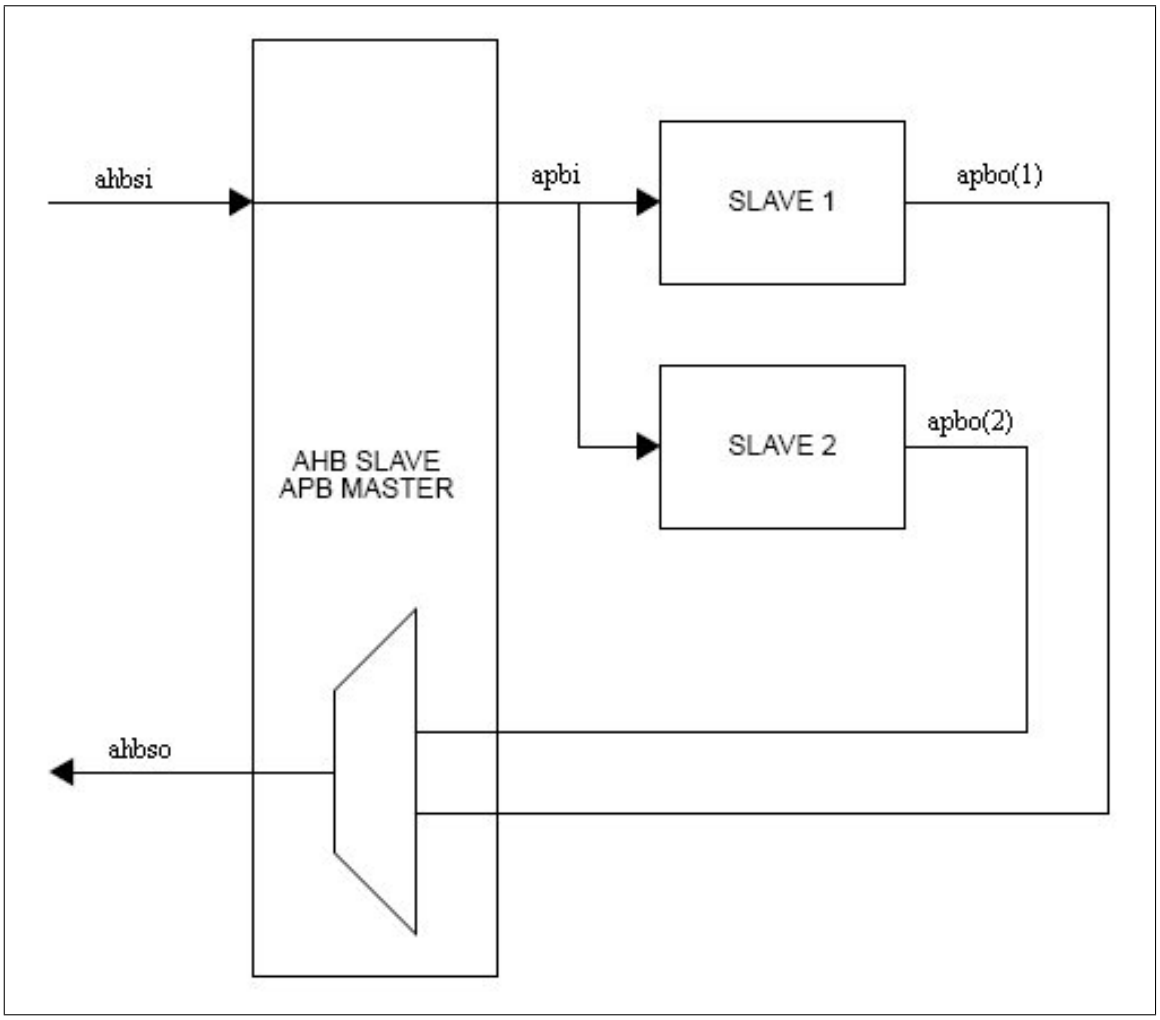

Abbildung A.2: Verbindungen des APB [grlibdoc]

Auf der linken Seite, der Abbildung A.2, sind die normalen Signalmengen „ahbso“ und „ahbsi“ eines AHB-Slave zu erkennen. Auf der rechten Seite wird eine „apbi“-Signalmenge zu allen APB-Slaves getrieben und die Ausgänge aller APB-Slaves („apbo“) münden einzeln in den APB.
Der APB-Arbiter dient dann als Multiplexer und leitet nur eine „apbo“-Signalmenge auf den AHB-Slave Ausgang des APB. Es wird also der anfordernde AHB-Master mit einem bestimmten APB-Slave verbunden.
Jede SoC-Komponente, die als Slave an den APB angeschlossen werden soll, muss eine bestimmte Menge von Signalen treiben oder lesen, d.h. die SoC-Komponente muss bestimmte Ports besitzen.

A.1.1 Das APB-Slave-Interface

Eine APB Slave Entity muss die Signalmenge „apb“ als Ports einbinden. Die „apb“-Signalmenge besteht aus der Vereinigung der Ausgangsportmenge „apbo“ und der Eingangsportmenge „apbi“. Dabei wird nur eine Signalmenge „apbi“ aus dem Bus an alle APB-Slaves getrieben.
Die Ausgangsports aller Slaves („apbo“) werden zu einem Signalvektor („ahbo(*)“) zusammengefasst. Dieser Vektor ist dann eine Signalmenge, die den Eingangsport des APB treibt.

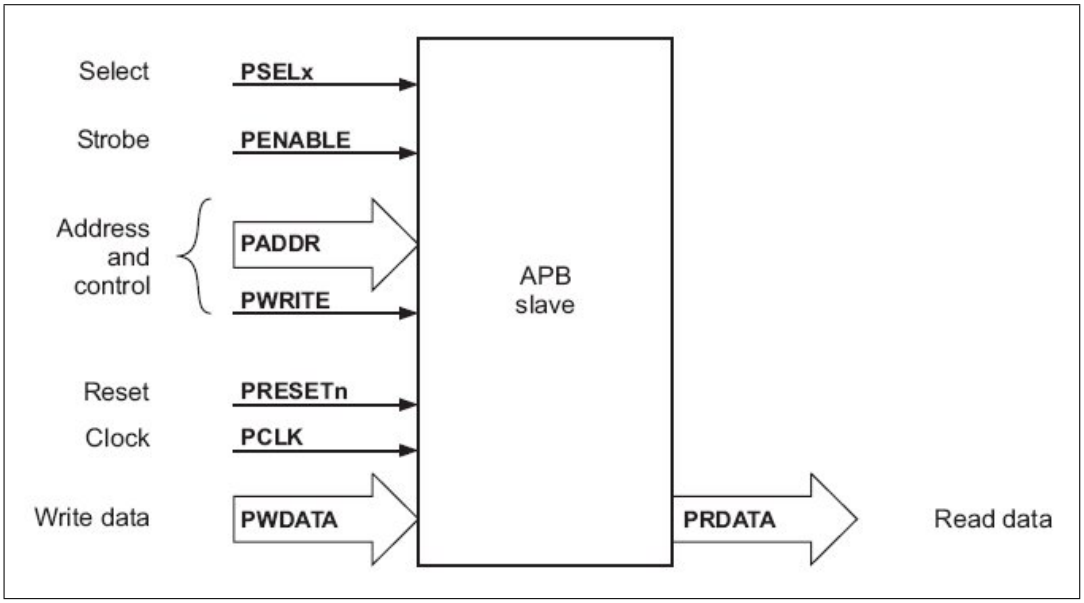

Abbildung A.3: APB-Slave Eingangs- und Ausgangsports nach [ambaspec]

In Abbildung A.3 sind alle Signalmengen aufgeschlüsselt. Links die APB-Slave Eingangsportmenge „apbi“. Rechts die APB-Slave Ausgangsportmenge „apbo“.

Die Namensbedeutungen der Signalmengen sind:

- „**apbo**“ - APB-Slave Output
- „**apbi**“ - APB-Slave Input

APB-Slave Output Ports

```
type apb_slv_out_type is record
  prdata  : std_logic_vector(31 downto 0);
  pirq    : std_logic_vector(NAHBIRQ-1 downto 0);
  pconfig : apb_config_type;
  pindex  : integer range 0 to NAPBSLV -1;
end record;
```

Abbildung A.4: APB-Slave Output Record der GRLIB

Abbildung A.4 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „apbo“-Signalmenge definiert.

Signalname	Bedeutung
prdata	Mit diesem Port schickt ein APB-Slave Daten an den verbundenen Master. Vergleichbar mit dem „Sl_DBus“-Port eines OPB-Slave.
pirq	Mit diesem Port kann der Slave einen Interrupt auslösen. Dieser Port wird nicht in der AMBA spezifiziert.
pconfig	Mit diesem Port übermittelt ein APB-Slave seine Konfigurationsdaten, wie (Systemadresse, benutze Interruptnummer, Adressmaske..), an den APB. Dieser Port wird nicht in der AMBA spezifiziert.
pindex	Hiermit übermittelt ein APB-Slave seine Index-Nummer an den APB. Dieser Port wird nicht in der AMBA spezifiziert.

Tabelle A.1: Signale des „apb_slv_out_type“ Record und ihre Bedeutung

Tabelle A.1 zeigt alle Siganle des „apb_slv_out_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle A.1 wird mit „apbo.*Signalname*“ als Port bezeichnet.

APB-Slave Input Ports

```
type apb_slv_in_type is record
  psel    : std_logic_vector(0 to NAPBSLV-1);
  penable : std_ulogic;
  paddr   : std_logic_vector(31 downto 0);
  pwrite  : std_ulogic;
  pwdata  : std_logic_vector(31 downto 0);
  pirq    : std_logic_vector(NAHBIRQ-1 downto 0);
end record;
```

Abbildung A.5: APB-Slave Input Record der GRLIB

Abbildung A.5 zeigt den Ausschnitt des „amba.vhd“ Pakets, der die „apbi“-Signalmenge definiert.

Signalname	Bedeutung
psel	Dieser Signalvektor beinhaltet alle Slave-Select-Signale. Ein APB-Slave überwacht das Select-Signal an seiner eigenen Index-Nummer. Dieser Port wird durch den APB getrieben. Der APB berechnet den kompletten „psel“-Vektor anhand der Adresse. Vergleichbar mit dem „OPB_select“-Port eines OPB-Slave.
penable	Dieser Port signalisiert dem APB-Slave, dass der Transfer der Daten beginnt. Dieser Port geht erst einen Takt später auf High-Prgel, nachdem „psel“ auf High-Pegel ging.
paddr	Dieser Port stellt die Zieladresse bereit, für die der Transfer gilt. Vergleichbar mit dem „OPB_ABus“-Port eines OPB-Slave.
pwrite	Signalisiert dem Slave, ob der verbundene AHB-Master schreiben oder lesen will. Vergleichbar mit dem „OPB_RNW“-Port eines OPB-Slave.
pwdata	Durch diesen Ports werden die Daten eines Transfers vom AHB-Master an den APB-Slave übermittelt. Vergleichbar mit „OPB_DBus“ eines OPB-Slave.
pirq	Zeigt an, an welcher APB-Index-Nummer ein Interrupt ausgelöst wurde.

Tabelle A.2: Signale des „apb_slv_in_type“ Record und ihre Bedeutung

Tabelle A.2 zeigt alle Siganle des „apb_slv_in_type“ Record und deren Bedeutung für die Bus Kommunikation. Ein Signal aus Tabelle A.2 wird mit „apbi.*Signalname*“ als Port bezeichnet.

Anbindung als APB-Slave

Soll eine Entity als Slave an den APB gebunden werden, so muss sie lediglich die oben definierten Signalmengen „apbo“ und „apbi“ als Ports besitzen. Dabei wird ausgenutzt, dass die Signalmengen „apbo“ und „apbi“ definiert wurden.

```
library grlib;
use grlib.amba.all;

entity apbslave is
  generic (pindex : integer := 0);
  port (
    rst   : in std_ulogic;
    clk   : in std_ulogic;
    apbi  : in apb_slv_in_type;
    apbo  : out apb_slv_out_type
  );
end entity;
```

Abbildung A.6: APB-Slave Definition

Abbildung A.6 zeigt die Entity Definition eines APB-Slave. Die Ports „apbo“ und „apbi“ müssen vorhanden sein (roter Rahmen).

Die Verbindung einer APB-Slave Entity mit dem APB erfolgt in folgender Weise. Zusätzlich ist die Anbindung des APB als AHB-Slave an den AHB dargestellt:

```
entity leon3mp is
  port (
    reset : in  std_ulogic;
    clk   : in  std_ulogic;
  );
end;

architecture struct of leon3mp is

signal ahbmi : ahb_mst_in_type;
signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
signal ahbsi : ahb_slv_in_type;
signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_vector := (others => apb_none);
begin

  ahb : ahbctrl      -- AHB arbiter/multiplexer
  generic map (nahbm => 0, nahbs => 1)
  port map (rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

  apb0 : apbctrl     -- APB
  generic map (hindex => 0, haddr => 0x800, nslaves => 16)
  port map (rst, clk, ahbsi, ahbso(0), apbi, apbo);

  apbslave0 : apbslave
  generic map (pindex => 0)
  port map (rst, clk, apbi, apbo(0));

end;
```

Abbildung A.7: Anschließen eines APB-Slave an den APB

Die Abbildung A.7 zeigt, dass die Entity „apbslave0“ an den Vektorindex 0 angeschlossen ist. Die anderen Vektorindizes werden mit der Konstante „apb_none“ getrieben. Die Konstante „apb_none“ ist in dem VHDL AMBA Paket definiert und füllt alle Signale mit logischen Nullen. Ebenfalls ist zu erkennen, dass der APB, der durch die Entity „apb0“ eingebunden wird, als AHB-Slave an dem Index 0 angeschlossen ist und zusätzlich die Signale „apbi“ und „apbo“ komplett einbindet.

In der GRLIB Umsetzung des APB wurden einige zusätzliche Signale für die Bus-Interfaces definiert. Es sollen nun einige wichtige Funktionsweisen des APB diskutiert werden, dabei wird auf die zusätzlichen Signale eingegangen und auf Unterschiede zu dem OPB aufmerksam gemacht.

Die „pindex“-Konstante

Die „pindex“-Konstante ist analog zur „hindex“-Konstante eine Generic-Konstante, die vor der Synthese des APB-Slave festgelegt werden muss. In den Abbildungen A.7 und Abbildung A.6 ist

die „pindex“-Konstante mit einem blauen Rahmen markiert.
Die „pindex“-Konstante muss mit dem APB-Vektorindex, des „apbo(*)“-Signalvektors, übereinstimmen, an den der APB-Slave seine „apbo“-Signalmenge einleitet.
Die „pindex“-Konstante wird verwendet, um den korrekten „psel“-Signalindex, vom APB-Slave, zu überwachen und damit zu erkennen wann der APB-Slave für einen Transfer ausgewählt wurde.

Eine genaue Erläuterung des Select-Prozesses beim APB erfolgt im Kapitelpunkt A.1.2.

Der „pconfig“-Port

Über diesen Port ist es analog zum „hconfig“-Port möglich, die APB-Slave Konfigurationsdaten an den APB zu übermitteln. Dieser Port gewährleistet also die „Plug and Play“ Funktionalität auch innerhalb des APB. Die Konfigurationsdaten werden hauptsächlich durch die Entity Generic-Konstanten festgelegt.

Zu den Konfigurations-Informationen einer APB-Entity gehört:

- Anbieter Identifikation
- Device Identifikation
- Device Version
- Device Interrupt

Sowie

- Adresse im Systemadressraum
- Adressmaske

Eine genaue Erläuterung des „Plug and Play“ Vorgangs beim APB erfolgt in Kapitelpunkt A.1.3.

Der „pirq“-Port

Der „pirq“-Port gehört zu den Ausgangsports einer APB-Slave Entity. Durch diesen Port kann die Entity einen für sie zugehörigen Interrupt auslösen.

Nähere Erläuterung der Interrupt-Steuerung beim APB erfolgt in Kapitelpunkt A.1.5.

A.1.2 Das APB-Select-Signal („psel“)

Das APB-Select-Signal („psel“) ist, wie das „hsel“-Signal, ein Signalvektor aus 16 Signalen. Jeder APB-Slave überwacht den „psel“-Signalvektor an dem Index, der durch seine „pindex“-Konstante vorgegeben ist. Wird das Signal „psel(index)“ durch den Bus auf High-Pegel gesetzt, so weiß der APB-Slave mit „index = pindex“, dass er als Ziel eines Transfers ausgewählt ist.

Selektierung im APB System

Bevor ein APB-Slave mit „psel“ aktiviert werden kann, muss erst der APB selbst selektiert werden. Genauso ist es im OPB System. Wenn der Benutzer beispielsweise eine OPB2OPB-Bridge einsetzt, um zwei OPB miteinander zu verbinden.

Ein APB-Slave braucht, genauso wie ein AHB-Slave, keinen Adressenvergleich vornehmen. Alle APB-Slave Adressen wurden dem APB, über die „Plug and Play“ Funktionalität des APB, mitgeteilt, so dass eine Berechnung des „psel“-Signals auf Seiten des APB möglich ist.
Der APB vergleicht dann, im Falle eines AHB-Master Lese- oder Schreibzugriff, an welchen APB-Slave der Transfer gerichtet ist und treibt den „psel“-Signalvektor an dem entsprechendem Index mit dem High-Pegel.
Dieser VHDL-Code ist im APB („apbctrl.vhd“) an der Zeile 125 zu finden. Es wird im folgenden nur das notwendigste, in der Abbildung A.8 eingeblendet:

```
if (((apbo(i).pconfig(1)(31 downto 20) and apbo(i).pconfig(1)(15 downto 4)) =
(HADDR(19 downto 8) and apbo(i).pconfig(1)(15 downto 4))) then
     psel(i) := '1';
end if;
```

Abbildung A.8: Selektierung im APB-System

Das Signal „HADDR“ in Abbildung A.8 beinhaltet die Adresse, die von einem AHB-Master, zu einem Transfer, ausgewählt wurde. Sie bildet mit der APB-Slave-Adressmaske (in Abbildung A.8 blau markiert) eine Konjuktion, da nicht immer alle Adressteile von Interesse sein sollen. Die Konjuktion wird dann mit den „Plug and Play“ „paddr“-Informationen aller APB-Slaves verglichen. Die „paddr“ Konfigurationen sind durch die „Plug and Play“ Funktionalität im „pconfig“-Signal abgelegt.
Wird ein Treffer-Index gefunden so treibt der APB-Arbiter an diesem Index im „psel“-Signalvektor den High-Pegel.

Ein wichtiger Unterschied zum Selektierungs-Prozess im AHB ist, dass von der Zugriffs-Adresse („HADDR“) nur die Bits 19 bis 8 benötigt werden. Dies hat folgenden Hintergrund:
Die Bits 31 bis 20 des Signals „HADDR“ werden benötigt um den AHB-Slave-Interface des APB

auszuwählen. Erst die darauf folgenden 12 Bits können für die Selektierung eines APB-Slaves verwendet werden.

A.1.3 „Plug and Play“

Die „Plug and Play“ Funktionalität des APB ist nur für Slaves möglich, da an den APB keine Masters angeschlossen werden können. Die Konfigurationsdaten aller APB-Slaves werden in dem „pconfig“-Signal des APB abgelegt.

Das „Plug and Play“ eines APB-Slaves besteht aus drei Teilen (nur für Slaves möglich):

- Identifikation der anliegenden SoC-Komponente
- Interrupt-Übermittlung („pirq“-Generic-Konstante)
- Adress-Übermittelung („paddr“-Generic-Konstante)

Will eine Entity ihre Konfigurationsdaten an den APB-Arbiter übermitteln, so muss sie den „pconfig“-Port treiben. Der „pconfig“-Port besteht aus 2 mal 32 Bit Registern und hat folgenden Aufbau:

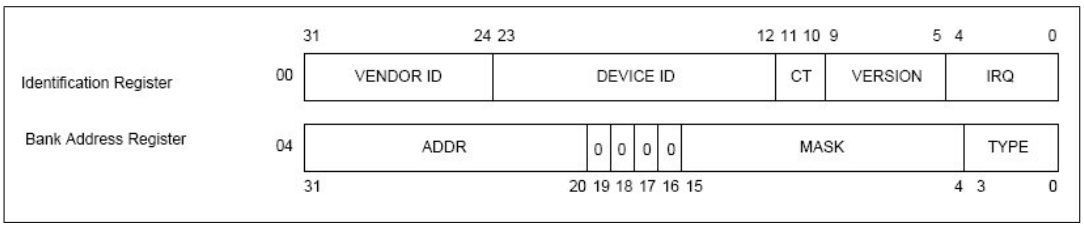

Abbildung A.9: APB - „pconfig“-Port Einteilung [grlibdoc]

Das **1. Register** in Abbildung A.9 ist das Identifikationsregister und beinhaltet die Identifikation des Anbieters der Entity und eine Identifikation der Entity selbst. In den Bits 4 bis 0 wird der verwendete Interrupt angegeben.
Das **2. Register** ist das „Bank Address Registers“. Hier kann ein Slave seine zur Verfügung gestellten Adressbereiche angeben. Die Bits 31 bis 20 beinhalten dabei die Basisadresse („paddr“). Um die Größe eines jeden Adressbereiches festzulegen muss die jeweilige Adressmaske bestimmt und, in den Bits 15 bis 4, angegeben werden.

Nähere Beschreibung des Adressierungs-Prozesses erfolgt in Kapitelpunkt A.1.4.

Haben nun alle Slaves, die an den APB angeschlossen sind, ihre Konfigurationsinformationen angelegt, so erstellt der APB, aus den Informationen, eine Konfigurationstabelle. Diese Tabelle besteht aus der Aneinanderkettung aller APB-Slave „pconfig“-Signalen.

Die Konfigurationstabelle wird standardmäßig an der Adresse 0x???FF000[1] zur Verfügung gestellt. Diese Adresse setzt sich aus den Generic-Konstanten der Entity „apbctrl“ wie folgt zusammen:

Startadresse der Konfigurationstabelle =
0x haddr & FF000 (Standardmäßig: 0x 800 & FF000)

Ab der Startadresse 0x800FF000 folgen 0x1000 Bytes, bis 0x800FFFFF, an Konfigurationsdaten der APB-Slaves:

$0x1000Bytes = 4096Bytes = 2\frac{Register}{Slave} \cdot 4\frac{Bytes}{Register} \cdot 512Slave$

Es könnten also maximal 512 Slaves ihre Konfigurationsdaten ablegen.

A.1.4 Der Adressierungs-Prozess

Der Adressierungs-Prozess muss für die APB-Slaves ebenfalls durchgeführt werden. Das bedeutet es müssen für alle SoC-Komponenten, die Slaves auf dem APB sind, entsprechende „paddr“-Generic-Konstanten festgelegt werden, so dass sie eindeutig vom APB selektiert werden können. Dabei dürfen ebenfalls keine Überlappungen der Adressbereiche entstehen. Dies kann sichergestellt werden, indem sich die „paddr“-Generic-Konstanten aller APB-Slaves unterscheiden.
Die Generic-Konstante „paddr“ ist vom Typ Integer im Bereich von 0 bis 4095. Dies entspricht einem hexadezimalen Bereich von 0x000 bis 0xFFF, also einen Signalvektor von 12 Bit. Es wird wieder eine Umwandlung zur hexadezimalen Zahl eingeführt.

Das EDK bietet für den Adressierungs-Prozess die spezielle „Addresses“-Ansicht, in der alle SoC-Komponenten aufgelistet sind und benutzerfreundlich alle Adressen vergeben werden können. Diese Ansicht konnte leider nicht für die GRLIB übernommen werden, da eine Abhängigkeit der Adressen vorliegt.

Adressierung im OPB-System

Für SoC-Komponenten die an dem OPB angeschlossen sind gibt es keine Adressenabhängigkeit. Beliebige OPB-Slaves können beliebige Basisadressen besitzen. Diese Adressen werden in der „Addresses“-Ansicht des EDK festgelegt oder mit „Generate Addresses“ automatisch erzeugt.

[1] Die drei „???“ stehen dabei für die AHB-Slave-Interface Adresse des APB.

Adressierung im APB-System

Hier sind die Basisadressen der APB-Slaves von der Adresse des AHB-Slave-Interfaces des APB abhängig. Da der APB selbst ein Slave auf dem AHB ist, so muss für ihn auch eine Adresse in der Generic-Konstante „haddr“ angegeben werden.
Es ist üblich für den APB die „haddr“-Generic-Konstante auf 0x800 festzulegen. Alle APB-Salves werden in diesem Adressraum eingeordnet.
Diese Einordnung ruft eine Adressabhängigkeit hervor, die in der „Addresses“-Ansicht nicht verarbeitet werden konnte. Aus diesem Grund ist es für das GRLIB System notwendig alle Adressen manuell festzulegen.

Wird für einen APB-Slave „paddr = 0x003“ festgelegt, so lautet die Basisadresse des Slave 0x80000300. Diese Adresse setzt sich zusammen aus 0x „haddr“ des APB & „paddr“ des APB-Slave. Ein Beispiel dafür, wie die Adressierung im Praktischen durchgeführt wird zeigt folgende Abbildung A.10:

```
component apbslave
  generic (
    pindex : integer := 0;
    paddr  : integer := 0;
    pmask  : integer := 16#fff#);
  port (
    rst  : in std_ulogic;
    clk  : in std_ulogic;
    apbi : in apb_slv_in_type;
    apbo : out apb_slv_out_type);
end component;

apb0 : apbctrl          -- APB
generic map (hindex => 0, haddr => 0x800, nslaves => 16)
port map (rst, clk, ahbsi, ahbso(0), apbi, apbo);

slave0 : apbslave
generic map (pindex => 1, paddr => 16#002#, pmask => 16#FFF#)
port map (rst, clk, apbi, apbo(1));
```

Abbildung A.10: Adressenvergabe an einen AHB-Slave

In Abbildung A.10 ist die Entity „slave0“ am „apbo“ Vektorindex 1 verbunden. Der Adressbereich beginnt bei 0x80000200, da „haddr“ des APB mit „0x800“ festgelegt wurde und die Adresse des APB-Slave mit „paddr = 0x002“.
Mit diesen Angaben reserviert sich der APB-Slave einen Adressbereich von 0x80000200 bis 0x800002FF. Dies entspricht 256 Bytes.

A.1.5 Interrupt-Steuerung

Die Interrupt-Steuerung eines APB-Slave erfolgt über den 32 Bit Vektor „pirq“. Dieser Signalvektor ist ein Ausgangsport einer jeden Entity, die an den APB angeschlossen ist. Dadurch ist es einfach möglich einen Interrupt auszulösen.

Ein APB-Slave, der einen Interrupt auslösen will, treibt den „pirq“-Signalvektor wie folgt:

```
apbo.pirq(pirq) <= '1';
```

Dabei ist der Index „pirq“ eine Generic-Konstante vom Typ Integer, die für den APB-Slave festgelegt werden muss.

Der APB verbindet die Signalvektoren aller APB-Slaves durch eine Disjunktion, d.h. es entsteht ein Ergebnis-Interrupt-Vektor, der alle geworfenen Interrupts des SoC enthält. Dies ist ab der Zeile 154 der APB-Entity „apbctrl“ zu erkennen und in der folgenden Abbildung A.11 dargestellt:

```
for i in 0 to nslaves-1 loop

    pirq := pirq or apbo(i).pirq;

end loop;
ahbso.hirq <= pirq;
```

Abbildung A.11: Disjuktion aller APB-Interrupts zum Ergebnis-Interrupt-Vektor

Die Konstante „nslaves“, in Abbildung A.11, ist eine Generic-Konstante der APB-Entity und steht für die Anzahl der, an den APB, angeschlossenen Slaves. Sie muss bei der Deklaration des APB übergeben werden. Später, nach der Einbindung des APB in das EDK, wird sie automatisch durch die EDK Tools berechnet.
An der Abbildung A.11 ist zu erkennen, dass alle Interrupts der APB-Slaves, mit einem logischen ODER, zu einem Ergebnis-Interrupt-Vektor „pirq“ vermischt werden.
Der „pirq“ wird dann in die AHB Interrupt-Steuerung eingespeist, indem der eigene „hirq“ Vektor, des APB, mit den „pirq“ Vektor getrieben wird.

A.2 Weitere Werkzeuge für die GRLIB

Dieser Kapitelpunkt stellt Werkzeuge vor mit denen es möglich ist, Software für ein GRLIB-SoC speziell für den LEON2 oder LEON3-Prozessor zu kompilieren, diese Software zu testen und letztendlich in ein GRLIB-SoC mit LEON3-Prozessor ein zu speisen.

A.2.1 Compiler

Damit alle Arbeitsschritte des EDK problemlos abgearbeitet werden können, ist es für den Arbeitsschritt 3 notwendig einen Compiler und einen Linker bereitzustellen. Für ein MicroBlaze-System wird für diese Zwecke der **M**icro**B**laze-**G**NU **C** **C**ompiler (kurz mb-gcc) verwendet.
Die Firma GR bietet auf ihrer Homepage zwei Compiler zum Download an.

Diese sind:

- **B**are-C **C**ross-**C**ompiler (kurz BCC)
- **R**TEMS LEON/ERC32 **C**ross-**C**ompiler (kurz RCC)

Diese Compiler kompilieren eine C-Datei in eine ELF-Datei, die speziell für die Prozessoren LEON2 und LEON3 konzipiert sind. Die Linker Tools werden zu den Compilern automatisch mitgeliefert.

Für diese Arbeit, speziell für die Einbindung in das EDK, wurde der BCC verwendet. Der BCC lag zur Anfertigung der Arbeit in der Version 1.0.29c vor. Grundsätzlich ist es aber auch mit dem RCC, der in der Version 1.0.12 vorlag, ein C Programm für den LEON3 zu kompilieren. Der BCC und RCC benutzen sogar gleichnamige Programmparameter für ihre Konfigurationen.

Der RTEMS Cross-Compiler steht für **R**eal-**T**ime Operating System and **E**nviro**m**ent**s** (kurz RTEMS). Mit dem RCC ist es möglich Anwendungen für bestimmte Real-Time Bedingungen umzusetzen.

A.2.1.1 Installation des BCC

Der BCC kann auf der GR Homepage in einem Archiv namens „sparc-elf-3.4.4-1.0.29c.tar.bz2“ heruntergeladen werden. Host-PCs die mit einem Windows und mit Cygwin ausgestattet sind, müssen das Archiv „sparc-elf-3.4.4-1.0.29d-cygwin.tar.bz2“ verwenden. Es ist allerdings zu beachten, dass die Compileraufrufe des EDK, mit dem EDK eigenen Cygwin durchgeführt werden und nicht mit der Cygwin-Version, deren Installation in Kapitelpunkt 3.1.2 beschrieben wurde.

Das Archiv des BCC muss mit dem folgenden Kommando in ein beliebiges Verzeichnis extrahiert werden:

```
$ tar -C /verzeichnis -xjf sparc-elf-3.4.4-1.0.29.tar.bz2
```

Anschließend ist darauf zu Achten, dass das „bin“-Verzeichnis des BCC in den Pfad exportiert wurde:

```
$ export PATH=/verzeichnis/sparc-elf-3.4.4/bin:$PATH
```

Der Pfad muss exportiert werden, da das EDK den Compiler in der „system.make“ ausführt und so ein Zugriff auf den Compiler aus jedem Verzeichnis möglich sein muss.

A.2.1.2 Handhabung des BCC

Damit der EDK Arbeitsschritt 3 jetzt vollständig ausgeführt werden kann, sind zwei Tools des BCC relevant. Diese sind:

- BCC Compiler-Tool „sparc-elf-gcc“
- BCC Linker Tool „sparc-elf-ld“

Zur Beschreibung der Handhabung des BCC wird angenommen, dass ein korrektes C-Programm für den LEON3 in der Datei „prog.c“ vorliegt.

Zur Kompilierung eines C-Programms wird folgendes Kommando verwendet:

```
$ sparc-elf-gcc [PARAMETER][2] prog.c -o prog.exe
```

die Ausgabedatei „prog.exe“ ist eine ELF-Datei.
Eine Zusammenfassung der wichtigsten Parameter, wird in der folgenden Tabelle A.3 dargestellt:

Parameter	Bedeutung
-g	fügt der ELF-Datei Debugginginformationen hinzu
-msoft-float	Dieser Parameter muss verwendet werden, wenn keine FPU im GRLIB-SoC eingebunden ist. Es wird eine Software Emulation der FPU vorgenommen.
-c	führt kein automatisches Linking der ELF-Datei durch
-mv8	generiert die SPARC V8 mul/div Instruktionen. Damit diese Instruktionen vom LEON3 ausgeführt werden können, muss die entsprechende Hardware im LEON3 vorhanden sein. Dies wird durch die Generic-Konstante „v8“ der LEON3-Entity bewerkstelligt. Genaue Beschreibung siehe Anhang B
-O2 oder -O3	Optimierungsgrad des Compilers für maximale Performance und minimale Code Größe
-qsvt	Die ELF-Datei benutzt das Single vector trapping Modell des LEON3. Damit das Modell verwendet werden kann, muss es im LEON3 zur Verfügung stehen. Dies wird durch die Generic-Konstante „svt“ der LEON3-Entity bewerkstelligt.
-mflat	Die ELF-Datei benutzt keine Register Fenster des LEON3.

Tabelle A.3: Parameterübersicht des BCC

[2] Der Ausdruck [PARAMETER] steht für die geeignete Wahl der BCC Parameter. Diese Parameter sind abhängig von der Zusammenstellung des GRLIB-SoC und einiger Compiler Richtlinien.

Eine Wahl geeigneter Parameter, für einen LEON3 ohne FPU, wäre beispielsweise:

```
$ sparc-elf-gcc -msoft-float -c -g -O2 prog.c -o prom.exe
```

Das Linking wird darauf mit folgendem Kommando durchgeführt:

```
$ sparc-elf-ld -T linkprom.ld prom.exe -o prom.elf
```

Der Linker erstellt aus der ELF-Datei („prom.exe“) die gelinkte ELF-Datei („prom.elf“). Eine gelinkte ELF-Datei ist bereit in ein GRLIB-SoC eingespeist zu werden. Im oben ausgeführten Linking Kommando ist der Linker Parameter „`-T linkprom.ld`“ (blau markiert) zu erkennen. Mit diesem Parameter wird ein Linkerskript an den Linker übergeben.
Standardmäßig werden die nativen C-Programme des EDK, für den LEON3, an die Adresse 0x00000000 gelinkt. Dies bedeutet, dass, in dem Linkerskript „linkprom.ld“, alle Sektionen der ELF-Datei auf die Adresse 0x00000000 abgebildet werden.

Für weitere Parameter und Funktionalitäten des BCC, wird auf dessen Dokumentation [bccdoc] verwiesen.

A.2.1.3 Erstellen einer „ahbrom“-Entity

Die „ahbrom“-Entity stellt eine einfache aber effektive Methode dar ein natives C-Programm in ein GRLIB-LEON3-SoC einzubinden. Die ELF-Datei eines kompilierten C-Programms wird in die VHDL-Entity („ahbrom“) eingebettet. Diese Entity fungiert in einem GRLIB-SoC als ROM und stellt den Programmcode für den LEON3 zur Verfügung.

Die Erstellung einer „ahbrom“-Entity in der Datei „ahbrom.vhd“ kann nur innerhalb eines Template-Design Verzeichnisses erfolgen. In diesem muss sich auch die C-Programmdatei befinden.
Ein C-Programm muss für diesen Zweck zuerst kompiliert werden. Für ein C-Programm, das durch einen LEON3 ohne FPU ausgeführt werden soll, gilt dieses Kommando als Beispiel:

```
$ sparc-elf-gcc -msoft-float -c -g -O2 prog.c -o prom.exe
```

Dabei ist zu beachten, dass die Ausgabedatei „prom.exe“ genannt wird, da diese für den nächsten Schritt verwendet wird.

Mit dem nun folgenden Kommando:

```
$ make ahbrom.vhd
```

wird aus der nicht gelinkten ELF-Datei („prom.exe“) die „ahbrom“-Entity erstellt. Ein vorheriges Linking der ELF-Datei ist nicht notwendig, da die „ahbrom“-Entity nun die Adresse bestimmt, an der die Programmdaten im GRLIB-SoC zur Verfügung stehen sollen. Dies wird durch die Generic-Konstante „haddr“ der „ahbrom“-Entity festgelegt.

Damit der LEON3 die Programmdaten bei seiner Inbetriebnahme findet, sollte die „ahbrom“-Entity die Systemadresse 0x00000000 haben. Dafür muss „haddr = 0x000“ sein.

Die folgende Abbildung A.12 zeigt den Kern-Prozess einer jeden „ahbrom“-Entity:

```
comb : process (addr)
begin
  case conv_integer(addr) is
  when 16#00000# => romdata <= X"03200000";
  when 16#00001# => romdata <= X"9A103FFF";
  when 16#00002# => romdata <= X"94106200";
  when 16#00003# => romdata <= X"DA22A008";
  when 16#00004# => romdata <= X"19004C4B";
  when 16#00005# => romdata <= X"981320FF";
  when 16#00006# => romdata <= X"96106100";
  when 16#00007# => romdata <= X"9A102000";
  when 16#00008# => romdata <= X"82032001";
  when others => romdata <= (others => '-');
  end case;
end process;
```

Abbildung A.12: Kern-Prozess der „ahbrom“-Entity

In der Abbildung A.12 wird das Signal „addr“ durch den Port „ahbsi.haddr“ getrieben und der Port „ahbso.hrdata“ durch das Signal „romdata“. Dadurch werden dem LEON3-Prozessor die unterschiedlichen Instruktionsdaten übermittelt, wenn er Adressen ab der Systemadresse 0x00000000 an den Bus legt.
Der große Vorteil der „ahbrom“-Entity ist, dass sie von dem XST als Inhalt eines BRAM erkannt wird. Wird also die „ahbrom“-Entity in einem GRLIB-SoC verwendet, so wird automatisch eine benötigte Menge an BRAMs für die Speicherung des „prom.exe“ Inhalts benutzt.

A.2.2 Gaisler Research Debugmonitor (GRMON)

Der **G**aisler **R**esearch Debug**mon**itor (kurz GRMON) ist ein wichtiges Tool für das Debugging eines LEON2/LEON3-GRLIB-SoC. Der GRMON ist das GRLIB Pendant zur **X**ilinx **M**icroprocessor **D**ebug (kurz XMD) Engine.

Der GRMON verfügt über folgende Funktionalitäten:

- Lesen und Schreiben an allen Systemadressen, dies umfasst (Register und Speicher)
- Auslesen von Tracepuffern[3] innerhalb des GRLIB-SoC
- Einspeisen und Ausführen von LEON Anwendungen auf dem SoC

[3] Tracepuffer sind Speicher, in denen AHB Transfers (Adresse, Datum und Kontrollsignale) gesichert werden. Dieser Systemtransferverlauf kann dann mittels GRMON, zur Analyse, abgerufen werden.

- Breakpoint und Watchpoint Management
- Unterstütz das Debugging über JTAG (Parallel III und IV Kabel, USB nur für Altera FPGAs), RS232, Ethernet, USB (nur in Linux).

A.2.2.1 Installation

Der GRMON kann auf der Homepage der Firma GR heruntergeladen werden. In dieser Arbeit wurde nur die Evaluation Version des GRMON (kurz GRMON-Eval) benutzt, eine uneingeschränkte Version des GRMON ist Lizenzpflichtig.
Eine einzelne Version des GRMON-Eval läuft nach 21 Tagen aus. Während der Erstellung dieser Arbeit erschienen allerdings regelmäßig Nachfolgerversionen, die mit gleicher Funktionalität sofort einsatzbereit waren.

Der GRMON-Eval ist eine Terminal-Anwendung, die mit einem GUI erweitert werden kann. Diese GUI nennt sich GrmonRCP. Damit GrmonRCP benutzt werden kann, muss eine Java Runtime Environment mit der Version 1.4.1 oder höher auf dem Host-PC installiert sein. In dieser Arbeit wurde nur die GRMON-Eval Terminal-Anwendung benutzt, da durch die GUI keine weitern Funktionalitäten erkennbar waren und die Terminal-Anwendung bereits ausreichend benutzerfreundlich war.

A.2.2.2 Handhabung

Der GRMON-Eval ist aus jedem Verzeichnis ausführbar und versucht sich, wenn er ohne weitere Parameter gestartet wird, über die serielle Schnittstelle mit einem GRLIB-SoC zu verbinden.
Wird es gewünscht eine Verbindung mit dem GRLIB-SoC aufzubauen und einen anderen Kommunikationsweg zu nutzen, so wird dies mit unterschiedlichen Startparametern bewirkt.

Startparameter

Es folgt eine Tabelle A.4 aller möglichen Kommunikationswege, der dafür notwendige Parameter und die notwendigen SoC-Komponenten sowie zusätzlichen Bemerkungen:

Kommunikationsweg	Parameter	SoC-Komponenten
Serielle Schnittstelle	(optional) -baud *baudrate*	ahbuart
Bemerkung: Die Initialisierung der Debugging Verbindung kann mehrere Versuche benötigen, da das Scalar Register der „ahbuart"-Entity während der Kommunikation bestimmt wird. Es wird also automatisch versucht die verwendete Baudrate zu erkennen.		
Ethernet	-eth (optional) -ip *IP-adresse des Boards*	greth mit EDCL
Bemerkung: Ermöglicht der Debugging mit bis zu theoretischen 100 MBits/s. Es werden dabei UDP-Pakete zwischen Board und Host-PC ausgetauscht.		
JTAG	-jtag	ahbjtag
Bemerkung: JTAG kann bei Xilinx FPGAs nur über Parallel III oder IV Kabel benutzt werden, da die Firma Xilinx die API für ihre USB Treiber nicht veröffentlichen möchte.		
JTAG	-altjtag	ahbjtag
Bemerkung: Das Debugging eines GRLIB-SoC auf einem Altera FPGA ist mit diesem Parameter über JTAG USB möglich.		
USB	-usb	usbdcl
Bemerkung: Mit dem IP-Core „usbdcl" ist das Debugging über USB möglich. Dies hat allerdings nichts mit einem Boundary Scan Test wie JTAG zu tun. Das Board muss über eine extra USB Schnittstelle verfügen, die außer der JTAG Funktion zusätzlich die Möglichkeit bietet USB Geräte anzuschließen. Dies ist beispielsweise auf dem XUP-Board nicht der Fall. Auf dem XUP-Board sind über USB nur die JTAG Funktionen möglich.		

Tabelle A.4: GRMON - Kommunikationswege

Zu Beachten A.1 (GRMON mit SnapGear-Linux 2.6)

*Soll auf einem GRLIB-SoC das Betriebssystem „SnapGear"-Linux mit dem Kernel 2.6 eingespeist und ausgeführt werden, so muss GRMON zusätzlich mit dem Parameter „**-nb**" gestartet werden.*

Um einen vollständigen Debugging Prozess eines GRLIB-SoC zu ermöglichen werden die IP-Cores **D**ebug **S**upport **U**nit (kurz DSU) und LEON3 in dem SoC benötigt.

Die DSU und der LEON3 ermöglichen folgende Debugging Funktionalitäten:

- Das Auslesen von Tracepuffern
- Das Auslesen der LEON3 Registern
- Das Ausführen von, in das SoC eingespeisten, Programmen
- Das setzen von Breakpoints und Watchpoints

Nachdem GRMON korrekt gestartet wurde und die Debuggingverbindung etabliert ist, wird eine kurze Zusammenfassen des ganzen SoC angezeigt.
Diese Zusammenfassung wird durch die „Plug and Play“ Funktionalität des AHB und APB möglich. GRMON erstellt zuerst eine Liste der AHB-Masters und AHB-Slaves, dazu muss GRMON lediglich die Systemadressen von 0xFFFF000 bis 0xFFFFF7FF auslesen und interpretieren. Dieser Adressbereich ist der „Plug and Play“ Konfigurationstabellenbereich des AHB, siehe Kapitelpunkt 3.1.5.2.

Wird eine APB in dem SoC gefunden, so könnten weitere SoC-Komponenten am APB als APB-Slaves verbunden sein. Um diese SoC-Komponenten in die Zusammenfassung aufzunehmen, wird der Konfigurationstabellenbereich des APB ausgelesen und interpretiert. Der Konfigurationstabellenbereich des APB erstreckt sich von 0x 800FF000 bis 0x800FFFFF. Dabei ist „0x8002’ die Adresse des APB am AHB.

Ist die Zusammenfassung des SoC erstellt und auf der Konsole ausgegeben, so wartet GRMON auf die Debuggingkommandos, des Benutzers.

GRMON Kommandos

Um alle, in dieser Arbeit, durchgeführten Vorgänge nach zu vollziehen wird eine Tabelle A.5 aller benötigten GRMON Kommandos mit Erläuterung gegeben:

`grlib> info sys`

Zeigt eine detaillierte Zusammenfassung des SoC an.

`grlib> load` *`ELF-Datei`*

Das Kommando „load“ läd eine, mit dem Ausdruck *`ELF-Datei`* bezeichnete, gelinkte Datei im ELF in den Systemspeicher des SoC. Dabei ist darauf zu achten, dass die ELF-Datei gelinkt sein muss. Nur dadurch kann GRMON entscheiden, an welche Adressen die einzelnen ELF-Sektionen geschrieben werden sollen.
Ebenfalls muss sichergestellt sein, dass sich an den zu beschreibenden Systemadressen eine SoC-Komponente befindet, die als Datenspeicher fungiert.
Befinden sich andernfalls Register einer SoC-Komponente an diesen Systemadressen so kann es zu Fehlfunktionen des SoC kommen, da Register nur Speicher zur Steuerung von SoC-Komponenten sind.
Würde sich an den zu beschreibenden Stellen im Adressraum allerdings keine SoC-Komponente befinden, so wären die übermittelten Daten der ELF-Datei verloren.

`grlib> verify` *`ELF-Datei`*

Mit diesem Kommando kann eine geladene ELF-Datei nochmals überprüft werden. Da ein SoC in der Entwicklungsphase fehlerhaft funktionieren kann, könnten Datentransfers über den Bus fehlerhaft verlaufen. Aus diesem Grund wäre auch das Laden einer ELF-Datei nicht vollständig möglich.
Bei einer Verifikation der ELF-Datei, werden alle ELF-Sektionen aus dem Systemspeicher gelesen und mit der ELF-Datei des Parameters verglichen.

`grlib> run`

Durch dieses Kommando wird der „program counter“ des LEON3-Prozessors auf den Einsprungspunkt der zuletzt geladenen ELF-Datei gesetzt und die Ausführung begonnen.

`grlib> go` *`Adresse`*

Hier wird der „program counter“ des LEON3 auf *`Adresse`* gesetzt und dort die Ausführung eines Programms begonnen.

`grlib> mem` *`Adresse Anzahl`*

Zeigt mit *`Anzahl`* bezeichnet viele Bytes aus dem Inhalt des Systemspeichers ab der *`Adresse`* an.

`grlib> wmem` *`Adresse Datum`*

Schreibt die mit *`Datum`* bezeichneten 32Bit an die *`Adresse`* im Systemspeicher.

Tabelle A.5: GRMON Kommandos - Ausschnitt [grmondoc]

Zu Beachten A.2 (Debugging über Ethernet)

Beim Debugging eines GRLIB-SoC über die Ethernet Schnittstelle, ist darauf zu achten, dass eine mögliche Firewall auf dem Host-PC entsprechend für die Kommunikation eingerichtet wird.

Weitere Startparameter und Kommandos für GRMON werden in dessen Dokumentation [grmondoc] beschrieben.

A.2.3 SPARC-Simulator (TSIM)

Der TSIM ist ein spezieller Simulator für Prozessoren der SPARC-Architektur. Die Evaluations-Version des TSIM (kurz TSIM-Eval) kann auf der Homepage der Firma GR heruntergeladen werden. Die kostenpflichtigen Professional-Versionen des TSIM ist für die Prozessoren LEON2, LEON3 und ERC32 erhältlich. Der ERC32 ist ein Vorgängerprozessor der LEON-Reihe. Der TSIM-Eval ist nur für den LEON3 verfügbar.
Der TSIM-Eval verfügt über ähnliche Kommandos wie GRMON. So ist das Laden und Ausführen von ELF-Dateien analog zum GRMON möglich. Der TSIM ist dafür geeignet, um ELF-Programme während ihrer Ausführung auf einem LEON-System zu überwachen. Dabei werden folgende Funktionalitäten unterstützt [tsimdoc]:

- Unterbrechung des Programmablaufes mit Breakpoints
- Überwachen des Systemspeichers mit Watchpoints
- Performance-Analyse bezüglich eines realen SoC-Prozessors
- Emulationen von UART, FPU, Prozessor, Cache Speicher und Interrupt-Controller können unter anderem durchgeführt werden.
- Emulationen von IP-Cores durch das Laden zusätzlicher Module

Da der TSIM für diese Arbeit nicht verwendet wurde, werden keine weiteren Details bezüglich der Handhabung erläutert. Für näheres Interesse sei auf die Dokumentation des TSIM verwiesen.

Anhang B

Der LEON3-Prozessor

In diesem Kapitel soll der LEON3-Prozessor vorgestellt werden, der in der GRLIB als Soft-Prozessor verfügbar ist. Der LEON3 wurde von der **E**urpean **S**pace **A**rgency (kurz ESA) entwickelt und von der Firma GR weitergeführt. Die ESA bietet eine fehlertolerante Version des LEON3 an, den LEON3-FT. Der LEON3-FT kann bei der Firma GR gebührenpflichtig angefordert werden.
Der LEON3 ist eine Implementierung nach der **S**calable **P**rocessor **Arc**hitekture (kurz SPARC). Im Kapitelpunkt B.1 sollen einige Spezifikationen der SPARC vorgestellt werden, um somit grundlegende Funktionsweisen und deren Umsetzungen, innerhalb des LEON3, zu beschreiben.
Da der LEON3 in der GRLIB in seiner Konfiguration sehr flexibel implementiert wurde, soll in Kapitelpunkt B.2 eine Erläuterung der Konfigurationsmöglichkeiten und deren Auswirkungen auf die Funktionsweise erfolgen. Der LEON3 ist in der GRLIB als Entity mit dem Namen „leon3s“ zu finden.
In Kapitel 8, wird im Rahmen der Ergebnisse dieser Arbeit, ein Vergleich der Prozessoren MicroBlaze und LEON3 durchgeführt. Als Vergleichskriterien sollen Performance, Größe und Funktionalität stehen. Bis zu Anfang des Jahres 2007 wurde auf der Homepage der LEON2 Prozessor zum Download angeboten. Da der Support des LEON2 von GR eingestellt wurde, wird kein Vergleich mit dem LEON2 Prozessor ausgeführt.

B.1 Die SPARC-Architektur Version 8

Die SPARC wurde 1985 von Sun Microsystems festgelegt und diente als Architekturvorlage für den LEON3-Prozessor. Durch die SPARC festgelegt, ist der LEON3 eine RISC-Architektur und soll mit einem kleinen Befehlssatz effektiv Programme abarbeiten. RISC-CPUs kommen gehäuft

in eingebetteten Systemen oder FPGAs zum Einsatz, da in diesen Bereichen auf besonders ressourcensparende Umsetzungen von CPUs wertgelegt wird. Durch die RISC besitzen CPUs nach der SPARC, Befehle mit einer einheitlichen Länge von einem Wort[1] (32Bit). Dadurch ist ein besonders kompakter Prozessorkern möglich, welcher sich im Vergleich zu einer CISC Architektur, effizienter auf einem FPGA integrieren lässt.
Ein SPARC-Prozessor setzt sich aus einer **I**nteger **U**nit (kurz IU), einer **F**loating-**P**oint **U**nit (kurz FPU) und weiteren Co-Prozessoren zusammen, welche jeweils über eigene Register mit der konstanten Länge von einem Wort verfügen. Soll eine höhere Genauigkeit bei den Über- oder Ausgabeoperanden benutzt werden, so werden mehrere Register zusammengeschlossen. Dadurch entstehen einfache Register (single word registers, single-precision), Register Paare (double word registers, double-precision) und vierfach Register (register quadruples).

Die SPARC definiert eine Harvard-Architektur. Das heißt, es soll eine Trennung von Programmspeicher und Datenspeicher stattfinden, um eine parallele und damit besonders schnelle Übertragung von Instruktionen und Daten zu ermöglichen. Programm- und Datenspeicher sollten, bei einer Harvard-Architektur, über getrennte Busse angesteuert werden.
Der LEON3 ist keine reine Harvard-Architektur. Eine parallele Ansteuerung von Programm- und Datenspeicher ist nicht möglich, da der LEON3 nur über ein AHB-Master-Interface verfügt und damit nur maximal ein Wort pro Takt entweder aus dem Programm- oder Datenspeicher lesen oder schreiben kann.
Dieser Nachteil wird dadurch abgeschwächt, dass der LEON3 über ein Cache-System verfügt, dass sowohl einen Programm-Cache als auch einen Daten-Cache beinhaltet. Der Zugriff auf diese Cache-Speicher erfolgt parallel. Dieser Geschwindigkeitsvorteil wird ausgenutzt, nachdem der LEON3 einige Instruktionen über den AHB in die Caches laden konnte. Das Cache-System ist nicht zwingend notwendig, wenn der LEON3-Prozessor die Instruktionen aus einem BRAM liest, da auf einen BRAM innerhalb von einem Takt lesend und schreibend zugegriffen werden kann. Zusätzlich werden auf einem Xilinx FPGA Dual-Port-BRAMs bereitgestellt. Auf diese BRAMs kann parallel zugegriffen werden zum Beispiel als Programm- und Datenspeicher.
Ein Cache-System wäre notwendiger, wenn beispielsweise die Daten aus dem DDR SDRAM gelesen würden. Allerdings unterscheidet der LEON3 nicht nach der Speicherzugriffszeit, so dass das Cache-System für alle AHB-Transfers aktiviert ist, wenn es vorhanden ist.

Der MicroBlaze-Prozessor ist eine echte Harvard-Architektur. Hier werden die nativen C-Programme aus einem BRAM gelesen, der über zwei parallel ansprechbare Ports verfügt.
Auf die beiden Ports des BRAMs kann ein paralleler Zugriff über die beiden anliegenden LMB ausgeführt werden. Der MicroBlaze-Prozessor verfügt zusätzlich über zwei LMB-Master-Interfaces um die Daten parallel in seine Register laden zu können.
Ein Cache-System für die Benutzung der nativen C-Programme ist, aufgrund der schnellen BRAM, nicht zwingend nötig. Das Cache-System des MicroBlaze-Prozessors kann nur bei der Verwendung

[1] Eine „Wort“ hat eine Länge von 32 Bit

von SDRAM und sonstigen „langsamen“ Speichern aktiviert werden.

Für mehr allgemeine Informationen über eine SPARC Version 8, sei auf deren Spezifikation [sparcv8sepc] verwiesen.

B.2 Ausgewählte Eigenschaften des LEON3

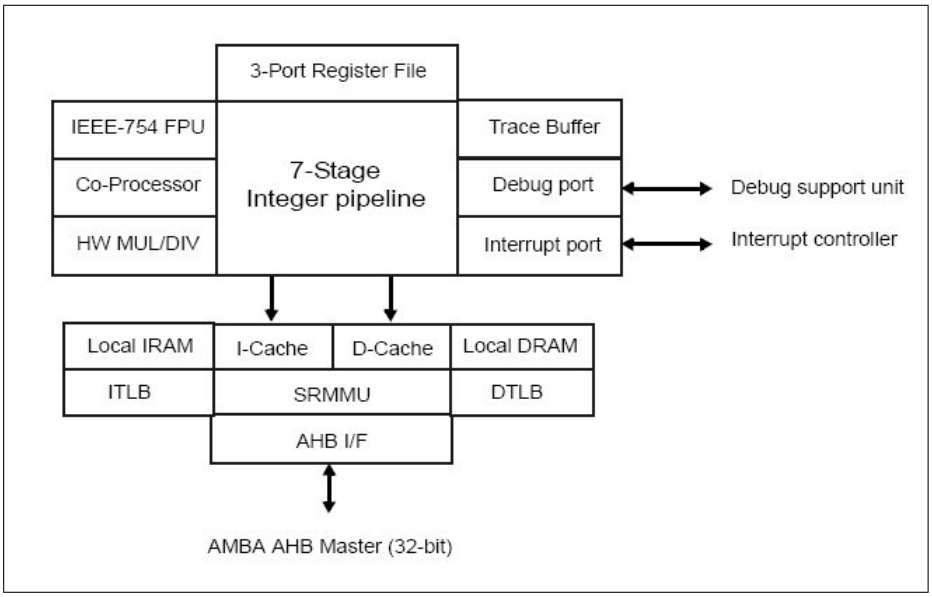

Abbildung B.1: Elemente des LEON3-Prozessors [gripdoc]

Abbildung B.1 zeigt eine Gesamtübersicht der Elemente des LEON3-Prozessors [gripdoc]. Es ist zu erkennen, dass sich neben dem AHB-Master-Interface ein Interface für die DSU und den Interrupt-Controller am LEON3 befindet. Jeder LEON3-Prozessor eines SoC wird direkt an die DSU oder an den Interrupt-Controller angeschlossen. DSU und Interrupt-Controller sind eigenständige IP-Cores der GRLIB.

Der LEON3 Kern verfügt über folgende Funktionalitäten [gripdoc]:

- 7 stufige Pipeline mit Harvard-Architektur
- getrennte Caches für Instruktionen und Daten
- Hardware Multiplizierer und Dividierer
- **D**ebug **S**upport **U**nit (kurz DSU)
- Multi-Prozessor Unterstützung

In den Unterkapitelpunkten des Kapitelpunkts B.2 soll die SPARC konkret am Beispiel des LEON3 genauer erläutert werden. Es wird dabei auf die konfigurierbaren Funktionalitäten des LEON3 eingegangen und auf Umsetzungen hingewiesen, welche nicht in der SPARC spezifiziert wurden.

Alle konfigurierbaren Funktionalitäten des LEON3 sind über spezielle Generic-Konstanten der LEON3-Entity einstellbar. Da die Anzahl der LEON3-Generic-Konstanten 45 beträgt, werden die entsprechenden Generic-Konstanten mit ihren Wirkungen genannt.
Nicht alle Generic-Konstanten der LEON3-Entity legen Funktionalitäten fest. Diese werden in diesem Kapitel nicht besprochen.

Zur Übersicht wird hier eine alphabetisch geordnete Auflistung der Generic-Konstanten mit Seitenzahl gegeben, die innerhalb des Kapitelpunktes B.2 besprochen werden:

broadcast (234), cp (230), dlinesize (240), drepl (240), dsetlock (241), dsets (240), dsetsize (240), dsnoop (234), dtlbnum (238), fpu (231), ilinesize (240), irepl (240), isetlock (241), isets (240), isetsize (240), itlbnum (238), mmuen (236), nwindows (226), rstaddr (233), smp (234), tlb_rep (239), tlb_type (238), v8 (225)

B.2.1 Integer Unit (IU)

Die **I**nteger **U**nit (kurz IU) einer SPARC verwaltet alle Register des Prozessors und übernimmt die Ausführung einfacher Integer Befehle. Ebenfalls werden die Speicheradressen berechnet und Load und Store Operationen ausgeführt. Die IU beinhaltet und verwaltet den „program counter“ und übernimmt die Ansteuerung von FPU und anderen Co-Prozessoren.

B.2.1.1 Multiplikation und Division

Laut der SPARC ist vorgeschrieben, dass der LEON3 folgende Befehle zur Integer Multiplikation und Integer Division besitzen muss:

- UMUL (UMULcc) zur Multiplikation von Integer Operanten
- SMUL (SMULcc) zur Multiplikation von vorzeichenbehafteten Integer Operanten
- UDIV (UDIVcc) zur Division von Integer Operanten
- SDIV (SDIVcc) zur Division von vorzeichenbehafteten Integer Operanten

Die IU kann bei Bedarf zwei zusätzliche Hilfsschaltungen ansteuern. Diese sind die MUL32-Schaltung, zur Multiplikation von zwei 32 Bit Operanten und die DIV32-Schaltung, zur Division eines 64 Bit Operanden durch einen 32 Bit Operand. Diese zusätzlichen Hilfsschaltungen können mit der Generic-Konstante „**v8**“, der LEON3-Entity, aktiviert werden.

Wählt der Benutzer „v8 = 1“, so werden die MUL/DIV32 Entitys in den LEON3 eingebunden. Mit „v8 = 2“ wird eine Pipeline zur Benutzung der MUL32-Schaltung eingerichtet. Die MUL32-Schaltung benötigt „ungepipet“ 4 Takte und „gepipet“ 5 Takte bis das Ergebnis der Operation vorliegt.

Bei der Verwendung einer Pipeline ist die Latenzzeit einen Takt länger, bis die erste Multiplikation ausgeführt ist, aber es können effektiv mehr Multiplikationen pro Takt ausgeführt werden.
Wird die MUL32 Schaltung beispielsweise für einem Xilinx FPGA synthetisiert, so wird automatisch ein Multiplikationsblock des FPGAs verwendet.
Die DIV32 Schaltung benötigt 35 Takte bis das Ergebnis der Operation vorliegt.

B.2.1.2 Register

Die SPARC spezifiziert, dass der LEON3 zwischen 40 und 520 Registern besitzen muss. Die Anzahl ist wiederum von der Anzahl der Register-Fenster abhängig. Für den LEON3 kann die Anzahl der Register-Fenster mit der Generic-Konstante „**nwindows**“, der LEON3-Entity, von 2-32 variiert werden. Die Register werden als Dual-Port-BRAM auf Xilinx FPGAs realisiert. Das Verhalten, der Register, ist in der Entity „regfile_3p“ beschrieben, die eine Unter-Entity der „leon3s“-Entity ist.

Der LEON3 besitzt aufgrund seiner festen Befehlslänge 32 Register. Um die Anzahl, laut SPARC, zu erhöhen bedient sich der LEON3 sogenannter Register-Fenster.
Von den 32 Registern sind 8 Register (r[0]-r[7]) die globalen Register, diese stehen zur Einführung von Register-Fenstern nicht zur Verfügung. Die übrigen 24 Register (r[8]-r[31]) bilden das momentane Register-Fenster. Dieses unterteilt sich in 3 Sektionen mit je 8 Registern. Diese Sektionen sind die „ins“, „locals“ und „outs“ Register.

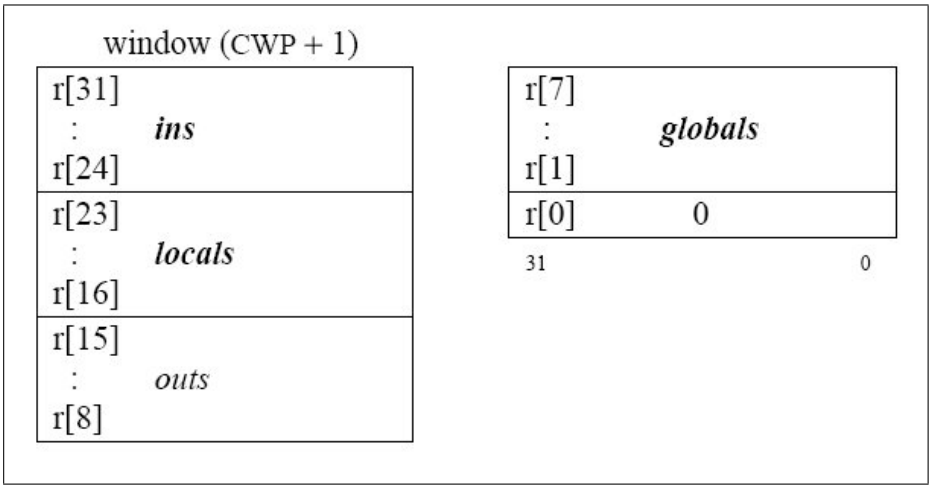

Abbildung B.2: Register des LEON3 mit globalen Registern aus [sparcv8spec]

Abbildung B.2 zeigt alle Register des LEON3. Links 24 Register zur Bildung der Register-Fenster. Rechts die 8 globalen Register.
Besitzt der LEON3 mehrere Register-Fenster, so überlappen sich die „ins“ Register, des Register-Fensters mit der Nummer (CWP mod nwindows), mit den „outs“ Registern des Register-Fensters ((CWP+1) mod nwindows).

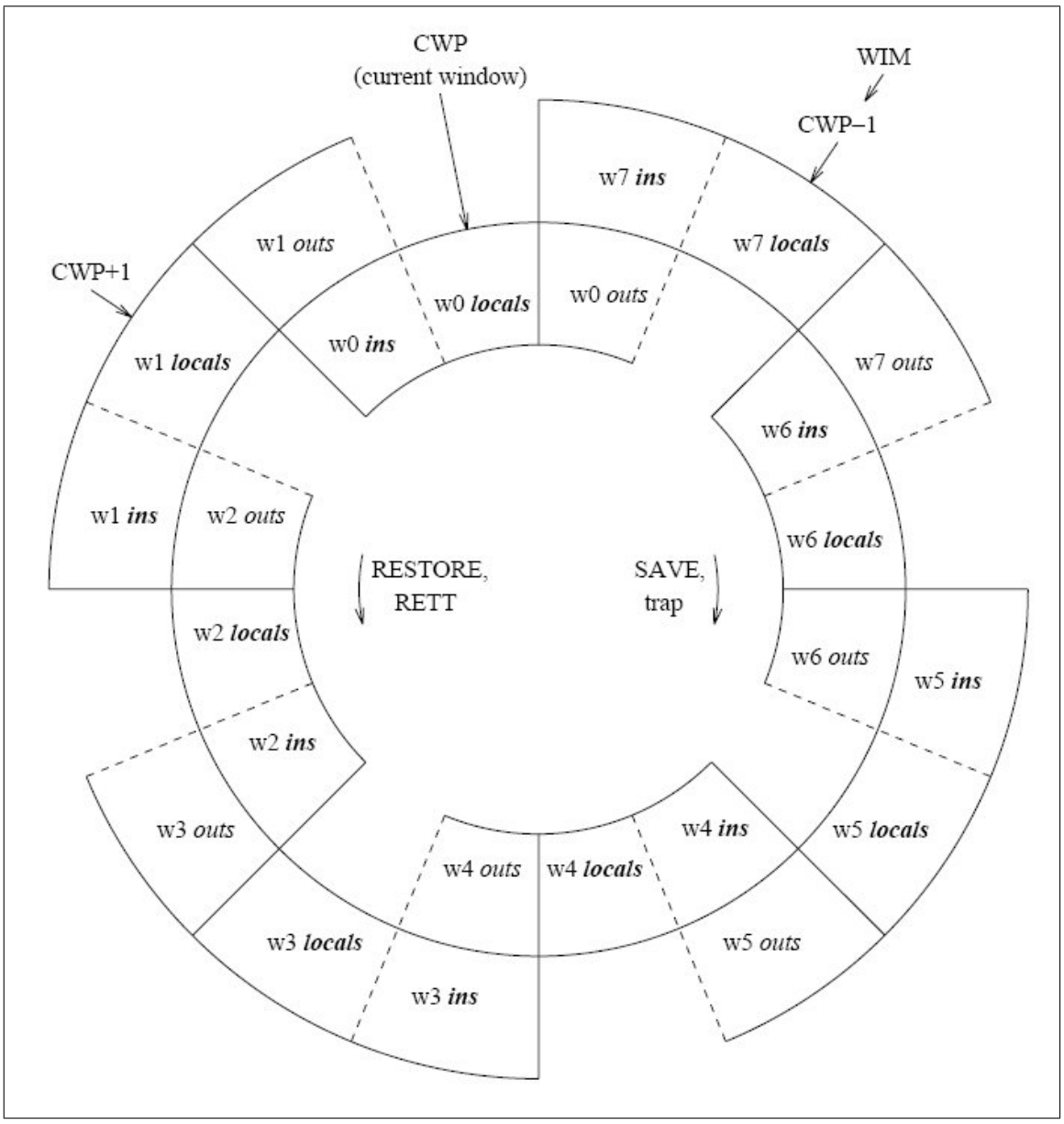

Abbildung B.3: Register-Fenster Struktur des LEON3-Prozessors bei 8 Register-Fenstern [spracv8spec]

Abbildung B.3 zeigt die Struktur der Register-Fenster bei „nwindows = 8“. Wenn CWP das aktuelle Register-Fenster ist und der LEON3 führt ein RESTORE aus so wird das Register-Fenster mit der Nummer CWP+1 das aktuelle Register-Fenster. Bei einer STORE Ausführung wäre CWP-1 das neue aktuelle Register-Fenster. Dadurch können dem LEON3 pro Register-Fenster effektiv 16 neue Register hinzugefügt werden.

Die gesamte Registeranzahl des LEON3 berechnet sich:

Gesamtanzahl der LEON3-Register = $16 Register \cdot nwindows + 8(globale\ Regsiter)$

Die Gesamtanzahl der LEON3-Register kann zwischen 40 und 520 in Schritten von 16 Registern variiert werden. Durch die Überlappung der Register-Fenster ist ein effizienter Aufruf von Unter-

prozeduren oder Unterprogrammen möglich.

B.2.1.3 Instruktion Pipeline

Das Ziel der Instruktion Pipeline ist, in jedem Takt einen Befehl abzuarbeiten. Dies gelingt allerdings nicht standardmäßig, da ein Befehl oft mehrere Taktzyklen benötigt, um vollständig bearbeitet zu werden. Erst nach der Bearbeitung des aktuellen Befehls kann der nächste Befehl, von der IU, abgearbeitet werden.
Mit einer Instruktion Pipeline wird ausgenutzt, dass die Befehlsabarbeitung über mehrere Stationen der IU durchgeführt wird. So können mehrere Befehle stückweise überlappend ausgeführt werden und im besten Fall wird ein Befehl pro Takt fertig bearbeitet.

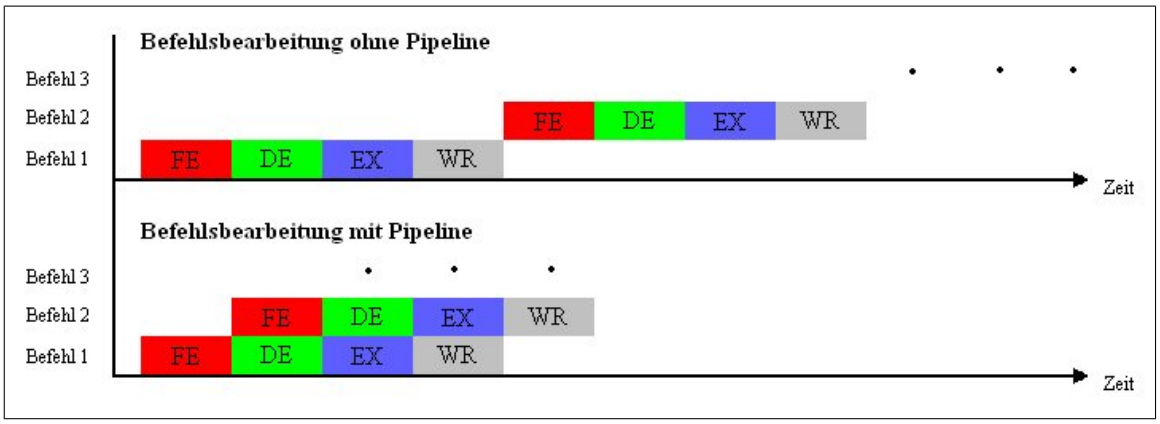

Abbildung B.4: Befehlsbearbeitung ohne/mit Pipeline

Abbildung B.4 vergleicht den zeitlichen Fortschritt der Befehlsbearbeitung ohne Pipeline und mit Pipeline. Die Pipeline in Abbildung B.4 hat 4 Stufen (FE (Instruction Fetch), DE (Decode), EX (Execute), WR (Write)).
Die Instruktion Pipeline des LEON2 war 5-stufig. Bei der Entwicklung des LEON3 wurde die Pipeline auf 7 Stufen erweitert. Durch diese Erhöhung werden im besten Fall effektiv weniger Takte, zur Abarbeitung eines Befehls, benötigt. Damit der Einsatz eine 7 stufigen Pipeline möglich wird, muss die Abarbeitung eines Befehls in sieben Teilschritte zerlegt werden. Diese Teilschritte sind die Bearbeitungsstationen der IU.

Der LEON3 realisiert eine IU-Zerlegung in folgende 7 Schritte:

Schritt	Beschreibung
1. FE (Instruction Fetch)	Die nächste Instruktion wird aus dem Cache oder aus dem RAM gelesen. Am Ende des Teilschritts steht die Instruktion bereit.
2. DE (Decode)	Die Instruktion wird interpretiert und für CALL- oder Branch-Befehle werden die Zieladressen generiert.
3. RA (Register access)	Die nötigen Operanten werden aus den Registern gelesen.
4. EX (Execute)	ALU, logische und shift Operationen werden jetzt umgesetzt. Für Speicher- und für RESTORE Operationen werden die Adressen generiert.
5. ME (Memory)	Alle Operationsergebnisse, des EX Teilschritts, werden in den Daten-Cache zurückgeschrieben.
6. XC (Exception)	Interrupts werden aufgelöst und gelesene Cache-Daten werden mit Operanten abgeglichen.
7. WR (Write)	Die Operationsergebnisse werden in den zugehörigen Registern abgespeichert.

Tabelle B.1: Die 7 Stufen der LON3 Pipeline

Bei der Pipeline des LEON2 waren die Teilschritte ME und XC noch nicht getrennt und der Teilschritt RA existierte nicht, da dieser Schritt mit dem Teilschritt WR erledigt wurde. Dadurch kam der LEON2 auf eine 5-stufige Pipeline.
Die Realisierung der Pipeline ist in der Entity „iu3“ des LEON3 VHDL Codes einzusehen. An der konkreten Umsetzung der Pipeline ist zu erkennen, dass die 7. Stufe der Pipeline auskommentiert ist, da die Umsetzung mit dem RA Teilschritt verbunden ist. Ob dem WR Teilschritt eine eigenständige Stufe effektiv zugeteilt wurde konnte in dieser Arbeit nicht festgestellt werden.

Instruktion	Takte (ohne MMU)	Takte (mit MMU)
JMPL, RETT	3	3
Double load	2	2
Single store	2	4
Double store	3	5
SMUL/UMUL	4*	4*
SDIV/UDIV	35	35
Taken Trap	5	5
Atomic load/store	3	5
All other instructions	1	1

*gilt für die „ungepipete“ MUL32-Schaltung

Tabelle B.2: benötigte Takte zur Abarbeitung einer Befehlsklasse [gripdoc]

Tabelle B.2 zeigt eine Übersicht aller Befehlsklassen des LEON3-Prozessors. In deren Abhängigkeit werden die zur Abarbeitung benötigten Taktzyklen angegeben.

B.2.1.4 Co-Prozessoren

Die SPARC spezifiziert zwei optionale Co-Prozessor-Interfaces für den LEON3. Die IU steuert den Einsatz der Co-Prozessoren je nach abzuarbeitenden Befehl. Ein Co-Prozessor-Interface kann mit einer FPU ausgestattet werden, das andere Interface mit einem benutzerdefinierten Co-Prozessor. Das Interface des benutzerdefinierten Co-Prozessor wird mit der Generic-Konstante „**cp**“ aktiviert oder deaktiviert.

B.2.2 Floating-Point Unit (FPU)

Eine **F**loating-**P**oint **U**nit (kurz FPU) muss laut SPARC an den LEON3 angeschlossen werden können. Die FPU muss einen vorgeschriebenen Satz von Befehlen, zur Gleitkommaberechnung, bearbeiten können. Die Ergebnisse der Berechnungen werden in den FPU-Registern zur Verfügung gestellt. Die FPU läuft parallel zu den Berechnungen der IU und wird durch diese gegebenenfalls aktiviert.
Eine FPU ist in der GRLIB nicht vorhanden. Die momentan existierenden FPUs müssen bei GR angefordert werden. Diese sind allerdings nur gegen eine Gebühr erhältlich.

Für den LEON3 existieren folgende FPUs:

Gaisler Research's Floating-Point Unit (kurz GRFPU)

Diese FPU ist eine Hochleistungs-FPU und unterstützt single- und double-precision. Sie wird direkt über einen FPU-Controller, an die Pipeline der IU, angeschlossen. Dadurch kann die FPU ihre Gleitkommaberechnungen simultan zur IU ausführen.
Die GRFPU ist „gepipet" und kann mit jedem Takt einen Befehl abarbeiten. Die FDIV- und FSQRT-Befehle (siehe Tabelle B.4) stören diese Pipeline nicht, da sie in einer parallelen Schaltung verarbeitet werden.

GRFPU-Lite

Die GRFPU-Lite ist eine kleinere Version der GRFPU. Sie ist weniger leistungsfähig und verfügt über keine Pipeline. Diese FPU arbeitet erst einen Befehl ab, bevor sie den nächsten beginnt.
Sie ist viel kleiner als die GRFPU und besonders für FPGAs mit stark begrenzten Ressourcen geeignet.

Meiko FPU

Die Meiko FPU ist eine einfache Schaltung mit einfacher und doppelter Präzision. Sie arbeitet nach dem SPARC „trap" Modell und kann nicht parallel zur IU laufen. Ihr Nachteil ist also, dass die IU stehen bleibt, wenn eine Gleitkommaoperation ausgeführt wird.

Die Wahl der FPU wird mit der LEON3 Generic-Konstante „**fpu**", von Typ Integer, festgelegt. Dabei gelten die Assoziationen:

„fpu"-Wert	**Bedeutung**
0	keine FPU
1 bis 7	GRFPU
8 bis 14	GRFPU-Lite
15	Meiko

Tabelle B.3: „fpu" Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

Die „fpu“-Werte aus Tabelle B.3 müssen, je nach der gewünschten Bedeutung, vor der Synthese des LEON3, als Integer-Wert der Generic-Konstante „fpu“ festgelegt werden. Für die genauere Erläuterung der „fpu“-Werte, ist in der Dokumentation [gripdoc] nachzuschlagen.

Die folgende Tabelle B.4 zeigt eine Übersicht aller SPARC V8 Befehle zur Gleitkommaberechnung und die Latenzzeiten in Takten der GRFPU und GRFPU-Lite für die jeweiligen Operationen:

Operation	**Latenzzeit**	
	GRFPU	GRFPU-Lite
FADDS, FADDD, FSUBS, FSUBD,FMULS, FMULD, FSMULD, FITOS, FITOD, FSTOI, FDTOI, FSTOD, FDTOS, FCMPS, FCMPD, FCMPES, FCMPED	4	8
FDIVS	16	31
FDIVD	17	57
FSQRTS	24	46
FSQRTD	25	65

Tabelle B.4: Latenzzeiten der GRFPU und GRFPU-Lite [gripdoc]

„fpu100“

Auf der Homepage des OpenCores-Projekts[2] ist eine freie, funktionstüchtige FPU in der Sprache VHDL erhältlich. Diese FPU könnte, wenn benötigt, für den LEON3 angepasst werden, um so die Lizenzkosten der GRFPU zu sparen. Diese FPU befindet sich im CVS-Verzeichnis „fpu100“.
Die „fpu100“ ist bis 100MHz taktbar und verfügt über die Befehle Addition, Subtraktion, Multiplikation und Division mit jeweils zwei 32 Bit Operanden. Zusätzlich kann die Wurzel aus einem Operanden gezogen werden. Die „fpu100“ wurde nicht für die speziellen SPARC V8 FPU Gleitkommaoperationen programmiert, könnte aber verhältnismäßig leicht mit fehlenden SPARC V8 Gleitkommaoperationen erweitert werden.

[2] http://www.opencores.org

Operation	Latenzzeit
Addition	7
Subtraktion	7
Multiplikation	12
Division	35
Square-root	35

Tabelle B.5: Latenzzeiten der „fpu100“ [fpu100doc]

Die Tabelle B.5 zeigt die Latenzzeiten in Taktzyklen der „fpu100“.

B.2.3 Power-down Modus

Der Power-down Modus ist eine Funktionalität des LEON3-Prozessors, um den Energiebedarf in bestimmten Modi herabzusetzen. Während sich der LEON3 in diesem Modus befindet, wird die Pipeline angehalten bis ein Interrupt geworfen wird. Der Power-down Modus kann mit der Generic-Konstante „**pwd**“ festgelegt werden.

Für den Wert der Konstante gelten folgende Assoziationen:

„pwd“ Wert	Bedeutung
0	Modus deaktiviert
1	Modus platzsparend aktivieren
2	Modus timingeffizient aktivieren

Tabelle B.6: „pwd“ Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

B.2.4 Multi-Prozessor Unterstützung

An einem AHB können bis zu 16 LEON3-Prozessoren verbunden sein. Wichtig ist dabei, dass alle Prozessoren ihre eigene „hindex“-Generic-Konstante besitzen müssen.
Geht ein System mit mehr als einem LEON3 in Betrieb, so befinden sich alle Prozessoren, bis auf den mit dem Index 0, im Power-down Modus. Dies gilt nur, falls der Power-down Modus für den entsprechenden LEON3-Prozessor synthetisiert wurde.
Der erste LEON3 des SoC (Index 0) beginnt mit der gewöhnlichen Abarbeitung seiner Instruktionen an der definierten Startadresse. Die Startadresse, an der ein LEON3 seine ersten Instruktionen sucht, lässt sich mit der Generic-Konstante „**rstaddr**“ der LEON3-Entity festlegen.

Alle übrigen LEON3-Prozessoren können mittels des MP Status Registers (Multiprozessoer Status Register), aus dem Power-down Modus geweckt werden. Die geweckten Prozessoren beginnen ebenfalls ihre Instruktionen an der Adresse zu suchen, die mit den jeweiligen „**rstaddr**“-Generic-Konstanten festgelegt wurde.

Die Multi-Prozessor Unterstützung muss im Falle eines Mehrprozessorsystems bei jedem einzelnen LEON3 aktiviert werden. Dazu muss die Generic-Konstante „**smp**“ (synchronous multiprocessing) für jeden LEON3 mit einem Wert ungleich 0 festgelegt werden.
Bei einer Verwendung von mehr als einem LEON3 auf einem AHB sollte das Daten-Cache Snooping aktiviert werden.

Zu Beachten B.1 (Interrupt Broadcast für SnapGear-Linux 2.6)

*Um das „SnapGear“-Linux 2.6 auf einem Multi-Prozessor System ausführen zu können, muss die „Broadcast“ Funktionalität des Interrupt-Controllers aktiviert werden, Diese Funktionalität wird mit der „**broadcast**“-Generic-Konstante des „irqmp“ IP-Cores eingestellt. Die Generic-Konstante „broadcast“ wird durch ein Tcl-Skript automatisch festgelegt.*

B.2.5 Daten-Cache Snooping

Befinden sich mehrere LEON3-Prozessoren an einen AHB, so verfügt jeder einzelne Prozessor über seinen eigenen Daten-Cache. Bearbeitet ein LEON3 einen Befehl, der zu einer Veränderung seines eigenen Daten-Cache führt, so sind diese Daten inkonsistent zu den Inhalten der Daten-Caches der anderen LEON3-Prozessoren. Um diese Inkonsistenz der Daten-Caches zu beheben muss das Daten-Cache Snooping bei jedem einzelnen LEON3 aktiviert werden.

Das Grundprinzip des Datan Snooping ist folgendes:

Legt ein Prozessor eine Adresse, zum Lesen oder zum Schreiben an den AHB an, so bemerken dies die anderen Prozessoren, wenn sie sich im Snooping Zustand befinden. Die LEON3-Prozessoren im Snooping Zustand, vergleichen nun ob es sich bei der beschriebenen Adresse um eine Adresse handelt die selbst in ihrem Daten-Cache vorliegt. Ist dies der Fall so aktualisieren die anderen Prozessoren dieses Datum in ihren eigenen Caches. So wird die Inkonsistenz des Daten-Caches vermieden.
Um die Snooping Funktionalität eines LEON3-Prozessors zu aktivieren, muss die Generic-Konstante „**dsnoop**“ jedes einzelnen LEON3 eingestellt werden. Die „**dsnoop**“ Konstante ist von Typ Integer von 0 bis 6, dies ergibt eine 3 Bit Konstante.

Für die „**dsnoop**“ Konstante sind folgende Assoziationen vorgesehen:

„**pwd**“ **Wert**	Bitvektor des Werts	**Bedeutung**
Bit 0 bis 1 ergeben 0	0b-00	Snooping deaktiviert
Bit 0 bis 1 ergeben 1	0b-01	Langsames Snooping
Bit 0 bis 1 ergeben 2	0b-10	Schnelles Snooping
disjuktiv kombiniert mit:		
Bit 2 ergibt 0	0b0- -	Einfaches Snooping (ohne MMU)
Bit 2 ergeben 1	0b1- -	MMU Snooping (mit MMU)

Tabelle B.7: „dsnoop“ Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

Möchte der Benutzer beispielsweise ein schnelles Snooping für einen LEON3 aktivieren, in dem eine MMU vorhanden ist, so muss er „dsnoop = 6“ setzen, was einem Bitvektor von 0b110[3] entspräche. Ist dagegen ein langsames Snooping mit geringem Platzbedarf ausreichend und darüber hinaus keine MMU vorhanden, so muss der Benutzer „dsnoop = 1“ setzen. Dies entspräche einem Bitvektor von 0b001.

B.2.6 Memory Managment Unit (MMU)

Die **M**emory **M**anagement **U**nit (kurz MMU) ist eine Speicherverwaltungseinheit, die für den Zugriff auf den Cache-Speicher, Hauptspeicher und andere Hardwarespeicher, die Umwandlung von virtuellen Adressen in physikalische Adressen durchführt.
Mikroprozessoren können, aufgrund ihres geringeren Programmumfangs auf eine MMU verzichten. Der MicroBlaze-Prozessor besitzt zum Beispiel keine MMU.
Die MMU sollte allerdings für das Betreiben von Betriebssystemen aktiviert werden, wenn dies nicht sogar vorgeschrieben ist. Möchte ein Benutzer ein „SnapGear“-Linux mit dem Kernel 2.6 auf dem LEON3 ausführen lassen, so wird unbedingt eine MMU benötigt.

Da die optionale Wahl einer MMU für den LEON3 neue interessante Einsatzmöglichkeiten bietet, wird die MMU der SPARC Spezifikation in diesem Kapitelpunkt genauer betrachtet.

Die LEON3 MMU besitzt laut SPARC Spezifikation folgende wichtige Eigenschaften:

- interpretiert 32 Bit virtuelle Adressen
- wandelt in 36 Bit physikalische Adressen um
- hat eine feste Seitengröße von 4 kByte

[3] Beginnt eine Zahl mit der Zeichenkette „0b“ so ist sie zur Basis 2 dargestellt

Die primären Funktionen einer SPARC V8 MMU sind folgende:

1. Umwandeln von virtuellen Adressen in physikalische Adressen. Das Speicher Mapping wird dabei mit Seiten vorgenommen, die eine feste Größe von 4 kByte haben.
2. Schützen von Speicherbereichen vor Schreib- oder Lesezugriffen. Dies ist wichtig für Betriebssysteme, die eine Trennung von Prozessen vornehmen und den User-Space verwalten.
3. Die MMU verwaltet den virtuellen Speicher. Sie führt einen „**t**ranslation **l**ookaside **B**uffer“ (kurz TLB), der alle übersetzbaren Adressen beinhaltet. Befindet sich eine Adresse nicht darin, so muss der Page Miss erkannt werden und nach einer Ersetzungsstrategie die physikalische Adresse eingelagert werden.

Der Vorteil des LEON3 ist, dass er optional über eine MMU verfügen kann oder nicht. Dazu muss lediglich die Generic-Konstante „**mmuen**“ der LEON3-Entity festgelegt werden.

Folgende Assoziationen stehen bereit:

„**mmuen**“ Wert	**Bedeutung**
0	keine MMU
1	MMU einbinden und benutzen

Tabelle B.8: „mmuen“ Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

Die MMU des LEON3 übernimmt, mit ihrer Einbindung, zusätzlich die Rolle des Cache-Controllers. Die MMU verwaltet also parallel Zugriffe auf Programm- und Daten-Cache. Bei einer Deaktivierung der MMU, wird ein alternativer Cache-Controller in den LEON3 eingebunden. Dieser Vorgang ist in der „proc3“-Entity ab der Zeile 152 zu erkennen.

Zu Beachten B.2 (Grösse der LEON3 MMU)

Es sei darauf hingewiesen, dass die MMU des LEON3, bei entsprechender Konfiguration, ca. 1200 bis 3200 Slices zusätzlich benötigt.

B.2.6.1 MMU im Gesamtkontext

Die folgende Abbildung B.5 zeigt ein Blockschaltbild des LEON3-Prozessors mit MMU:

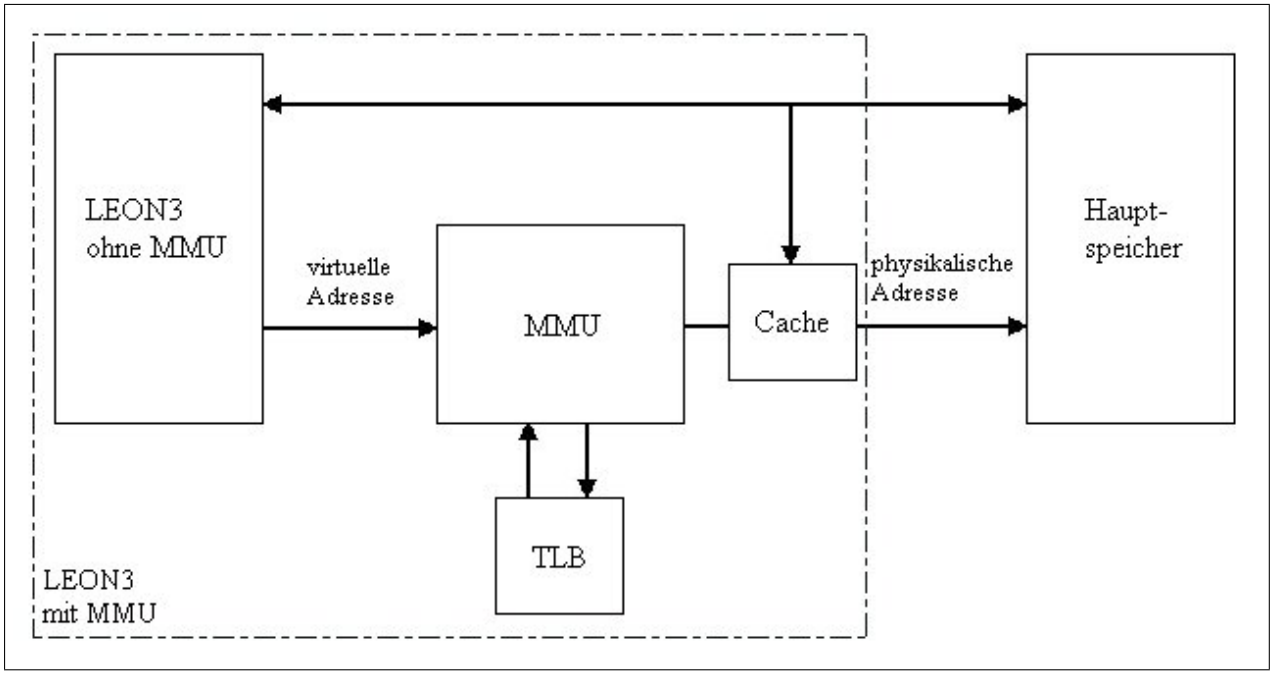

Abbildung B.5: Blockschaltbild des LEON3 mit MMU

Es ist zu erkennen, dass die MMU lediglich die Adressen des CPU umwandelt und an den AHB legt. Die Daten werden zwischen LEON3 und AHB direkt ausgetauscht.
Die SPARC-Spezifikation definiert neben dem herkömmlichen 32 Bit Adressen, die einen Adressraum von 4 GByte ermöglichen, weitere 8 Bit, die 256 weitere separate Adressräume schaffen. Der LEON3 benutzt von diesen 8 Bit nur 4 Bit. Dadurch entsteht eine physikalische Adresse von 32 + 4 = 36 Bit.

B.2.6.2 Konfiguraton

Die möglichen Konfigurationen in der sich eine MMU des LEON3 befinden kann, sind spezielle Einstellungen des TLB. Es muss ein TLB in Verbindung mit der MMU benutzt werden. Der TLB speichert alle in der Vergangenheit umgewandelten Adressen. Die MMU ist somit in der Lage die Adresse schneller an den AHB anzulegen.
Es können die TLB Einstellungen **Typ des TLB**, **Größe des TLB** und **Ersetzungsstrategie** vorgenommen werden.

Typ des TLB

Für jeden Cache kann ein eigener TLB eingerichtet werden. Dies wird mit der Generic-Konstante „**tlb_type**“, der LEON3-Entity festgelegt. Wird der Wert der Konstante auf 0 gesetzt so wird für jeden Cache ein eigener TLB benutzt. Ist der Wert dagegen 1 so wird nur ein TLB, für beide Caches, angelegt. Der TLB wird dabei entweder in zwei („tlb_type = 0“) oder einem („tlb_type = 1“) BRAM gespeichert. Die Entscheidung, für die Anzahl der BRAMs, kann in der Entity „mmu“ ab Zeile 559 beobachtet werden.

Zu Beachten B.3 (MMU TLB Typ)

Da die MMU die Steuerung des Cache übernimmt, kann es bei der Verwendung eines einzelnen TLB („tlb_type = 1“) zu folgender Verzögerung kommen.
Legt der LEON3 eine virtuelle Adresse an die MMU so versucht sie diese Adresse im TLB zu finden. Ist kein Eintrag im TLB vorhanden (TLB miss) so können mehr Takte benötigt werden, um die virtuelle Adresse umzuwandeln.
Wenn nun ein einzelner TLB benutzt wird, so muss die Ansteuerung des parallelen Caches stoppen, bis die virtuelle Adresse übersetzt ist. Der TBL ist also in Benutzung und kann erst wieder freigegeben werden, wenn die Adresse vollständig umgewandelt ist.
Würde die MMU dagegen zwei getrennte TLBs besitzen („tlb_type = 0“), so könnte eine parallele Cache-Ansteuerung auch bei einem „TLB miss“, des einen TLB, erfolgen. Es würde also ein Geschwindigkeitsvorteil entstehen.

Größe des TLB

Die Größe einer TLB wird in Einträgen angegeben. Ein Eintrag umfasst dabei die Umwandlung einer virtuellen Adresse mit Verwaltungsdaten.
Für die Größeneinstellungen stehen die Generic-Konstanten „**itlbnum**“, für die Einträgeanzahl der Instruktion-TLB (Programm-TLB) und „**dtlbnum**“, für die Einträgeanzahl der Daten-TLB, zur Verfügung. Es kann für jeden TLB eine Anzahl von 2 bis 64 Einträgen gewählt werden. In der Dokumentation [gripdoc] wird von 2 bis 32 Einträgen gesprochen. Diese Angabe ist jedoch falsch, was in der „leon3s“-Entity, ab der Zeile 75, und in allen anderen Unter-Entitys überprüft werden kann. Bei der Synthese zeigen sich ebenfalls mehr als 32 Einträge als möglich.
Wird für die MMU ein gemeinsamer TLB („tlb_type = 1“) gewählt so ist nur die Konstante „itlbnum“ für die Größe des TLB relevant.

Ersetzungsstrategie

Ist die Kapazität eines TLB erschöpft, da entsprechend viele virtuelle Adressen umgewandelt und gespeichert wurden, so werden die Einträge des TLB nach einer bestimmten Ersetzungsstrategie überschrieben. Folgende Strategien können mittels der Generic-Konstante „**tlb_rep**“ der LEON3-Entity gewählt werden:

„tlb_rep“ Wert	Bedeutung
0	zufällig gewählte Einträge werden überschrieben
1	der am längsten nicht verwendet Eintrag wird überschrieben (LRU)

Tabelle B.9: „tlb_rep“ Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

Die im Kapitelpunkt B.2.6 aufgeführten Einstellungsmöglichkeiten der MMU haben erheblichen Einfluss auf die Größe des LEON3. Siehe Zu Beachten B.2.

B.2.7 Cache-System

Wie zu Beginn des Kapitelpunktes B.2 erwähnt, besitzt der LEON3 zwei Caches, jeweils einen Programm- und Daten-Cache. Durch diese zwei Cache-Speicher auf die parallel zugegriffen werden kann, wird der LEON3 zu einer Art Harvard-Architektur. Den daraus resultierenden Geschwindigkeitsvorteil kann der LEON3 erst ausschöpfen, wenn die Caches nach vielen Buszugriffen gefüllt sind.

Ein großer Unterschied zum Cache-Verwalten eines MicroBlaze-Systems ist, dass jeder Zugriff auf den Systemspeicher prinzipiell „gecachet“ werden kann. Es ist also nicht relevant welche SoC-Komponente sich hinter dem angesprochenem Systemadressbereich befindet oder, ob der Zugriff auf Register oder auf externem Speicher stattfindet. Alle Bus-Transfers können gecachet werden.
Bei einem MicroBlaze-System muss dagegen die zu cachende SoC-Komponente speziell konfiguriert werden. Zusätzlich wird eine spezielle „Initiator-Target“-Verbindung zwischen dem zu cachenden IP-Core und dem MicroBlaze-Prozessor eingerichtet.
Diese Verbindung besitzt den Busstandard „XIL_MEMORY_CHANNEL“. Dieses Vorgehen hat den Nachteil, dass nur der Systemspeicher des zucachenden IP-Cores gecachet werden kann und andere Zugriffe nicht.

B.2.7.1 Konfiguration

Der Programm- und Daten-Cache, des LEON3-Prozessors, können aus jeweils 1 bis 4 Sets bestehen. Je Set ist eine Größe von 1 bis 256 kByte möglich. Insgesamt ist also eine Cachegröße von 1

MByte je Programm- und Daten-Cache erlaubt. Dies reicht zum vollständigen cachen von größeren Programmen und kleineren Betriebsystemen aus. Um einen LEON3 mit beliebiger Cache-Größe zu synthetisieren, werden allerdings beliebig viele BRAM benötigt, zusätzlich wird eine erhebliche Rechenkapazität des Host-PC gebraucht.

Die Anzahl der Programm-Cache-Sets lässt sich mit der Generic-Konstante „**isets**“ festlegen und analog die des Daten-Caches mit der „**dsets**“ Konstante der LEON3-Entity. Die Größe der Sets konfiguriert der Benutzer mit der „**isetsize**“, für den Programm-Cache, und „**dsetsize**“ für den Daten-Cache.
Die Anfangsbuchstaben der Generic-Konstanten stehen für den jeweiligen Cache, „i“ für Programm (Instruktion)-Cache und „d“ für Daten-Cache.

Zu Beachten B.4 (LEON3 Cache Pagegrösse mit MMU)

Die MMU des LEON3 handelt mit einer Pagegröße von maximal 4 kByte. Wird die MMU aktiviert so müssen die Größen der Cache-Sets kleiner gleich 4 kByte sein. [gripdoc]

Jeder Cache-Set wird in Lines unterteilt. Eine Cache-Line ist die kleinste Einheit, in der Daten in den Cache eingelagert oder in den Hauptspeicher zurückgeschrieben werden können.
Die Größe einer Cache-Line kann mit den Generic-Konstanten „**ilinesize**“, für eine Programm-Cache-Line und mit "**dlinesize**", für eine Daten-Cache-Line festgelegt werden. Die Größe wird in der Einheit „Wort“ angegeben. Es sind Cache-Line-Größen von 16 bis 32 Bytes möglich, das entspricht 4 bis 8 Worten.

Wenn die Kapazität eines Cache-Sets erschöpft ist wird mit der gewählten Überschreibungsstrategie eine alte Cache-Line gelöscht. Die Strategie wird mit den Konstanten „**irepl**“ oder „**drepl**“ eingestellt.

Folgende Strategien stehen zur Auswahl:

„irepl“, „drepl“ Wert	Bedeutung
0	LRU
1	LRR
2	zufällige Wahl einer Cache-Line

Tabelle B.10: „irepl“, „drepl“ Werte der LEON3-Entity und ihre Bedeutung [gripdoc]

Soll es möglich sein innerhalb eines Caches eine Line zu markieren, so dass sie nicht überschrieben werden kann (Line-Lock), so muss diese Möglichkeit, für den jeweiligen Cache, zuvor aktiviert

werden. Dies wird durch die LEON3 Generic-Konstanten „**isetlock**“ und „**dsetlock**“, vom Typ Integer, festgelegt. Dabei steht 0 für Line-Locking deaktiviert und 1 für eine Aktivierung.

B.2.7.2 Programm-Cache

Der Programm-Cache, auch Instruktion-Cache genannt, wird während eines Instruktion-Bursts gefüllt. Initiiert der LEON3-Prozessor einen Transfer zum Lesen einer Instruktion aus dem System-Speicher. So treibt er seinen Ausgangsport „ahbmo.hport(3 downto 0)“ mit dem Signalvektor 0b---0[4] [ambaspec]. Ist also „hport(0) = ’0’“, so kennzeichnet dies, dass der LEON3-Prozessor eine Instruktion lesen will. Zeitgleich wird erkannt, dass Daten aus dem Programm-Cache gelesen werden könnten, wenn sie im Cache vorhanden wären.

Wird ein „Cache miss“ festgestellt, so wird eine Cache-Line aus dem Systemspeicher in den Programm-Cache geschrieben. Der Schreibvorgang beginnt an der Systemadresse, an welcher der „Cache miss“ festgestellt wurde. Gleichzeitig werden die gelesenen Instruktionen wenn benötigt an die IU weitergeleitet („streaming“ genannt).

Angenommen die IU erhält ihre angeforderte Instruktion zu Beginn des Einlesens der Cache-Line und versucht bereits im nächsten Takt wieder eine Instruktion aus dem Systemspeicher zu lesen. In diesem Fall würde die Pipeline der IU angehalten werden, bis das Einlesen der momentanen Cache-Line abgeschlossen ist. Aus diesem Grund ist es vorteilhaft einen größtmöglichen Programm-Cache, in den LEON3, einzubinden, da umso seltener „Cache misses“ auftreten können.

B.2.7.3 Daten-Cache

Der Daten-Cache besitzt die gleiche Arbeitsweise, wie der Programm-Cache. Die Benutzung des Daten-Cache erfolgt allerdings durch das Treiben des „ahbmo.hport“ Ports mit dem Signal 0b---1. Für den Zugriff auf den Daten-Cache verwendet der LEON3 das „Write-through“ Verfahren. Bei diesem Verfahren werden Cachedaten bei Veränderung sofort in den Hauptspeicher zurückgeschrieben, so dass sie dort immer aktuell sind. Das selbe Verfahren wird auch beim MicroBlaze verwendet.
Ein Anderes Verfahren wird zum Beispiel „Write-back“ genannt. Hier werden die Daten nur im Cache verändert und zusätzlich ein „Dirty-Flag“ gesetzt, dadurch sind die Daten als inkonsistent gegenüber dem Hauptspeicher markiert. Zu einem späteren Zeitpunkt werden die Daten in den Hauptspeicher zurückgeschrieben.

Der Daten-Cache verfügt über drei zusätzliche 32 Bit Register und besitzt spezielle Verhaltensanpassungen für Snooping-Operationen in einem **M**ulti **P**rozessor **S**ystem (kurz MPS). Diese Verhaltensanpassungen, im Snooping Fall, oder ein „Write-through“ Zugriffsverfahren sind für den

[4] Die mit „-“ markierten Signale für diese Betrachtung nicht relevant („Dont-care“).

Programm-Cache nicht nötig, da Instruktionsdaten nicht von einzelnen Prozessoren geschrieben sondern nur gelesen werden.

B.2.7.4 Cache-Flush

Ein „Cache-Flush“ ist das Zurückschreiben des gesamten Cache Inhaltes in den Hauptspeicher. Dies wird immer dann durchgeführt, wenn die Speicherinhalte von anderen SoC-Komponenten benötigt werden, da andere SoC-Komponenten nicht im Daten-Cache des LEON3 lesen können. Das Flushing des Caches benötigt einen „Cache-Line Transferzyklus“[5] pro Cache-Line. Währenddessen sind die Caches des LEON3 deaktiviert. Die Pipeline der IU wird jedoch nicht zwingend angehalten. Sollte die IU allerdings Daten aus den Caches benötigen, so muss die Pipeline warten bis das Flushing abgeschlossen ist.

Für Anwendungen oder Operationen die häufige Cache-Flushs auslösen, wirkt es sich negativ aus, wenn der Cache-Speicher zu groß gewählt wurde. Für jede bestimmte Anwendung gibt es eine optimale Einstellung der Cache-Speichergröße, so dass selten „Cache misses“ auftreten und ein Cache-Flush nicht zuviel Transferzeit benötigt.

[5] Es konnte in dieser Arbeit nicht festgestellt werden wie viele Taktzyklen ein „Cache-Line Transferzyklus“ benötigt.

Anhang C

Das LEON3-GRLIB-System als eingebettetes System

In diesem Kapitel soll der LEON3-Prozessor und andere IP-Cores der GRLIB aus der Sicht der eingebetteten Systeme betrachtet werden.
Eingebettete **S**ysteme (kurz ES) sind elektronische Systeme, welche meist als Untersystem in ein größeres System eingeordnet sind. ES können ebenfalls aus Hardware- und Software-Anteilen bestehen. ES übernehmen im eigentlichen Sinn nur kleinere Teilaufgaben des gesamten Systems, wie das Kontrollieren von Sensoren oder das Ansteuern von Motoren.
Sie können allerdings je nach Flexibilität zu größeres Systemen anwachsen, auf denen beispielsweise ein Betriebssystem läuft und die Hardware eines gewöhnlichen PC benutzt werden kann. Der LEON3 bietet, durch seine umfangreiche Konfigurierbarkeit, diese flexiblen Möglichkeiten. Ein gutes Beispiel für eine zweckbedingte Konfigurationsmöglichkeit ist die optionale MMU. Für einfache ES mit Controller-Funktionen ohne Betriebssystem kann sie deaktiviert werden, dadurch werden logische Ressourcen gespart und können anderweitig eingesetzt werden.
Soll allerdings ein PC-ähnliches System konstruiert werden, auf dem ein komplexes Betriebssystem ausgeführt werden soll, so kann die MMU aktiviert werden.

ES unterliegen allerdings noch weiteren Anforderungen. Diese Anforderungen sind je nach Einsatzgebiet des ES unterschiedlich und müssen erfüllt werden.

Solche Anforderungen können beispielsweise sein:

Zuverlässigkeit und Fehlertoleranz Das ES soll seine Aufgabe unter allen Umständen ununterbrochen ausführen. Dies soll auch bei auftretenden Systemfehlern, durch äußere Einflüsse, gelten.
Robustheit Ein ES muss unter Umständen gewisse Behandlungen fehlerfrei überstehen. Diese Behandlungen können extreme physikalische Einflüsse wie Kräfte, Temperaturen, Strahlung, elektro-magnetische Felder oder Feuchtigkeit sein.
Wartbarkeit Meist unzugängliche ES müssen spezielle Anforderungen an ihre Wartbarkeit erfüllen. Dies ist durch den Einsatz von FPGAs im großen Maß erfüllt.
elektrische Eigenschaften Ein ES muss eventuell bestimmte elektrische Eigenschaften erfüllen. Diese können ein maximaler Stromverbrauch oder eine maximale Abwärmeleistung sein. Die Anforderungen an die elektrischen Eigenschaften müssen dann nicht nur im Betrieb erfüllt werden, sondern auch in jedem möglichen Fehlerfall.
Performance, Latenz und Echtzeitverhalten Das ES soll darüber hinaus mindest Anforderungen bezüglich seiner Performance erfüllen, eine maximale Latenzzeit zur Bereitstellung von Daten einhalten oder bei seinen Ausführungen Echtzeitbedingungen erfüllen. Unter Echtzeitverhalten ist das Einhalten von festgelegten Zeitschranken, bei der Programmausführung, gemeint. Solche Zeitschranken müssen unter allen Umständen eingehalten werden.

Tabelle C.1: typische Anforderungen an ein Eingebettetes System

Die Anforderungen aus Tabelle C.1 scheinen sehr streng, sind aber verständlicherweise gerechtfertigt, wenn ein ES in medizinischen Geräten, wie Herzschrittmachern, oder in Sicherheitseinrichtungen, wie ABS oder ESP eingesetzt werden soll. Wird eine gegebene Anforderung nicht durch das ES erfüllt werden, so kann dies zur Unverwendbarkeit des jeweiligen ES führen.

Ein anderes Beispiel für die unbedingte Zuverlässigkeit, ist der Einsatz eines ES in einem Satelliten. Dort kann die Reparatur eines ES nahezu unmöglich sein und extreme physikalische Bedingungen (Strahlung, Temperatur) müssen ausgehalten werden.
Der LEON3-Prozessor wird von der ESA in Satelliten eingesetzt. Dazu ist eine spezielle fehlertolerante Version des LEON3 verfügbar, auch genannt LEON3FT.

In den folgenden Kapitelpunkten soll die GRLIB, einschließlich des LEON3FT, betrachtet werden.

Es sollen Funktionalitäten der GRLIB gezeigt werden, welche ihre Möglichkeiten, zum Einsatz als ES erweitern.

C.1 Fehlertoleranz der GRLIB

Die Fehlertoleranz der GRLIB betrifft hauptsächlich sogenannte **S**ingle **E**vent **U**pset (kurz SEU) Fehler. Bei SEU-Fehlern handelt es sich um „soft errors“ welche einen Bit-Fehler (engl.: bitflip) bezeichnen. Solche Fehler können beim Durchgang ionisierender Strahlung durch Halbleiterelemente auftreten.
Ein anderer sogenannter „hard error“ ist der **S**ingle **E**vent **L**atch-up (kurz SEL) Fehler. Bei diesem Fehler werden Kurzschlüsse erzeugt, die gegebenenfalls zum Defekt des Halbleiterelementes führen können.

Die GRLIB bietet verschiedene IP-Cores an, welche eine bestimmte Fehlertoleranz gegenüber SEU-Fehlern besitzen. Zu diesen IP-Cores gehören, der:

IP-Core	
Bezeichung	**Name**
On-Chip SRAM mit EDAC	ftahbram
8/16/32-Bit Memory Controller mit EDAC	ftmctrl
32/64-Bit PC133 SDRAM Controller mit EDAC	ftsdctrl
Fehlertoleranter 32-Bit PROM/SRAM/IO Controller	ftsrctrl
8-Bit SRAM/16-Bit IO Memory Controller mit EDAC	ftsrctrl8
Fehlertoleranter LEON3 SPARC V8 Prozessor	leon3ft

Tabelle C.2: fehlertolerante IP-Cores der GRLIB [gripdoc]

Bei den IP-Cores der Tabelle C.2, bis auf den Prozessor, handelt es sich um Memory Controller oder On-Chip-Memory mit einer fehlertoleranten Datenübertragung. Um diese Funktionalität umzusetzen verfügen diese IP-Cores über eine „**E**rror **D**etection **A**nd **C**orrection“ Funktion (kurz EDAC).
Die EDAC-Funktion ermöglich das Korrigieren von einem Bit-Fehler und das Erkennen von zwei Bit-Fehlern. Die IP-Cores werden wie gewöhnliche SoC-Komponenten mit einem GRLIB-Bus verbunden und führen die EDAC-Funktionen während eines normalen Transfers aus.
Die EDAC-Funktion verwendet den BCH-Code zur Detektierung und Korrigierung von Bitfehlern. Der BCH-Code steht für **B**ose-**C**haudhuri-**H**ocquenghem-Codes (kurz BCH) und wurde nach den Entwicklern benannt.

Alle fehlertoleranten IP-Cores einschließlich des LEON3FT sind unter der speziellen „FT“-Lizenz erhältlich und werden nicht mit der normalen GRLIB ausgeliefert. Darüber hinaus werden die IP-Cores nur als Netzlisten veröffentlicht und nicht als frei zugängliche VHDL-Codes. In den Kapitelpunkten C.1.1 bis C.1.3 sollen die IP-Cores: „ftahbram“, „ftsdctrl“ und „leon3ft“ etwas genauer vorgestellt werden.

C.1.1 On-Chip SRAM mit EDAC („ftahbram“)

Der „ftahbram“-IP-Core ist eine fehlertolerante Version des „ahbram“-IP-Cores und besitzt die gleiche Hauptfunktionalität. Die Hauptfunktionalität ist das Speichern und Auslesen von Daten in On-Chip-Speichern. Im Fall der FPGAs wären dies die BRAMs.

Alle Transfers, die mit dem BRAMs ablaufen sind durch die EDAC-Funktion geschützt. Der „ftahbram“-IP-Core wird für den Daten-Transfer als Slave an den AHB angeschlossen. Ein zusätzliches APB-Slave-Interface ermöglicht die Konfiguration während der Laufzeit über Register. Die EDAC-Funktion kann mit der Generic-Konstante „**edacen**“ aktiviert oder vollständig aus dem Design entfernt (deaktiviert) werden. Wird die EDAC-Funktion entfernt so ist auch das APB-Slave-Interface, zur Konfiguration, überflüssig.

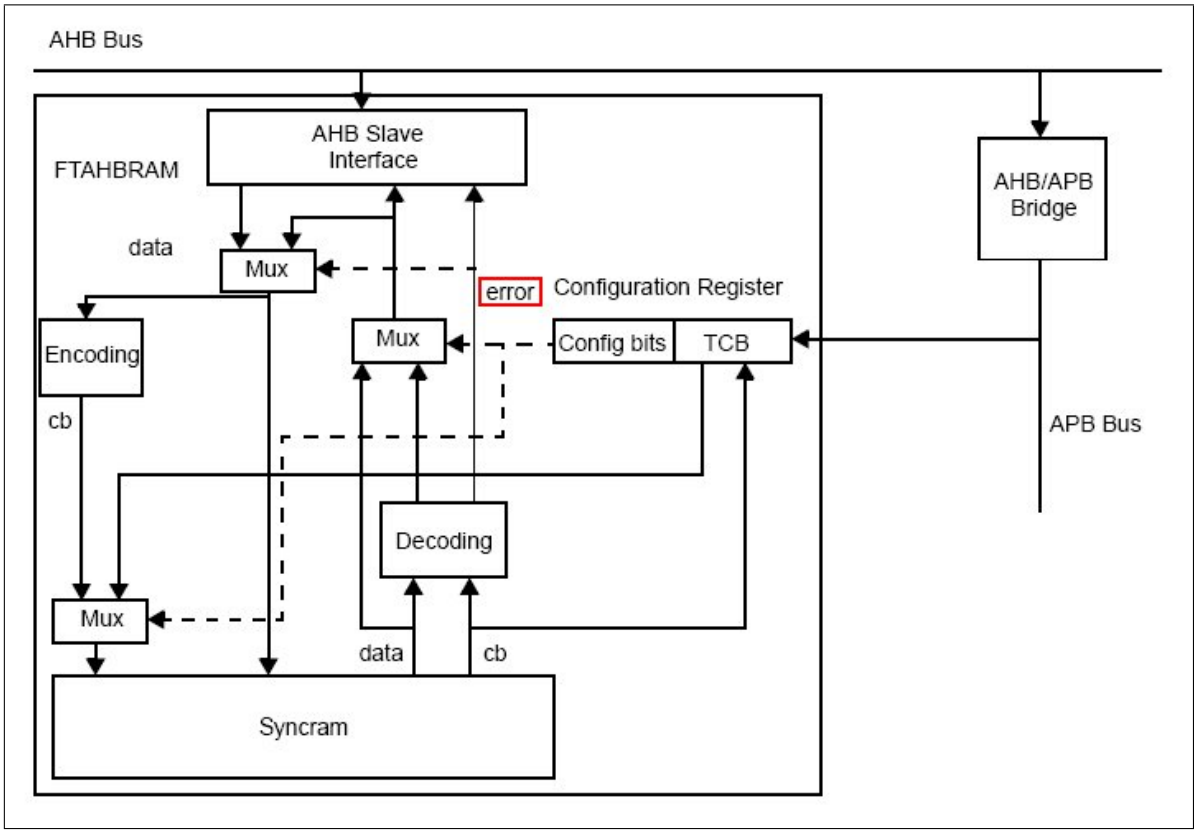

Abbildung C.1: Blockschaltbild des fehlertoleranten AHB RAM IP-Cores „ftahbram“ [gripdoc]

Abbildung C.1 zeigt ein Blockschaltbild des „ftahbram“-IP-Core. Es ist das zusätzliche APB-Slave-Interface zu erkennen, sowie die Logik zur Fehlerkorrektur.
Der IP-Core verfügt über einen separat getriebenen Port, der in der Abbildung C.1 mit „error“ gekennzeichnet ist (roter Rahmen). Dieser Port dient als direkte Anzeige eines erkannten Fehlers und kann durch das SoC als Interrupt für eine spezielle Fehlerroutine genutzt werden.

Zugriffe auf den RAM

Wird, über den AHB, ein gewöhnlicher Schreib-Transfer auf dem „ftahbram“ eingeleitet, so wird zu jedem Wort eine Checksumme von 7 Bit generiert. Die Checksumme wird zusätzlich zu dem, zu speicherndem, Wort im BRAM abgelegt.
Soll aus dem BAM gelesen werden, so wird auch die Checksumme eines Wortes gelesen. Das Datum wird danach mit der Checksumme verglichen. Sollte ein Bitfehler aufgetreten sein so wird dieser automatisch korrigiert. Dieser Vorgang beeinflusst den Lese-Transfer nicht und kann, durch das AHB-Interface, nicht festgestellt werden. Konnte der Fehler korrigiert werden, da nur ein Bit fehlerhaft war, so wird das korrigierte Wort in den Speicher zurückgeschrieben.
Wurden zwei oder mehr Bitfehler festgestellt, so können diese nicht korrigiert werden. Der Lesefehler wird intern entsprechend verarbeitet und es wird ein Error-Response über das AHB-Slave-Interface an den Bus zurückgegeben. Optional können solche Lesefehler über einen „Error-Counter“ gezählt werden. Wird ein Lese-Fehler festgestellt, so wird dieser um eins erhöht.

Durch die spezielle Behandlung der Bus-Transfer-Worte ergeben sich andere Zugriffszeiten als bei dem herkömmlichen „ahbram_if“-IP-Core.

Operation	Taktzyklen ohne EDAC	Taktzyklen mit EDAC
Lesen	1 - 2	1 - 3
Schreiben	1	1

Tabelle C.3: Transfer-Taktzyklen des „ftahbram“-IP-Core [gripdoc]

Die Tabelle C.3 zeigt eine Übersicht der benötigten Taktzyklen des „ftahbram“-IP-Core für einen bestimmten Transfertyp bei aktivierter oder deaktivierter EDAC. Ist die EDAC aktiviert so wird beim Lesen je nach Fehler ein Takt mehr benötigt um eine Fehlerkorrektur durchzuführen.
Zu bemerken ist, dass das Speichern von Wörtern in den „ftahbram“-IP-Core im Gegensatz zum gewöhnlichen „ahbram_if“-IP-Core einen Taktzyklus kürzer dauert. Also statt zwei Taktzyklen des „ahbram_if“-IP-Cores nur einen Taktzyklus.
Dies liegt daran, dass das zu speichernde Wort nicht direkt an den BRAM weitergeleitet wird, sondern erst, zur Erstellung der Checksumme, zwischengespeichert werden muss. Dieser Prozess

kann schneller erfolgen. Das Speichern des Wortes in den BRAM dauert trotzdem zwei Taktzyklen, dies ist aber auf dem Bus nicht erkennbar.

C.1.2 32/64-Bit PC133 SDRAM Controller mit EDAC („ftsdctrl“)

Der „ftsdctrl“-IP-Core ist eine fehlertolerante Variante des „sdctrl_if“-IP-Cores. Die EDAC-Funktion ist wie beim „ftahbram“-IP-Core optional aktivierbar oder deaktivierbar.
Ist die EDAC aktiviert, so unterstützt der „ftsdctrl“-IP-Core nur Daten-Transfers von je 32 Bit. Die EDAC-Funktion kann ebenfalls während der Laufzeit über Register aktiviert oder deaktiviert werden. Die EDAC-Funktion funktioniert gleichermaßen wie beim „ftahbram“-IP-Core verursacht jedoch keine zusätzlichen Warte-Taktzyklen, da ein Zugriff auf einen SDRAM mehr Taktzyklen benötigt als Taktzyklen, zur Umsetzung, der EDAC-Funktion bebraucht werden.
Ein weiterer interessanter IP-Core mit Fehlertoleranz ist der 8/16/32-Bit Memory Controller mit EDAC der „ftmctrl“-IP-Core. Mit diesem IP-Core ist es möglich gleichzeitig die EDAC-Funktion für PROM, I/O, SRAM und SDRAM bereitzustellen. Dadurch wird die EDAC-Logik nur einmal eingebunden.

Eine fehlertolerante Version des „ddrspa_if“-IP-Cores, also eine Schnittstelle zum DDR SDRAM mit EDAC existierte zur Entstehung dieser Arbeit noch nicht.

C.1.3 Der fehlertolerante LEON3 SPARC V8 Prozessor („leon3ft“)

Der LEON3FT-Prozessor ist funktionell identisch mit den LEON3-Prozessor. Dieser Kapitelpunkt hebt die fehlertoleranten Eigenschaften des LEON3FT-Prozessors hervor.
In erster Linie soll der LEON3-Prozessor gegen SEU-Fehler geschützt werden. Prozessorelemente, welche mit diesem Schutz ausgestattet wurden, sind die Register der IU und FPU sowie die Lines der Cache-Speicher. Diese werden durch den LEON3-Prozessor in BRAMs abgelegt.

C.1.3.1 IU-Register Schutz

Folgende Möglichkeiten können zum Schutz der IU-Register vorgenommen werden. Die Einstellungen müssen vor der Synthese mit der Generic-Konstante „**iuft**“, nach der Tabelle C.4 festgelegt werden:

„iuft“ Wert	Bezeichung	Bedeutung
0	Hardened Flip-Flops	Die Register der IU werden in Flip-Flops abgespeichert statt in BRAMs. Kein weiterer Schutz.
1	4 Bit Parität mit Neustart	4 Bit zur Korrektur von einem 32 Bit Wort. Detektiert und korrigiert 4 Bits pro Wort. Die Pipeline wird bei einer Korrektur neu gestartet.
2	8 Bit Parität ohne Neustart	8 Bit zur Korrektur von einem 32 Bit Wort. Detektiert und korrigiert 4 Bits pro Wort. Die Pipeline muss nicht neu gestartet werden. Fehler werden während der Pipelinestages korrigiert.
3	7 Bit BCH mit Neustart	Detektiert 2 Bits und korrigiert 1 Bit pro Wort. Die Pipeline wird bei einer Korrektur neu gestartet.

Tabelle C.4: „iuft“-Werte der LEON3FT-Entity und ihre Bedeutung [gripdoc]

C.1.3.2 FPU-Register Schutz

Die Register der FPU werden ähnlich wie bei der IU geschützt, allerdings gibt es weniger Möglichkeiten, welche zusätzlich FPU-abhängig sind. Tabelle C.5 stellt alle Schutzmöglichkeiten in Abhängigkeit der verwendeten FPU dar:

FPU	Schutzmöglichkeit
GRFPU	7 Bit BCH Checksumme pro Wort. Detektiert 2 Bits und korrigiert 1 Bit pro Wort. Die Pipeline wird bei einer Korrektur neu gestartet.
GRFPU-Lite	4 Bit zur Korrektur von einem 32 Bit Wort. Detektiert und korrigiert 4 Bits pro Wort. Die Pipeline wird bei einer Korrektur neu gestartet.

Tabelle C.5: Schutzmöglichkeiten der FPU-Register [gripdoc]

C.1.3.3 Cache-Speicher Schutz

Jedes Wort des Cache-Speichers wird mit zusätzlichen 4 Bits geschützt. Wird ein Fehler beim Zugriff auf den Cache-Speicher detektiert, so wird die betroffene Cache-Line verworfen und mit den entsprechenden Daten aus dem Hauptspeicher ersetzt. Eine Art EDAC wurde für den Cache-Speicher nicht implementiert, da dies permanente zusätzliche Taktzyklen für einen Lese-Transfer benötigen würde. Dadurch würde die Performance zu sehr beeinträchtigt.
Ein weiterer Schutz ist nicht unbedingt nötig, da Cache-Speicher sehr kurzlebige Speicher sind und

bei größeren Anwendungen, wie Betriebssystemen, ohnehin ständig bei Taskwechseln mit dem Hauptspeicher abgeglichen werden müssen. In diesem Fall wäre dann der Hauptspeicher wieder durch eine EDAC-Funktion geschützt.

C.2 Möglichkeiten für Echtzeit-Umsetzungen

Dieser Kapitelpunkt stellt Wege vor, mit denen es möglich ist Echtzeitverhalten eines LEON3-GRLIB-SoC zu erzeugen.

Echtzeit, im englischen auch „realtime“ genannt, ist ein verbreitete Möglichkeit um die Reaktionszeit eines ES unter einer festgelegten Schranke zu halten. Bei harten Echtzeit-Anforderungen darf diese Zeitschranke unter keinen Umständen übertreten werden, da sonst mit einem maximalen Schaden gerechnet werden muss.
Die Echtzeit-Anforderungen eines ES betreffen meist nur kleine Programmteile, auch Tasks genannt. Diese Programmteile müssen allerdings mit größter Zuverlässigkeit und Priorität auf dem ES ausgeführt werden, um die Zeitschranke nicht zu überschreiten.
Die Echtzeit-Anforderung macht aus einem gewöhnlichen Task einen Echtzeit-Task. Durch das herkömmliche SnapGear-Linux wird die Echtzeit-Anforderung nicht unbedingt automatisch erfüllt.
Das grundlegendes Prinzip der Umsetzungsmöglichkeiten, welche hier vorgestellt werden ist, dass der Echtzeit-Programmteil entsprechend kompiliert und aufbereitet wird, um die Einhaltung der Zeitschranke zu erfüllen.

Folgende dieser Umsetzungsmöglichkeiten werden für den LEON3 und die GRLIB angeboten:

- „eCos“ ein Echtzeitbetriebssystem für den LEON-Prozessor
- „VxWorks“ ein eigenständiges Echtzeitbetriebssystem
- „RTAI“ eine Echtzeit-Kernelerweiterung für den SnapGear-Linux-Kernel

Ein wichtiger Teil eines Echtzeit-Betriebssystems oder einer Echtzeit-Erweiterung ist, neben dem Scheduling der Echtzeit-Tasks, das Interrupt-Handling.
Interrupts werden durch die externe Hardware, wie Sensoren, getrieben um das System über Änderungen oder Ereignisse zu informieren. Dadurch kann auf zeitkritische Ereignisse schnell reagiert werden. Die Einführung von Interrupts entlastet zusätzlich die CPU, da ohne Interrupts ein periodisches Überprüfen (Polling genannt) der Sensoren durchgeführt werden müsste. Dieses Polling würde in den meisten Überprüfungsfällen unnötige CPU-Zeit benötigen.

Aufgrund der Bedeutung von Interrupts soll der Multi-Prozessor-Interrupt-Controller des LEON3-Systems im Kapitelpunkt C.2.4 genauer vorgestellt werden.

C.2.1 „eCos“ Betriebssystem

eCos, die Abkürzung für „**e**mbeded **C**onfiguration **o**perating **s**ystem“, ist ein Open-Source Echtzeit-Betriebssystem, welches hauptsächlich in ES Verwendung findet.
Das Betriebssystem eCos ist ein Nicht-Linux Betriebssystem und bietet in erster Linie hohe Konfigurierbarkeit und Portabilität. Es wird eine **H**ardware **A**bstraction **L**ayer (kurz HAL) benutzt, um eine unabhängige Schnittstelle von Software zur Hardware zu erhalten. Dies erhöht die Portabilität.
eCos wird auf einem Host-PC, mit Linux oder Windows, konfiguriert und kompiliert, um dann auf das Zielsystem übertragen zu werden. Eine Richtlinie bei der Entwicklung des eCos Betriebssystems ist der geringe Speicherbedarf.

C.2.2 „VxWorks“ Betriebssystem

VxWorks ist ein weiteres Echtzeit-Betriebssystem, welches die SPARC-Architektur unterstützt und speziell für den LEON-Prozessor aufbereitet wurde. Das Betriebssystem VxWorks 6.3 ist mit Treibern für alle GRLIB-Peripherie-Komponenten erhältlich und wird auf einem externen Host-PC erstellt und kompiliert.
VxWorks wird mit einem BSP ausgeliefert, mit dem das Betriebssystem für MMU- und nicht-MMU-LEON-Prozessoren kompiliert werden kann.

C.2.3 RTAI-Erweiterung

Das **R**eal **T**ime **A**pplication **I**nterface (kurz RTAI) ist ein Open-Source-Projekt zur Erweiterung eines gewöhnlichen Linux-Betriebssystems zu einem Echtzeit-Betriebssystem.

Einem gewöhnlichen Linux-Betriebssystem wird mit dem RTAI-Patch eine zusätzlicher RT-Kernel hinzugefügt. Dieser RT-Kernel empfängt nun zuerst alle Interrupts des Systems, vor dem normalen Linux-Kernel und verwaltet Echtzeit-Tasks, welche am RT-Kernel angemeldet werden müssen. Der normale Linux-Kernel wird in einem RTAI-System als eine Echtzeit-Task mit der niedrigsten Priorität behandelt und wird nur dann ausgeführt, wenn kein anderer Echtzeit-Task die CPU benötigt.

Die Interrupt-Verwaltung wird in erster Linie von dem neuen RT-Kernel durchgeführt. Um die Interrupt-Verwaltung optional zwischen dem RT-Kernel und dem normalen Linux-Kernel umzuschalten wird eine **R**eal **T**ime **H**ardware **A**bstraction **L**ayer (kurz RTHAL) dem normalen Linux-Kernel hinzugefügt. Für diese Veränderung muss der Quellcode des normalen Linux-Kernel um wenige Zeilen erweitert werden. Dies erledigt das RTAI-Patch.

Da der SnapGear-Linux-Kernel vorwiegend aus den Quellcode eines Linux-Kernels für einen gewöhnlichen PC besteht, könnte das RTAI-Patch auch für ein SnapGear-Linux anwendbar sein. Die Echtzeitfähigkeit des SnapGear-Linux könnte dann durch das Laden von speziellen RTAI-Modulen zur Laufzeit hergestellt werden. Eine Arbeit in der das RTAI-Patch auf einem gewöhnlichen PC angewendet wurde, um ihn für eine Regelung vorzubereiten ist [GentooRTAI].

C.2.4 Der Multi-Prozessor-Interrupt-Controller

Für ein GRLIB-System existieren systemweit 32 Interrupts. Um diese Interrupts zu treiben besitzt jeder AHB-Slave, AHB-Master oder APB-Slave den Ausgangsport „ahbo.hirq“ oder „apbo.pirq“. Diese Ausgangsports sind jeweils 32-Bitvektoren.
Möchte nun eine SoC-Komponente ein Interrupt auslösen so treibt sie den entsprechenden „a?bo.?irq“ Port, an einen zuvor festgelegten Index. Dieser Index wird, in der Regel, mittels einer Generic-Konstante, wie „**hirq**“ für die AHB-Slaves oder „**pirq**“ für die APB-Slaves, vor der Synthese festgelegt.

Die Busse AHB oder APB verknüpfen alle Interrupt-Vektoren, welche durch ihre Komponenten getrieben werden mit einem logischen Oder (siehe Abbildung 3.27 für den AHB und Abbildung A.11 für den APB).

Zu Beachten C.1 (Interrupts eindeutig vergeben)

Da die Interrupt-Vektoren logisch mit Oder verknüpft werden, muss darauf geachtet werden, dass jeder Inderruptindex nur einmal in dem GRLIB-System vergeben wird.
Wird dies nicht eingehalten so kann es passieren, dass die interruptauslösende Komponente nicht eindeutig identifiziert werden kann.

Jede SoC-Komponente eines GRLIB-Systems, welche an einen Bus angeschlossen ist, verfügt entweder über den Eingangsport „ahbi.hirq“ oder „apbi.pirq“. Dies ist davon abhängig, ob die SoC-Komponente an den AHB oder APB angeschlossen ist. Diese Eingangsports werden durch die Busse getrieben und stellen einen vollständigen Interrupt-Vektor aller Systeminterrupts dar.

Der Interrupt-Controller, dessen Entity in der VHD-Datei „irqmp.vhd“ gespeichert ist, wird als Slave an den APB angeschlossen. Durch seinen Eingangsport „apbi.pirq“ erhält er so eine vollständige Übersicht aller systemweit auftretenden Interrupts.
Der Interrupt-Controller ordnet alle auftretenden Interrupts nach Priorität, maskiert und leitet sie, über den Interrupt-Bus, an alle LEON3-Prozessoren weiter. Da die Anzahl der LEON3-Prozessoren für ein GRLIB System unbestimmt ist, wurde der Interrupt-Bus durch das EDK ebenfalls mit einem Bus (Busstandard: GR_IRQLEON3) assoziiert.

Folgende Abbildung C.2 zeigt eine Gesamtübersicht des Multi-Prozessor-Interrupt-Controllers in einem GRLIB System:

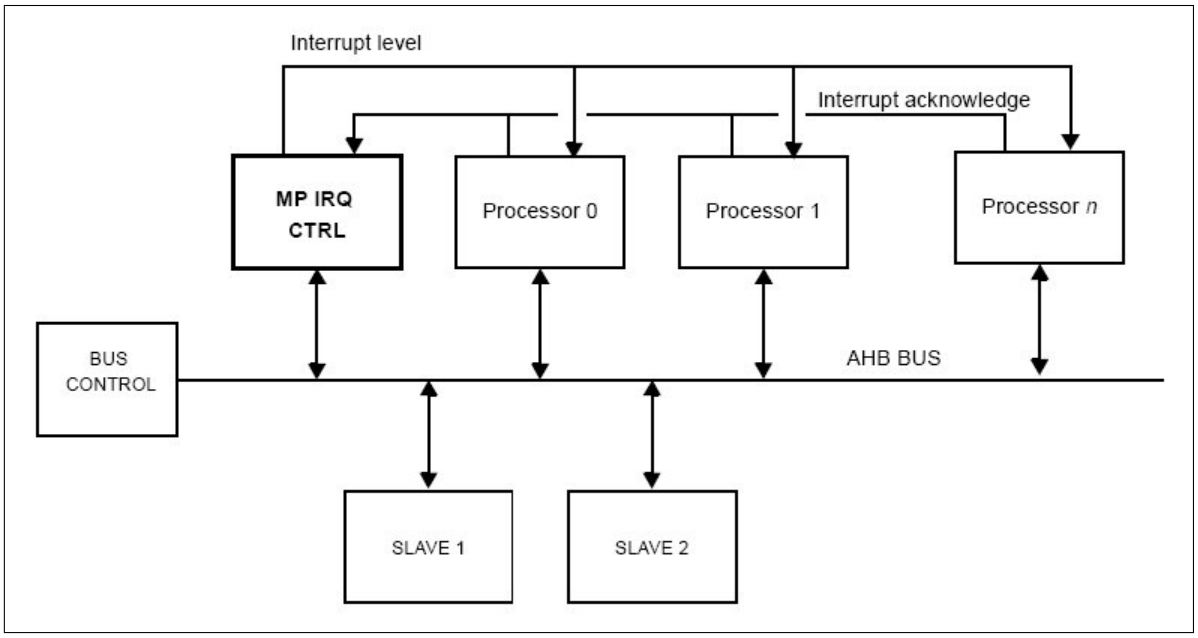

Abbildung C.2: Blockschaltbild eines GRLIB-SoC mit Multi-Prozessor-Interrupt-Controller [gripdoc]

In Abbildung C.2 ist zu erkennen, dass bei einem auftretenden Interrupt ein „Interrupt Level“ durch den Interrupt-Controller erzeugt wird, der dann zu allen LEON3-Prozessoren getrieben wird. Die Interrupt Acknowledgements (deutsch.: Interrupt Bestätigungen) werden von jedem LEON3-Prozessor einzeln in den Interrupt-Controller getrieben, so dass der entsprechend reagierende Prozessor identifiziert werden kann.

C.2.4.1 Interrupt Prioritäten

Der Multi-Prozessor-Interrupt-Controller überwacht die Interrupts 1-15. Dies entspricht einer Überwachung des Eingangsports „apbi.pirq(15 downto 1)“.

Zu Beachten C.2 (Verarbeitung der Interruptnummer durch interrupt-Controller)

Soll ein Interrupt einer SoC-Komponente durch den Interrupt-Controller an einen LEON3-Prozessor weitergeleitet werden, so darf durch die Generic-Konstante, der SoC-Komponente, nur ein Interrupt Index von 1 bis 15 festgelegt werden. Die anderen Interrupts (0, 16-31) werden nicht vom Interrupt-Controller überwacht und führen zu keiner Reaktion der LEON3-Prozessoren.
Dies hat den Vorteil, dass noch freie Interrupts zur Verfügung stehen. Dadurch können eigene SoC-Komponenten konstruiert werden, welche bei einem entsprechendem Interrupt reagieren können.

Das Interrupt-Signal mit dem Index 15 hat die höchste Priorität und wird bei einem logischen Pegel von ’1’ an alle Prozessoren weitergeleitet. Danach folgen die anderen Interrupts mit absteigendem Index. Das Interrupt-Signal mit dem Index 1 hat die niedrigste Priorität.
Bevor der vollständige Interrupt-Vektor an einen bestimmten Prozessor eines MPS getrieben wird, wird er für jeden Prozessor speziell maskiert. Dadurch kann festgelegt werden, dass für bestimmte Interrupts nur ein bestimmter Prozessor reagieren soll.

Zu Beachten C.3 (Interrupt 15)

Der Interrupt 15 hat die höchste Priorität und kann nicht maskiert werden. Er gelangt gleichzeitig an alle LEON3-Prozessoren eines GRLIB Systems.

Zu Beachten C.4 (Interrupt auslösen)

Interrupts müssen nicht unbedingt über die Ausgangsports („a?bo.?irq“) einer SoC-Komponente ausgelöst werden. Sie können auch in das „Interrupt force register“ des Interrupt-Controllers über einen APB-Schreib-Transfer ausgelöst werden.

C.2.4.2 Interrupt-Broadcasting

Ein Interrupt der in das „Broadcast Register“ des Interrupt Controllers geschrieben wird, wird bei aktivierten Interrupt-Broadcasting an alle LEON3-Prozessoren weiterleitet. Dies kann zum Beispiel genutzt werden, um einen Interrupt an alle Prozessoren unmaskiert weiterzuleiten.
Das Interrupt-Broadcasting des Interrupt-Controllers wird mit der Generic-Konstante „**broadcast**“ aktiviert. Da die Funktion nur benötigt wird, wenn mehr als ein LEON3-Prozessor an den Interrupt-Bus angeschlossen ist und durch die Aktivierung der Funktion keine herkömmlichen Interrupt-Controller Funktionen überschrieben werden, wird die „**broadcast**“-Generic-Konstante durch ein Tcl-Skript automatisch festgelegt. Dies erhöht die Benutzerfreundlichkeit.

Anhang D

Handhabung der DVD, zu dieser Arbeit

Diese Arbeit wird mit einer DVD ausgeliefert. Mit der DVD zu dieser Arbeit können alle Funktionalitäten der GRLIB genutzt und alle Ergebnisse dieser Arbeit weiterverwendet werden. Die DVD beherbergt auch Software der Firma GR wie den GRMON, TSIM oder das SnapGear-Linux. Bei diesen Software-Elementen ist allerdings zu beachten, dass sie ständig weiterentwickelt werden und fortwährend neue Versionen erscheinen. Auf der DVD befinden sich dagegen nur die Versionen, welche auch in dieser Arbeit verwendet wurden. Um die neusten Versionen zu erhalten wird auf die Homepage von GR[1] verwiesen.
Auf der DVD befinden sich ebenfalls die eingebundenen GRLIB-IP-Cores. Mit ihnen kann ein normales MicroBlaze-EDK-Projekt, welches beispielsweise mit dem „Base System Builder wizard“ erstellt wurde, so aufbereitet werden, dass es die GRLIB mit dem LEON3-Prozessor verarbeiten kann.

In diesem Anhang soll der Umgang mit der DVD erläutert werden. Ein Schwerpunkt ist dabei die Aufbereitung eines MicroBlaze-EDK-Projekts mit dem eingebundenen GRLIB-IP-Cores.

[1] http://www.gaisler.com

Die DVD zu dieser Arbeit hat folgende Verzeichnisstruktur:

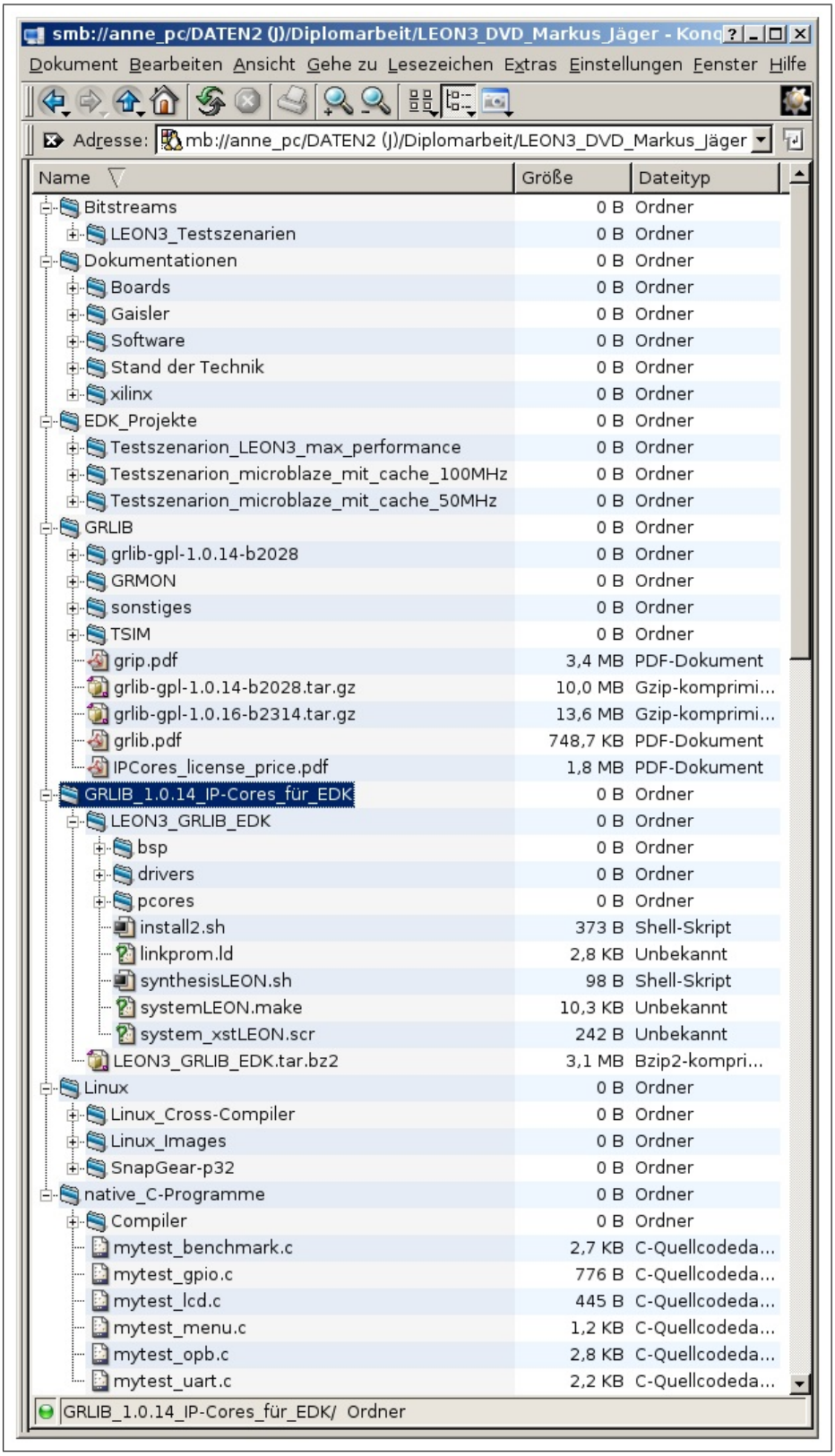

Abbildung D.1: Inhalt der DVD zu dieser Arbeit

Abbildung D.1 zeigt einen Einblick in den Inhalt der DVD zu dieser Arbeit.

Folgende Hauptverzeichnisse sind zu erkennen:

- Bitstreams
- Dokumentationen
- EDK_Projekte
- GRLIB
- GRLIB_1.0.14_IP-Cores_für_EDK
- Linux
- native_C-Programme

D.1 „Bitstreams“-Verzeichnis

Das „Bitstreams“-Verzeichnis enthält fertig synthetisierte FPGA-Bitstreams für den „Virtex-II Pro 30 FF896“, speziell für das XUP-Board. Bei diesen Bitstreams handelt es sich um die LEON3-Testszenarien aus Kapitelpunkt 8.3.2. Grundlage für die verschiedenen Bitstreams sind die Variationen der Testszenarien. Bei diesen Variationen wurde der Cache-Speicher (jeweils 8 kByte), die MMU sowie der V8 Hardware-Multiplizierer und Dividierer aktiviert oder deaktiviert. Eine dadurch neu entstandene Variation wurde synthetisiert und dann dem Dhrystone-Benchmark unterzogen. Damit die Ergebnisse nachvollzogen werden können, wurden die Synthese-Ergebnis-Dateien in diesem Verzeichnis abgespeichert.

Ein Testszenario-Verzeichnis beinhaltet jeweils:

- „download.bit“
- „system.bis“
- den Synthesebericht des LEON3-Prozessors „leon3s_if_0_wrapper_xst.srp“
- den Synthesebericht des gesamten LEON3-SoC „system_xstLEON.srp“

D.2 „Dokumentationen“-Verzeichnis

In diesem Hauptverzeichnis befinden sich alle Dokumentationen, welche sich im Laufe des Entstehungsprozesses dieser Arbeit angesammelt haben.

- Im Unterverzeichnis „**Boards**“ befinden sich alle Dokumentationen der FPGA-Boards, auf denen die GRLIB erfolgreich, während dieser Arbeit, getestet wurde. Dies ist nicht nur das XUP-Board sondern auch das „Spartan 3E Starter Kit“-Board und das „Virtex-II Pro (P4/P7) Development“-Board von Memec. Das XUP-Board war jedoch das einzige Board, welches den größten FPGA besitzt und mit dem ein GRLIB-SoC großzügig getestet werden konnte. Aus diesem Grund wurde die Arbeit mit dem XUP-Board vorgestellt.
- Das Unterverzeichnis „**Gaisler**“ beinhaltet alle Dokumentationen, welche direkt mit der GRLIB in Verbindung stehen. Dies sind nicht nur von der Firma GR selbst entworfene Dokumentationen, sondern auch die AMBA-Spezifikation oder die SPARC.
- Im „**xilinx**“-Unterverzeichnis sind alle Dokumentationen und Data-Sheets enthalten, welche Produkte der Firma Xilinx erläutern. Dies sind zum Beispiel Dokumentationen zu den FPGA-Familien „Virtex 2,4 oder 5“ aber auch Dokumentationen der EDK-Konstrukte und der Tcl.

D.3 „EDK_Projekte“-Verzeichnis

Dieses Hauptverzeichnis beinhaltet vollständig synthetisierte Projekte des EDK. Jedes Projekt ist in einem weiteren Unterverzeichnis und zeigt Testszenarien von LEON3-SoCs oder MicroBlaze-SoCs. Zusätzliche Projekte sind beispielsweise ein Projekt mit der AHB2OPB-Bridge oder ein Projekt, welches die VGA- und die Keyboard-PS2-Schnittstelle des XUP-Boards verwendet. In das letztgenannte Projekt kann beispielsweise das Linux-Image „image-2.6_svga_640_keyb.dsu“ im „Linux“-Hauptverzeichnis geladen werden. Dadurch wird das SnapGear-Linux gebootet und macht seine grafischen Ausgaben auf einem Monitor und kann mit einer Tastatur bedient werden. Das EDK-Projekt „LEON3_Latenzzeiten“ ist ein spezielles LEON3-SoC zur Messung der Latenzzeiten der Busse AHB, APB und OPB vom LEON3-Prozessor aus. Dazu wurden spezielle Latenz-IP-Cores implementiert, welche zu einem Tansfer ständig bereit sind. Mit dem zugehörigen nativen C-Programm „Latenz_benchmark.c“ können dann die Latenzzeiten mit einem Benchmark gemessen werden. Dabei ist zu bemerken, dass die gemessenen Latenzzeiten immer 2 Takte länger sind als die „echten“ Latenzzeiten der Busse. Dies liegt daran, dass der LEON3-Prozessor nach jeden Transfer zusätzlich 2 Takte, zum Lesen der nächsten Instruktion, benötigt. Die Simulation der Latenzzeiten zeigt dies deutlich, zu erkennen an den Bildern im „Bilder“-Verzeichnis des EDK-Projekts.

D.4 „GRLIB“-Verzeichnis

Dieses Hauptverzeichnis beinhaltet die GRLIB in der Form, wie sie von GR ausgeliefert wird. Hier sind also alle Verzeichnisse aus Kapitelpunkt 3.1 einsehbar. Zusätzlich wird aber auch das GRLIB-Archiv abgespeichert. Die Archive und Verzeichnisse für den GRMON und TSIM sind hier ebenfalls abgespeichert.

D.5 „GRLIB_1.0.14_IP-Cores_für_EDK“-Verzeichnis

Dieses Hauptverzeichnis beinhaltet die Kern-Ergebnisse dieser Arbeit. Hier wird das Archiv „LEON3_GRLIB_EDK.tar.bz2“ gespeichert sowie in seinem extrahierten Zustand im Unterverzeichnis „LEON3_GRLIB_EDK“. Das Archiv „LEON3_GRLIB_EDK.tar.bz2“ beinhaltet alle eingebundenen GRLIB-IP-Cores im „pcores“-Unterverzeichnis sowie die BSP und Drivers. Wichtig sind auch die Dateien „synthesisLEON.sh“, „system_xstLEON.scr“ und der, für die GRLIB, modifizierte EDK-Arbeitsplan „systemLEON.make“.

D.5.1 Vorbereitung eines Host-PC zur Benutzung der GRLIB-IP-Cores

Um ein Host-PC zur Synthese der GRLIB vorzubereiten muss zuerst ein LEON3-Software-Compiler installiert werden, beispielsweise der BCC, siehe Kapitelpunkt A.2.1. Nach dieser Installation muss das Archiv „LEON3_GRLIB_EDK.tar.bz2“ auf dem Host-PC entpackt werden. Für einen Host-PC mit Linux-Betriebssystem wird das Archiv empfohlen, da das Archiv spezielle Dateirechte und Eigenschaften beinhaltet. Für einen Host-PC mit Windows kann auch das extrahierte Archiv im „LEON3_GRLIB_EDK“-Unterverzeichnis verwendet werden.

Ist das Archiv extrahiert so muss ein normales MicroBlaze-EDK-Projekt aufbereitet werden um in diesem Projekt die eingebundenen GRLIB-IP-Cores nutzen zu können.

Dazu sind folgende Schritte notwendig:

1. Kopieren der Archiv-Unterverzeichnisse in das Verzeichnis „[EDK_PROJECT]“[2].
2. Kopieren der Dateien „synthesisLEON.sh“, „system_xstLEON.scr“ in das Verzeichnis „[EDK_PROJECT]/synthesis“. Existiert es nicht muss es erstellt werden.
3. Kopieren des modifizierten EDK-Arbeitsplans („systemLEON.make“) und des Linkerskripts „linkprom.ld“ in das Verzeichnis „[EDK_PROJECT]“.

[2] Der Ausdruck [EDK_PROJECT] steht für das Verzeichnis in dem das EDK-Projekt abgespeichert ist

4. Der modifizierte EDK-Arbeitsplan („systemLEON.make“) muss im MicroBlaze-EDK-Projekt in den „Project Options...“ unter „Custom Makefile“ eingetragen werden.

   Wirkung: Eintrag in der „system.xmp“ mit „`UserMakeFile: systemLEON.make`“

5. Der Name des Software-Projekts muss zu „TestApp“ geändert werden. Siehe Zu Beachten 5.9.

   Wirkung: Eintrag in der „system.xmp“ mit „`SwProj: TestApp`“

6. Im Software-Projekt muss bei den Compiler-Optionen das neue Linkerskript eingetragen werden.

   Wirkung: Eintrag in der „system.xmp“ mit „`LinkerScript: linkprom.ld`“

7. Da der LEON3-Prozessor standardmäßig keine FPU besitzt muss der Parameter „-msoft-float“ in „Set Compiler Options“ bei „Other Compiler Options to Append“ eingetragen werden.

   Wirkung: Eintrag in der „system.xmp“ mit „`ProgCCFlags: -msoft-float`“

Die Schritte 1-3 werden durch das Shell-Skript „install2.sh“ (für Host-PCs mit Linux) oder durch die Batch-Datei „install2.bat“ (für Host-PCs mit Windows) ausgeführt.

Unter Linux kann ein MicroBlaze-EDK-Projekt mit dem Kommando:

```
$ ./install2.sh [EDK_PROJECT]
```

aufbereitet werden. Unter Windows ist das analoge Kommando:

```
> install2.bat [EDK_PROJECT]
```

Die Schritte 4-7 müssen manuell ausgeführt werden. Es ist allerdings denkbar, das die Skripte „install2.*“ so ergänzt werden können, dass die Schritte 4-7 automatisch durchgeführt werden. Die Skripte könnten dabei auf der EDK-Projekt-Datei „system.xmp“ arbeiten.

Nachdem die oben genannten Schritte ausgeführt wurden, ist das EDK-Projekt im Verzeichnis „[EDK_PROJECT]“ zur Erstellung und Synthese eines GRLIB-SoC bereit. Da der modifizierte EDK-Arbeitsplan angewendet wird, wird automatisch das native C-Programm mit dem LEON3-Software-Compiler kompiliert und in die BRAMs geladen.

D.6 „Linux“-Verzeichnis

Das „Linux“-Hauptverzeichnis speichert alle Daten, die mit dem SnapGear-Linux zusammenhängen. Dies sind die Archive des „LEON Cross-Compiler für Linux“ und des SnapGear-Linux selbst. Zusätzlich beinhaltet das „Linux_Images“-Unterverzeichnis Linux-Image-Dateien, die während dieser Arbeit kompiliert und erfolgreich, auf einem LEON3-SoC und dem XUP-Board, getestet wurden.

D.7 „native_C-Programme“-Verzeichnis

Dieses Hauptverzeichnis beinhaltet alle nativen C-Programme, die während dieser Arbeit entstanden sind. Mit ihnen wurden die SoC-Komponenten eines LEON3-SoC und der LEON3-Prozessor selbst getestet. Diese nativen C-Programme können im „Applications“-Tab des EDK, als Source-Code, eingetragen werden. Das Unterverzeichnis „Compiler“ liefert die Archive der Software-Compiler (BCC und RCC) für den LEON3-Prozessor. Diese Versionen, der hier abgespeicherten Compiler, wurden in dieser Arbeit verwendet.

Literaturverzeichnis

[ambaspec] „AMBA Specification 2.02“,
http://www.gaisler.com/doc/amba.pdf

[bccdoc] „BCC - Bare-C Cross-Compiler User’s Manual“,
http://gaisler.com/doc/bcc.pdf

[bitgendoc] „Development System Reference Guide, Bitgen“ (Chapter 16),
http://www.xilinx.com/support/sw_manuals/2_1i/download/dev_ref.pdf

[CES96] Sanjaya Kumar: „The Codesign of Embedded Systems“, Massachusetts 1996

[EIS05] Peter Nauth: „Embedded Intelligent Systems“, Oldenbourg 2005

[estdoc] „Embedded System Tools Reference Manual“,
http://www.xilinx.com/ise/embedded/edk91i_docs/est_rm.pdf

[EvalCPU] Daniel Mattsson und Marcus Christensson:
„Evaluation of synthesizable CPU cores“, Gothenburg 2004

[fpu100doc] Jidan Al-Eryani: „Floating Point Unit“,
http://www.opencores.org/cvsweb.shtml/fpu100/doc/FPU_doc.pdf

[GentooRTAI] Markus Jäger: „Erweitern eines Gentoo Linux mit RTAI, LabVIEW und
Comedi als digitaler Regler“,
Leipzig 2007

[gripdoc] „GRLIB IP Core User’s Manual“,
http://www.gaisler.com/products/grlib/grip.pdf

[grlibdoc] „GRLIB IP Library User’s Manual“,
http://www.gaisler.com/products/grlib/grlib.pdf

[grmondoc] „GRMON User's Manual“,
http://www.gaisler.com/doc/grmon.pdf

[IBMCore] IBM: „IBM CoreConnect and CPU support cores“, 2006

[LeonRob] Rainer Findenig und Robert Priewasser:
„Einsatz von embedded Linux auf dem Leon3 am Beispiel eines Hexapod-Roboters“,
Hagenberg 2006

[LinuxLeon] Adrian Meier und Patrick Stählin:
„Linux auf Sparc Leon3 Softcore Processor“, Winterthur 2007

[onpulhp] Homepage der Onpulson.de GbR Online Lexikon,
http://www.onpulson.de/lexikon/right-first-time.htm

[platrefdoc] „Platform Specification Format Reference Manual“,
http://www.xilinx.com/ise/embedded/edk91i_docs/psf_rm.pdf

[snaplinuxdoc] „Manual SnapGear Linux for LEON“
ftp://gaisler.com/gaisler.com/linux/snapgear

[sparcv8spec] „The SPARC Architecture Manual Version 8“,
http://www.gaisler.com/doc/sparcv8.pdf

[tsimdoc] „TSIM2 Simulator User's Manual“,
ftp://ftp.gaisler.com/gaisler.com/tsim/tsim-eval-2.0.8.tar.gz

[Xilfam1] „Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Introduction and Overview“,
http://www.xilinx.com/bvdocs/publications/ds083.pdf

[Xilfam2] „Virtex-4 Family Overview“,
http://www.xilinx.com/bvdocs/publications/ds112.pdf

[Xilfam3] „Virtex-5 Family Overview“,
http://www.xilinx.com/bvdocs/publications/ds100.pdf

Abbildungsverzeichnis

Tabellenverzeichnis

Zu Beachten Verzeichnis

Abkürzungen

μC	**Mikroc**ontroller
μP	**Mikrop**rozessor
AHB	**A**dvanced **H**igh-performance **B**us
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
APB	**A**dvanced **P**eripheral **B**us
BCC	**B**are-C **C**ross-**C**ompiler
BCH	**B**ose-**C**haudhuri-**H**ocquenghem
BMM	**B**RAM **M**emory **M**ap
BRAM	**B**lock-**RAM**
BSCAN	**B**oundary **Scan**
BSP	**B**oard **S**upport **P**ackage
CISC	**C**omplex **I**nstruction **S**et **C**omputer
DCM	**D**igital **C**lock **M**anager
DIPS	**D**hrystone **I**terationen **p**ro **S**ekunde
DSU	**D**ebug **S**upport **U**nit
E/A	**E**ingabe/**A**usgabe
eCos	**e**mbeded **C**onfiguration **o**perating **s**ystem
EDAC	**E**rror **D**etection **A**nd **C**orrection
EDCL	**E**thernet **D**ebug **C**ommunication **L**ink
EDK	**E**mbedded **D**evelopment **K**it
ELF	**E**xecutable and **L**inking **F**ormat
ES	**E**ingebettetes **S**ystem
ESA	**E**urpean **S**pace **A**rgency
FPGA	**F**ield **P**rogrammable **G**ate **A**rrays
FPGAs	Plural von FPGA
FPU	**F**loating-**P**oint **U**nit
FSM	**F**inite **S**tate **M**achine
GCC	**G**NU **C** **C**ompiler
GPIO	**G**eneral **P**urpose **I/O**

GPL	**G**eneral **P**ublic **L**icense
GR	**G**aisler **R**esearch
GRFPU	**G**aisler **R**esearch´s **F**loating-**P**oint **U**nit
GRLIB	**G**aisler **R**esearch **Lib**rary
GRMON	**G**aisler **R**esearch Debug**mon**itor
HAL	**H**ardware **A**bstraction **L**ayer
HDL	**H**ardware **D**escription **L**anguage
IF	**I**nter**f**ace
IOBUF	**IO**-**Buf**fer
IPs	**I**ntellectual **P**roperti**e**s
IU	**I**nteger **U**nit
Libgen	**Lib**rary **Gen**erator
logan	**Log**ic **An**alyzer
LUT	**L**ook-**U**p-**T**able
mb-gcc	**M**icro**B**laze-**G**NU **C** **C**ompiler
MDD	**M**icroprocessor **D**river **D**efinition
MDM	**M**icroprocessor **D**ebug **M**odule
MHS	**M**icroprocessor **H**ardware **S**pecification
MLD	**M**icroprocessor **L**ibrary **D**efinition
MMU	**M**emory **M**anagement **U**nit
MPD	**M**icroprocessor **P**eripheral **D**efinition
MPS	**M**ulti **P**rozessor **S**ystem
MSS	**M**icroprozessor **S**oftware **S**pecification
PAO	**P**eripheral **A**nalyze **O**rder
PC	**P**ersonal **C**omputer
RCC	**R**TEMS LEON/ERC32 **C**ross-**C**ompiler
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTAI	**R**eal **T**ime **A**pplication **I**nterface
RT-Ebene	**R**egister-**T**ransfer-Ebene
RTEMS	**R**eal-**T**ime Operating System and **E**nviro**m**ent**s**
RTHAL	**R**eal **T**ime **H**ardware **A**bstraction **L**ayer
RTOS	**R**eal**t**ime**o**perating**s**ystems
SEL	**S**ingle **E**vent **L**atch-up
SEU	**S**ingle **E**vent **U**pset
SMP	**S**ynchronous **M**ulti-**P**rocessing
SoC	**S**ystem-**o**n-**C**hip
SoCs	Plural von SoC
SPARC	**S**calable **P**rocessor **Arc**hitekture
tar	**t**ape **ar**chiver

Tcl	**T**ool **C**ommand **L**anguage
TLB	**t**ranslation **l**ookaside **B**uffer
TSIM	SPARC-Simulator
UCF	**U**ser **C**onstraint **F**ile
XBD	**X**ilinx **B**oard **D**efinition
XMD	**X**ilinx **M**icroprocessor **D**ebug
XST	**X**ilinx **S**ynthese **T**ool

Erklärung

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

Leipzig, den 04.02.2008 Markus Jäger