RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

KIRCHHOFF-INSTITUT FÜR PHYSIK

**Fakultät für Physik und Astronomie**
**Ruprecht-Karls-Universität Heidelberg**

Diplomarbeit
im Studiengang Physik

vorgelegt von
**Torsten Alt**
aus Saarlouis
2002

# Entwicklung eines autonomen FPGA-basierten Steuerrechners

Die Diplomarbeit wurde von Torsten Alt ausgeführt am
Kirchhoff-Institut für Physik
unter der Betreuung von
Herrn Prof. Dr. Volker Lindenstruth

## Inhalt

Inhalt dieser Arbeit ist die Entwicklung eines FPGA-basierten, autonomen und netztwerkfähigen Steuerrechners in Form einer PCI-Karte zur Kontrolle von Standard-PCs. Der Steuerrechner basiert auf einem konfigurierbaren 32-Bit RISC-Prozessor und ist komplett im FPGA untergebracht. Als Betriebssystem kommt ein netzwerkfähiges $\mu$C-Linux zum Einsatz. Aufbau, Funktionsweise und Konfiguration des Prozessor-Systems werden beschrieben und an den Prototypen einer PCI-Steckkarte angepaßt. Weiterhin wird ein PCI-Design erstellt, über welches der Steuerrechner auf den PCI-Bus des zu kontrollierenden Rechners zugreifen kann.



## Abstract

This thesis describes the development of a FPGA-based, autonomic and network capable controller implemented as PCI-card for controlling commercial-off-the-shelf computers. The controller is based on a configurable, 32-Bit RISC-processor and runs completely inside the FPGA. The operating system is a network capable $\mu$C-Linux. Structure, functionality and configuration of the processor system are described and adapted to a prototype of a PCI-card. Furthermore, a PCI-design is developed, which allows the controller to access the PCI-bus of the controlled node.

# Inhaltsverzeichnis

**1 Motivation** **7**
1.1 Das ALICE-Experiment . . . . . 7
1.1.1 Der ALICE-Detektor . . . . . 8
1.1.2 Der ALICE High Level Trigger . . . . . 9
1.2 Cluster Computing . . . . . 9

**2 Der Cluster Interface Adapter** **11**
2.1 Anforderungen an den C.I.A. . . . . . 11
2.2 Konzept . . . . . 12
2.3 Implementation . . . . . 13

**3 FPGAs** **15**
3.1 Funktionsweise eines FPGAs . . . . . 15
3.2 Design-Flow für einen FPGA . . . . . 15
3.3 ALTERA FPGAs . . . . . 17
3.4 QUARTUS II . . . . . 17

**4 Das NIOS-System** **19**
4.1 Das EXCALIBUR Development-Kit . . . . . 19
4.1.1 Das Development-Board . . . . . 19
4.1.2 Die Development-Software . . . . . 21
4.2 Der SOPC-Builder . . . . . 21
4.3 NIOS-System-Modul . . . . . 22
4.3.1 Der NIOS-Prozessor . . . . . 23
4.3.2 Der AVALON Bus . . . . . 24
4.3.3 Peripherie-Module . . . . . 25
4.3.4 User-Module . . . . . 25
4.4 G.E.R.M.S. Monitor . . . . . 25

**5 Das $\mu$C-Linux System** **27**
5.1 Überblick . . . . . 27
5.2 Hardware . . . . . 27
5.3 Software . . . . . 28

**6 PCI Designs** **31**
6.1 Überblick . . . . . 31
6.2 Anforderungen . . . . . 32
6.3 Der PCI-Bus . . . . . 32
6.3.1 Aufbau . . . . . 32
6.3.2 Initialisierung der PCI-Devices: Configuration-Cyles . . . . . 34

6.3.3 Datenübertragung . . . 35
6.4 PCI-Megacore . . . 35
6.5 Register-File und PCI-Controller . . . 38
6.5.1 Schnittstellen zum NIOS : PIOs . . . 38
6.5.2 Register-File . . . 39
6.5.3 PCI-Controller . . . 41
6.6 Zusammenfassung PCI-Design . . . 45

**7 PCI-Design - Messungen 47**
7.1 Configuration-Read . . . 48
7.2 Memory-Transaktions . . . 50
7.2.1 Memory-Read . . . 50
7.2.2 Memory-Write . . . 51

**8 Das C.I.A.-Board 53**
8.1 Überblick . . . 53
8.2 Ressourcen des C.I.A.-Boards . . . 54
8.2.1 FPGA . . . 54
8.2.2 64-Bit-Memory-Unit . . . 55
8.2.3 32-Bit-Memory-Unit . . . 56
8.2.4 Schnittstellen . . . 57
8.2.5 Clock-Distribution . . . 58
8.3 Konfiguration des FPGAs und CPLD . . . 58
8.3.1 Konfiguration über JTAG . . . 58
8.3.2 Konfiguration über FLASH & CPLD . . . 58

**9 System-Logik 61**
9.1 Überblick . . . 61
9.2 Aktuelle Implementation . . . 61
9.2.1 Das NIOS-System-Modul des C.I.A. Boards . . . 61
9.2.2 Das Interface zur 32-bit Memory-Unit . . . 64
9.2.3 PCI-Designs . . . 64
9.3 Geplante Implementation . . . 65
9.4 VGA-Implementation . . . 65

**10 Zusammenfassung und Ausblick 67**

**A CD-Rom 69**

**B NIOS-Tools 71**

**C Glossar 73**

**Literaturverzeichnis 75**

**Abbildungsverzeichnis 78**

**Tabellenverzeichnis 79**

# Kapitel 1

# Motivation

In den Jahren 1936-38 entwickelte Konrad Zuse die Z1, den ersten, noch mechanischen Rechner der Welt und leitete damit den Beginn der automatischen Datenverarbeitung ein. Die, im Laufe der Jahre immer komplexeren und leistungsfähigeren System, entwickelten sich schnell zu einem Hilfsmittel, vor allem für Naturwissenschaftler, insbesondere für Physiker. Ob Ozeanströmungen, Atmosphärendynamik oder die Suche nach den elementaren Bausteinen der Materie, viele physikalische Disziplinen haben eins gemeinsam: Enorme Mengen an Daten müssen erfaßt, analysiert und weiterverarbeitet werden.

## 1.1 Das ALICE-Experiment

Das ALICE[1]-Experiment wird eins von 4 Experimenten des geplanten LHC[2] am CERN[3] sein. Ziel ist die Untersuchung eines Zustands der Materie, der als Quark-Gluonen-Plasma bezeichnet wird. In Kernmaterie normaler Dichte wechselwirken Quarks nach dem Modell der starken Wechselwirkung über ihre Austauschteilchenen, die Gluonen, und bilden so die Nukleonen. Während es bei ausreichend hohen Energien möglich ist Atome in Atomkerne und Elektronen und Atomkerne in Protonen und Neutronen zu trennen, ist es nach der Theorie des *Confinement* nicht möglich Nukleonen in einzelne Quarks zu trennen, da hierbei die Energie eines isolierten Quarks ins Unendliche steigt. Erhöht man jedoch die Dichte der Kernmaterie, so brechen die Nukleonen auf und es bildet sich ein Quark-Gluonen-Plasma. Innerhalb des QGP können Quarks und Gluonen nicht mehr einzelnen Nukleonen zugeordnet werden und quasi als freie Teilchen angesehen werden. Man spricht vom *Deconfinement.* Beim Abkühlen des QGP lagern sich Quarks wieder zu Teilchen, Hadronen, zusammen, das QGP *hadronisiert.* Anhand der bei der Hadronisierung entstandenen Teilchen lassen sich Rückschlüsse auf die Wechselwirkungen zwischen den Quarks ziehen. Die Theorie, welche diese *starke Wechselwirkunge* beschreibt, ist die Quantenchromodynamik (QCD), ein Teilbereich des Standardmodells.
Hinweise auf die Existenz des QGP finden sich in den Produktionsraten, der bei einer Kollision entstehenden Teilchen, insbesondere in der Unterdrückung von Charmonium-Zuständen[4], sowie einer erhöhten Produktion von Hadronen, welche Strange Quarks enthalten[5].
Um detaillierte Aussagen über das QGP machen zu können, müssen Schwerionenexperimente bei höheren Energien durchgeführt werden. Im geplanten LHC werden hierzu Blei-Ionen[6] auf

[1] **A** **L**arge **I**on **C**ollider **E**xperiment
[2] **L**arge **H**adron **C**ollider
[3] **C**onseil **E**uropeéen pour la **R**echerche **N**ucléaire
[4] Schwefel-Uran-Kollisions-Experiment NA38/CERN und Blei-Blei-Kollisions-Experiment NA50/CERN
[5] NA49/NA50/WA97 CERN
[6] Pb 208

Schwerpunktsenergien von ca. 1150 TeV beschleunigt und anschließend im Detektor zur Kollision gebracht. Am Kollisionspunkt steigt die Dichte auf das 20-facher der Kerndichte an und es bildet sich ein QGP. Nach ungefähr $8 \cdot 10^{-23}$ s hadronisiert das Plasma aus und es bilden sich bis zu 20000 Teilchen, welche im Detektor-System identifiziert werden müsssen. Zur Identifikation der bei der Kollision entstehenden Partikel benötigt man deren Spur, ausgehend vom Kollisionspunkt, sowie ihre Energie und Impuls. Die Aquirierung der dazu benötigten Daten geschieht innerhalb des ALICE-Detektors, die Analyse und Spurrekonstruktion im High-Level-Trigger (HLT).

### 1.1.1 Der ALICE-Detektor

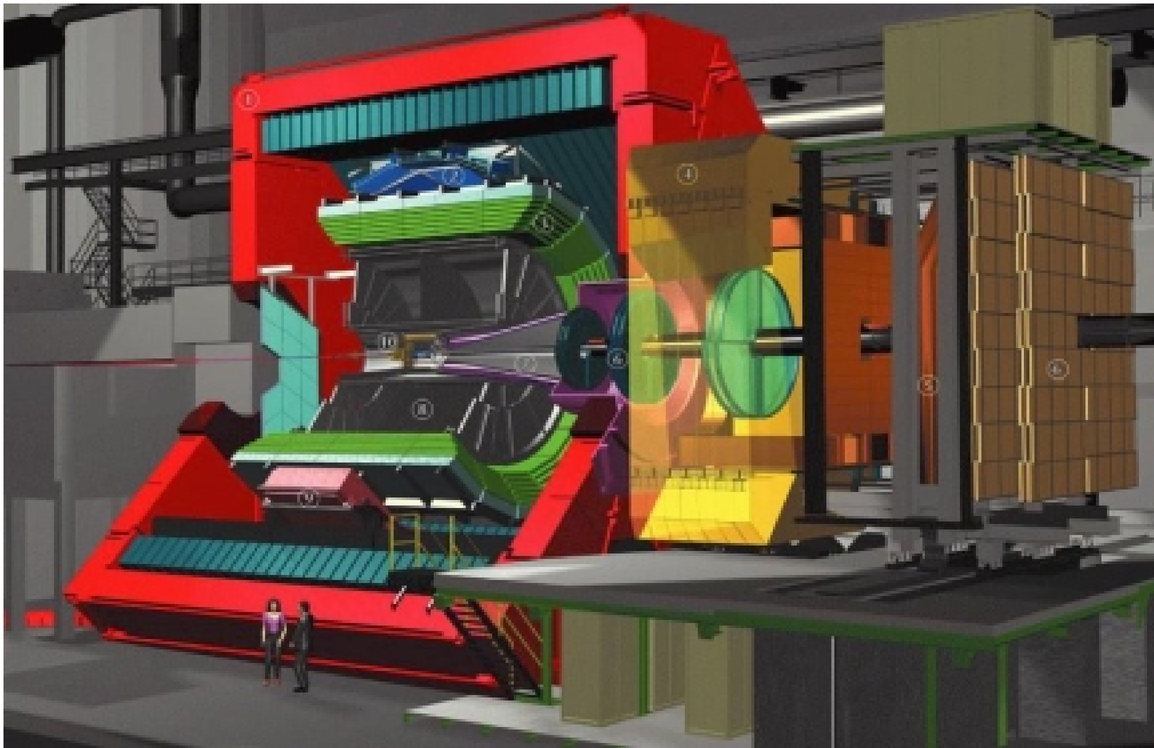

Abbildung 1.1: Aufbau des ALICE-Detektors : 1:L3-Magnet, 2:HMPID, 3: TRD/TOF, 4: DipolMagnet, 5:Myon-Filter, 6: Spurkammern, 6':Trigger-Kammern, 7: Absorber, 8:TPC, 9:PHOS, 10:ITS

Abb. 1.1 zeigt den ALICE-Detektor. Die Hauptkomponente zur Datenerfassung ist die Time Projection Chamber (TPC), welche durch verschieden Sub- und Triggersysteme unterstützt wird. Der Vollständigkeit halber werden diese System hier aufgelistet, für vertiefende Informationen sei auf die entsprechenden Referenzen verwiesen.

- ITS - Inner Tracking System[1]
- TPC - Time Projection Chamber[2]
- TRD - Transition Radiation Detector[3]
- TOF - Time Of Flight Detector[5]
- PHOS - PHOton Spectrometer[6]
- HMPID - High Momentum Particle IDentification[7]

- FMD - Forward Multiplicity Detector[8]
- PMD - Photon Multiplicity Detector[9]
- CASTOR - Centauro And STrange Object Research[10]
- ZDC - Zero Degree Calorimeter[11]

Die maximale Auslesefrequenz der TPC wird 200Hz betragen bei einer Datensatzgröße von ungefähr 82MByte, was eine Datenrate von über 16GByte/s ergibt. Diese Daten werden zum ALICE HLT[7] übertragen, analysiert, komprimiert und anschließend gespeichert.

### 1.1.2 Der ALICE High Level Trigger

Der High Level Trigger (HLT) dient dazu, eine erste Selektion der Rohdaten der TPC vorzunehmen. Physikalisch relevante Daten sollen herausgefiltert und komprimiert werden, anschließend können sie gespeichert und weiterverarbeitet werden. Um die enormen Datenmengen ausreichend schnell bearbeiten zu können, soll der HLT in Form einer PC-Farm, eines Clusters, realisiert werden. Hierbei werden Rechner über schnelle Netzwerke zu einer geeigneten Topologie verbunden. Das Betriebssystem und entsprechend angepaßte Software übernehmen das Verteilen der Rohdaten auf einzelne Rechner, dort wird eine lokale Analyse durchgeführt und anschließend werden die aufbereiteten Daten wieder eingesammelt.
Der Cluster des HLT erfüllt zwei Aufgaben. Während der Strahlzeit dient er der Datenaufnahme. Hierzu werden die Daten über 180 optisch GigaBit-Links direkt in die Hauptspeicher einzelner Nodes[8] geschrieben. Diese nehmen eine erste Analyse vor und reichen die Daten anschließend zur Weiterverarbeitung an die nächst Hierachiestufe. Während dieses Modus werden vor allem Spurrekonstruktionen durchgeführt. Besteht keine Strahlzeit, so kann der Cluster als High-Performance PC-Farm genutzt werden.
Für den Betrieb ab Ende 2005 wurde die Größe des Cluster auf ca. 1000 PCs berechnet, unter Berücksichtigung von Moores Law[9]. Die dabei zum Einsatz kommenden PCs, wie auch die Netzwerktechnologie, werden handelsübliche Komponenten sein und dadurch deutlich günstiger als Spezialrechner oder Supercomputer. Dabei stellt sich neben der Anpassung der Software aber auch die Frage nach der Kontrolle und Wartung eines solchen Clusters.

## 1.2 Cluster Computing

Wie oben erwähnt werden beim Cluster Computing einzelne Rechner über entsprechende Netzwerk zusammengeschlossen und treten nach außen als eine Einheit auf. Dies erfordert intern jedoch einen erheblichen Verwaltungsaufwand. Neben einer fairen Verteilung der Daten auf die einzelnen Nodes (Lastausgleich) muss vor allem die Ausfallsicherheit garantiert sein. Sollte ein Node ausfallen, so darf der Benutzer davon nichts merken, ein anderer Node muss folglich seine Aufgabe übernehmen. Während dies alles eine Aufgabe der Software ist, stellt sich aber auch die Frage nach der Kontrolle und Wartung der einzelnen Rechner. Bei Spezialrechnern und Supercomputern existieren für solche Aufgaben meist Monitoring-Systeme, welche einen Defekt anzeigen, und redundante Systeme, die bei einem Ausfall einspringen. Bei einem Cluster hat man jedoch nicht ein System, sondern viele, deren Hardware unabhängig voneinander ist. Darüberhinaus existiert bei einem Cluster aus normalen PCs eine Vielzahl an

[7] High Level Trigger
[8] einzelner PC eines Clusters
[9] Aussage über die Entwicklung der Leistungsfähigkeit eines Chips : "Alle 18 Monate verdoppelt sich die Anzahl der Transistoren eines Chips"

redundanter Hardware. So werden VGA, Floppy, Maus und Tastatur nur einmalig zur Installation eines netzwerkfähigen Betriebssystems benötigt, sowie bei der Wartung vor Ort. Beide Punkte, Installation und Wartung, sind ab einer bestimmten Größe des Clusters ein nicht zu unterschätzendes, administratives Problem. Wünschenswert wäre eine Lösung, welche es erlaubt die einzelnen PCs auf der Ferne zu kontrollieren und zu warten, sowie eine Ferninstallation der Software gestattet. Der im Rahmen dieser, und der Diplomarbeit von Stefan Philipp entwickelte Cluster Interface Adapter (C.I.A.) zeigt eine mögliche Implementation eines solchen Kontroll-Systems.

# Kapitel 2

# Der Cluster Interface Adapter

Der C.I.A. ist ein FPGA[1]-basierter, autonomer, netztwerkfähiger Steuerrechner in Form einer PCI-Steckkarte zur Ferndiagnose und -kontrolle eines Standard-PC (Intel-PC mit PCI-Bussystem), im folgenden Host genannet. Sein primärer Einsatz ist die Kontrolle und Wartung von PC-Farmen, sogenannte Cluster. Dies wird realisiert durch die Fähigkeit des C.I.A. den Host über PCI zu scannen, Mouse, Keyboard, FDD[2] zu emulieren, sowie die VGA-Karte zu ersetzten und den Bildschirm zu exportieren. Das folgende Unterkapitel behandelt alle Anforderungen, welche an den C.I.A. gestellt werden.

## 2.1 Anforderungen an den C.I.A.

Um eine zufriedenstellende Kontrolle des Hosts zu gewährleisten, werden folgende Bedingungen an den C.I.A. gestellt:

- Autonomie des C.I.A : der C.I.A. muss jederzeit in der Lage sein vollkommen unabhängig vom Host zu operieren. Dies bedeutet, dass er über eine eigene Schnittstelle zur Außenwelt, sowie eine autonome Stromversorgung verfügen muss.

- PCI-Scan des Hostsystems : der PCI-Bus ermöglicht es, das komplette Host-System zu scannen und damit die aktuelle System-Hardware zu ermitteln und ihre Konfiguration auszulesen. Der C.I.A. soll in der Lage sein, den Bus anzufordern und Bus-Transaktionen zu initialisieren, eine Fähigkeit, welche man als Bus-Master fähig bezeichnet. Dies ermöglicht es dem C.I.A. die Hardware des Host-Systems zu scannen, auszulesen und gegebenfalls zu beschreiben. Dies geschieht direkt über den PCI-Bus und geht nicht über das Betriebssystem des Hosts.

- FDD-Emulation : einige Betriebssystem, darunter vor allem Linux, bieten bei der Installation eines Rechners die Möglichkeit einer Netz-Installation. Dazu wird zuerst ein minimales Betriebssystem über das FDD geladen, welches die Netzfunktionalität implementiert, und danach das eigentliche System über das Netz installiert. Der C.I.A. soll zur Fern-Installation eines Betriebssystems, oder generell zur Datenübertragung, die Möglichkeit bieten, dem Host ein FDD zu emulieren. Die Datenübertragung vom C.I.A. zum Host geschieht aus der Sicht des Hosts über eine normale Floppy, d.h. die Datenübertragung ist transparent.

- Mouse/Keyboard-Emulation : der C.I.A. soll dem Host beide Eingabegeräte vortäuschen. Somit ist es möglich, Eingaben an den Host zu tätigen, wenn keine Möglichkeit mehr

[1] Field Programmable Gate Array
[2] Floppy Disc Drive

über das Netzwerk besteht, z.B. im Falle eines Netzwerk-Problems oder bei der Konfiguration des BIOS.

- Steuerung der Control-Lines : moderne Rechner verfügen über steuerbare Netzteile, welche es erlauben, den Host über bestimmte Control-Lines an- und abzuschalten, oder zu reseten. Hierüber soll der C.I.A. in der Lage sein, den Host zu rebooten oder, im schlimmsten Fall, komplett abzuschalten.

- VGA-Emulation : um den Host aus der Ferne kontrollieren zu können, insbesondere bei der Konfiguration des BIOS, ist es nötig, den Bildschirm zu exportieren. Dazu ist es prinzipiell möglich, über den PCI-Scan, den Bildschirm-Speicher einer vorhandenen VGA-Karte direkt auszulesen und über das Netzwerk des C.I.A. zu exportieren. Da jedoch bei Cluster-PCs die VGA-Karte nach der Installation eine unnötige Komponente im System darstellt, kann man diese konsequenter weise durch eine entsprechende VGA-Emulation des C.I.A. ersetzten. In diesem Fall gibt sich die C.I.A.-Karte dem Host als VGA-Karte zu erkennen. Bildschirm-Daten werden dann direkt in den Speicher der C.I.A. geschrieben und können von dort exportiert werden.

- Netzwerkfähigkeit : da die Kontrolle des C.I.A. nicht über den Host laufen darf, muss der C.I.A. ein eigenes Netzwerkinterface besitzen.

- Flexibilität : zukünftige Schnittstellen und Sensoren sollen möglichst einfach in den bestehenden C.I.A. integriert werden können.

## 2.2 Konzept

Betrachtet man die oben genannten Anforderungen, so wird einem schnell klar, dass die Realisierung in reiner Elektronik unmöglich ist, sondern ein komplexes, digitales System mit den entsprechenden Schnittstellen erfordert. Weiterhin muss dieses System sich komplett auf einer PCI-Steckkarte unterbringen lassen. Unter diesen Gesichtspunkten kristallisieren sich zwei Möglichkeiten heraus: ASICs - Application Specific Integrated Circuits - oder FPGAs - Field Programmable Gate Arrays. Für die Implementation des C.I.A. wurde sich aus folgenden Gründen für die zweite Möglichkeit, den FPGA, entschieden :

- Kosten: FPGAs sind im Vergleich zu ASICs bei kleinen bis mittleren Stückzahlen deutlich billiger als ASICs.

- Hohe Integrationsdichte: Heutige FPGAs sind ausreichend komplex um das komplette Design aufnehmen zu können.

- Flexibilität: Während ASICs für eine bestimmte Aufgabe entwickelt werden und danach nicht mehr geändert werden können, bieten FPGAs die Möglichkeit beliebig oft rekonfiguriert zu werden. Dies ermöglicht zum einen die Verifikation eines Designs über die Simulation hinaus, zum anderen ist ein Update des Systems sehr einfach.

- SOPC : System-On-a-Programmable-Chip. Einige Firmen bieten sogenannte SOPC-Solutions an. Dies sind komplette, vorgefertigte und modular aufgebaute Systeme, welche nach dem Baukasten-Prinzip zu eigenen Systemen zusammengestellt werden können. Darüber hinaus könne eigene Module entwickelt und in ein System integriert werden. Das komplette System kann, nachdem es konfiguriert und synthetisiert wurde, in einen FPGA geladen werden.

Vor allem die letzten beiden Punkte, die Flexibilität und das Konzept des SOPC geben dem FPGA den Vorrang gegenüber einem ASIC. Das SOPC ermöglicht es, die komplette Logik in einem einzigen Chip unterzubringen, welcher beliebig oft rekonfigurierbar ist. Außerdem muss nicht ein komplettes System von Grund auf neu entwickelt werden, sondern es können vorgefertigte Systemkomponenten genutzt und mit eigenen Designs kombiniert werden. Dieses noch etwas abstrakte Konzept wird im nächsten Abschnitt konkret für den C.I.A. vorgestellt.

## 2.3 Implementation

Bei der Wahl des Systems für den C.I.A. wurde sich für das NIOS-System der Firma ALTERA entschieden. ALTERA FPGAs werden seit einigen Jahren im Rahmen der Forschungsprojekte der TI-Gruppe eingesetzt und man verfügt über einen große Erfahrungschatz in dieser Technologie. Desweiteren wurde eine erste Version des C.I.A. in Stefan Philippps Diplomarbeit entwickelt, dort befanden sich jedoch nur dessen C.I.A.-Designs im FPGA, das $\mu$C-Linux System war auf einer eigenen Platine untergebracht, verfügte über einen Hardcore-Prozessor und war nicht erweiterbar. Mit dem NIOS-System konnte nun nicht nur die komplette System-Logik in den FPGA gebracht werden, sondern es existiert auch ein $\mu$C-Linux-Betriebssystem für das System inklusive frei verfügbarem Source-Code.

Das System besteht aus einem 32-Bit RISC Prozessor, dem NIOS, mehreren Peripheriemodulen zur Ansteuerung von externen Bausteinen wie RAM, FLASH, usw. und dem AVALON-Bus, der alle Systemkomponenten untereinander verbindet. Das System wird über den SOPC-Builder zusammengestellt und konfiguriert, weiterhin können eigene Module über entsprechende Schnittstellen integriert werden. Auf dieser Hardware kann nun ein Betriebssystem aufbauen, im Falle des C.I.A. ein Linux-Betriebssystem, das $\mu$C-Linux[3]. Das $\mu$C-Linux ist netzwerkfähig, gestattet also ein Einloggen in das System über ein Netzwerk-Interface. Weiterhin können eigene Appliktionen geschrieben und in das System eingebunden werden. Diese Applikationen können über das NIOS-System auf die restlichen C.I.A.-Designs zugreifen. Somit ist sowohl eine Kontrolle des C.I.A. als auch des Hosts über ein Netzwerk möglich.

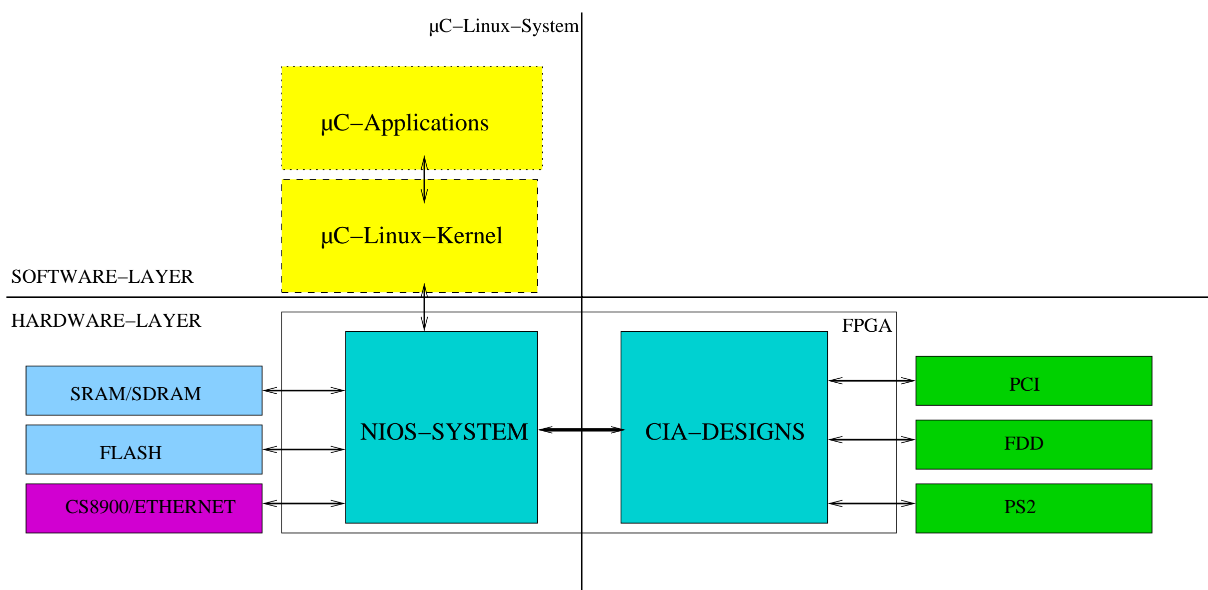


Abbildung 2.1: Konzept des kompletten Systems

[3]Micro Controller Linux

Abb. 2.1 gibt einen Überblick über das komplette System. Auf der Hardware-Ebene, im Innern des FPGA, befindet sich das NIOS-System mit seinen Schnittstellen zur Peripherie, sowie die übrigen C.I.A.-Designs zur Steuerung und Kontrolle des Hosts. Außerhalb des FPGAs befinden sich der Speicher (SRAM[4], SDRAM[5], FLASH) und der Netzwerk-Chip des NIOS-Systems, sowie die FDD-, PCI-, PS/2-Connectoren der C.I.A.-Designs. Auf das NIOS-System baut der Linux-Kernel und darauf die Applikationen auf. Bei den Applikationen kann es sich sowohl um Standard-Applikationen des $\mu$C-Linux handeln, als auch um Eigene zur Steuerung der C.I.A.-Designs.

[4]Static Random Access Memory

[5]Synchronous Dynamic Random Access Memory

# Kapitel 3

# FPGAs

Dieses Kapitel geht in recht knapper Form auf die Grundlage des C.I.A. ein, den FPGA. Neben der allgemeinen Funktionsweise und dem Designflow für FPGAs wird abschließend die FPGA-Familie vorgestellt, welche auf dem C.I.A. zum Einsatz kommt, die APEX20KE von ALTERA.

## 3.1 Funktionsweise eines FPGAs

Obwohl sich die exakten Funktionsweisen der FPGAs von Hersteller zu Hersteller unterscheiden ist der grundsätzliche Aufbau bei allen der gleiche: Einzelne Funktionsblöcke, je nach Hersteller LogicCells, LogicElements, usw. genannt, dienen zur Aufnahme der logischen Funktionen. Ein Block kann entweder eine logische Funktion implementieren, oder mit mehreren anderen Blöcken zu einer Funktion zusammengeschaltet werden, aber nie kann ein Block mehr als eine Funktion enthalten. Daher sind diese Blöcke meist klein gehalten, um eine höhere Ausnutzung zu gewährleisten. Man spricht auch von einer feinen Granularität des FPGAs. Alle Blöcke sind durch ein oder mehrere Verbindungsnetze verbunden, welche wie die einzelnen Blöcke konfiguriert werden können. Somit ist der Aufbau komplexer logischer Systeme möglich. Da ein System ohne Daten, welche es verarbeiten kann, keinen Sinn macht, verfügen die FPGAs über I/O-Blöcke. Diese stellen die Schnittstelle zwischen den I/O-Pins der FPGAs und den Verbindungsnetzen dar. Signale von außen werden so über die I/O-Pins und die I/O-Blöcke über das Verbindungsnetz zu den einzelnen Funktionsblöcken geleitet. Moderne FPGAs sind außerdem in der Lage, die I/O-Blöcke auf die entsprechenden Pegel der von außen angelegten Signale einzustellen. I/O-Blöcke, Interconnektions-Netze und Funktionsblöcke bilden die generische Struktur der FPGAs. Um diese Struktur nutzen zu können, muss der FPGA konfiguriert werden. Dies geschieht über einen Bitstrom, welcher auf verschiedene Arten in den FPGA übertragen werden kann. Mögliche Varianten der Programmierung sind über JTAG[1], EEPROMs, Configuration Controller, usw. Nachdem ein FPGA konfiguriert wurde, behält er seine Konfiguration bis zum nächsten Power-Up. Eine erschöpfendere Beschreibung der Funktionsweise von FPGAs und Logik-Bausteinen im allgemeinen findet sich in [12].

## 3.2 Design-Flow für einen FPGA

Abb. 3.1 skizziert den prinzipiellen Design-Flow für einen FPGA.
Die Grundlage eines jeden guten Designs ist ein entsprechendes Konzept zur Implementation der gewünschten Funktionalität. Mit einem Konzept sollte daher auch das Design für einen

[1] Joint Test Action Grout

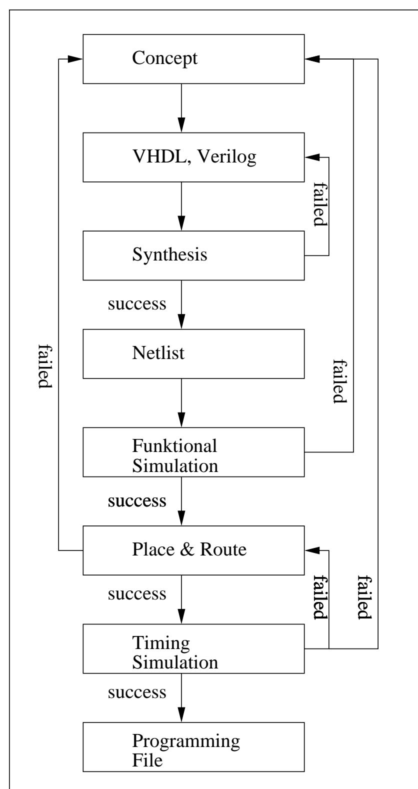


Abbildung 3.1: Design-Flow für FPGAs

FPGA starten. Nachdem das Konzept ausformuliert wurde, beginnt die Umsetzung in eine Hardware-Beschreibungssprache, meist VHDL[2] oder Verilog. Diese stellt eine Abstraktionsebene zwischen der späteren Logik, welche nur aus Gattern besteht, und menschenlesbarem Code dar, ähnlich C oder Assembler. Anschließend wird das Design synthetisiert und auf eine Netzliste abgebildet, welche das Design entweder auf Gattereben beschreibt oder schon auf die Zieltechnologie abbildet. Misslingt diese Synthese, so gibt es hierfür 2 Gründe. Entweder ein Syntaxfehler im Code, welcher recht einfach zu beheben ist, oder das Design ist nicht synthetisierbar. Im Gegensatz zu Software-Programmiersprachen kann nicht jeder Code synthetisiert werden, da er auf Gatter/Technologien abgebildet, welche physikalischen Gesetzen unterliegen. Ist ein Code nicht synthetisierbar, so muss das Konzept neu überdacht und codiert werden. Liegt die Netzliste vor, so kann eine erste, funktionale Simulation des Designs durchgeführt werden. Ist diese erfolgreich, so wird mit Place & Route die Netzliste auf den gewählten Baustein abgebildet, ist sie es nicht, so muß der Fehler im Konzept gesucht werden. Nach dem Place & Route liegt die Konfiguration des FPGAs fest und es kann ein Timing-Simulation durchgeführt werden, da die Signallaufzeiten zwischen den einzelnen Funktionsblöcken und der Propagationdelay[3] innerhalb der Funktionsblöcke bekannt ist. Sollte die Timing-Simulation negativ ausfallen, so kann eventuell ein neuer Place & Route Versuch zum Erfolg führen, falls nicht, muss das Konzept geändert werden. Ist die Timing-Simulation erfolgreich, so ist es sehr wahrscheinlich, dass das Design das gleiche Verhalten im FPGA zeigt und das entsprechende Programming-File kann zur Konfiguration verwendet werden. Über eine entsprechende Schnittstelle zwischen Designs-Software und FPGA kann dieser konfiguriert und das Design getestet werden.

[2]VHSIC Hardware Describtion Language

[3]Zeitdifferenz zwischen Eingangssignalen und stabilen Ausgangssignalen einer Funktion

## 3.3 ALTERA FPGAs

Bei dem, auf dem C.I.A. verwendeten FPGA handelt es sich um einen APEX 20K400EFC aus der APEX20KE-Family der Firma ALTERA. Die APEX20KE-Family unterscheidet zwei unterschiedliche Arten von Funktionsblöcken, die Logic-Elements(LEs) und Embedded System Blocks(ESBs), welche über verschiedene Interconnections-Netze verbunden sind. Für die Verbindung nach außen sorgen spezielle I/O-Cells.

**Logic Elements (LE)** bestehen jeweils aus einer Look-up-table(LUT) auf SRAM-Basis mit 4 Eingängen und einem programmierbaren Register, welches wahlweise ein D-,T-,JK- oder S/R-Latch sein kann. Zwei schnelle Netzwerke, Carry- und Cascade-Chain verbinden jeweils benachbarte LEs und dienen zur Realisierung großer Funktionen,welche sich über mehrere LEs erstrecken. Jede LE verfügt über zwei Ausgänge welche die verschiedenen Interconnectionsnetzwerke treiben. Spezielle Bypässe ermöglichen es, die LUT und das Register einer LE gleichzeitig, aber unabhängig voneinander zu betreiben. Dies erhöht die Flexibilität und den Nutzungsgrad einer LE. Weiterhin verfügt jede LE über 4 Eingänge für globale Steuersignale, wie *Clock* und *Reset*. Die globalen Steuersignale zeichnen sich dadurch aus, daß sie jeden Funktionsblock (LE, ESB, I/O) mit minimaler Zeitverzögerung erreichen, d.h. minimale Skew.

**Embedded System Blocks (ESB)** dagegen bieten die Möglicheit entweder Memory zu implementieren (RAM, ROM, FIFO, DUALPORT-RAM) oder "Product-term-logic" für State-Machines und komplexe, kombinatorische Logik.

**I/O-Cells** stellen die Schnittstelle zwischen der Logik des FPGAs und seinen I/O-Pins da. Sie können entweder als Eingang, Ausgang oder Bidirektional genutzt werden. Jede I/O-Zelle ist gepuffert und verfügt über Input- und Output-Register um Timing-Requirements wie Setup- und Hold-Time einhalten zu können. Weiterhin unterstützt jede I/O-Zelle folgende Standards : LVTTL, LVCMOS, 1.8-V I/O, 2.5-V I/O, 3.3-V PCI, PCI-X, 3.3-V AGP, LVDS, LVPECL, GTL+, CTT, HSTL Class I, SSTL-3 Class I & II, und SSTL-2 Class I & II.
Durch diese Architektur ist es möglich komplette Systeme auf einem Chip zu entwickeln, da Datenpfade, Kontrollstrukturen und Speicher/Register leicht umsetzbar sind. Für vertiefende Informationen sei auf den entsprechenden Datasheet verwiesen [13]

## 3.4 QUARTUS II

Für das Place & Route eines Designs auf den FPGA ist eine herstellerspezifische Software nötig, im Falle von ALTERA FPGAs ist dies QUARTUS II. QUARTUS II ist eine komplette Entwicklungsumgebung und umfaßt einen Editor (AHDL[4], VHDL, Verilog), einen Compiler, einen Simulator zur Funktional/Timing-Simulation, einen Fitter zum Place & Route und einen Programmer zum Programmieren des FPGAs über eine JTAG-Schnittstelle. Damit läßt sich theoretisch der komplette Design-Flow abdecken. Theoretisch ! In der Praxis wurde, soweit möglich, zur Simulation ModelSim und zur Synthese LeonardoSpektrum verwendet, da QUARTUS in diesen Punkten einige Schwachpunkte aufweist. Lediglich zum Place & Route muss QUARTUS verwendet werden.

[4]ALTERA Hardware Describtion Language

# Kapitel 4

# Das NIOS-System

Das NIOS-System stellt ein komplettes System dar, bestehend aus einer 32-Bit RISC-CPU, AVALON-Bussystem und Peripherie-Modulen, welches vollständig im FPGA untergebracht ist. Zusammen mit externer Peripherie wie SRAM, SDRAM, FLASH und RS232 bildet es einen kleinen, aber vollständigen Rechner. Das System wird über ein entsprechendes Tool, den SOPC-Builder, zusammengestellt und anschließend synthetisiert. Der SOPC-Builder kann dabei nicht nur die vorgefertigten System-Module konfigurieren, sondern gestattet außerdem die Integration eigener Designs in das System. Das so generiert System liegt als einzelne Design-Entity vor, mit den entsprechenden Ports nach außen. Diese Entity kann entweder weiter in andere Designs eingebunden werden, oder als Top-Level-Entity benutzt werden.

## 4.1 Das EXCALIBUR Development-Kit

Das EXCALIBUR Development-Kit ist eine integrierte Entwicklungsumgebung für den NIOS Embedded Prozessor. Es besteht aus dem EXCALIBUR Development-Board und einem Development-Softwarepaket zur Systementwicklung. Die folgenden Abschnitte gehen jeweils auf das Board und die Software ein.

### 4.1.1 Das Development-Board

Das Development-Board stellt die Hardware-Plattform für das NIOS-System dar. Neben einem APEX20K200 FPGA von ALTERA verfügt es über die folgenden Ressourcen:

- 256KByte SRAM : 2 SRAM-Chips(64Kx16) wurden parallel geschaltet und können so dem NIOS als 32-Bit breiter Systemspeicher dienen.
- JTAG : Über die JTAG-Schnittstelle kann der FPGA direkt über den Parallelport konfiguriert werden.
- CPLD : Ein Baustein der MAX-Familie dient als Configuration Controller. Er konfiguriert bei Power-Up den FPGA über ein im FLASH-Memory gespeichertes Konfigurationsfile.
- 1MByte FLASH : der FLASH-Baustein(512Kx16) kann sowohl zur Konfiguration des FPGAs verwendet werden, als auch vom NIOS als nichtflüchtiger 16-Bit breiter Systemspeicher. Da das FLASH zwei Funktionen erfüllt und auch von 2 unterschiedlichen Device angesprochen wird, nämlich vom NIOS und dem Configuration-Controller, wurde es in 2 Bereiche untergliedert. Die untersten 512KByte stehen dem NIOS als Systemspeicher zur Verfügung, die oberen 512KByte dienen zur Aufnahme des FPGA-Konfigurationfiles. Aus Sicherheitsgründen wurde der Konfigurationsbereich wieder in 2

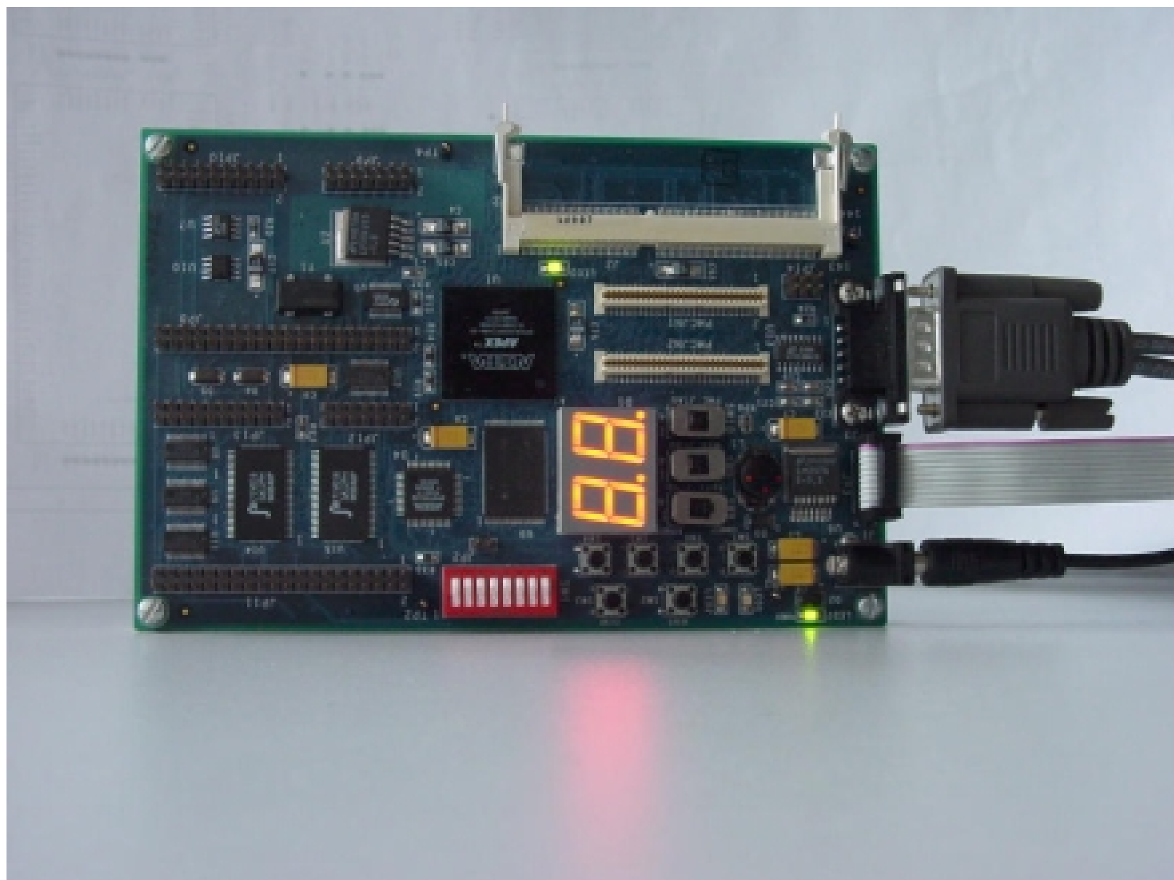

Abbildung 4.1: EXCALIBUR Development Board

Bereiche unterteilt, den Factory- und den Userspace. Normalerweise wird der FPGA aus dem Userspace konfiguriert. Sollte dabei etwas schieflaufen, so enthält der Factoryspace ein auf das Development-Board zugeschnittenes Referenzdesign als Fall-back Lösung. Das Überschreiben dieses Bereichs ist aus Sicherheitsgründen nur durch Setzten eines Jumpers möglich.

- RS232-Connector : dient zur Kommunikation mit dem Board über ein Terminal und zum Debuggen des NIOS. Dazu können über den RS232-Connector zwei serielle Schnittstellen (UARTs) nach außen geführt werden.

- 40 3.3V-I/O Ports: Dienen zur Aufnahme von Erweiterungs-Kits (Ethernet-Kit, Linux-Kit) oder können als freie I/O-Ports verwendet werden.

- 40 5.0V-I/0 Ports: Dienen zur Aufnahme von Erweiterungs-Kits (Ethernet-Kit, Linux-Kit) oder können als freie I/O-Ports verwendet werden.

- S0DIMM-Connector: Dient zur Aufnahme von Standard S0DIMM-Speichermodulen oder zur Aufnahme des $\mu$C-Linux Memory-Moduls.

- Oszillator 33MHz. Versorgt den NIOS und die restliche Peripherie mit einem globalen Taktsignal.

- LEDs, Buttons, 2x 7Seg-LEDs, DIPSwitches : Haben für das NIOS-System keine wirkliche Bedeutung, können jedoch zum Testen von eigenen Designs verwendet werden.

- Zwei IEEE-1386 PCI-Mezzanine-Connectors : ermöglichen eine Erweiterung der Systemhardware über das Board hinaus.

Anhand dieses Boards war es möglich, einen Einstieg in die Systementwicklung zu finden und ein grundlegendes Verständniss für die Funktionsweise des NIOS zu bekommen, sowie für das Zusammenspiel der einzelnen Komponenten. Die hier gesammelten Erfahrungen bildeten die Grundlage für den späteren Export des NIOS-Systems vom dem EXCALIBUR Development-Board auf das C.I.A.-Board. Zugleich stand eine Referenz zur Verfügung, welche durch vergleichende Messungen geholfen hat, den Prototypen des C.I.A.-Boards zu debuggen. Abb. 4.1 zeigt das EXCALIBUR Development Board.

### 4.1.2 Die Development-Software

Das Softwarepaket umfaßt alle Tools die zur Generierung eines eigenen NIOS-Systems und zur Erstellung von NIOS-Applikationen nötig sind.

- SOPC-Builder : Mittels des SOPC-Builder wird das NIOS-System-Modul zusammengestellt und man erhält wahlweise ein AHDL-, VHDL-, oder Verilog-File, oder eine synthetisierte Netzliste. In beiden Fällen erhält man ein Designfile für die weiter Verwendung. Zusätzlich erzeugt der SOPC-Builder für alle im System-Modul enthaltenen Komponenten Library-Files für den GNUPro Cross-Compiler. Diese sind zur Erstellung eigener Applikationen zwingend notwendig. Der SOPC-Builder ist eine zentrale Instanz in der Generierung eigener Systeme, weshalb ihm ein eigener Abschnitt gewidmet ist.

- QUARTUS II : QUARTUS ist keine NIOS-spezifische Software, sondern eine allgemeine Design-Software für ALTERA FPGAs. Das mit dem SOPC-Builder erzeugte System-Modul kann hier in andere Designs eingebunden werden oder seperat verwendet werden. Nach der Synthese und dem Place&Route durch QUARTUS kann der FPGA mit dem Design programmiert werden.

- GNUPro : GNUPro ist ein sogenannter Cross-Compiler. Dieser erlaubt es Software für anderer Prozessor-Architekturen zu kompilieren, als die, auf der der Compiler läuft. Somit ist es möglich auf einem x86-Rechner oder einer Unix-Workstation Applikationen für den NIOS zu kompilieren. Dazu benötigt der Cross-Compiler allerdings bestimmte Librarys, welche das Ziel-System beschreiben. Dies sind die Library-Files, welche vom SOPC-Builder bei der Generierung des NIOS-System-Modul erzeugt werden. Sie enthalten das Adress-Mapping der einzelnen Komponenten, sowie komponenten-spezifische Routinen zur Ansteuerung.

- NIOS-Bash : Die NIOS-Bash ist eine Shell, welche zur Entwicklung des NIOS-Systems konfiguriert wurde und über entsprechende Tools zur Kommunikation mit dem NIOS verfügt. Mit ihnen kann man die Ein- und Ausgabe des NIOS in ein Terminal umleiten, Software in den Speicher des NIOS laden und ausführen, sowie das FLASH-Memory über den NIOS beschreiben. Eine Auflistung der einzelnen Tools, sowie ihrer Funktion findet sich im Anhang wieder.

## 4.2 Der SOPC-Builder

Das Erstellen des NIOS-System Moduls geschieht durch eine entsprechende Software, den SOPC-Builder. Dieser gestatte es dem User sich sein System aus einzelnen Komponenten zusammenzubauen. Abb. 4.2 illustriert die Vorgehensweise bei der System-Generierung mittels des SOPC-Builder. Grundlage eines jeden Systems ist der NIOS-Prozessor und der

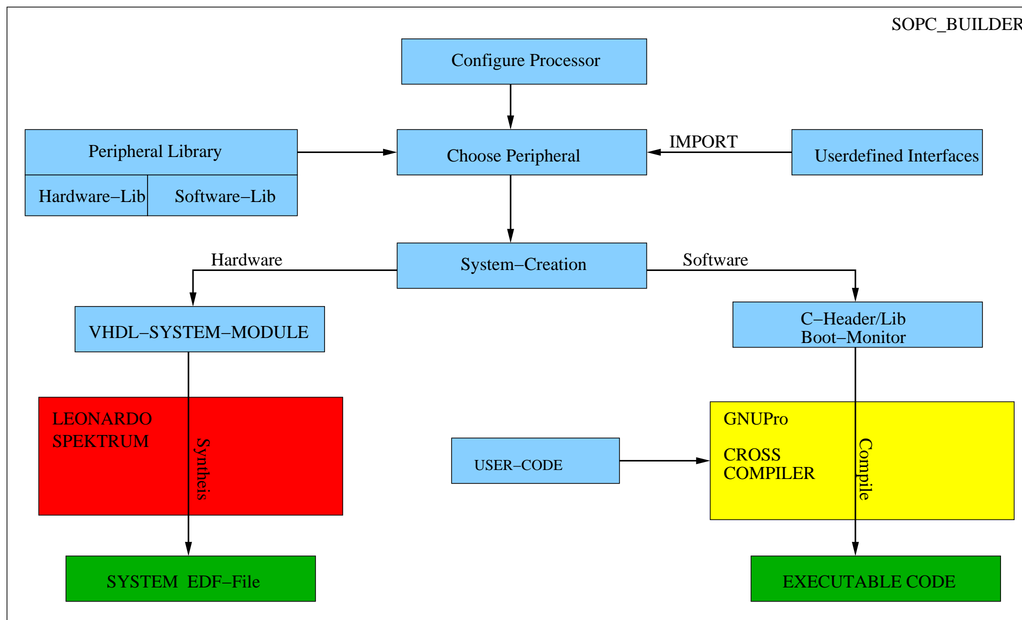


Abbildung 4.2: SOPC-Builder

AVALON-Bus, welcher den NIOS mit der übrigen Peripherie verbindet. Zuerst wird der Prozessor konfiguriert. Danach können die einzelnen Peripherie-Module in den Adress-Raum des NIOS-Prozessors gemappt werden. Peripherie-Module können entweder aus einer vordefinierten Peripherie-Library gewählt werden, oder eigene User-Interfaces zur Hardware sein. Die Peripherie-Library enthält für jedes Modul sowohl eine Hardware-Lib, als auch, falls das Modul dies benötigt, eine Software-Lib. Die Hardware-Lib besteht immer aus einem *class.ptf* File, welches die Schnittstellen des Moduls zu AVALON-Bus und das Timing spezifiziert, zusätzlich kann ein Modul seinen eigenen AHDL-,VHDL-, oder Verilog-Code mitbringen, dies wird ebenfalls in dem File vermerkt. Das File ist in einer menschenlesbaren Skriptsprache geschrieben, der genaue Befehlssyntax findet sich in [16] wieder. Anhand dieser Hardware-Lib bindet der SOPC-Builder die Peripherie-Module in das System-Modul ein. Weiterhin werden aus der Software-Lib die Header- bzw. Library-Files für das erzeugte System erstellt. Diese können zusammen mit eigenem Code von dem GNUPro Cross-Compiler zu Applikationen für das NIOS-System compiliert werden. Will man eigene Hardware in das System integrieren, so bieten sich zwei Möglichkeiten. Entweder man schreibt ein entsprechendes *class.ptf* File, oder man nutzt die Import-Funktion des SOPC-Builders. Diese scannt die Top-Level Entity des Designs, importiert die Ports und überläßt dann dem User die Zuweisung dieser Ports zu den entsprechenden Ports des AVALON-Busses und das Timing. In dieser Diplomarbeit wurde überwiegend von erster Möglichkeit gebrauch gemacht, d.h. *class.ptf* Files geschrieben. Diese Methode ist etwas komplexer, als der Import der Ports, bietet aber eine wesentlich größere Flexibilität bei der Nutzung der Ports und bei den Spezifikationen des Timings. Für ein tiefergehendes Verständniss des SOPC-Builders und des AVALON-Busses sei auf entsprechende Literatur verwiesen: [16] und [21].

## 4.3 NIOS-System-Modul

Abb. 4.3 zeigt das NIOS-System-Modul und die Verbindungen der einzelnen Komponenten.

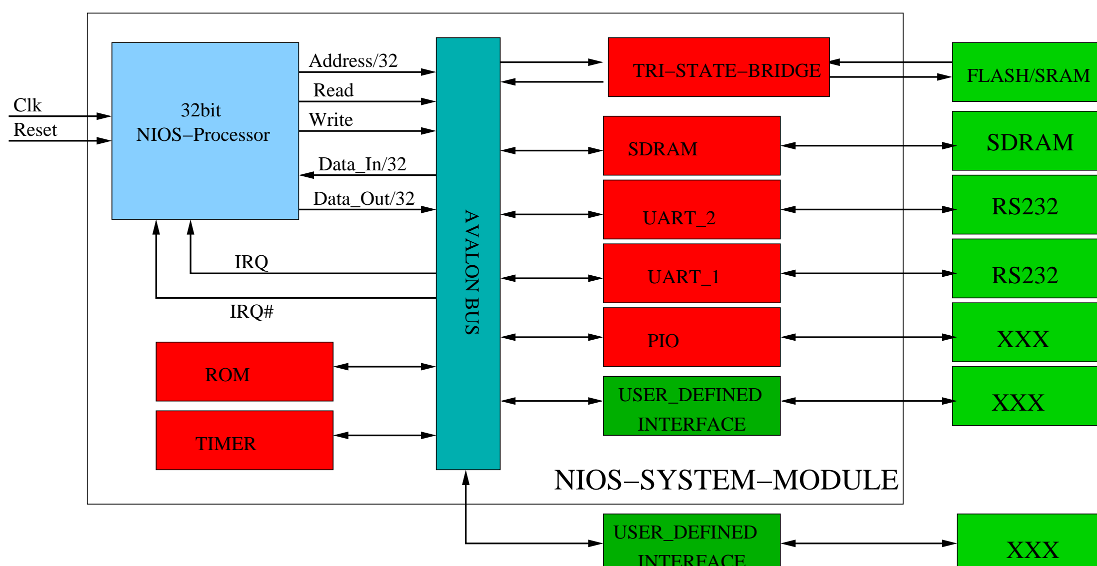


Abbildung 4.3: Das NIOS System-Modul

### 4.3.1 Der NIOS-Prozessor

Der NIOS-Prozessor ist ein 32-Bit RISC Prozessor, welcher als Source-Code vorliegt und nach seiner Konfiguration synthetisiert wird. Er verfügt über folgende Eigenschaften:

- Harvard-Architektur : Instruktions- und Datenspeicher sind voneinander getrennt.
- Alle Daten- und Addresspfade sind intern registert und dadurch voll synchron.
- 4-stufige Pipeline zur Erhöhung des Instruktionsdurchsatzes
  - Instruction Fetch
  - Instruction Decode / Operand Fetch
  - Execute
  - Write-Back

  Diese bietet außerdem die Möglichkeit, unmittelbar nach einer Branch-Instruktion, aber noch vor dem eigentlichen Branch, in einem sogenannten Branch-Delay-Slot eine Instruktion auszuführen.
- Register-File mit 128, 256 oder 512 Registern. Das Register-File ist als "Windowed register file" angelegt. Durch ein Fenster werden immer nur 32 Register gleichzeitig dargestellt, dieses Fenster kann jedoch verschoben werden. Durch einen Überlab der Fenster ist es möglich, beim Aufruf oder Rücksprung einer Subroutine Werte sehr schnell zu übergeben.
- Interrupthandler für 64 (6-Bit) Hardware Interrupts.
- Interrupthandler für Software Interrupts.
- Custom Instructions : Die ALU kann durch eigene Designs ergänzt werden. Der SOPC-Builder generiert automatisch den Op-Code, sowie Assembler- und C-Makros.

Abb. 4.4 gibt abschließend eine Überblick über den Prozessor. Die Pipeline-Struktur, sowie die registrierten Daten- und Addresspfade sind gut anhand der DFFs zu erkennen.

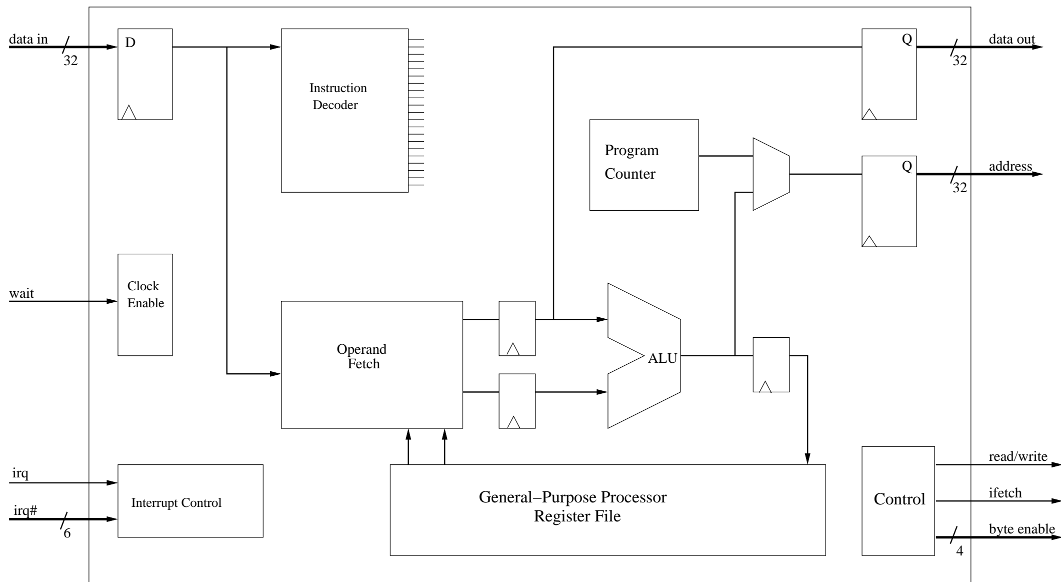


Abbildung 4.4: NIOS 32b-RISC-Processor

### 4.3.2 Der AVALON Bus

Der AVALON-Bus stellt den Backbone des System-Moduls dar, er übernimmt die komplette Kommunikation zwischen dem Master (NIOS) und den Slaves (Peripherie-/User-Module). Jedes Modul hat eine eigens für sich definierte Schnittstelle zum Bus, den sogenannten Port. Diese Ports werden vom SOPC-Builder generiert, entweder aus mitgelieferten Librarys, wie bei den Peripherie-Modulen, oder über eigene Librarys bei User-Modulen. Über diese Ports steuert der AVALON-Bus den Datenfluss und übernimmt dabei folgende Aufgaben:

- Arbitrierung : Da das NIOS-System-Modul nur einen Master enthält, fällt die Arbitrierung weg, ist aber prinzipiell möglich.
- Address-Mapping : Peripherie-/User-Module können fast beliebig im 32-Bit Addressraum gemappt werden. Der SOPC-Builder generiert automatisch ein Mapping-File für den Assembler/C-Compiler.
- Chipselect decoding : für die einzelnen Module wird entsprechend ihrer Adresse ein CS-Signal generiert, sodaß keine Address-Decodierung auf seiten der Peripherie nötig ist.
- Dynamik Bus sizing : der AVALON-Bus stellt jedem Master/Slave die Daten in der vollen Breite des jeweiligen Datenbusses bereit. Liest z.B. ein 32-Bit Master von einem 16-Bit Slave, so werden 2x ein 16-Bit Wort vom Slave gelesen und an den Master ein 32-Bit Wort übergeben.
- Wait-State generation : Benötigt ein Slave-Module mehrere CPU-Zyklen oder bestimmte Setup/Hold-Times, so kann der Master für diese Zeit in eine Wait-State versetzt werden (STALL).
- IRQ generation : Jedem Modul kann einer der 64 möglichen Hardware-IRQs zugeordnet werden.

Abb. 4.5 skizziert den Anschluß eines Master und mehrer Slaves an den AVALON-Bus. Für die exakte Spezifikation der Ports sei auf die Avalon-Bus-Specifikation [21] verwiesen.

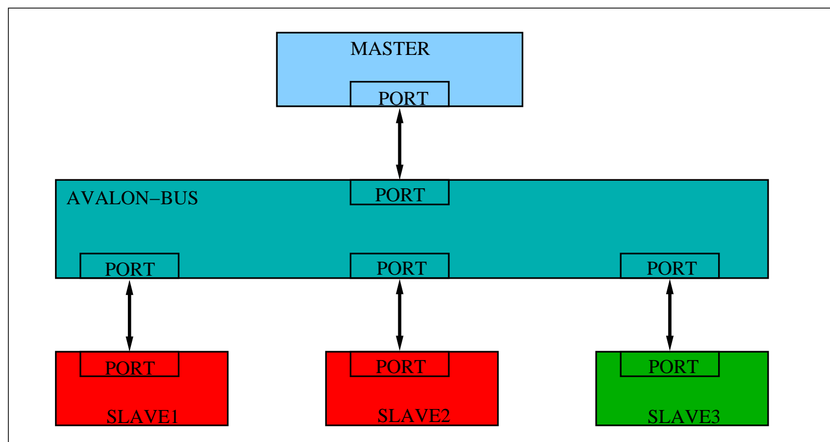


Abbildung 4.5: AVALON-Bus mit einem Master und mehreren Slaves

### 4.3.3 Peripherie-Module

Die Peripherie-Module werden mit dem SOPC-Builder mitgeliefert und können in 3 Klassen eingeordnet werden:

- Peripherie innerhalb des System-Moduls: Timer, Onchip RAM/ROM
- Peripherie außerhalb des System-Moduls aber innerhalb des FPGAs (Onchip) : z.B. Megafunctions.
- Peripherie außerhalb des System-Moduls und des FPGAs : UART, SDRAM-Controller, SRAM, FLASH.

Bis auf das ROM soll hier nicht weiter auf die Peripherie-Module eingegangen werden, ihre Funktionen erklären sich aus ihren Namen. Das ROM dagegen enthält den G.E.R.M.S.-Monitor, ein 1KByte großes "Betriebssystem" für den NIOS. Dieser wird ausführlicher in 4.3.5 "G.E.R.M.S. Monitor" behandelt.

### 4.3.4 User-Module

Wie erwähnt bietet der SOPC-Builder bietet zwei Möglichkeit, eigene Designs in das NIOS-System zu integrieren:

- *class.ptf* Library-Files : definiert die Port-Schnittstellen. Kann in lesbarem Text geschrieben werden und enthält alle Portzuweisungen zwischen den Ports der Design-Entity und möglichen AVALON-Ports, sowie die Timing-Requirements.
- Import der Design-Ports mittels SOPC

## 4.4 G.E.R.M.S. Monitor

Der G.E.R.M.S Monitor stellt eine Art rudimentäres Betriebssystem für den NIOS dar und ermöglicht es dem User, sich Speicherinhalte anzusehen, zu manipulieren, Programme in den Speicher oder ins FLASH zu laden, das FLASH zu löschen, sowie die Übergabe der Kontrolle an Programme. Dabei steht G.E.R.M.S für

- G - GO : Programmaufruf

- E - ERASE : Löscht das FLASH
- R - RELOCATE : legt Zieladdresse für nächsten Download fest
- M - MONITOR : zeigt den Speicherinhalt ab einer bestimmten Adresse an
- S - STORE : speichert ein S-Record ab einer bestimmten Adresse

Der genaue Befehlssynthax findet sich in [14] wieder.
Bei der Erstellung des Systemmoduls mittels des SOPC-Builders besteht die Möglichkeit ein ROM innerhalb des Systemmoduls anzulegen. Wird dieses ROM an Adresse 0 gemapped, so führ der NIOS bei einem Reboot oder Reset automatisch den im ROM enthaltenen Code aus. Der G.E.R.M.S.-Monitor ist ein solcher im ROM enthaltener Code. Er liegt als Source-Code vor, wird bei den Systemerstellung durch den SOPC-Builder compiliert und in das ROM eingebunden. Dies ermöglicht es den Monitor zu verändern, um ihn an eigene Systeme anzupassen.

# Kapitel 5

# Das $\mu$C-Linux System

## 5.1 Überblick

Das $\mu$C-Linux System stellt ein komplettes Linux System dar, welches auf dem NIOS Embedded Processor aufbaut und auch für diesen optimiert wurde. Es liegt vollständig als Source-Code vor und biete sowohl die Möglichkeit den Kernel zu rekompilieren, als auch eigene Anwendungen zu schreiben und in das System zu integrieren. Wie alle Linux-Systeme bietet es Netzwerk-Funktionalität, hier über einen Ethernetanschluß. Dies mach es zu einem geeigneten Betriebssystem für den C.I.A., da es möglich ist, sich auf dem C.I.A. einzuloggen und über entsprechende Applikationen die Schnittstellen zu den übrigen C.I.A.-Designs anzusprechen. Das $\mu$C-Linux-System gliedert sich in 2 Teile, einen Hardware-Teil, welcher die benötigte Hardware zur Verfügung stellt, und einen Softwareteil, das eigentliche $\mu$C-Linux. Leider lag bis zum Ende dieser Niederschrift noch keine aktuelle Version, welche auf dem C.I.A.-Board hätte getestet werden können[1]. Entsprechende Tests zur Ansteuerung von eigenen Designs unter $\mu$C-Linux und über Ethernet konnten mittels des EXCALIBUR Development Kits trotzdem zufriedenstellend durchgeführt werden.

## 5.2 Hardware

Folgende Hardware wird zum Betrieb des $\mu$C-Systems benötigt:

- SDRAM/SRAM : da der NIOS über keinen Massenspeicher verfügt, wird das komplette System im RAM abgelegt und von dort ausgeführt.

- FLASH : enthält den kompletten System-Code. Beim Booten wird der Code vom FLASH in das RAM kopiert und dann die Kontrolle an das System übergeben.

- CS8900-Ethernet-Chip : stellt die Ethernet-Schnittstelle zur Verfügung. Der CS8900 erledigt alle Aufgaben des Physical-Layer und des Data-Link-Layer des TCP-Netzwerk-Modells und bietet dem NIOS eine einfache ISA-Schnittstelle zur Kommunikation mit dem Netz. Der CS8900 stellt für den C.I.A. nur eine temporäre Lösung dar und wird in zukünftigen Versionen durch ein entsprechendes Design im FPGA ersetzt werden. Lediglich die analoge Signalformung wird dann über eine eigenen Chip außerhalb des FPGAs stattfinden.

[1]Zwar existierte eine $\mu$C-Linux Version, diese lief aber nur auf einer alten Version des NIOS Prozessors. Für den C.I.A. wurde allerding eine neuerer Version des Prozessor angepaßt

## 5.3 Software

Die Software des $\mu$C-Linux Systems gliedert sich in zwei Bereiche, den Kernel und die ROM-DISK. Der Kernel ist die Grundlage des Systems, die ROM-DISK enthält das Filesystem und die Applikationen. Beide können unabhängig voneinander kompiliert und in des System integriert werden. Zur Kompilieren wird der GNUPro CROSS-Compiler verwendet, der die NIOS-System spezifischen Librarys einbindet und entsprechenden Code für den NIOS generiert. Dieser Code kann anschließend in einem FLASH gespeichert werden und beim Booten des Systems aus dem FLASH zur Ausführung in das RAM geladen werden. Diese Kopierroutine übernimmt ein modifizierter G.E.R.M.S.-Monitor im Boot-ROM des System. Er kann so konfiguriert werden, dass er beim Booten des System im FLASH nach Kernel und ROM-Disk sucht und diese ins RAM kopiert. Anschließend wird die Kontrolle dem $\mu$C-Linux-Kernel übergeben. Dieser initialisiert den Rest des Linux-Systems und legt eine RAM-Disk an. Abb. 5.2 zeigt den Initialisierungsprozess.

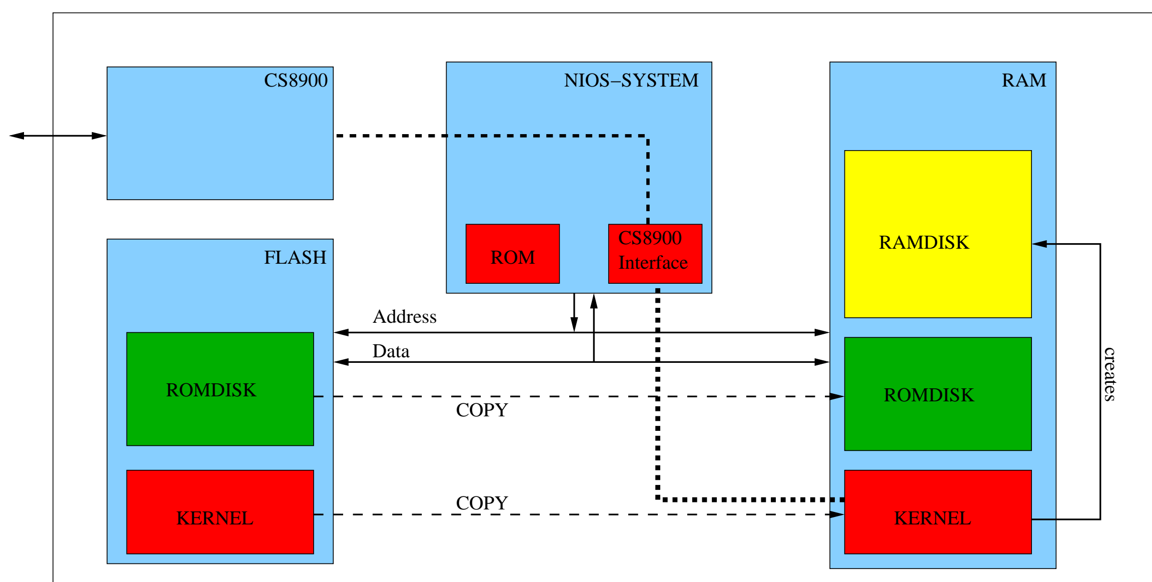


Abbildung 5.1: Initialisierung des $\mu$C-LINUX-SYSTEM

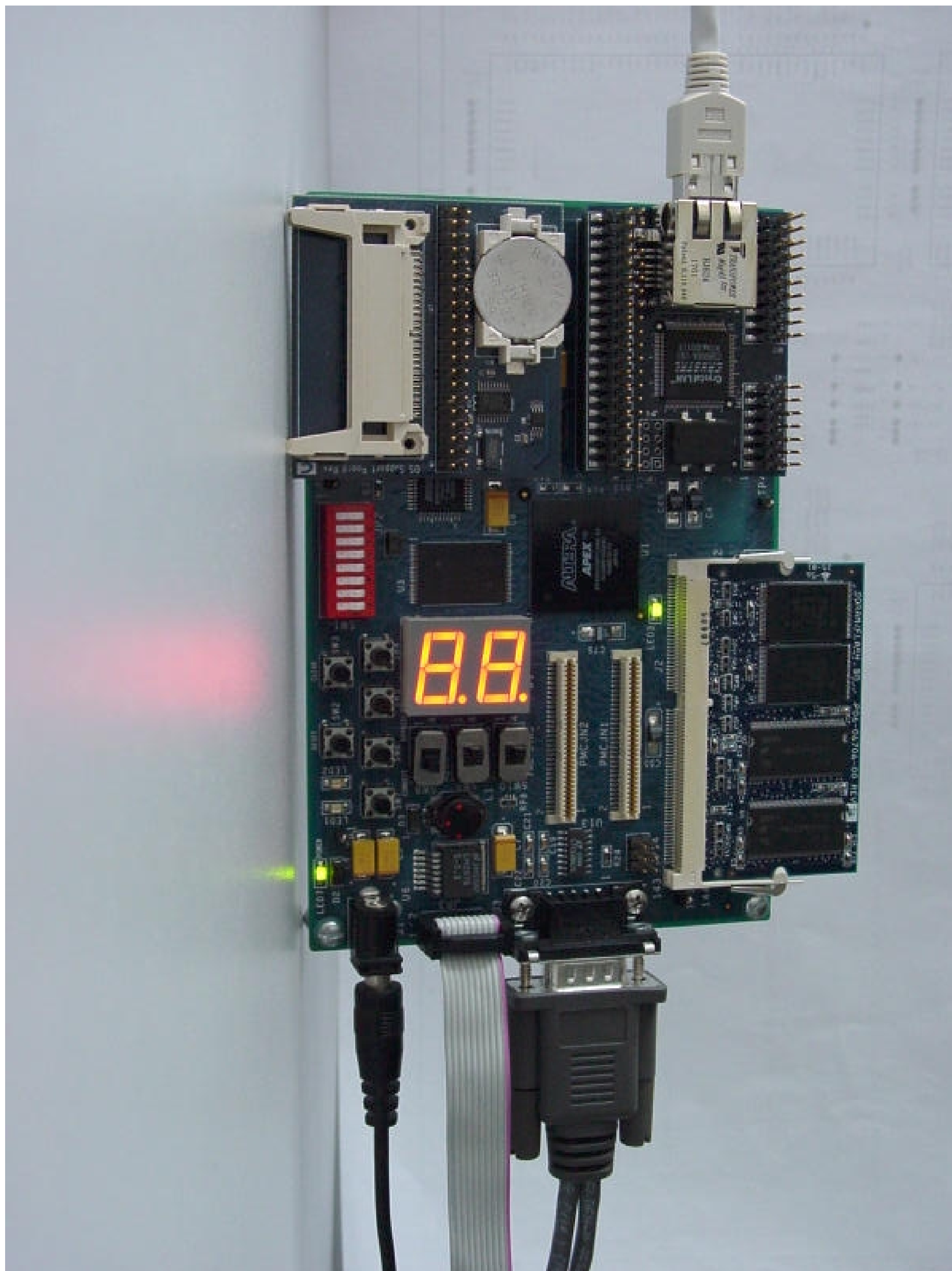

Abbildung 5.2: EXCALIBUR Development Board mit $\mu$C-Linux Kit, bestehend aus Ethernet-Board, OS-System-Board und S0DIMM-Memory-Modul

# Kapitel 6

# PCI Designs

## 6.1 Überblick

Eine der Hauptaufgaben, neben der Inbetriebnahme und der Portierung des NIOS-Systems auf das C.I.A.-Board, bestand in der Entwicklung eines PCI-Designs zum Scannen des Hostrechners. Hierzu wurde ein Design entwickelt, welches es dem C.I.A. ermöglicht als PCI-Master zu agieren und damit Zugang zum Memory-, Configuration- und I/O-Space des Hosts zu erlangen. Bevor auf das eigentliche Design eingegangen wird, erfolgt zunächst eine kurze Einführung in die Grundlagen des PCI-Busses. Das eigentliche Design gliedert sich in zwei Hauptteile. Auf der einen Seite ein PCI-Core von ALTERA zur Generierung der Signals auf dem Bus. Auf der anderen Seite die Schnittstelle zwischen NIOS-System und PCI-Core, die PCI-Control-Unit.

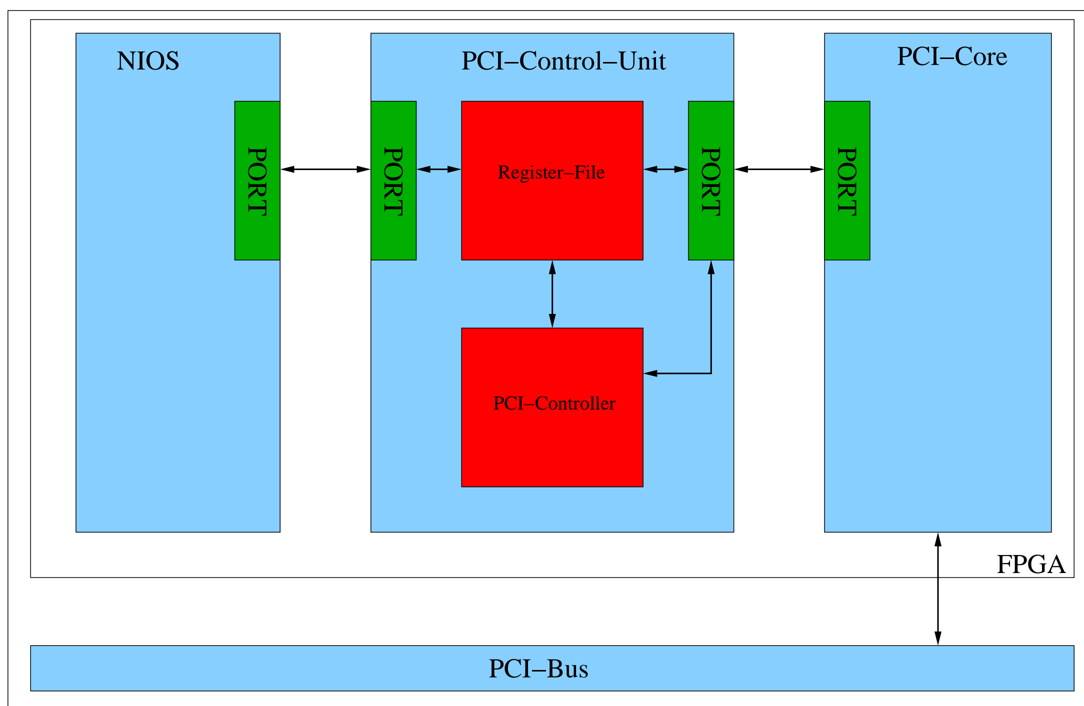


Abbildung 6.1: PCI-Design, bestehend aus PCI-Control-Unit und PCI-Core

## 6.2 Anforderungen

Das Design soll in der Lage sein die Konfiguration des Hosts zu ermitteln. Dafür würde theoretisch das Auslesen der Konfigurationsregister der PCI-Komponenten genügen, also Configuration-Reads. Da aber Memory-, Configuration- und I/O-Zyklen sehr ähnlich aufgebaut sind, wurden alle Zyklen implementiert. Dafür wurde auf die Unterstützung von Burst-Zyklen verzichtet, da für keine Aufgaben des C.I.A. ein hoher Datendurchsatz zwischen PCI-Bus und C.I.A. benötigt wird.

## 6.3 Der PCI-Bus

### 6.3.1 Aufbau

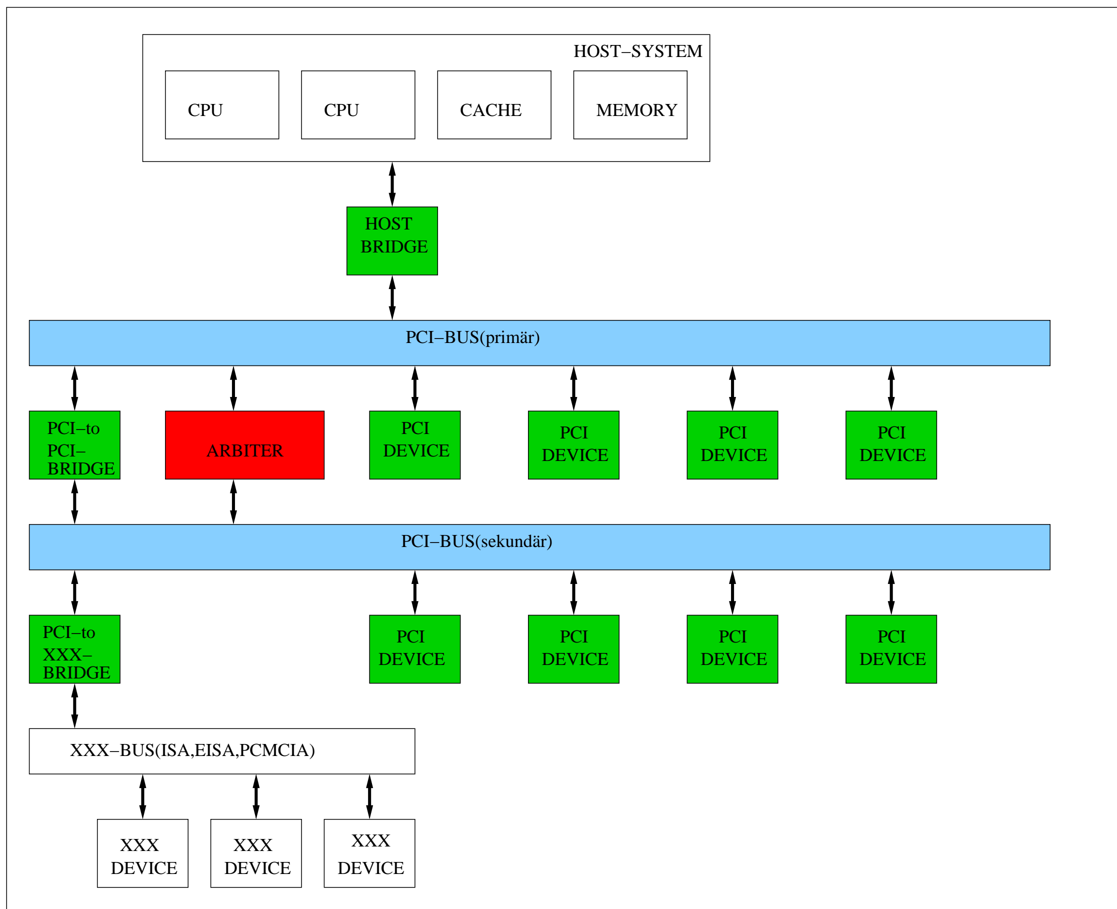


Abbildung 6.2: Hierarchischer Aufbau des PCI-Busses

Der 1992 von Intel ins Leben gerufene PCI-Bus ist ein synchrones, hierarchisches, prozessorunabhängiges Bussystem, welches das Host-System mit der übrigen Peripherie eines Rechners verbindet. Dafür wurde das Bussystem in 3 Teile untergliedert:

- PCI-Bus : das eigentliche Bussystem, bestehend aus Address-, Daten- und Steuerleitungen, sowie die damit verbunden elektrischen, mechanischen und prozeduralen Spezifikationen. Der Bus verfügt über 32 bzw. 64 Address/Datenleitungen, welche zeitlich gemultiplext werden. Dies gewährleistet eine höhere Bandbreite bei gleicher Leitungsanzahl. Der Mehraufwand an Bus-Zyklen für ein solches Zeitmultiplexing wird durch die

Fähigkeit zu *Bursts* (ein Address-Zyklus, mehrere Daten-Zyklen) wieder wettgemacht.

- PCI-Device : ein an den PCI-Bus angeschlossenes Device, welches über entsprechende Schnittstellen mit dem Bus kommuniziert. Hier unterscheidet man zwischen master- und targetfähigen Devices: Ein Master kann den Bus anfordern und, nach erfolgreicher Arbitration, Bus-Transfers initialisieren → Initiator. Ein Target dagegen ist nicht in der Lage, den Bus anzufordern, sondern nimmt Anforderungen durch einen Master entgegen und führt diese aus. Es werden 32/64-Bit, sowie 5V/3.3V Devices unterstützt.

- PCI-Arbiter : dieser verwaltet die Zugriffe auf den Bus und stellt damit sicher, daß zu jeder Zeit immer nur ein Master Zugriff auf den Bus erhält, da es sonst zu Bus-Konflikten kommt. Die Arbitration beim PCI-Bus ist zentral. Will ein Master den Bus für seine Transaktionen, so sendet er einen Request an den Arbiter und dieser gewährt ihm den Bus (Grant) oder nicht. Falls nicht, so kann zu einem späteren Zeitpunkt ein erneuter Request erfolgen. Die Arbitration erfolgt im Hintergrund (Overlapped Arbitration) und ein Watchdog-Timer entzieht nach einer gewissen Zeitspanne dem Master den Bus. Somit werden Deadlocks vermieden, falls ein Master aus irgendwelchen Gründen den Bus nicht mehr freigibt.

Somit ist ein hierarchischer, prozessor-unabhängiger Bus möglich. Über sogenannte Bridges können Hostsystem und Bus, oder Bus und Bus miteinander verbunden werden. Aus der Sicht des PCI-Busses handelt es sich dabei um ein PCI-Device.

Tabelle 6.1 zeigt die Signal-Leitungen, über welche die Initialisierung der Devices, die Arbitration der Master und die Datenübertragung zwischen Master und Target erfolgt:

| Signal | Name |
|---|---|
| clk | Clock |
| rst# | Reset |
| gnt# | Grant |
| req# | Request |
| ad[31..0] | Address/Data |
| ad[63..32] | Address/Data |
| cben[3..0] | Command/Byteenable |
| cben[7..4] | Command/Byteenable |
| par | Parity |
| par64 | Parity |
| idsel | Initialization device select |
| frame# | Frame |
| req64# | Request 64-Bit |
| irdy# | Initiator ready |
| devsel# | Device select |
| ack64# | Acknowledge 64-Bit |
| trdy# | Target ready |
| stop# | Stop |

Tabelle 6.1: Signale des PCI-Bus

Eine ausführliche Beschreibung der einzelnen Signale und deren Funktionaliät befindet sich in [22] und [23].

### 6.3.2 Initialisierung der PCI-Devices: Configuration-Cyles

Jedes Device am Bus wird in den Address-Raum des Busses gemappt und kann dann über seinen Addressbereich angesprochen werden (Logische Adressierung). Aus Gründen der Flexibilität und zur Vermeidung von Addresskonflikten besitzen die Devices keine festen Adressen, sondern bekommen bei ihrer Initialisierung einen Addressbereich zugeteilt. Dies geschieht in den Configuration-Zyklen. Da ein noch nicht eingebundenes Device nicht über seine logische Adresse angesprochen werden kann, wird es über ein eigenes Signal, IDSEL, ausgewählt (Geographische Adressierung). Hierzu verfügt jeder PCI-Slot über ein eigenes IDSEL Signal. Das so ausgewählte Device wird anschließend über seinen Configuration-Space initialisiert.

#### Configuration Space

Jedes Device verfügt über eine 256 Byte großen Configuration Space, wobei 64 Byte auf einen vordefinierten Header fallen und die restlichen 192 Byte deviceabhängig sind und hier nicht weiter betrachtet werden. Abb. 6.3 zeigt die Struktur des Headers. Dieser enthält alle wichtigen Informationen über das Device und für dessen Initialisierung. Durch Auslesen dieser 64 Byte durch den C.I.A. ist es möglich, sowohl die einzelnen Devices zu identifizieren, als auch ihre aktuelle Konfiguration zu bestimmen. Zu den einzelnen Feldern des Header finden sich in [22] und [23] ausführlichere Informationen.

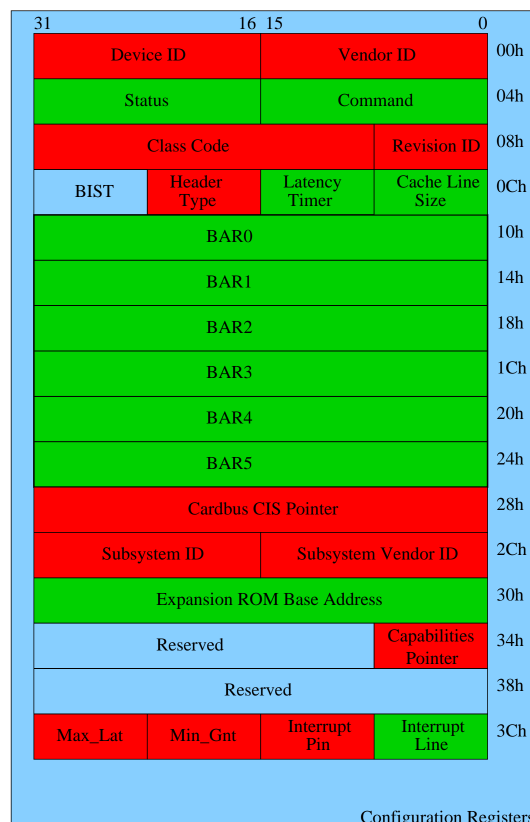


Abbildung 6.3: 64-Bit Configuration Space

#### BARs - Base Address Registers

Die BARs sind ein Teil des 64 Byte Headers und für die logische Adressierung des Device zuständig. Sie enthalten vor der Initialisierung den Typ (Memory oder I/O) und die Größe

des zu reservierenden Addressraum. Während der Initialisierung liest das System die BARs aus, mapped das Device in den entsprechenden Addressraum und schreibt seine Basisadresse wieder zurück in die BARs. Von jetzt an kann das Device bis zum nächsten Reboot über seine logische Adresse angesprochen werden.

### 6.3.3 Datenübertragung

Jede Datenübertragung beginnt mit einem REQUEST-Signal eines Masters an den Arbiter. Ist der Bus frei, so erhält der Master einen PCI-Zyklus später die Bestätigung über das GRANT-Signal. Falls nicht, muss gewartet werden, bis der Bus frei ist. Beide Signale, REQUEST und GRANT, sind Point-to-Point-Signale und verlaufen vom Arbiter zu jedem PCI-Slot. Ist der Master im Besitz des Busses, so kann er folgende Transaktionen ausführen:

- **Memory-Zyklen** : 32/64-Bit Single/Burst Read/Write
- **I/O-Zyklen** : 32-Bit Single Read/Write
- **Configuration-Zyklen** : 32-Bit Single Read/Write

Zur Illustration einer Datenübertragung wird abschließend kurz auf den Ablauf eines Read-Zyklus eingegangen : der Master beantragt den Bus und wartet auf die Bestätigung durch den Arbiter. Erhält er diesem, so legt er die Adresse und das Kommando auf den Bus. Danach erfolgt ein *Turnaround*-Zyklus, d.h. der Master schaltet seinen Adress- und Datenausgang hochohmig (tristate) um dem Target das Senden der Daten auf dem gemeinsamen Adress-/Datenbus zu ermöglichen. Das Target überprüft anhand seiner BARs, ob die Adresse auf dem Bus in seinen Adress-Bereich fällt. Falls ja, nimmt es die Transaktion an und legt die Daten auf den Bus. Nach dem Empfang der Daten beendet der Master die Transaktion und gibt den Bus wirder frei. Abb. 6.4 skizziert den zeitlichen Ablauf einer solchen Transaktion.

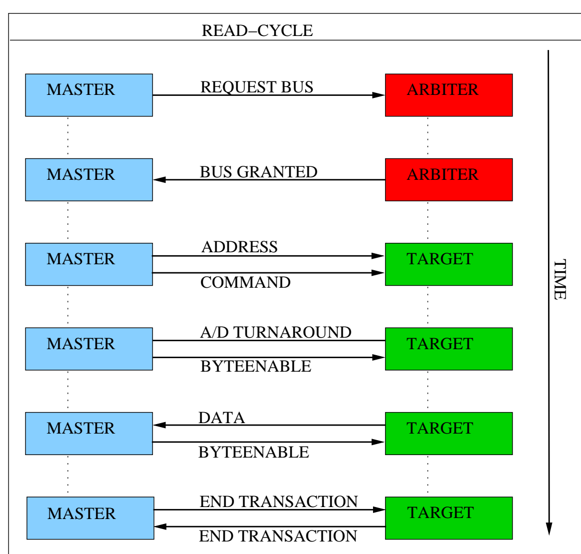


Abbildung 6.4: Ablauf eines READ-CYCLES

## 6.4 PCI-Megacore

Für die Kommunikation mit dem PCI-Bus wurde ein PCI-Core der Firma ALTERA verwendet, welcher den Benutzer von der Pflicht entbindet, alle Bussignale selbst zu steuern

und ihm stattdessen ein Interface zur Verfügung stellt. Der PCI-Core liegt als verschlüsselter VHDL-Code vor, wird über ein entsprechendes Konfigurationstool, den MEGAWIZARD konfiguriert und man erhält ein VHDL- oder ein graphisches File (.bsf) für QUARTUS, welches anschließend in ein bestehendes Design eingebunden werden kann. Insgesamt existieren 4 verschieden Varianten des PCI-Cores:

- 32-Bit Target-Core : 32-Bit, nur Target-fähig
- 32-Bit Master-Core : 32-Bit, Master/Target-fähig
- 64-Bit Target-Core : 64-Bit, nur Target-fähig
- 64-Bit Master-Core : 64-Bit, Master/Target-fähig

Für jede Variante kann man zwischen 33MHz und 66MHz PCI-Takt wählen. Alle 64-Bit Varianten sind abwärtskompatibel, d.h. sie verfügen über alle Signalleitungen wie die 32-Bit Varianten. Für die 64-Bit Funktionalität wurden zusätzlichen Steuersignale eingeführt, bzw. die Busbreite bei AD von 32 auf 64 und bei CBEN von 4 auf 8 erhöht. In dieser Diplomarbeit wurde sich für einen 64-Bit Master-Core entschieden, welcher im 32-Bit-Mode betrieben wird. Der Grund ist folgender. Der 64-Bit/66MHz-Core stellt die obere Grenze des Resourcenverbrauchs dar, da er der komplexeste der PCI-Cores ist. Somit ist eine Abschätzung der maximalen Ressourcen für die Core möglich. Da das C.I.A.-Board nicht allein für den C.I.A. entwickelt wurde, sondern noch in verschieden anderen Projekten verwendet wird (RORC, RCU), welche den 64-Bit/66MHz-Core verwenden werden, konnte so ein erster Funktionstest und eine Ressourcen-Abschätzung durchgeführt werden. Abb. 6.5 zeigt einen 64-Bit Master-fähigen PCI-Core und die Signalleitungen zum PCI-BUS (links) und zum User (rechts).

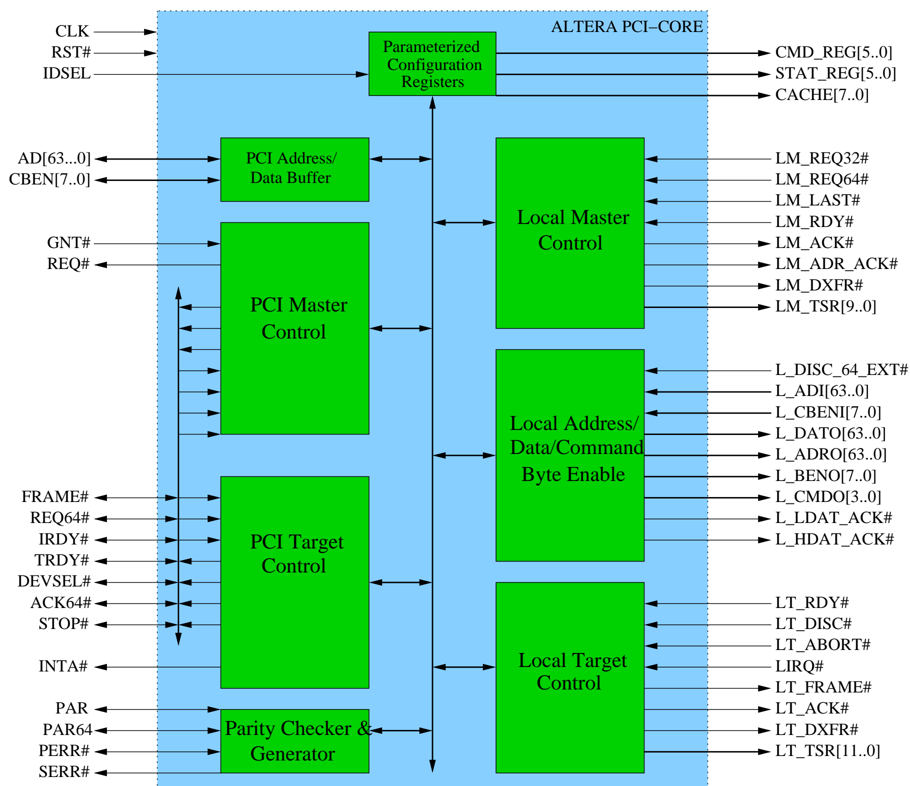


Abbildung 6.5: 64-Bit-PCI Master/Target-Megacore

Die Signalleitungen zum PCI-BUS sollten zum Einhalten der strengen Timing-Requirements der PCI-SPECS direkt über entsprechende Pins nach außen geroutet werden und keine zusätzliche Logik dazwischen plaziert werden. Weiterhin macht das Verwenden eines schnellen PCI-Core (66MHz) die Verwendung spezieller Constrains-Files nötig. Diese Constrains-Files (.csf, .esf) enthalten Zusatzinformationen für das PLACE & ROUTE um den PCI-Core möglichst nicht über den FPGA zu "verstreuen" und so die Signal-Laufzeiten im Core zu minimieren. Die Signalleitungen zum User, die Local-Side Signale, können entsprechend ihrer Funktion in mehrere Gruppen unterteilt werden:

- Status-Signale : cmd_reg, stat_reg, cache. Geben Auskunft über den Zustand des PCI-Cores.
- Master-Control-Signale : lm_xxx. Dienen zur Steuerung von Master-Transaktionen.
- Address/Daten/Commando-Signale : l_xxx. Dienen der Address- und Datenübertragung, selektieren die gültigen Bytes (Byteenable) im Datum und legen die Art der Transaktion fest (MEM, I/O, CFG, Read, Write).
- Target-Control-Signale : lt_xxx. Dienen der Steuerung von Target-Transaktionen.

Der PCI-Core dient nur als Vermittler zur Steuerung des Datenflusses zwischen PCI-Bus und Local-Side und garantiert das Einhalten der PCI-SPECS was das Timing und den Ablauf der Steuersignale angeht. Für den exakten Ablauf einer Transaktion und das Bereitstellen und Puffern von Adressen und Daten ist die Local-Side und damit der User zuständig. Daher wurde ein entsprechendes Interface, bestehend aus einem REGISTER-File und einem PCI-CONTROLLER, entwickelt, welche es dem NIOS, oder anderen Systemen, auf einfache Weise ermöglicht, auf den PCI-Bus zu schreiben oder von ihm zu lesen.

## 6.5 Register-File und PCI-Controller

Das Register-File und der PCI-Controller stellen die Schnittstelle zwischen NIOS und PCI-Core dar und ermöglichen dem NIOS einen einfachen Zugriff auf den PCI-Bus. Der Schwerpunkt bei der Entwicklung des Designs lag auf der Unabhängigkeit des NIOS vom PCI-Core und nicht in der Performance. Auf keinen Fall darf der PCI-Core in der Lage sein, das NIOS-System zu beeinflussen, z.B. zu stallen. Dies würde das Prinzip der Autonomie des C.I.A. zunichte machen. Stattdessen wurde der Zugriff über 3 Input/Output-Register des NIOS, den PIOs, realisiert. Abb. 6.6 zeigt das komplette Design und die Signalverläufe zwischen den einzelnen Komponenten.

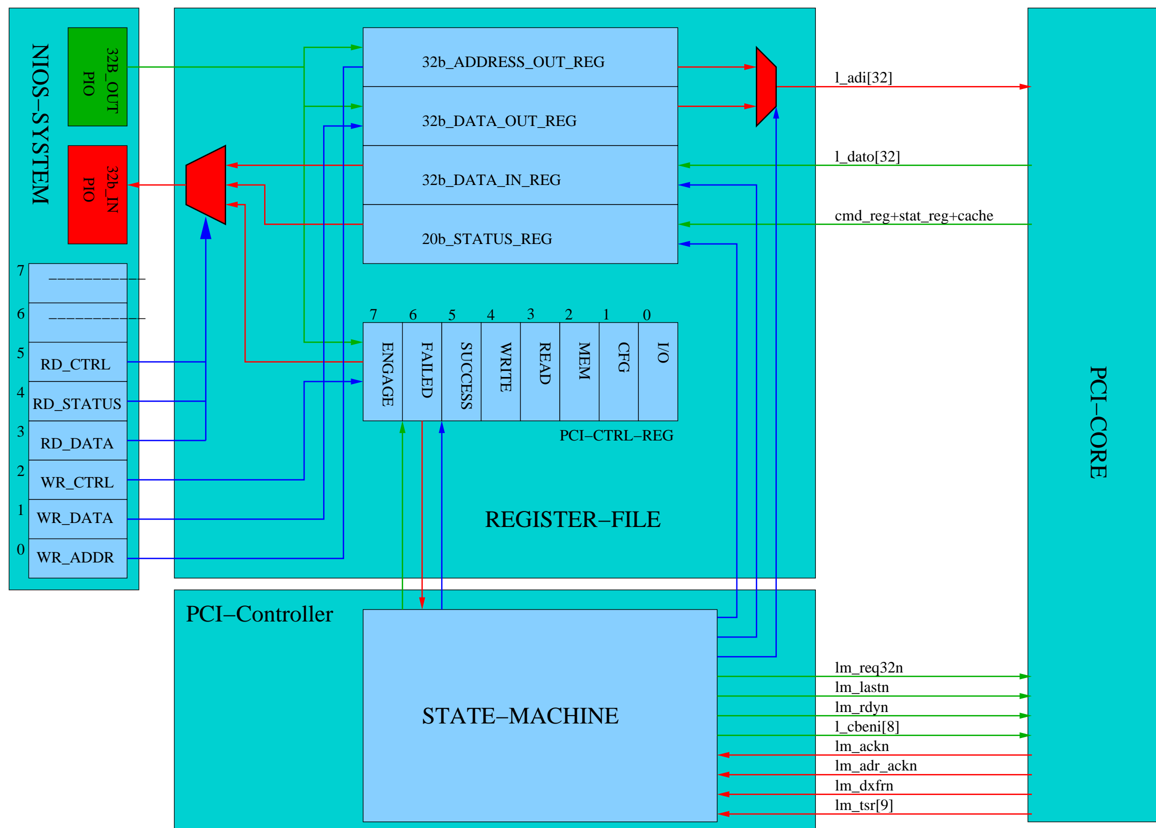


Abbildung 6.6: Datenpfade des PCI-Designs

### 6.5.1 Schnittstellen zum NIOS : PIOs

Die Schnittstellen des NIOS zum Register-File bestehen aus registrierten PIOs. Diese wurden vom SOPC-Builder generiert und in den Addressraum des NIOS gemappt. Da der NIOS über keine Memory Management Unit (MMU) verfügt, können die Schnittstellen direkt über ihrer Adresse angesprochen und gelesen bzw. beschrieben werden. Zum Beschreiben und Auslesen des Register-Files wurden entsprechende Funktionen in C geschrieben. Damit war es möglich Testsoftware zu schreiben und PCI-Transaktionen zu testen. Diese Schnittstellen sind die einzige Zugriffsmöglichkeit des NIOS auf das PCI-Design und gewährleisten die Unabhänigigkeit des NIOS. Eine weiter Überlegung war, das PCI-Design so zu konzipieren, dass es im Falle eines Wechsels vom NIOS auf einen anderen Prozessor, z.B. einen hardcodierten ARM-Prozessors, einfach übernommen werden kann. Folgende PIOs übernehmen die Kommunikation mit der PCI-Control-Unit:

- 32-Bit Input PIO : Empfängt Daten aus dem Register-File
- 32-Bit Output PIO : Enthält Daten zum Schreiben in das Register File.
- 8-Bit Control PIO : Steuert das Lesen oder das Schreiben in das Register-File.

Über diese 3 PIOs kann in das Register-File geschrieben und gelesen werden und indirekt der PCI-Controller gesteuert werden. Tabelle 6.2 enthält die Beschreibung des Control-PIO mit welchem die Datenübertragung zum Register-File gesteuert wird.

| BIT | NAME | FUNKTION |
|---|---|---|
| 0 | Write Address | Schreibt den Inhalt des OUTPUT-PIO in das ADDRESS-OUT-Reg |
| 1 | Write Data | Schreibt den Inhalt des OUTPUT-PIO in das DATA-OUT-Reg |
| 2 | Write Control | Schreibt den Inhalt des OUTPUT-PIO in das PCI-CTRL-Reg |
| 3 | Read Data | Liest den Inhalt des DATA-IN-Reg in das INPUT-PIO |
| 4 | Read Status | Liest den Inhalt das STATUS-Reg in das INPUT-PIO |
| 5 | Read Control | Liest den Inhalt des PCI-CTRL-Reg in das INPUT-PIO |
| 6 | Unused | — |
| 7 | Unused | — |

Tabelle 6.2: NIOS-CONTROL-PIO

### 6.5.2 Register-File

In der gegenwärtigen Version besteht das Register-File aus 5 Registern :

- 32-Bit ADDRESS-Reg : Enthält die Adresse zu der geschrieben oder von der gelesen werden soll.
- 32-Bit DATA-IN-Reg : Speichert bei READ-Zyklen das ankommende Datum.
- 32-Bit DATA-OUT-Reg : Enthält bei WRITE-Zyklen das zu schreibende Datum.
- 20-Bit Status-Reg : Enthält die Status-Signale cmd-reg, stat-reg und cache des PCI-Cores. Diese geben Auskunft über den gegenwärtigen Status des PCI-Core.
- 8-Bit PCI-CONTROL-Reg : Enthält Bitmuster zur Steuerung das PCI-Controllers.

Das NIOS-System kann in die Register ADDRESS, DATA-OUT und PCI-CTRL schreiben und von den Registern DATA-IN, STATUS und PCI-CTRL lesen. Dies geschieht durch die entsprechenden Bits des 8b-Control-PIO. Das Register-File dient zur Initialisierung einer PCI-Übertragung. Die entsprechenden Daten werden in das Register-File geschrieben und dann über das PCI-CONTROL-Reg die Kontrolle an den PCI-Controller übergeben. Dieser führt die Transaktion durch, schreibt bei Read-Zyklen das Datum in des DATA-IN-Reg und liefert eine SUCCESS/FAILURE-Meldung über das PCI-CTRL-Reg zurück.

#### Das PCI-CTRL-Reg

Das 8-Bit breite PCI-CTRL-Reg dient zur Steuerung des PCI-Controllers, welcher die Transaktionen des PCI-Cores kontrolliert und regelt. Tabelle 6.3 erklärt die Bedeutung der einzelnen Bits.

| BIT | NAME | FUNKTION |
|---|---|---|
| 0 | I/O | Die nächste Transaktion wird eine I/O-Transaktion. |
| 1 | CFG | Die nächste Transaktion wird eine CFG-Transaktion. |
| 2 | MEM | Die nächste Transaktion wird eine MEM-Transaktion. |
| 3 | READ | Die nächste Transaktion wird eine Read-Transaktion. Die Art der Transaktion wird durch die Bits 0,1,2 festgelegt. |
| 4 | WRITE | Die nächste Transaktion wird eine Write-Transaktion. Die Art der Transaktion wird durch die Bits 0,1,2 festgelegt. |
| 5 | SUCCESS | Wird vom PCI-Controller gesetzt, falls eine Transaktion erfolgreich abgeschlossen werden konnte. |
| 6 | FAILURE | Wird vom PCI-Controller gesetzt, falls eine Transaktion nicht erfolgreich abgeschlossen werden konnte. |
| 7 | ENGAGE | Wird vom NIOS gesetzt und aktiviert die Statemachines des PCI-Controllers, aktiviert also eine Transaktion. Wird vom PCI-Controller nach einer Transaktion zurückgesetzt. |

Tabelle 6.3: PCI-CTRL-Register

#### NIOS liest aus dem Register-File

Das Auslesen des Register-Files geschieht durch das Setzten der entsprechenden Bits des 8-Bit Control-PIO. Diese schalten im Register-File einen 32-Bit 3fach-Multiplexer, welcher wahlweise das DATA-IN, das STATUS-REGISTER oder das PCI-CONTROL-Reg auf den Ausgang zum NIOS schaltet. Da alle PIOs des NIOS aus Gründen der Synchronität registert sind, stehen die Daten nach Setzten der Auslese-Bits einen Takt später zur Verfügung.

#### NIOS schreibt in das Register-File

Das Schreiben in das Register-File geschieht ebenfalls durch Setzen der entsprechenden Bits des 8-Bit Control-PIO. Das im 32b OUT-PIO stehende Datum wird dann mit der nächsten Taktflanke in des Register-File geschrieben. Hierzu führt der Eingang vom NIOS zu jedem der Register ADDRESS-OUT, DATA-OUT und PCI-CTRL. Jedes dieser Register verfügt über ein Input-Enable-Signal, welches von den Bits des Control-PIO getrieben wird. An allen Register liegt also das selbe Datum an, es wird nur selektiert in welches das Datum geschrieben wird.

#### PCI-Controller liest aus dem Register-File

Der PCI-Controller liest genaugenommen nur ein Register aus, das PCI-CTRL-Reg, steuert jedoch zur Datenübertragung zum PCI-Core die Lese-Zugriffe des ADDRESS-OUT-Reg und des DATA-OUT-Reg. Beide dienen der Datenübertragung zum PCI-Core. Beide werden über einen 32-Bit 2fach Multiplexer zum Address/Daten-Eingang (l_adi[32]) des PCI-Cores geschaltet. Die Steuerung des Mutliplexers geschieht über die Statemachine des PCI-Controllers. Je nachdem in welcher Phase sich der PCI-Core befindet, Address- oder Daten-Phase, wird das entsprechende Register gelesen. Das PCI-CTRL-Reg wird permanent gelesen, es enthält alle Steuerbits für den PCI-Controller. Der Ausgang des PCI-CTRL-Reg ist direkt mit dem Steuereingang des PCI-Controllers verbunden.

#### PCI-Controller schreibt in das Register-File

Ähnlich wie mit dem Lesen verhält es sich mit dem Schreiben in das Register-File. Der PCI-Controller steuert die Datenübertragung zwischen PCI-Core und dem DATA-IN-Reg und dem STATUS-Reg. Die Dateneingänge beider Register sind direkt mit dem PCI-Core verbunden, lediglich das Input-Enable wird über die Statemachine des PCI-Controllers gesteuert. Direkt geschrieben werden kann nur in einen Teil des PCI-CTRL-Reg, nämlich in die oberen 3 Bits (ENGAGE, SUCCESS, FAILURE). Diese dienen zur Rückmeldung zum NIOS nach einer Transaktion.

### 6.5.3 PCI-Controller

Der PCI-Controller initialisiert über den PCI-CORE entsprechende Transaktionen auf dem PCI-Bus und regelt die Datenübertragung zwischen dem Register-File und PCI-CORE. Dazu wurden zwei StateMachines entwickelt, eine für READ- und eine für WRITE-Zyklen. Diese werden durch das PCI-CTRL-Reg kontrolliert und aktiviert, steuern die Kontrollsignale des PCI-Cores und die Datenübertragung zwischen PCI-CORE und Register-File.

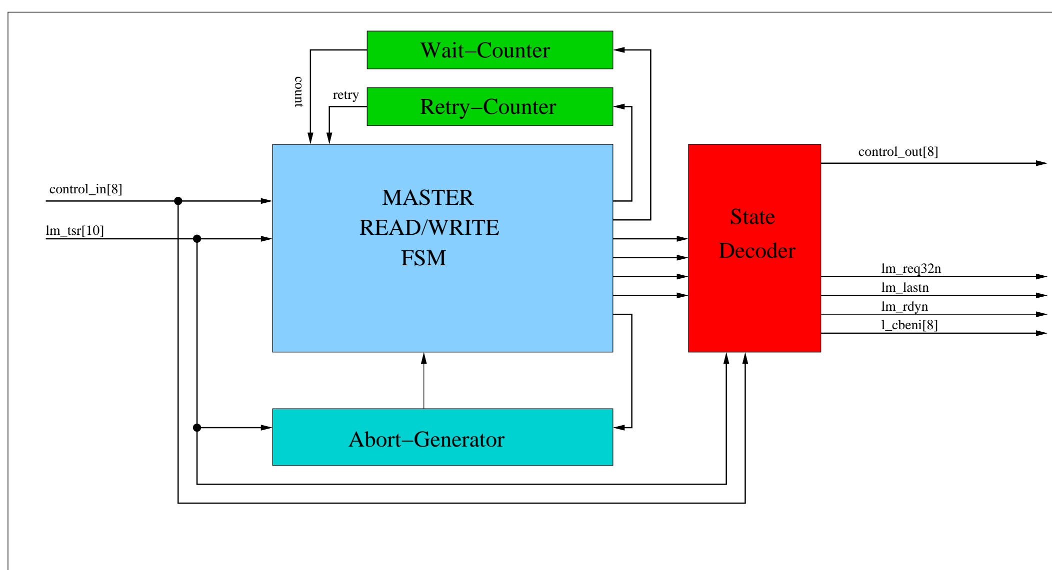


Abbildung 6.7: PCI-Controller

Abb 6.7 zeigt die innerer Struktur des PCI-Controllers und die Eingangs- und Ausgangssignale. Vom Register-File führt das 8-Bit breite *control_in* Signal in den PCI-Controller, vom PCI-Core das 10-Bit breite *lm_tsr* Signal. Aus dem PCI-Controller zum Register-File führt das 8-Bit breite *control_out* Signal, zum PCI-Core die Signale *lm_req32, lm_lastn, lm_rdyn, l_cbeni*. Die *control_in* Signale kommen vom PCI-CTRL-Reg und steuert die Transaktionen des PCI-Controllers, die *lm_tsr* Signale geben den Status des PCI-Cores bei der Datenübertragung an. Beide steuern sowohl die FSMs als auch den State-Decoder, die *lm_tsr* Signale zusätzlich noch den Abort-Generator. Die Ausgangssignale werden vom State-Decoder aus den Zustandsvektoren der FSMs und den Eingangssignalen generiert, es handelt sich also um einen Mealy-Automaten. Die *control_out* Signale steuern das Register-File, die Signale *lm_req32, lm_lastn, lm_rdyn* und *l_cbeni* steuern die Datenübertragung zwischen PCI-Core und PCI-Bus.
Tabelle 6.5 gibt die *control-out* Signale zur Steuerung des Register-Files wieder, Tabelle 6.4 die Statussignale des PCI-Cores. Die Bedeutung und Verwendung der einzelnen Signale wird im Kontext erklärt.

| BIT | NAME | FUNKTION |
|---|---|---|
| 0 | request | PCI-Core hat PCI-Bus beantragt( *request*) und wartet auf Bestätigung durch den Arbiter(*grant*) |
| 1 | grant | PCI-Core hat Bestätigung vom Arbiter erhalten(*grant*) und verfügt nun über den Bus |
| 2 | adr_phase | Address-Phase. Der PCI-Core legt als Master die Adresse(*l_adi*) und das Commando(*l_cbeni*) auf den Bus |
| 3 | dat_xfr | Daten-Transfer. Beginnt nach der Address-Phase und dauert bis zum Ende der Transaktion |
| 4 | lat_exp | Latency timer expired. Der Latency Timer des PCI-Cores ist abgelaufen und dieser muss den Bus freigeben |
| 5 | retry | Target hat Transaktion mit retry beendet. Nach PCI-SPECs muss der Master die Transaktion zu einem späteren Zeitpunkt wiederholen |
| 6 | disc_wod | Disconnect without data. Target hat Transaktion mit disconnect without data beendet |
| 7 | disc_wd | Disconnect with data. Target hat Transaktion mit disconnect with data beendet |
| 8 | dat_phase | Data-Phase. Zeigt eine erfolgreiche Datenübertragung im vorangegangen Takt an. |
| 9 | trans64 | 64–Bit Transaction. Zeigt eine 64–Bit Transaktion an |

Tabelle 6.4: *lm_tsr* - Local Master Transaction Status Register

Die eigentliche Steuereinheit gliedert sich in zwei Teile: die FSMs zusammen mit den Wait- und Retry-Countern sowie dem Abort-Generator, zur Generierung der Zustände und dem nachgeschalteten State-Decoder zur Erzeugung des Ausgangssignale. Im folgenden werden die einzelnen Komponenten ausführlicher beschrieben, den beiden FSMs ist jeweils ein eigener Unterabschnitt gewidmet.

- **Finite State Machines - FSMs**. Zur Steuerung des PCI-Cores und der Datenübertragung wurden zwei FSMs entwickelt, eine MasterRead-FSM und eine MasterWrite-FSM.

- **Wait-Counter**. Der Wait-Counter dient als Zähler für Warte-Zyklen. Er wird durch die FSM aktiviert und resetet und zählt im Takt der PCI-Clock. Sollte eine FSM in einem bestimmten Zustand zu lange verweilen, so wird der Abort-Generator aktiviert, welcher die Transaktion abbricht.

- **Retry-Counter**. Der Retry-Counter zählt die Anzahl der Retry-Versuche nach einem Target-Retry. Falls ein Target eine Transaktion des Masters mit einem Target-Retry abbricht, so verlangt die PCI-SPECs, dass der Master es zu einem späteren Zeitpunkt nochmal versucht. Der Retry-Counter zählt die Anzahl dieser Versuche. Wird die Anzahl zu groß, kann der Abort-Generator die Transaktion abbrechen. So wird ein Deadlock vermieden.

- **Abort-Generator**. Der Abort-Generator überwacht die Transaktionen der FSM und den Zustand des PCI-Cores und kann gegebenfalls eine Transaktion abbrechen und die FSM in den IDLE-Zustand zurücksetzten.

| BIT | NAME | FUNKTION |
|---|---|---|
| 0 | Write DATA-IN | Schreibt mit der nächsten steigenden Taktflanke die Daten vom PCI-Core in das DATA-IN-Reg des Register-Files. |
| 1 | Read ADDRESS | Legt die Address aus dem ADDRESS-OUT-Reg des Register-Files an den Address/Daten-Eingang des PCI-Core. |
| 2 | Read DATA-OUT | Legt das Datum aus dem DATA-OUT-Reg des Register-Files an den Address/Daten-Eingang des PCI-Core. |
| 3 | — | Unused |
| 4 | Clear DATA-IN | Löscht den Inhalt des DATA-IN-Reg des Register-Files. |
| 5 | Set SUCCESS | Setzt das SUCCESS-Bit und resetet das ENGAGE-Bit des PCI-CTRL-Reg. |
| 6 | Set FAILURE | Setzt das FAILURE-Bit und resetet das ENGAGE-Bit des PCI-CTRL-Reg. |
| 7 | — | Unused |

Tabelle 6.5: *control_out*-Signale des PCI-Controllers zur Steuerung des Register-File

- **State Decoder**. Der State-Decoder dekodiert die Zustände der FSM unter Berücksichtigung der Eingangssignale von PCI-CTRL-Reg und PCI-Core.

**MasterRead-STATEMACHINE**

Abb. 6.8 zeigt das Zustandsdiagramm der FSM für Read-Transaktionen. Der genaue Ablauf der Transaktion ist durch die verschiedenen Zustände festgelegt, welche im folgenden näher beschrieben werden. Für ein komplettes Verständniss der FSM sei auf das VHDL-File *pci-controller.vhd* verwiesen, welches der CD beliegt.

- **MR-IDLE** : Ausgangszustand. Durch setzten der untersten 5 Bit im PCI-CTRL-Reg wird eine Transaktion spezifiziert und anschließend durch setzten des ENGAGE-Bit( Bit 7) aktiviert. Sobald die FSM Bit 3(READ) und Bit 7(ENGAGE) sieht, wird die Transaktion gestartet und im nächsten Takt zu MR-REQ übergegangen.

- **MR-REQ** : Bus-request. Der PCI-Core wird durch Setzten des *lm_req32* Signals veranlasst den Bus zu beantragen. Dieser Zustand ist ein reiner Übergangszustand ohne äußere Einflüsse und geht im nächsten Takt zu MR-RDY über.

- **MR-RDY** : Ready for Transaction. Es wird auf die Bestätigung des Bus-Requests gewartet, welche der PCI-Core durch *lm_tsr(1)* anzeigt. Die PCI-SPEC verlangen, dass noch im gleichen Takt, in dem der PCI-Core das *grant* durch *lm_tsr(1)* anzeigt, die Zieladdresse und das Commando an den PCI-Core übergeben wird. Daher werden sowohl Adresse als auch Commando während der ganzen Dauer des Zustandes MR-RDY an die entsprechenden Eingänge des PCI-Cores angelegt. Sobald die FSM das *grant*-Signal *lm_tsr(1)* vom Core sieht, wird im nächsten Takt zu MR-DATA übergegangen. Sollte der Abort-Generator die Transaktion an dieser Stelle abbrechen, so wird mit der nächsten Taktflanke in den Zustand MR-IDLE gewechselt.

- **MR-DATA** : Data-Phase. Es wird auf die Datenübertragung vom Target gewartet. Sobald der PCI-Core Daten vom Target erhält, signalisiert der dies durch Setzten von

*lm_tsr(8)* an die FSM. Diese veranlaßt über *control_out(0)* das Schreiben des Datums in das Register-File und geht mit der nächsten Taktflanke zu MR-SUCCESS über. Sollte das Target die Transaktion mit einem *retry* abbrechen, so signalisiert der PCI-Core dies durch *lm_tsr(5)*. Das die PCI-SPEC eine Wiederholung der Transaktion fordert, geht die FSM zum nächsten Takt in den Zustand MR-BUSWAIT über. Dort wird eine bestimmte Zeit gewartet und die Transaktion wiederholt. Für den Fall, dass ein Target gar nicht antwortet, wurde ein Timer eingebaut, der Wait-Counter. Dieser wird bei Eintreten von MR-DATA gestartet und zählt die PCI-Takte. Ist ein bestimmter Zeitpunkt erreicht und ein Target hat nicht geantwortet, wird die Transaktion mit einem Fehler beendet. Die FSM geht dann in den Zustand MR-FAILURE über. Aktuell liegt die Abbruchbedingung bei 32 PCI-Takten, d.h. *count(5)=1*, dies kann jedoch einfach im Design geändert werden. Ebenfalls in zu MR-FAILURE übergegangen wird bei einem Abbruch durch den Abort-Generator.

- **MR-BUSWAIT** : Warte-Zyklen. Bei Eintritt in den Zustand MR-BUSWAIT wird sowohl der Wait-Counter, als auch der Retry-Counter gestartet. Erreicht der Wait-Counter einen bestimmten Wert, aktuell 8 PCI-Zyklen, so wird die PCI-Transaktion mit einem Bus-Request wiederholt: MR-REQ. Der Retry-Counter zählt die Anzahl der wiederholten PCI-Transaktionen nach einem Target-Retry. Wird ein bestimmter Wert erreicht, aktuell 8 Versuche, so wird die Transaktion mit einem Fehler abgebrochen : MR-FAILURE.

- **MR-SUCCESS** : Dieser Zustand ist wieder ein Übergangszustand ohne äußere Einflüße. Er dient dazu, eine Rückmeldung an das PCI-CTRL-Reg zu machen und dort das SUCCESS-Flag zu setzten und das ENGAGE-Flag zu reseten, da ansonst die Transaktion wieder durchlaufen würde. MR-SUCCESS geht nach einem Takt in MR-IDLE über.

- **MR-FAILURE** : Ebenfalls ein Übergangszustand. Dieser setzt das FAILURE-Flag und resetet das ENGAGE-Flag im PCI-CTRL-Reg und geht danach zu MR-IDLE.

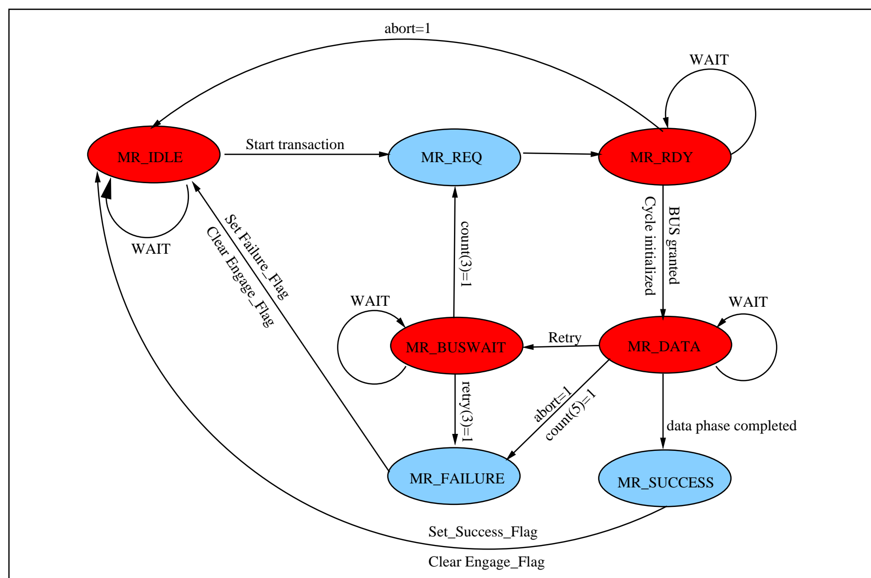


Abbildung 6.8: MasterRead-StateMachine

### MasterWrite-STATEMACHINE

Abb. 6.9 zeigt das Zustandsdiagramm der MasterWrite-STATEMACHINE. Einige der Zustände sind dieselben wie bei der MasterRead-Statemachine. Auf diese wird an dieser Stelle nicht mehr explizit eingegangen, da an entsprechender Stelle bei der MasterRead-Statemachine nachgelesen werden kann.

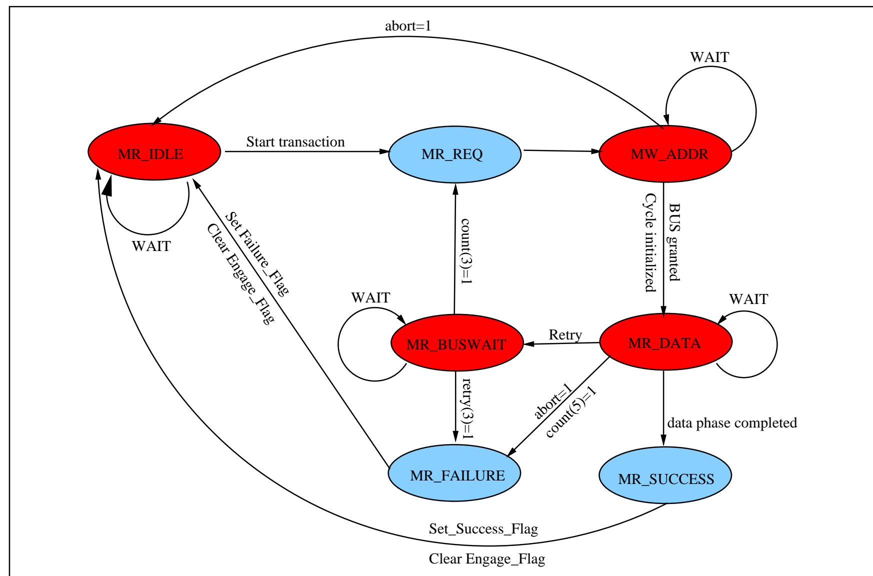


Abbildung 6.9: MasterWrite-StateMachine

- **MW-IDLE** : Identisch zu MR-IDLE
- **MW-REQ** : Identisch zu MR-REQ
- **MW-ADDR** : Address-Phase. Bis auf das angelegte Commando ist diese Phase identisch mit MR-RDY.
- **MW-DATA** : Data-Phase. Im Gegensatz zu MR-DATA muss nicht auf das Datum vom Target gewartet werden, sondern dieses wird unmittelbar zu Beginn der Data-Phase aus dem DATA-OUT-Reg des Register-File an den PCI-Core übergeben . Danach wird auf die Bestätigung vom Target gewartet. Was die Folgezustände angeht, ist dieser Zustand mit MR-DATA identisch.
- **MW-BUSWAIT** : Identisch zu MR-BUSWAIT
- **MW-SUCCESS** : Identisch zu MR-SUCCESS
- **MW-FAILURE** : Identisch zu MR-FAILURE

## 6.6 Zusammenfassung PCI-Design

Das PCI-Design, bestehend aus Register-File, PCI-Controller und PCI-Core, ist in der Lage, alle Arten von 32-Bit Single-Zugriffe (I/O, CFG, MEM) auf den PCI-Addressraum durchzuführen. Das Design wurde sowohl unter ModelSim mittels einer PCI-Testbench getestet als auch im Prototypen des C.I.A.-Boards unter realen Bedingungen. Zur Überprüfung der realen

PCI-Transaktionen wurden die BUS-Signale und -Zyklen mit einem PCI-Tracer mitgeschnitten. Als reale PCI-Transaktionen wurden das Auslesen des Configuration-Space verschiedener PCI-Karten durchgeführt und in den Bildschirmspeicher der VGA-Karte geschrieben und gelesen. Im Falle der VGA-Karte konnte der Erfolg der Transaktion unmittelbar am Bildschirm beobachtet werden. Umgekehrt ermöglicht das Auslesen des Bildschirmspeichers den Export des Bildschirms, solange bis der C.I.A. in der Lage sein wird, die VGA-Karte vollständig zu ersetzten. Leider war es im Rahmen dieser Diplomarbeit nicht möglich, die VGA-Implementation vollständig durchzuführen, jedoch wurde dazu ein Konzept erarbeitet. Der Grund dafür ist der hierarchische Aufbau des PCI-Busses. Zum Test der Designs standen vor Entwicklung des C.I.A. Board nur 3.3V PCI-Entwicklungskarten zur Verfügung. Der 3.3V PCI-Bus ist jedoch durch eine PCI-to-PCI-Bridge vom 5V PCI-Bus getrennt und liegt hierarchisch unterhalb. Diese Bridge läßt nun keine VGA-Zyklen zum 3.3V-Bus durch, was eine Entwicklung mit 3.3V-PCI-Karten unmöglich macht. Für die Entwicklung des C.I.A. wurde auf die Implementation von 64-Bit Transaktionen, sowie dem Burst-Mode verzichtet, da sie nicht benötigt werden. Allerdings ist das Design so konzipiert, dass eine Modifikation leicht möglich ist. Da Daten - und Steuerpfade getrennt gehalten wurden, reichen zur Unterstützung des 64-Bit-Modus einfach breitere Register und ein paar zusätzliche Steuersignale. Die StateMachines müssen nicht verändert werden, da sie nichts von der Datenbreite wissen, sondern nur zwischen den verschiedenen Phasen eines Zyklus umschalten. Auch der Burst-Mode sollte kein größeres Problem darstellen. Ein zusätzlicher Zustand der StateMaschine, ein Dual-Port-RAM mit Address-Counter im Registerfile anstelle von DATA-IN-REG und DATA-OUT-REG und zusätzliche Steuersignale zwischen Register-File und PCI-Controller.

# Kapitel 7

# PCI-Design - Messungen

Für den Test des PCI-Designs wurde folgende Konfiguration des NIOS gewählt.

- 32-Bit CPU
- Timer
- UART
- Boot-ROM mit G.E.R.M.S.-Monitor
- 8KByte OnChip-RAM
- 1MByte Synchrones SRAM
- 32-Bit Input-PIO
- 32-Bit Output-PIO
- 8-Bit Control-PIO

Das System wurde so konfiguriert, daß es ausreichend Systemspeicher zur Entwicklung eigener Test-Programme besitzt, gleichzeitig aber noch einen minimalen OnChip-Speicher im FPGA als Fall-Back Lösung. Der Timer kann zum Erstellung von Benchmarks genutzt werden und die UART zusammen mit dem G.E.R.M.S. ermöglichen ein Terminal-Kommunikation mit dem NIOS. Die 3 PIOs dienen zur Ansteuerung des PCI-Designs.

Das PCI-Design besteht aus dem Register-File, dem PCI-Controller und einem 64-Bit Master-Core. Tabelle 7.1 gibt einen Überblick über den Ressourcen-Verbrauch der einzelnen Komponenten.

| Komponente | Logic-Elements(LE) | Register |
|---|---|---|
| NIOS-System | 2692 | 1162 |
| Register-File | 200 | 123 |
| PCI-Controller | 77 | 25 |
| PCI-Core | 1289 | 477 |

Tabelle 7.1: Ressourcen-Verbrauch des PCI-Designs

Die maximale Frequenz des PCI-Controllers und des Register-Files wurden von dem Timing-Analyser der QUARTUS-Software nach der Synthese zu 256MHz bestimmt.

Um das Design zu testen, wurde das C.I.A.-Board zusammen mit einem PCI-Tracer in einem Test-Rechner installiert. Anschließend wurde der PCI-Bus gescannt (CFG-Read) und Write/Read-Zyklen gegen den Bildschirmspeicher der VGA-Karte gefahren (MEM-Read/Write). Auf die Darstellung von CFG-Write- und I/O-Zyklen wurde hier bewußt verzichtet. Ursprünglich bestand keine Notwendigkeit diese Zyklen zu implementieren, daher wurden sie und ihre Wirkung auf das System nicht ausreichend untersucht, obwohl das PCI-Design sie unterstützt. Dies wurde sowohl in der Simulation als auch mit dem Tracer bestätigt.

## 7.1 Configuration-Read

Abb. 7.1 zeigt mehrere CFG-Reads durch den NIOS zum Scannen des Busses. Dabei wurde an Adresse 0x00080100 die VGA-Karte gefunden. Ausgelesen wurden Vendor- und Device-ID, in diesem Falle 0x47501002, d.h Vendor-ID : 1002, Device-ID : 4750. Korrekt erkannt wurde eine *Rage 3D Pro PCI*-Karte der Firma ATI. Abb. 7.2 zeigt die Transaktion auf dem PCI-Bus.

```
~/src
[SOPC Builder]$ nr pci_control.srec

nios-run: Downloading...............................
....................................................
nios-run: Terminal mode (Control-C exits)
==========================================
Reading...    Addresse : 0x900    => 0x0
Reading...    Addresse : 0x1100   => 0x0
Reading...    Addresse : 0x2100   => 0x0
Reading...    Addresse : 0x4100   => 0x0
Reading...    Addresse : 0x8100   => 0x0
Reading...    Addresse : 0x10100   => 0x0
Reading...    Addresse : 0x20100   => 0x0
Reading...    Addresse : 0x40100   => 0x0
Reading...    Addresse : 0x80100   => 0x47501002
Reading...    Addresse : 0x100100   => 0x0
Reading...    Addresse : 0x200100   => 0x0
Reading...    Addresse : 0x400100   => 0x0
Reading...    Addresse : 0x800100   => 0x0
Reading...    Addresse : 0x1000100   => 0x0
Reading...    Addresse : 0x2000100   => 0x0
Reading...    Addresse : 0x4000100   => 0x0
Reading...    Addresse : 0x8000100   => 0x0
Reading...    Addresse : 0x10000100   => 0x0
Reading...    Addresse : 0x20000100   => 0x0
Reading...    Addresse : 0x40000100   => 0x0
#66D12008
CIA_PCI
+
```

Abbildung 7.1: Vom NIOS initialisierte Configuration-Read-Zyklen vom TYP-0

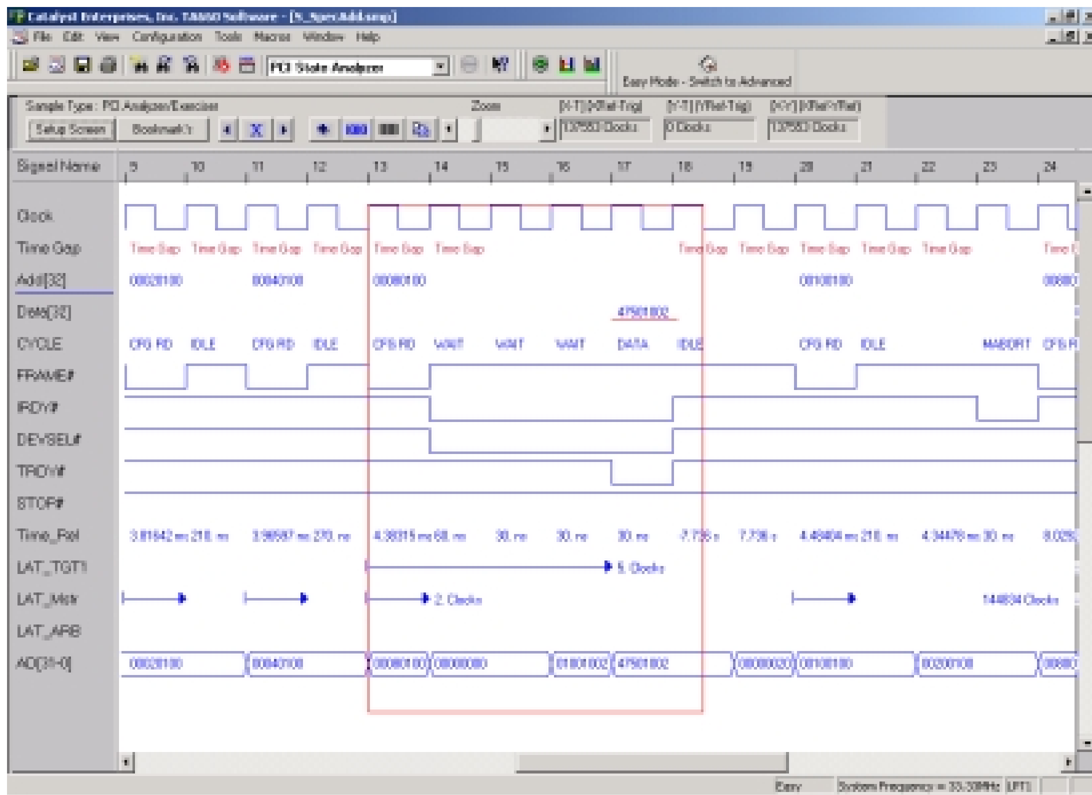


Abbildung 7.2: PCI-Trace der Configuration-Read-Zyklen. Rot umrahmt die Antwort der VGA-Karte auf den CFG-Read

## 7.2 Memory-Transaktions

### 7.2.1 Memory-Read

Abb. 7.3 zeigt mehrere Lese-Zyklen durch den NIOS auf einen Bereich des Bildschirmspeichers der VGA-Karte. Die entsprechenden Transaktionen auf dem PCI-Bus sind in 7.4 zu sehen.

```
Reading...   Reading from Address : 0xB807C => 0xFF96
Reading...   Reading from Address : 0xB8080 => 0xFF97
Reading...   Reading from Address : 0xB8084 => 0xFF98
Reading...   Reading from Address : 0xB8088 => 0xFF99
Reading...   Reading from Address : 0xB808C => 0xFF9A
Reading...   Reading from Address : 0xB8090 => 0xFF9B
Reading...   Reading from Address : 0xB8094 => 0xFF9C
Reading...   Reading from Address : 0xB8098 => 0xFF9D
Reading...   Reading from Address : 0xB809C => 0xFF9E
Reading...   Reading from Address : 0xB80A0 => 0xFF9F
Reading...   Reading from Address : 0xB80A4 => 0xFFA0
Reading...   Reading from Address : 0xB80A8 => 0xFFA1
Reading...   Reading from Address : 0xB80AC => 0xFFA2
Reading...   Reading from Address : 0xB80B0 => 0xFFA3
```

Abbildung 7.3: Vom NIOS initialisierte Memory-Read-Zyklen

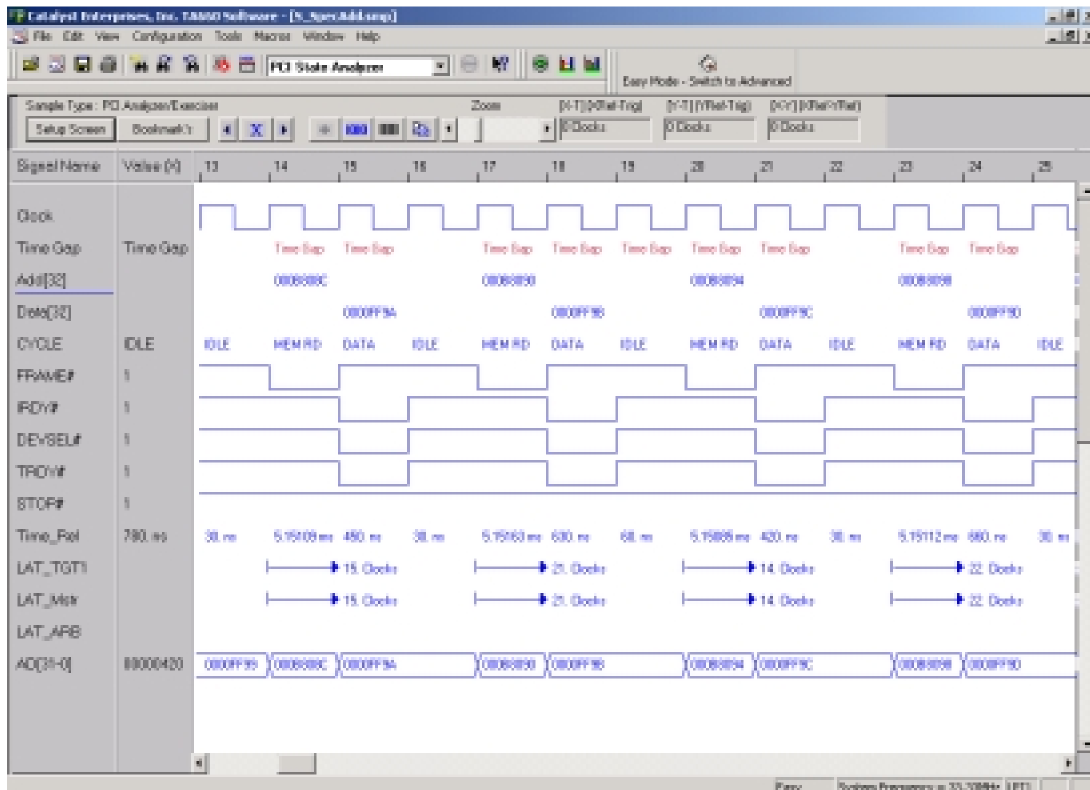


Abbildung 7.4: PCI-Trace der Memory-Read-Zyklen

### 7.2.2 Memory-Write

Abb. 7.5 zeigt mehrere Schreib-Zyklen durch den NIOS auf einen Bereich des Bildschirmspeichers der VGA-Karte. Die entsprechenden Transaktionen auf dem PCI-Bus sind in 7.6 zu sehen.

```
Writing to Address : 0xB8364 => 0x10050
Writing to Address : 0xB8368 => 0x10051
Writing to Address : 0xB836C => 0x10052
Writing to Address : 0xB8370 => 0x10053
Writing to Address : 0xB8374 => 0x10054
Writing to Address : 0xB8378 => 0x10055
Writing to Address : 0xB837C => 0x10056
Writing to Address : 0xB8380 => 0x10057
Writing to Address : 0xB8384 => 0x10058
Writing to Address : 0xB8388 => 0x10059
Writing to Address : 0xB838C => 0x1005A
Writing to Address : 0xB8390 => 0x1005B
Writing to Address : 0xB8394 => 0x1005C
Writing to Address : 0xB8398 => 0x1005D
Writing to Address : 0xB839C => 0x1005E
```

Abbildung 7.5: Vom NIOS initialisierte Memory-Write-Zyklen

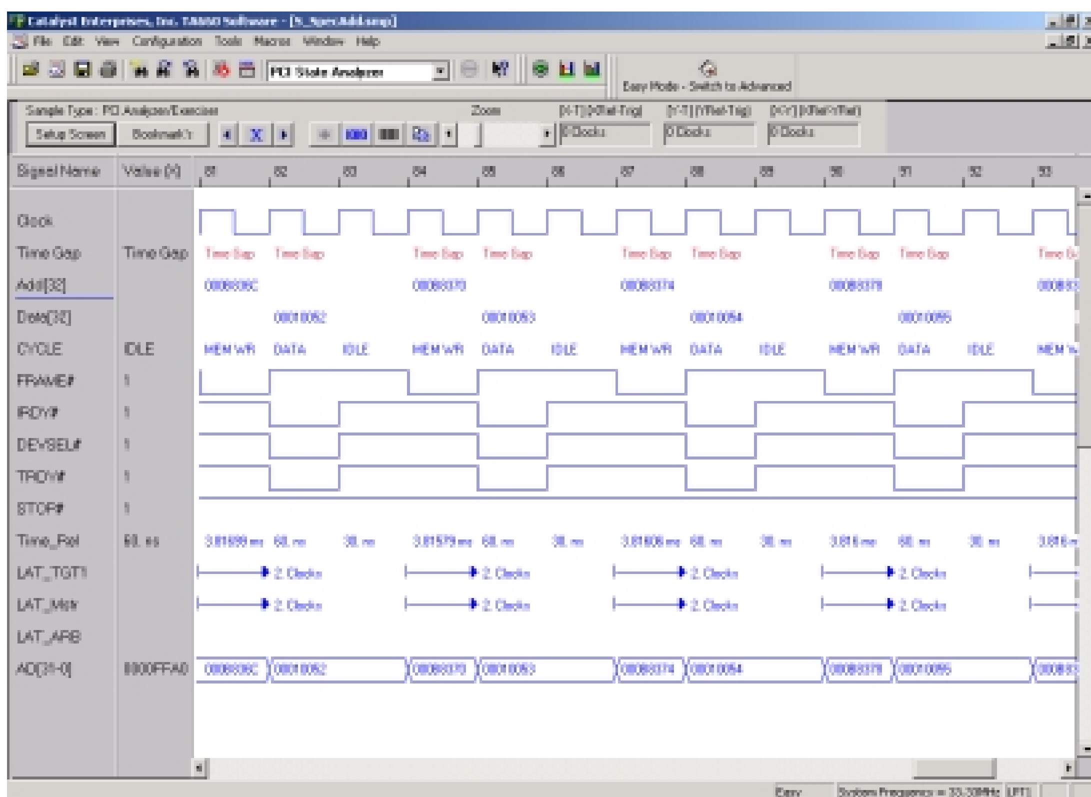


Abbildung 7.6: PCI-Trace der Memory-Write-Zyklen

# Kapitel 8

# Das C.I.A.-Board

## 8.1 Überblick

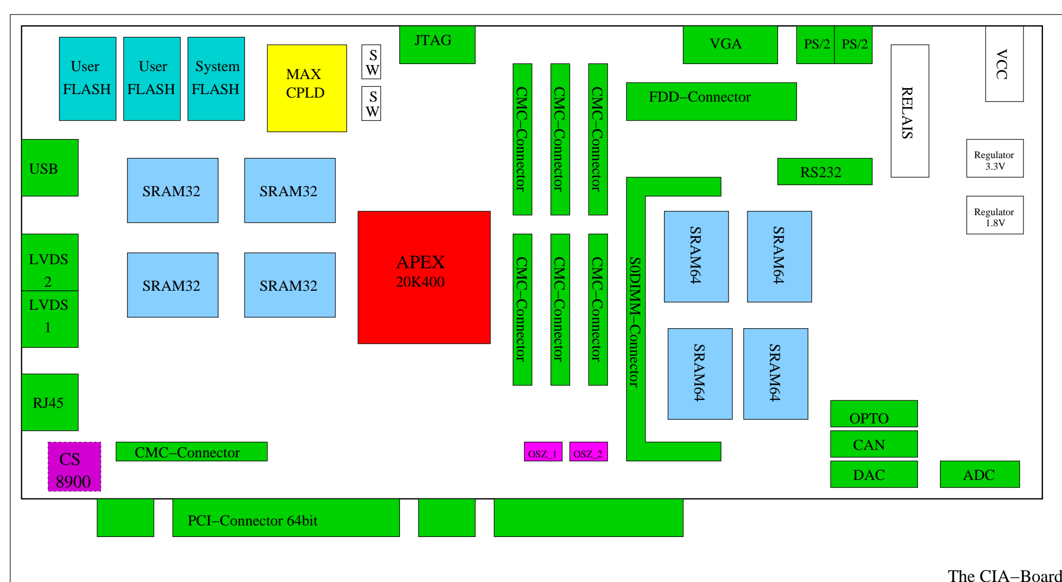


Abbildung 8.1: Überblick über das C.I.A.-Board

Bei dem C.I.A.-Board handelt es sich um eine FPGA-basierte PCI-Steckkarte, welche als Hardware-Grundlage für verschiedene Anwendungen gedacht ist. Eine Anwendung ist der **C**luster **I**nterface **A**dapter, dessen Name mittlerweile stellvertretend für das komplette Board steht. Unter funktionalen Gesichtspunkten kann das C.I.A.-Board in 3 Bereiche gegliedert werden:

- **SYSTEM-LOGIK** : Die komplette System-Logik ist im FPGA/CPLD untergebracht. Dies gewährleistet die größtmögliche Flexibiliät. Neue Designs oder Updates des bestehenden Systems können einfach in den FPGA geladen werden. In Falle des C.I.A. besteht die System-Logik aus dem NIOS-System und den C.I.A.-Designs.
- **Memory-Units** : Zwei Units dienen als Systemspeicher. Eine 32-Bit-Unit bestehend aus nichtflüchtigem FLASH und SRAM sowie eine 64-Bit-Unit mit SRAM. Die 64-Bit Unit bietet weiterhin einen S0DIMM Connector zur Aufnahme von SDRAM.
- **Schnittstellen** : Zur Kommunikation der Designs des FPGAs verfügt das C.I.A.-Board über verschieden Schnittstellen zur Außenwelt. Alle sind direkt an die I/O-Ports des

FPGAs angeschlossen, falls nötig befinden sich Level-Shifter und Bus-Switches in den Signalpfaden um die unterschiedlichen Potentiale der einzelnen Standards auf ein einheitliches, für den FPGA verträgliches Niveau zu bringen.

Dieses Kapitel beschäftig sich mit der eigentlichen Hardware, den Ressourcen des C.I.A.-Boards, sowie der Konfiguration des FPGA/CPLDs. Der System-Logik, d.h. den Designs innerhalb des FPGAs und des CPLDs, ist ein eigenes Kapitel, Kapitel 9, gewidmet. Alle Schaltpläne des C.I.A.Board finden sich auf der CD wieder.
Abb. 8.1 gibt einen Überblick über das Board und die auf ihm vorhanden Komponenten.

## 8.2 Ressourcen des C.I.A.-Boards

In den Unterkapiteln werden die einzelnen Ressourcen des C.I.A.-Boards und deren Zusammenspiel behandelt. Abbildung 8.2 gibt hierzu einen globalen Überblick über die Komponenten und deren Signalpfade.

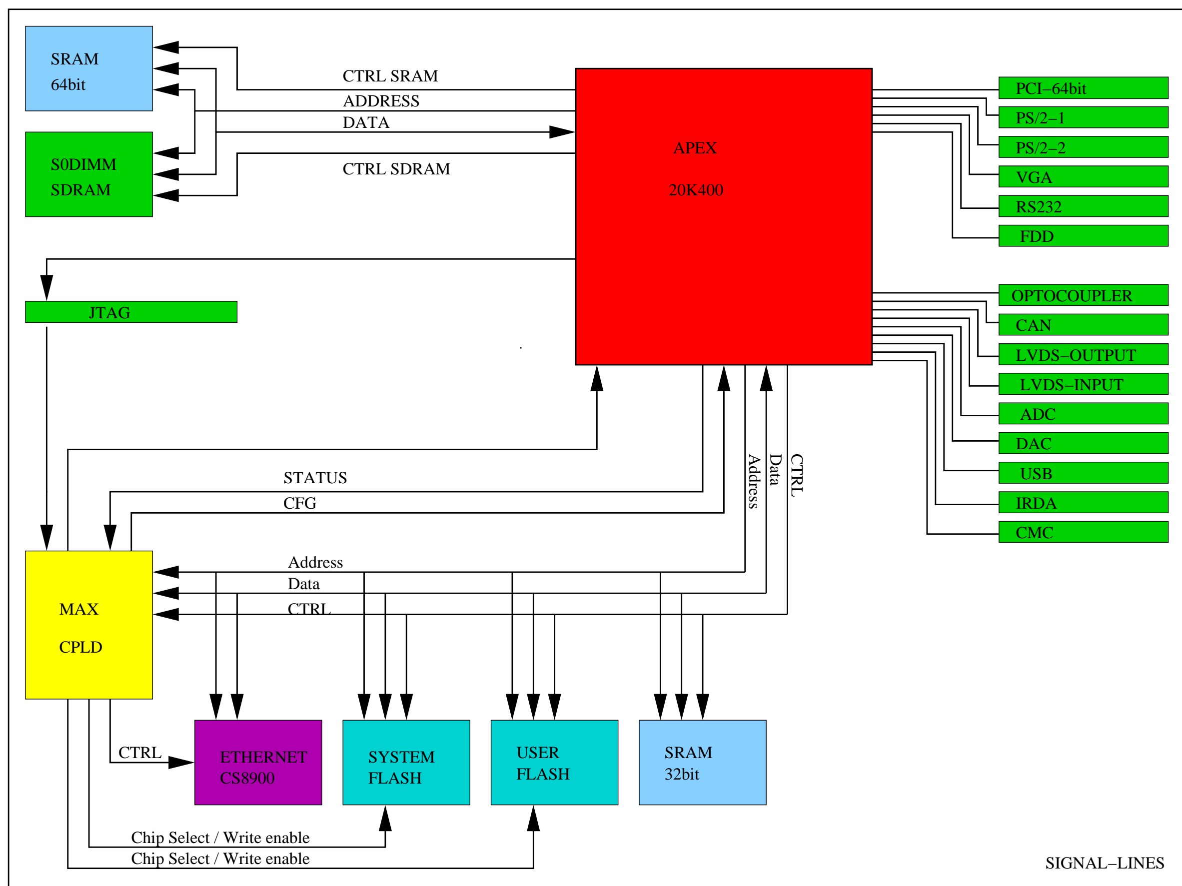


Abbildung 8.2: C.I.A.-Board - Ressourcen und Signalpfade

### 8.2.1 FPGA

Das Rückgrat des C.I.A.-Boards bildet ein FPGA der Firma ALTERA. Dieser dient zur Aufnahme der kompletten System-Logik, im Falle des C.I.A. ist dies das NIOS-System-Modul und die C.I.A.-Designs. Für die Entwicklung und das Debuggen von Designs kann es beliebig oft und sehr einfach über eine JTAG-Schnittstelle programmiert werden. Desweiteren besteht

die Möglichkeit es über ein FLASH-Memory zu programmieren und so Stand-Alone, also ohne entsprechende Soft- und Hardware, zu konfigurieren. Auf der aktuellen Version des Boards befindet sich ein APEX20K400EFC, welches über folgende Resourcen verfügt:

- 400.000 Gates
- 16.640 LogicCells(LC)
- 502 User-I/Os

Das FPGA wurde so gewählt, dass es pinkompatibel mit dem APEX20K200EFC672, sowie dem APEX20K600EFC672 ist. Somit ist sowohl ein Upgrade aus Platzgründen (APEX20K600), als auch ein Downgrade aus Preisgründen möglich (APEX20K200).

### 8.2.2 64-Bit-Memory-Unit

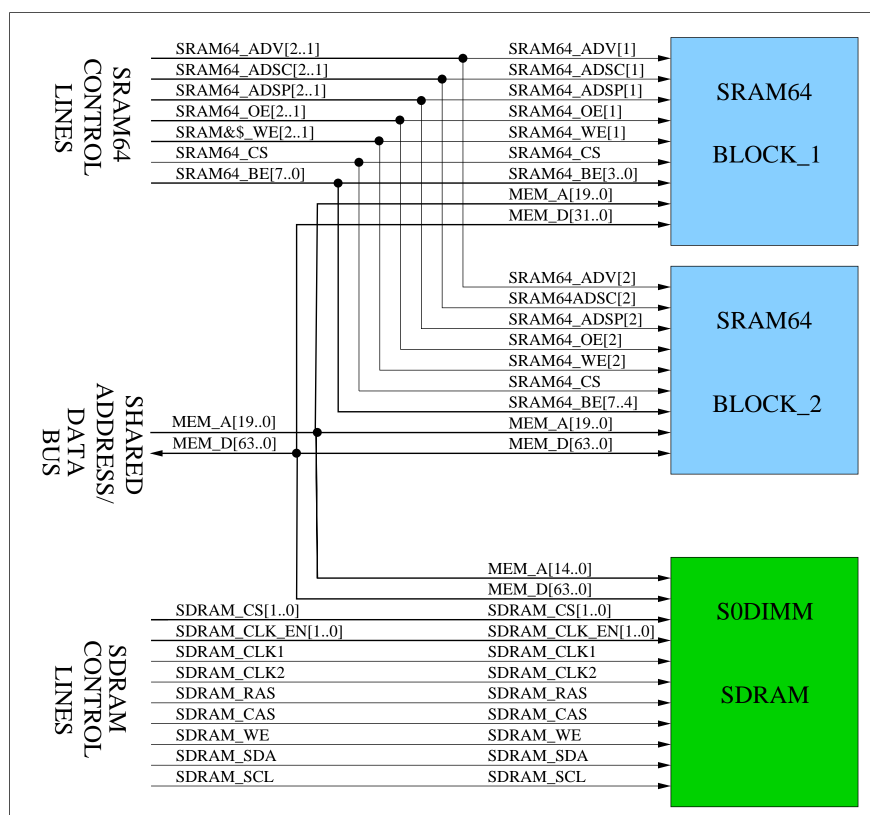


Abbildung 8.3: Signalpfade der 64-Bit-Memory-Unit. Alle Signale führen direkt zu den entsprechenden I/O-Pins des FPGAs

Die 64-Bit-Memory-Unit untergliedert sich in zwei Einheiten: eine SRAM-Unit und eine, optionale, SDRAM-Unit welche über einen 144poligen S0DIMM-Connector in des System integriert werden kann.Beide Units teilen sich den Daten- und Address-Bus, verfügen jedoch über separate Steuerleitungen.

**SRAM-Unit** : 4 SRAM-Chips bilden die SRAM-Unit. Jeweils 2 Chips bilden einen 32-Bit Block. Beide Blöcke sind parallel geschaltet um auf die Daten-Busbreite von 64-Bit zu kommen, können jedoch einzeln angesteuert werden.

**SDRAM-Unit** : durch den S0DIMM-Connector ist es möglich Standard SDRAM-Module in das System einzubinden und dieses dadurch bis auf 512MByte aufzurüsten.

Die 64-Bit-Memory-Unit stellt den Systemspeicher des NIOS-Systems dar. In der aktuellen Version ist dies durch die SRAM-Unit realisiert. Da auf dem C.I.A. jedoch keine Performance-kritischen Applikationen laufen werden, ist geplant die SRAM-Unit in zukünftigen Versionen durch billigeres und langsameres SDRAM zu ersetzten.
Abbildung 8.3 zeigt die Datenpfade der 64b-Memory-Unit.

### 8.2.3 32-Bit-Memory-Unit

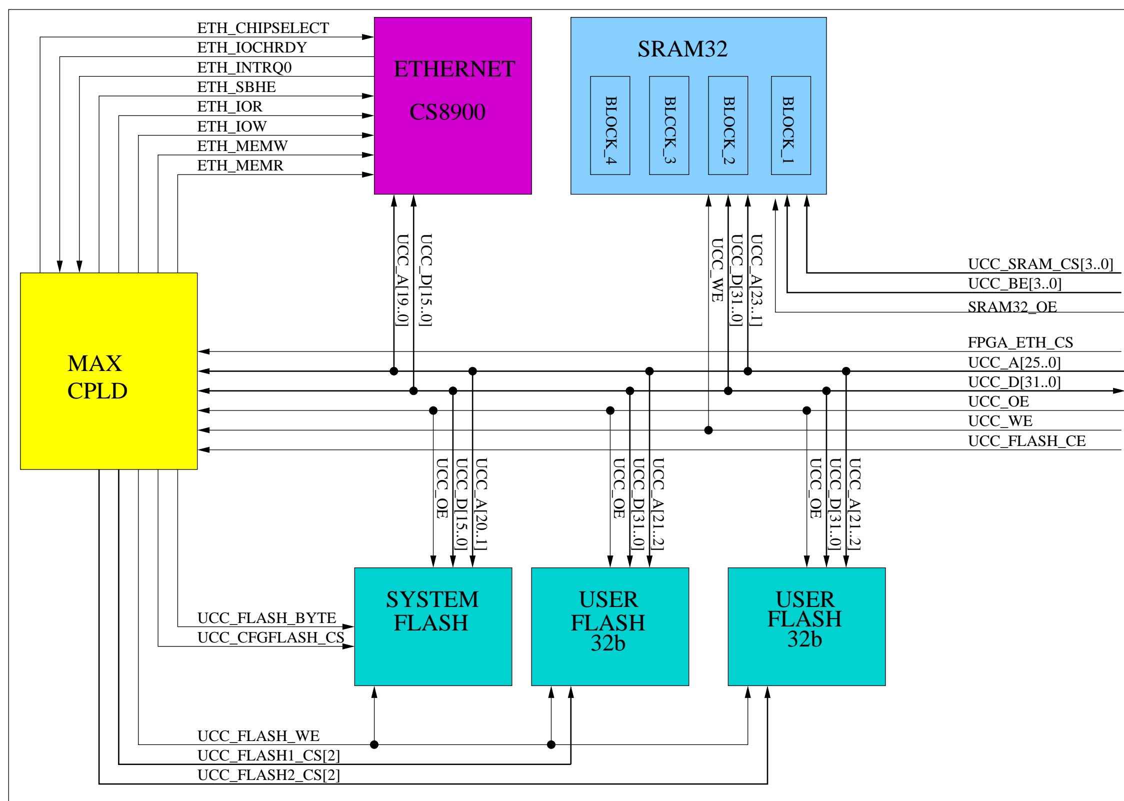


Abbildung 8.4: Signalpfade der 32B-Memory-Unit

Während die 64-Bit-Memory-Unit als reiner Arbeitsspeicher konzipiert ist und dementsprechend nur aus RAM besteht, beinhaltet die 32-Bit-Memory-Unit weitaus mehr. Neben reinem SRAM verfügt sie über verschiedene FLASH-Memorys, einen ALTERA MAX7000B CPLD, sowie einen Ethernet Chip, den Crystal-LAN CS8900:

**SRAM-Unit** : Insgesamt 8 Chips bilden die SRAM-Unit. Jeweils 2 Chips bilden einen 32-Bit Block, 4 Blöcke die komplette SRAM-Unit. Alle Blöcke teilen sich Daten-,Address-Bus, Write-Enable, Byte-Enable und Output-Enable, können jedoch über Chip-Select einzeln angesprochen werden. Aktueller Chip: CY1383B.

**FLASH-Unit** : 5 Chips bilden die FLASH-Unit. 1 Chip dient als Configuration-FLASH zur Konfiguration des FPGAs über den CPLD. Die restlichen 4 bilden das User-FLASH zur Speicherung des Linux-Systems oder für nichtflüchtige Daten. Dazu wurden wieder 2 Chips zu einem 32-Bit Block parallel geschaltet, sodaß dem User insgesamt 2 32-Bit Blöcke zur

Verfügung stehen. Sowohl System- als auch User-FLASH teilen sich mit den übrigen Units den Address- und Datenbus, sowie Output-Enable. Die übrigen Signale, Chip-Select und Write-Enable laufen über den CPLD und erfordern eine Addressdekodierung.

**CPLD** : der ALTERA MAX7000B CPLD dient sowohl der Konfiguration des FPGAs, als auch der Ansteuerung der FLASH-Unit und des Crystal-LAN CS8900. Der Grund dafür ist folgender: der CS8900 ist nur eine temporäre Lösung und soll auf Dauer durch ein entsprechendes Design im FPGA ersetzt werden. Würde man den CS8900 direkt über den FPGA ansteuern, so wären diese I/O-Pins später ungenutzt. Um dies zu vermeiden lagert man die Ansteuerung auf den CPLD aus. Ebenso werden die einzelnen Chip-Select Signal für die FLASHs aus der Adresse dekodiert. Dies erfordert wenig Logik, spart dafür aber wieder I/O-Pins am FPGA, welche bei einem Multi-Purpose-Board eine wichtige Ressource darstellen.

**CRYSTAL-LAN CS8900** : Der Crystal LAN CS8900 von Cirrus Logic ist ein Ethernet (IEEE 802.3)-Controller mit ISA-Bus Interface und stellt die Schnittstelle zwischen dem $\mu$C-Linux-System und einem Ethernet-Netzwerk dar. Er unterstützt 2 Modi, Memory-Mode und I/O-Mode, wobei vom $\mu$C-Linux z.Z. nur der I/O-Mode unterstützt wird. Auf eine vertiefende Beschreibung wird hier verzichtet, da der CS8900 nur eine temporäre Lösung darstellt und in naher Zukunft durch ein entsprechendes Design im FPGA ersetzt wird. Dies ist Ziel eines Projekts der Nachrichtentechnik der FH Köln in Zusammenarbeit mit dem KIP Heidelberg.

### 8.2.4 Schnittstellen

Folgende Schnittstellen sind direkt mit den I/O-Pins des FPGAs verbunden. Im Anhang ist ein Beschaltungsplan der einzelnen Schnittstellen, sowie die Pin-Belegung am FPGA. Soweit zu den Schnittstellen Datenblätter erhältlich sind, befinden sich diese auf der CD.

- 64-Bit PCI-Connector
- S0DIMM-Connector
- FDD-Connector
- VGA-Port
- 2 x PS/2-Port
- 10BaseT over RJ45
- RS232
- IrDA : 115kbps
- USB : 1.5 Mbps/12Mbps
- 2 x LVDS : 400 Mbps
- 4 x Optokoppler
- CMC-Connectors
- ADC : 10-Bit 4 Kanal
- DAC : 10-Bit Dual
- Realtime clock

Von diesen Schnittstellen wird das C.I.A. in seiner aktuellen Version folgende nutzten: PCI-Connector, FDD-Connector, RS232, RJ45, S0DIMM, DAC.

### 8.2.5 Clock-Distribution

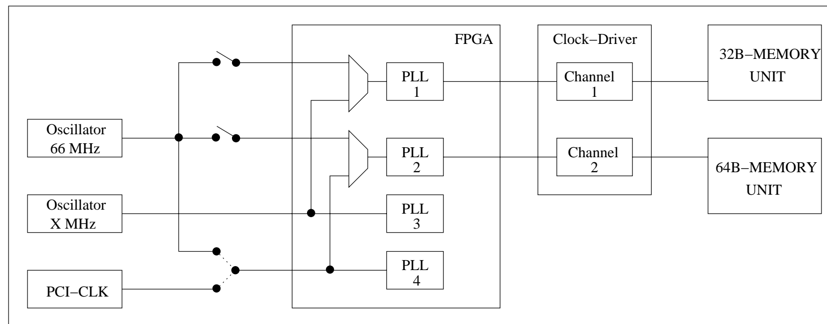


Abbildung 8.5: CLOCK-Distribution des C.I.A.-Boards

Das C.I.A.-Board verfügt über 2 Onboard Oszillatoren und einen Clock-Eingang über den PCI-Connector. Die entsprechenden Clock-Signale können über Jumper in die 4 Clock-Eingänge des FPGA eingespeist werden. Die 4 Clock-Eingänge des FPGAs sind intern sowohl mit dem Global-Clock-Network, als auch mit 4 PLLs verbunden. Zwei dieser PLLs besitzen die Möglichkeit ihre Ausgänge nach außen zu leiten und somit die beiden Memory-Units zu versorgen. Abb. 8.5 gibt einen Überblick über die Clock-Distribution. Exakte Information über die Ansteuerung und Verwendung der FPGA-internen Phase Locked Loops (PLLs) bietet der folgende Datasheet von ALTERA : [15]
Vorgesehen ist, die PCI-Designs über die PCI-Clock zu takten und das NIOS-System-Modul und die Memory-Units über die Onboard-Oszillatoren. Dies ist nötig um eine Clock auch bei ausgeschaltetem Host zu erhalten und so die Autonomie des C.I.A. zu gewährleisten.

## 8.3 Konfiguration des FPGAs und CPLD

Für das CPLD gibt nur eine Möglichkeit der Konfiguration, nämlich die über die JTAG-Schnittstelle. Für das FPGA dagegen existieren zwei Möglichkeiten, über JTAG und aus dem FLASH-Memory über den CPLD. Auf beide Möglichkeiten wird im folgenden eingegangen.

### 8.3.1 Konfiguration über JTAG

Erstere Möglickeit, via JTAG, bietet sich bei der Systementwicklung an und ermöglicht eine schnelle, unkomplizierte Programmierung des CPLD/FPGAs über die JTAG Schnittstelle und ein sogenanntes ByteBlaster Kabel. Das ByteBlaster-Kabel ermöglicht eine Datenübertragung zwischen dem Parallelport des Entwicklungsrechner und dem JTAG des C.I.A. Gesteuert wird die Datenübertragung über einen Teil der Design-Software QUARTUS II, bzw. MAX+PLUS, den Programmer. Dieser lädt die entsprechenden Configuration-Files (.sof, .pof) in den ausgewählten Baustein.
Die zweite Möglichkeit, die Konfiguration im FLASH zu speichern und den FPGA über den CPLD zu programmieren erfordert etwas mehr Aufwand, bietet jedoch die Möglichkeit, ein fertiges Design auf dem Board zu speichern und ohne zusätzliches technisches Equipment bei Power-Up innerhalb von Millisekunden in den FPGA zu laden.

### 8.3.2 Konfiguration über FLASH & CPLD

Abb. 8.6 zeigt, wie der APEX aus dem FLASH über den CPLD konfiguriert wird. Der FPGA kann über eine 8-Bit breiten Datenbus und einige Steuersignale programmiert werden. Das

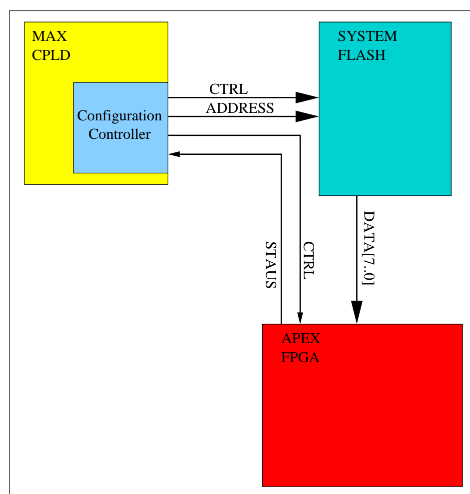


Abbildung 8.6: FPGA-Konfigurationsmechanismus über FLASH und CPLD

CPLD enthält ein entsprechendes Design zur Konfiguration, welches bei Power-up sowohl die Programmierung des FPGAs, als auch das Auslesen des SYSTEM-FLASHs initialisiert und für eine synchrone Datenübertragung vom FLASH zum APEX sorgt.

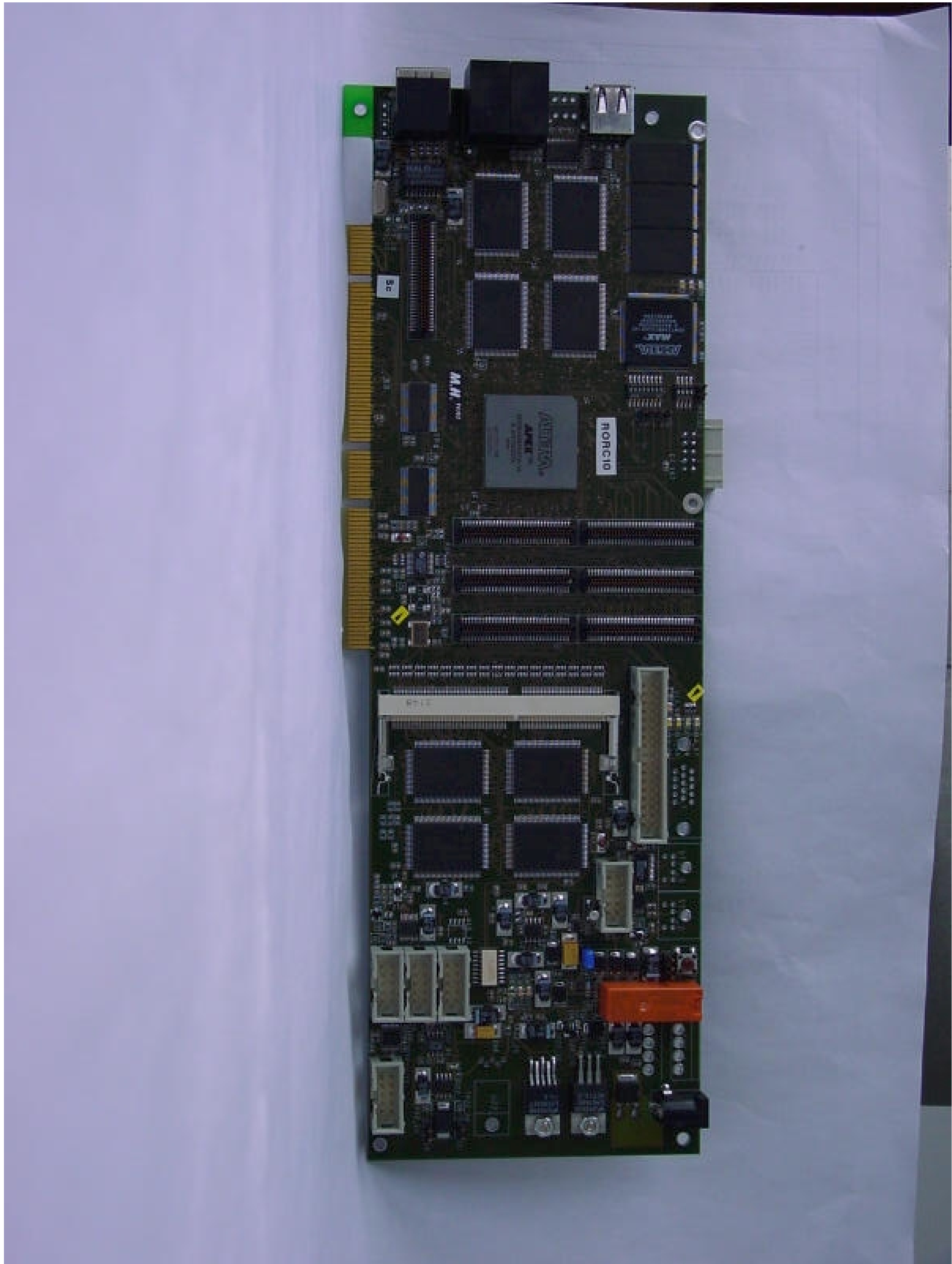


Abbildung 8.7: Das C.I.A.-Board

# Kapitel 9

# System-Logik

## 9.1 Überblick

Die System-Logik bezeichnet die komplette Logik im FPGA. Im Falle des C.I.A. ist sie modular aufbaut. Abb. 9.1 zeigt die Gliederung für den C.I.A.. Im Rahmen dieser Diplomarbeit wurde der Export des NIOS-Systems vom EXCALIBUR Development-Board auf das C.I.A.-Board durchgeführt, sowie das PCI-Design, das Interface zu den Memory-Units und ein Konzept für die VGA-Implementierung entwickelt. Das FDD- und das PS/2-Design wurde im Rahmen der Diplomarbeit von Stefan Philipp implementiert und werden ausführlich in dessen Diplomarbeit behandelt[24]. Abschnitt 9.2 gibt den aktuellen Stand der Entwicklung wieder, Abschnitt 9.3 das Ziel der Entwicklung und Abschnitt 9.4 ein Vorschlag für die Implementation des VGA-Designs.

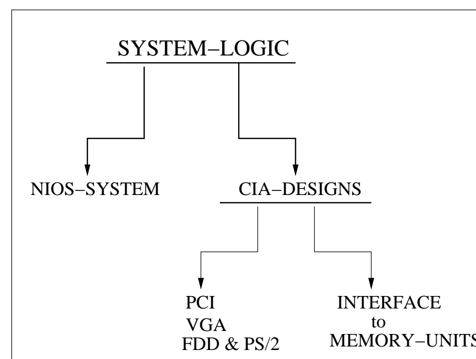


Abbildung 9.1: System-Logik

## 9.2 Aktuelle Implementation

Die aktuelle Implementation der C.I.A.-System-Logik besteht aus einem, auf das C.I.A.-Board angepaßten NIOS-System-Modul, den in Kapitel 6 beschriebenen PCI-Designs, sowie zwei Memory-Interfaces, eins zur 32–Bit Memory-Unit und eins zur 64–Bit Memory-Unit. Die folgenden Unterabschnitte behandeln jeweils die einzelnen Designs.

### 9.2.1 Das NIOS-System-Modul des C.I.A. Boards

Ziel der Entwicklung ist es, nicht nur das NIOS-System auf das C.I.A.-Board zu portieren, sondern das komplette $\mu$C-Linux. Daher wurde das System-Modul so konfiguriert, wie es

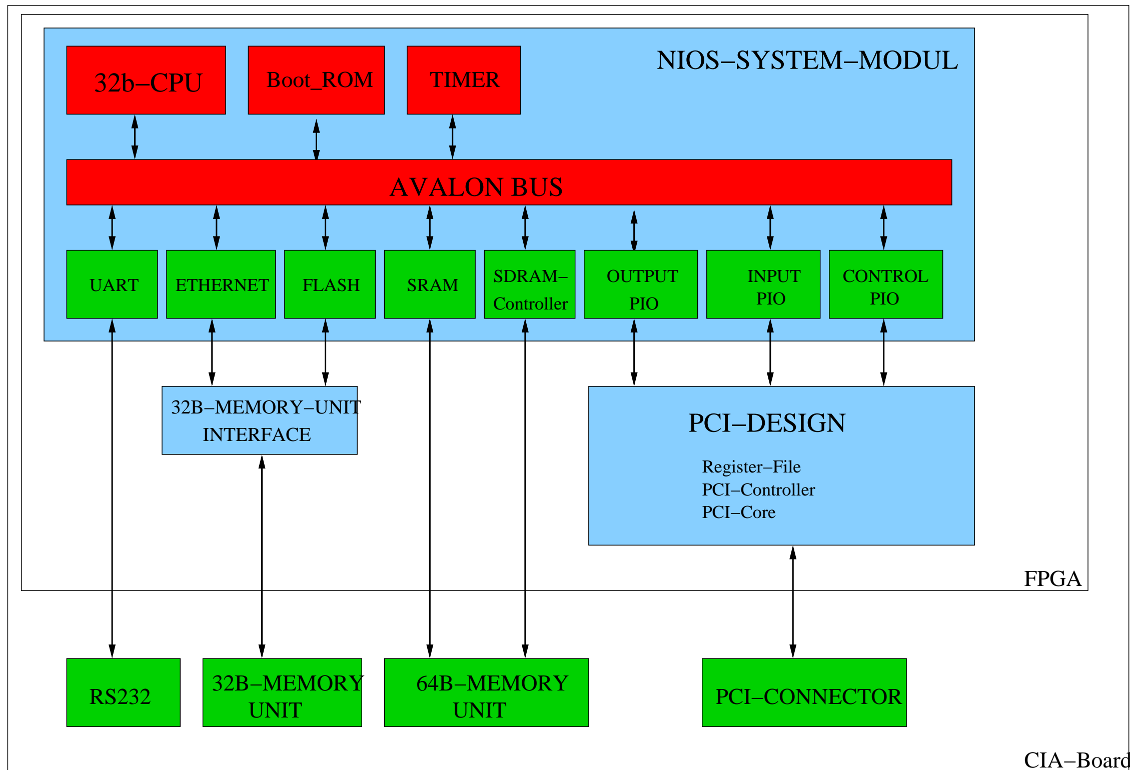


Abbildung 9.2: Aktuelle Implementation der C.I.A.-Logik

auch zum Betrieb des $\mu$C-Linux Systems auf dem EXCALIBUR Development-Board zu finden ist. Hierbei tritt jedoch folgendes Problem auf. Die Schnittstellen zur Peripherie sind auf die jeweils verwendetem Peripherie-Bausteine, wie SRAM, FLASH, usw. und deren Bustopologie zugeschnitten. Peripherie-Module des EXCALIBUR Boards können also ohne Modifikation nicht auf dem C.I.A.-Board eingesetzt werden. Ebenso ist der G.E.R.M.S.-Monitor, das Betriebssystem des NIOS, speziell auf das Development-Board angepasst. Somit entstand die Notwendigkeit, einige Peripherie-Module neu zu schreiben und zu testen, sowie den G.E.R.M.S.-Monitor so zu modifizieren, dass er den Ansprüchen des C.I.A.-Boards genügt. Die folgenden Abschnitte gehen auf die verschiedenen Probleme und deren Lösung ein:

### SRAM-Modul

Das NIOS-Development-Board verfügt über asynchrones statisches RAM (asynch.SRAM) das C.I.A.-Board dagegen über synchrones, statisches RAM (synch.SRAM). Beide haben nicht nur unterschiedliche Address-,Daten- und Steuerleitungen, sondern auch unterschiedliche Timings. Für das C.I.A.-Board wurde also ein neues Peripherie-Modul geschrieben, das *cia_ssram*- Modul. Das Modul ist in einer menschenlesbaren Scriptsprache geschrieben und kann vom SOPC-Builder in ein System eingebunden werden. Innerhalb des Moduls werden sowohl die Ports zum SSRAM, als auch das Timing definiert. Beides erhält man aus dem entsprechenden Datenblatt[27]. Der genauen Befehlssynthax der Scriptsprache findet sich in [16] wieder. Alle Quellen zur Erstellung des Moduls, sowie das Modul selbst finden sich auf der CD wieder.

### FLASH-Modul

Bei der Entwicklung des FLASH-Moduls trat, neben der Anpassung der Address-, Daten- und Steuerleitungen, sowie der Timings, ein weiteres Problem auf. Auf dem C.I.A.-Board werden jeweils 2 FLASHs mit 16-Bit Datenbreite gemeinsam angesteuert und bilden so einen 32-Bit Datenbus. Dies ist jedoch von den Tools zur Systementwicklung des NIOS nicht vorgesehen. Dort werden nur 16-Bit FLASHs unterstützt. Der Grund dafür ist folgender. FLASHs können nicht direkt beschrieben und gelöscht werden, wie normale RAMs, sondern lediglich direkt ausgelesen werden. Stattdessen verfügen sie über interne Programmieralgorithmen, welche dadurch initialisiert werden, dass man bestimmte Werte an bestimmte Adressen des FLASHs schreibt. Der interne Programmieralgorihmus beschreibt oder löscht dann das FLASH. Daher wurden zwei Module für das FLASH-Memory der C.I.A. entwickelt, ein *cia_lowerflash* und ein *cia_upperflash* Modul. Beide Module erzeugen gemeinsame Address-und Steuerleitungen(bis auf das Chip-Select), jedoch einen getrennten Datenbus. Somit können beide 16-Bit FLASHs separat angesteuert werden.

### Der G.E.R.M.S.-Monitor des C.I.A.-Boards

Wie in Kapitel 4.4 erläutert, stellt der G.E.R.M.S.-Monitor eine Art Minimal-Betriebssystem des NIOS dar. Er ist in der Lage, den Speicherinhalt des gesamten Addressraums anzuzeigen, Daten direkt an eine bestimmte Adresse zu schreiben, sowie FLASHs zu löschen und zu beschreiben. Während normale Speicher, wie SRAM, SDRAM, direkt über die Hardware, d.h. den AVALON-Bus und die Peripherie-Module beschrieben werden können, indem diese die Address-, Daten- und Steuersignale generieren, sind bei FLASHs bestimmte Routinen notwendig, um ein Datum in das FLASH zu schreiben. Der G.E.R.M.S.-Monitor stellte nun die Schnittstelle zwischen User und FLASH dar, sodaß die Datenübertragung für den User transparent bleibt. Der User übergibt das zu schreibende Datum, der G.E.R.M.S.-Monitor führt intern die Write/Erase-Routinen aus und speichert das Datum im FLASH. Diese Write/Erase-Routinen sind jedoch keine generischen Routinen, sondern speziell an die FLASHs des EXCALIBUR Development-Boards angepaßt. Daher war es nötig, den Monitor entsprechend umzuschreiben und in das ROM des NIOS-System Modul einzubinden. Das Einbinden in das ROM ist relativ einfach über den SOPC-Builder möglich, da dieser die Möglichkeit bietet, vorhanden System-Speicher bei der Generierung zu initialisieren. Bei der Wahl des Peripherie-Moduls ROM, zur Erstellung des System-ROMs, kann angegeben werden, ob das ROM mit einem Memory-Imagefile (.mif, .srec, .hex) geladen werden soll, oder ob dieses Imagefile aus vorhandenem Source-Code erstellte werden soll. Entscheidet man sich für die Compilierung des Source-Codes, so wird dieser bei der Generierung des System-Moduls automatisch mit der aktuellen System-Map compiliert, in ein Memory-Imagefile umgewandelt (. mif) und in das System-ROM eingebunden. Der Source-Code des Monitor ist komplett in Assembler geschrieben und enthält neben verschieden Routinen zur Ein/ Ausgabe über die UART-Schnittstelle und zum Lesen/ Schreiben des RAMS auch die Write/Erase-Routinen des FLASHs. Diese teilen sich in 2 Sequenzen. Eine Routine zur Erkennung, ob auf das FLASH zugegriffen wird, und eine weitere Routine, welche die eigentlichen Write/Erase Operationen durchführt. Beim Schreiben/Löschen eines bestimmten Addressbereichs wird also zuerst überprüft, ob ein FLASH-Zugriff stattfindet, falls ja wird die entsprechende Routine ausgeführt. Zur Anpassung an das C.I.A.-Board wurde die FLASH-Bereichserkennung verändert, da das C.I.A.-Board über 3, das EXCALIBUR Board nur über 1 Device verfügt. Die Write/Erase-Routine musste nicht verändert werden, da die FLASHs des C.I.A.- und des EXCALIBUR-Board in dieser Beziehung kompatibel sind. Der angepaßte G.E.R.M.S.-Monitor befindet sich als kommentierter Source-Code auf der beiliegenden CD.

### 9.2.2 Das Interface zur 32-bit Memory-Unit

Das Interface der 32-bit Memory-Unit steuert in der aktuellen Version den Zugriff auf das FLASH-Memory und den Ethernet-Chip CS8900 und besteht aus zwei Teilen, einem Interface im FPGA und einem im CPLD. Das Interface im FPGA multiplext die Adress-Busse von FLASH und CS8900 auf den Adress-Bus der 32-Bit-Memory-Unit. Dieser Adress-Bus führt, wie Abb. 8.4 zeigt, zu allen Devices der Unit. Der Multiplexer wird über das *chip-select_n* Signal des FLASH-Moduls geschaltet. Während eines Zugriffs auf das FLASH wird dessen Adresse durchgeschaltet, ansonsten die Adresse das CS8900. Es ist zwar möglich, die Peripherie-Module von FLASH und CS8900 so zu konfigurieren, daß der SOPC-Builder einen gemeinsamen Adress-Bus generiert, doch diese Möglichkeit kann hier nicht genutzt werden. Der Grund dafür ist der Operationsmode des CS8900. Dieser wird im sogenannten I/O-Mode betrieben, in dem er über 8 16-bit breite I/O-Ports angesprochen wird. Diese I/O-Ports liegen aufeinander folgend ab Adresse 300h im Adress-Raum des CS8900. Das CS8900-Peripherie-Modul stellt nun nur die Adress-Leitungen für den Offset ab 300h zur Verfügung, was bedeutet, daß die übrigen Adress-Leitungen zum CS8900 auf Adresse 300h festgelegt werden müssen. Daher muss zwischen dem FLASH Adress-Bus und dem auf 300h+Offset festgelegten CS8900-Adress-Bus umgeschaltet werden. Es ist nicht möglich, die Adresse das CS8900 in der NIOS-System-Map an die entsprechende Stelle zu legen, da der AVALON-Bus diese Adressen ummapped. Eine weitere Aufgabe des Interfaces ist das Encoden/Decoden der Steuersignal für den CS8900 und der Chipselect-Signale für die FLASHs. Die Steuersignale für den CS8900 werden im FPGA auf die oberen, unbenutzten Addressleitungen *ucc_a* codiert und im CPLD wieder decodiert und an den CS8900 geführt. Ebenfalls decodiert werden die einzelnen *chipselect* Signale der FLASHs aus der Adresse und dem globalen *chipselect* Signal der FLASHs.

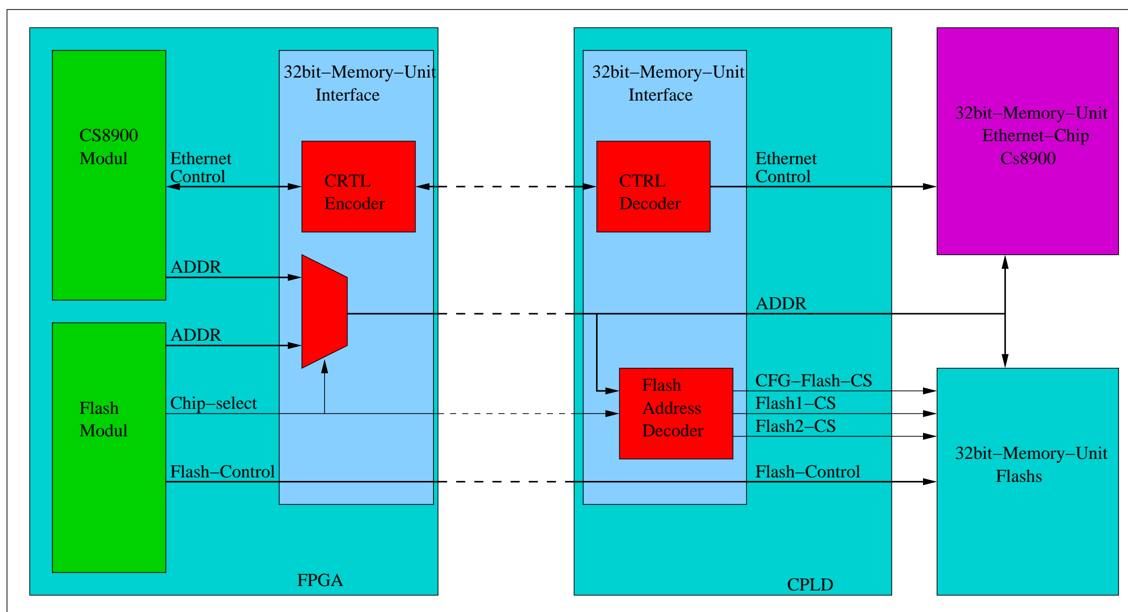


Abbildung 9.3: 32-Bit-Memory-Unit-Interface

### 9.2.3 PCI-Designs

Die während dieser Diplomarbeit entwickelten PCI-Designs werden ausführlich in Kapitel 6 behandelt. Auf sie wird an dieser Stelle nicht weiter eingegangen.

## 9.3 Geplante Implementation

Abb. 9.4 zeigt die geplante, finale Version des C.I.A.-Boards. Alle blauen Komponenten existieren bereits, das NIOS-System, die Interfaces und das PCI-Design wurden in dieser Diplomarbeit portiert und entwickelt, die PORT80h-, FDD- und PS/2-Designs von Stefan Philipp in dessen Diplomarbeit. Es fehlen die Schnittstellen zu den Designs von Stefan Philipp, sowie das VGA-Design. Beides konnte aus Zeitgründen nicht mehr in Angriff genommen werden, für Letzteres wurde jedoch ein Konzept entwickelt, welches eine mögliche Umsetztung beschreibt.

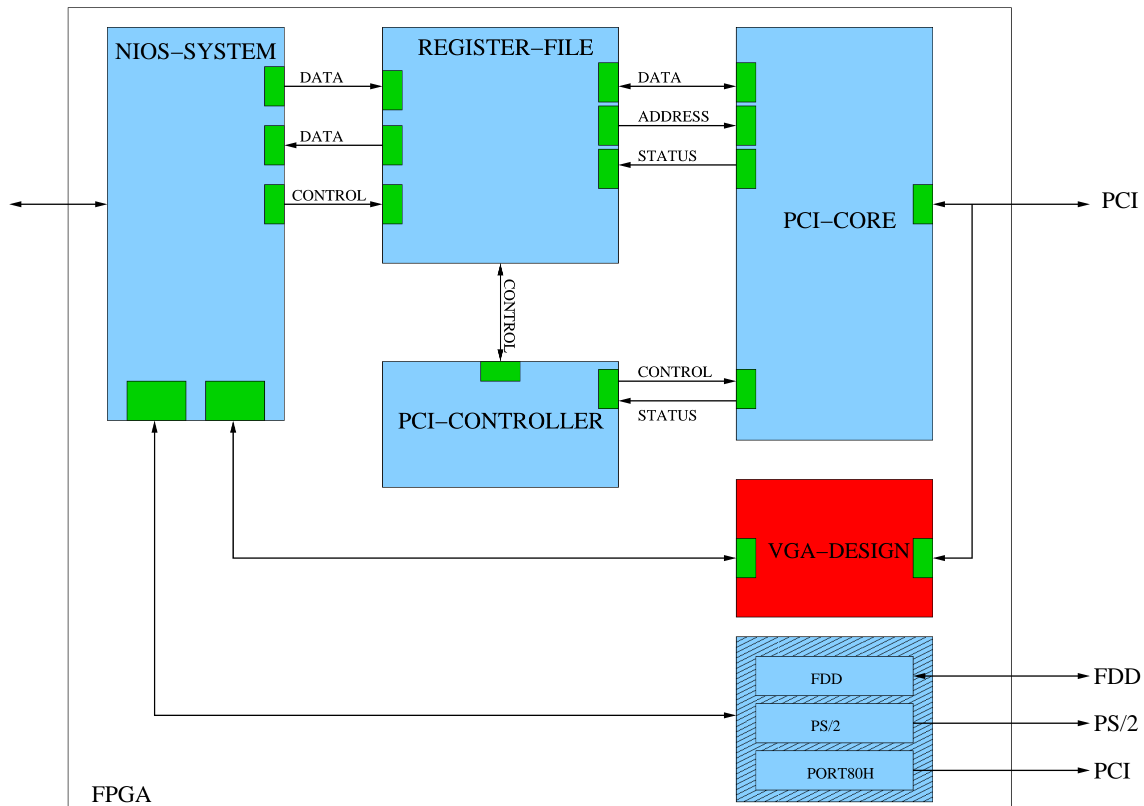


Abbildung 9.4: Zukünftige Implementation

## 9.4 VGA-Implementation

Eine der Anforderungen an den C.I.A. ist das Ersetzten der VGA-Karte im Host. Dabei soll der C.I.A. lediglich die Daten das Bildschirm-Speichers exportieren und nicht die Ansteuerung eines Monitors übernehmen. Folglich muss nicht die komplette Funktionalität nachgebildet werden. Bevor auf das eigentlich Konzept eingegangen wird, erfolgt eine kurze Einführung in die Funktionsweise einer VGA-Karte, dabei werden hier nur die Schnittstellen zum PC erklärt:

Jede VGA-Karte verfügt über ein VGA-Bios, ein VGA-RAM und mehrere I/O-Register, welche in den Adress-Raum des PCI-Busses gemappt werden:
Das **VGA-Bios** enthält Routinen zur Ansteuerung der VGA-Karte und wird, im Gegensatz zu anderen PCI-Devices, an eine festen Adresse im Adress-Raum gemappt. Der Grund dafür liegt in der Vergangenheit des VGA-Standards. Das BIOS ruft Grafik-Routinen durch bestimmte Interrupts auf, welche wiederum die entsprechenden Funktionen des VGA-Bios aufrufen. Dazu müssen sich diese Funktionen aber an bestimmten Adressen befinden.

Das **VGA-RAM** dient zur Aufnahme der Bildschirm-Daten. Diese werden in das RAM geschrieben und von dort für den Monitor aufbereitet. Um an die Bildschirm-Daten heranzukommen reicht also ein Auslesen des VGA-RAMs.
Die **I/O-Register** steueren den Zugriff auf die VGA-Karte. Sofern der Host keine Rückmeldung erwartet könnte man dies Register bereitstellen, aber keine Logik damit verbinden.

Abb. 9.5 zeigt eine mögliche Variante des VGA-Designs. Damit die PCI-Karte vom Host als VGA-Karte akzeptiert wird, muss sie sich als solche bei der Konfiguration zu erkennen geben. Hierzu ist ein Eintrag im Configuration Space, in der Device-ID nötig. Für das VGA-ROM, das VGA-RAM und die I/O-Register müssen jeweils die entsprechenden BARs bei der Erstellung des PCI-Cores reserviert werden. VGA-RAM und I/O-Register sind hierbei normale MEM- bzw. I/O-BARs, besitzen also eine konfigurierbarer Adresse. Das VGA-ROM dagegen muss an eine bestimmte Adresse hardcodiert werden. Der PCI-Core verfügt hierzu über eine entsprechende Möglicheit. Die VGA-Logik muss nun das Lesen des VGA-ROMs, das Schreiben des VGA-RAMs und das Schreiben/Lesen der I/O-Register durch den PCI-Core ermöglichen. Dies sollte durch einfache Statemachines möglich sein. Für das VGA-ROM existiert ein frei verfügbares Image im Internet, welches aber erst noch getestet werden müsste.

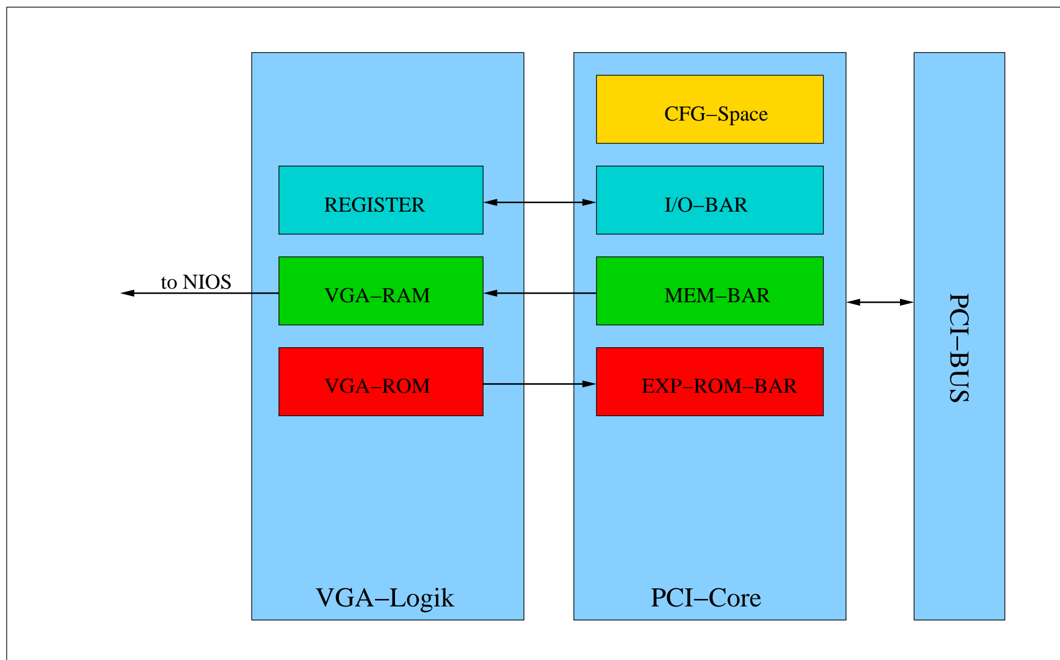


Abbildung 9.5: Konzept für zukünftige VGA-Implementation

# Kapitel 10

# Zusammenfassung und Ausblick

Der Inhalt dieser Arbeit kann in drei Themenbereiche unterteilt werden:

- Das NIOS-System incl. $\mu$C-Linux
- Die PCI-Control-Unit
- Das C.I.A.-Board

Jeder Bereich für sich stellt ein unabhängiges System dar, kann also unabhäbgig von den anderen betrieben werden. Die Schnittstellen der einzelnen Systeme wurden aneinander angepaßt, bzw. so entwickelt, daß sie zusammen den C.I.A. ergeben. Nachfolgend eine kurze Reflektion der einzelnen Systeme:
Das **NIOS-System** ist ein individuell konfigurierbares, netzwerkfähiges Embedded System, welches komplett als Source-Code vorliegt und erst durch seine Generierung auf eine Zieltechnologie abgebildet wird. Kern des Systems ist der NIOS 32-bit RISC-Prozessor zusammen mit dem AVALON-Bus. Der AVALON-Bus gestattet es, sowohl vorhanden, als auch eigene Peripherie-Module zu integrieren. Der Ausbau des Systems kann damit an die Anforderungen angepaßt werden und es können eigene Schnittstellen/Designs integriert werden. Mit dem $\mu$C-Linux System existiert ein netzwerkfähiges, linuxartiges Betriebssystem für den NIOS. Auch dieses liegt komplett als Source-Code vor und ermöglicht so eine Anpassung des Kernels und der Applikationen an das jeweils zugrunde liegende System. Ebenso wird das Erstellen und Einbinden eigener Anwendungen unterstützt. Im Rahmen dieser Arbeit wurde das NIOS-System von dem EXCALIBUR Development Kit auf das C.I.A.-Board exportiert und für das $\mu$C-Linux vorbereitet. Für die Komponenten SRAM, FLASH und Ethernet des C.I.A.-Boards wurden die entsprechenden Peripherie-Module entwickelt und getestet. Das $\mu$C-Linux selbst konnte nur auf dem Development Board in Betrieb genommen werden, da das Update für das NIOS-System des C.I.A.-Boards erst gegen Ende der Arbeit zur Verfügung stand. Es gibt jedoch keine Gründe, die gegen einen korrekten Betrieb auf dem C.I.A. sprechen. Über Ethernet ist ein Einloggen auf dem C.I.A. und eine Steuerung der restlichen C.I.A.-Designs möglich.
Die **PCI-Control-Unit** wurde während dieser Arbeit entwickelt und stellt die Schnittstelle zwischen dem PCI-Bus und dem User dar. Das Design ist nicht auf den NIOS beschränkt, sondern besitzt ein bewußt möglichst einfach gehaltenes Interface. Ein Eingangs-, ein Ausgangs- und ein Kontrollregister steuern die komplette Datenübertragung zum PCI-Bus, im Falle des NIOS geschieht die Anbindung durch 3 PIOs. Es werden alle 32-bit Single-Zyklen unterstützt: Memory-Read, Memory-Write, I/O-Read, I/O-Write, CFG-Read sowie CFG-Write. Zusätzlich kann der Status des PCI-Cores ausgelesen werden. Das Design wurde sowohl unter ModelSim mit einer PCI-Testbench getestet, als auch unter realen Bedingungen in einem Test-Rechner. Hierbei wurden ein Tracer zum Mitschneiden der PCI-Bussignale verwendet. Alle

Transaktionen werden laut Tracer korrekt abgewickelt, die Überprüfung der übertragenen Daten wurden jedoch nur bei CFG-Read, Memory-Read und Memory-Write durchgeführt, da diese Zyklen zur Kontrolle des Hosts ausreichen.
Das **C.I.A.-Board** stellt die eigentliche Hardware-Plattform dar. Das Board beinhaltet alle Komponenten zur Realisierung eines Systems : einen FPGA für die Logik, SRAM, FLASH, SDRAM-Interface, Ethernet, sowie eine Vielzahl an Schnittstellen. Sein Einsatzbereich ist nicht auf den C.I.A. beschränkt, sondern es bietet eine generelle, gut ausgestattete Entwicklungsumgebung für Prototypen im Bereich FPGA-Design. Das Board wurde am Lehrstuhl für Technische Informatik entwickelt, insbesonders von Deyan Atanasow (Scematics) und Volker Kiworra (PCB-Design). Die Funktionalität des einzelnen Komponenten wurden mit Hilfe des NIOS überprüft, indem die Komponenten in das System integriert wurden. Entsprechende Test-Designs finden sich auf der CD.

Die Ersetzung der VGA-Karte und die Anbindung der Designs von Stefan Philipp konnten im Rahmen dieser Diplomarbeit aus Zeitgründen nicht mehr durchgeführt werden, da das Testen des C.I.A.-Prototypen sehr viel Zeit in Anspruch genommen hat. Die Anbindung der übrigen C.I.A.-Designs an den NIOS sollte keine Probleme bereiten, da alle Schnittstellen in der Diplomarbeit von S.Philipp [24] gut dokumentiert sind. Für das VGA-Design stellt sich die Frage, inwieweit es Sinn macht. Für Motherboards im Serverbereich zeichnet sich ein Trend ab, die VGA-Karte durch einen Onboard-Chip zu ersetzten. Der C.I.A. müßte also nur den Bildschirm-Speicher über PCI auslesen und exportieren, was heute schon möglich ist. Zusammen mit den übrigen C.I.A.-Designs sollte es damit möglich sein, einen Host zu booten, sein BIOS zu konfigurieren und ein Betriebssystem aufzuspielen, bzw. ein Diagnose-Programm auszuführen. Alle hardwareseitigen Anforderungen an den C.I.A. sind damit erfüllt, lediglich ein praktikables Software-Interface steht noch aus. Dessen Realisierung kann beispielsweise in einem oder mehreren Software-Praktikas geschehen, oder auch in einer eigenen Diplomarbeit.

# Anhang A

# CD-Rom

Die beiliegende CD-ROM enthält neben eine Postscript-Version dieser Arbeit (*/thesis*) die komplette Dokumentation des NIOS, $\mu$C-Linux und C.I.A.-Boards (*/doc*), sowie die Designs (*/design*). Ein ausführliches Inhaltsverzeichnis und vertiefende Informationen finden sich in den README Dateien.

# Anhang B

# NIOS-Tools

Hier ein Überblick über die verschiedenen Tools zur Software-Entwicklung für den NIOS.

- **nios-build** : Compiliert, assembliert und linkt C/C++ Sourcecode für den NIOS zu einem ausführbaren .srec-File.
- **nios-elf-as** : GNU Assembler für NIOS
- **nios-elf-gcc** : GNU C/C++ Compiler für NIOS
- **nios-elf-gdb** : GNU Debugger für NIOS
- **nios-elf-ld** : GNU linker für NIOS
- **nios-run** : Terminalprogramm zur Kommunikation mit dem NIOS
- **srec2flash** : Konvertiert .srec-Files in .flash-Files um, welche über den G.E.R.M.S.-Monitor direkt ins FLASH geschrieben werden können.
- **hexout2flash** : Konvertiert .hexout-Files in .flash-Files um,

# Anhang C

# Glossar

- **SOPC-Builder**. JAVA-basiertes Tool zur Generierung eines System-On-a-Programmable-Chip. Der SOPC-Builder erlaubt die Zusammenstellung des NIOS-System Moduls aus konfigurierbaren Komponenten und die Anbindung eigener Hardware an das NIOS-System. Erzeugt wahlweise ein vorsynthetisiertes EDIF-File oder ein AHDL, VHDL oder Verlog-File des NIOS-System Moduls zur weiteren Verwendung.

- **NIOS-System-Modul**. Beinhaltet das komplette NIOS-System, bestehend aus dem NIOS Prozessor, dem AVALON-Bus und den Peripherie-Modulen.

- **PCI-Core**. Konfigurierbares Design, welches eine Schnittstelle zum PCI-Bus bereitstellt. Wird über ein ähnliches Tool wie der SOPC-Builder konfiguriert, den MEGA-WIZARD. Der PCI-Core bietet dem User eine Schnittstelle zur Datenübertragung auf den PCI-Bus und erzeugt auf dem Bus die dafür notwendigen Signale und Timings.

- **Register-File**. Teil der PCI-Control-Unit. Dient zur Speicherung von Adressen und Daten, sowie einem Register zur Kontrolle des PCI-Controllers.

- **PCI-Controller**. Teil der PCI-Control-Unit. Regelt die Datenübertragung zwischen Register-File und PCI-Core und steuert die Transaktionen des PCI-Cores.

- **C.I.A.**. Cluster Interface Adapter. FPGA-basierter, autonomer, netzwerkfähiger Steuerrechner in Form einer PCI-Karte. Wird in einen Host-Rechner gesteckt und kann diesen scannen und kontrollieren.

- **C.I.A.-Board**. PCI-Karte mit der kompletten Systemhardware : FPGA, SRAM, FLASH, ETHERNET, usw.

- **EXCALIBUR Development Board**. Entwicklungs-Board für das NIOS-System. Verfügt über einen ALTERA APEX20K200EFC484-2X FPGA, 256KByte SRAM, 1MByte FLASH, sowie diverse Connectoren.

- **$\mu$C-Linux**. Linux-Betriebssystem für Systeme ohne Memory-Manage-Unit (MMU) wie Embedded Systems oder Mikrocontroller.

- **FGPA**. Field Programmable Gate Arrays. Programmierbare Logikbausteine mit meist hoher Komplexität und feiner Granularität.

- **CPLD**. Complex Programmable Logic Device. Programmierbare Logikbausteine , allerdings grobe Granularität der Logikelemente, meist Macrozellen genannt.

- **JTAG**. Joint Test Action Group. Schnittstelle zum Testen von Bausteinen. Kann bei ALTERA CPLDs und FPGAs zum Programmieren verwendet werden. Besteht aus 4 Anschlüssen : TCK, TMS, TDI, TDO.
- **SRAM**. Statisches RAM, besteht intern aus FlipFlops. Sehr schnell, sehr teuer, einfach Anzusteuern.
- **DRAM**. Dynamisches RAM, besteht intern aus Kapazitäten, welche ihre Ladung mit der Zeit verlieren und deswegen regelmäßig "refreshed" werden müssen. Langsam, billig. Erfordert einen Controller zur Ansteuerung.
- **Configuration-Space**. Speicherbereich eines PCI-Devices, welcher alle nötigen Informationen über das Device und zu seiner Konfigurierung enthält.

# Literaturverzeichnis


[1] ALICE Collaboration. *Technical Design Report of the Inner Tracking System.* CERN/LHCC 99-12, Jun.1999

[2] ALICE Collaboration. *Technical Design Report of the Time Projection Chamber.* CERN/LHCC 2000-001, Jan.200

[3] ALICE Collaboration. *A Transition Radiation Detector for Electron Identifikation within the ALICE Central Detector (TRD).* CERN/LHCC, 99-13.

[4] ALTERA. *Simulating the PCI MegaCore Function Behavioral Models.* Appl. Note 169. ALTERA. Aug.2001

[5] ALICE Collaboration. *Technical Design Report of the Time Of Flight System (TOF).* CERN/LHCC 2000-12, Feb.2000

[6] ALICE Collaboration. *Technical Design Report of the Photon Spectrometer (PHOS).* CERN/LHCC 99-4, Mar.1999

[7] ALICE Collaboration. *Technical Design Report of a High Momentum Particle Identifikation Detector (HMPID).* CERN/LHCC 98-19, Aug.1998

[8] ALICE Collaboration. *Technical Design Report of the Forward Multiplicity Detector based on Micr Channel Plates.* Feb.1999

[9] ALICE Collaboration. *Technical Design Report of the Photon Multiplicity Detector (PMD).* CERN/LHCC 99-32. Sep.1999

[10] ALICE Collaboration. *Centauro And STrange Object Research (CASTOR). A detector for ALICE at very forward rapidity dedicated to the identification and study of 'centaurons' and 'strangelets'.* Technical proposal. ALICE/97-07 Internal Note. Mar.1997

[11] ALICE Collaboration. *Technical Design Report of the Zero Degree Calorimeter (ZDC).* CERN/LHCC 99-5, Mar.1999

[12] Markus Wannemacher. *Das FPGA-Kochbuch.* Internat. Thomson Publ. 1998

[13] ALTERA. *APEX20K Datasheet.*

[14] ALTERA. *NIOS Embedded Processor Software Development Reference Ver.1.1.* ALTERA. Mar.2001

[15] ALTERA. *Application Note AN115.*

[16] ALTERA. *SOPC-Builder PTF File Reference Manual Ver.1.0.* ALTERA

[17] CIRRUS LOGIC. *CS8900A Product Data Sheet.* CIRRUS LOGIC. Apr.2001

[18] ALTERA. *PCI_MT64 Megacore Function, Reference Design.* ALTERA. Nov.2000

[19] ALTERA. *PCI-Testbench.* ALTERA. Aug. 2001

[20] ALTERA. *NIOS Embedded Processor Programmers Reference Manual Ver. 1.1.* ALTERA. Mar.2001

[21] ALTERA. *Avalon Bus Specification Ver. 2.0.* ALTERA. Jan. 2002

[22] PCI Special Interest Group. *PCI Specifikation Revision 2.2.* PCI Special Interest Group. Dez. 1998

[23] ALTERA. *PCI-Megacore Function User Guide.* ALTERA. Aug. 2001

[24] Stepfan Philipp *Entwicklung einer PCI-Karte zur Steuerung und Überwachung nichtlokaler Rechner für den Einsatz in Computerclustern.* Diplomarbeit. KIP 2001.

[25] CYGNUS. *GNUPro Toolkit ver. 99r1.* CYGNUS. 1999

[26] Kernighan/Ritchie. *Programmieren in C.* 2.Ausgabe. Hanser

[27] CYPRESS. *CY7C1383B Datasheet.* CYPRESS Jul.2001

# Abbildungsverzeichnis


1.1 Aufbau des ALICE-Detektors . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1 Konzept des kompletten Systems . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Design-Flow für FPGAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

4.1 EXCALIBUR Development Board . . . . . . . . . . . . . . . . . . . . . . . 20
4.2 SOPC-Builder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3 Das NIOS System-Modul . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.4 NIOS 32b-RISC-Processor . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.5 AVALON-Bus mit einem Master und mehreren Slaves . . . . . . . . . . . . 25

5.1 Initialisierung des $\mu$C-LINUX-SYSTEM . . . . . . . . . . . . . . . . . . . 28
5.2 EXCALIBUR mit $\mu$C-Linux Kit . . . . . . . . . . . . . . . . . . . . . . . 29

6.1 PCI-DESIGN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
6.2 PCI-BUS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
6.3 CFG-Registers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4 Ablauf eines READ-CYCLES . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.5 64-Bit-PCI Master/Target-Megacore . . . . . . . . . . . . . . . . . . . . . 36
6.6 Datenpfade des PCI-Designs . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.7 PCI-Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.8 MasterRead-StateMachine . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.9 MasterWrite-StateMachine . . . . . . . . . . . . . . . . . . . . . . . . . . 45

7.1 Vom NIOS initialisierte Configuration-Read-Zyklen vom TYP-0 . . . . . . . . 48
7.2 PCI-Trace der Configuration-Read-Zyklen . . . . . . . . . . . . . . . . . . 49
7.3 Vom NIOS initialisierte Memory-Read-Zyklen . . . . . . . . . . . . . . . . . 50
7.4 PCI-Trace der Memory-Read-Zyklen . . . . . . . . . . . . . . . . . . . . . 50
7.5 Vom NIOS initialisierte Memory-Write-Zyklen . . . . . . . . . . . . . . . . 51
7.6 PCI-Trace der Memory-Write-Zyklen . . . . . . . . . . . . . . . . . . . . . 51

8.1 Überblick über das C.I.A.-Board . . . . . . . . . . . . . . . . . . . . . . . 53
8.2 C.I.A.-Board - Ressourcen und Signalpfade . . . . . . . . . . . . . . . . . 54
8.3 64-Bit Memory-Unit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
8.4 Signalpfade der 32B-Memory-Unit . . . . . . . . . . . . . . . . . . . . . . 56
8.5 CLOCK-Distribution des C.I.A.-Boards . . . . . . . . . . . . . . . . . . . 58
8.6 FPGA-Konfigurationsmechanismus über FLASH und CPLD . . . . . . . . . . 59
8.7 Das C.I.A.-Board . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

9.1 System-Logik . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
9.2 Aktuelle Implementation der C.I.A.-Logik . . . . . . . . . . . . . . . . . . 62

9.3 32-Bit-Memory-Unit-Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
9.4 Zukünftige Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
9.5 Konzept für zukünftige VGA-Implementation . . . . . . . . . . . . . . . . . . . . 66

# Tabellenverzeichnis

6.1 Signale des PCI-Bus . . . . . . . . . . 33
6.2 NIOS-CONTROL-PIO . . . . . . . . . . 39
6.3 PCI-CTRL-Register . . . . . . . . . . 40
6.4 *lm_tsr* - Local Master Transaction Status Register . . . . . . . . . . 42
6.5 *control_out*-Signale des PCI-Controllers zur Steuerung des Register-File . . . . 43

7.1 Ressourcen-Verbrauch des PCI-Designs . . . . . . . . . . 47